Toward Load-Balanced Redundancy Transitioning for Erasure-Coded Storage

Keyun Cheng, Huancheng Puyang, Xiaolu Li, Patrick P. C. Lee, Yuchong Hu, Jie Li, and Ting-Yi Wu

Abstract—Redundancy transitioning enables erasure-coded storage to adapt to varying performance and reliability requirements by re-encoding data with new coding parameters on-the-fly. Existing studies focus on bandwidth-driven redundancy transitioning that reduces the transitioning bandwidth across storage nodes, yet the actual redundancy transitioning performance remains bottlenecked by the most loaded node. We present BART, a load-balanced redundancy transitioning scheme that aims to reduce the redundancy transitioning time via carefully scheduled parallelization. We show that finding an optimal load-balanced solution is difficult due to the large solution space. Given this challenge, BART decomposes the redundancy transitioning problem into multiple sub-problems and solves the sub-problems via efficient heuristics. We evaluate BART using both simulations for large-scale storage and HDFS prototype experiments on Alibaba Cloud. We show that BART significantly reduces the redundancy transitioning time compared with the bandwidth-driven approach.

1 Introduction

Practical storage systems must distribute data across nodes, yet node failures become prevalent as the storage size continues to scale [15]. *Erasure coding* is a well-known space-efficient redundancy technique for fault tolerance and achieves higher reliability than traditional replication under the same redundancy overhead [42]. In particular, Reed-Solomon (RS) codes [39] are the most popular erasure code construction deployed in production [5], [6], [11], [15], [35], [36]. At a high level, RS codes encode data into multiple erasure-coded *stripes*, each of which is independently encoded/decoded over a collection of blocks. A storage system distributes the stripes across distinct nodes, such that it provides fault tolerance against a subset of lost blocks of each stripe (see §2.1 for details).

The configuration of coding parameters in erasure coding presents a trade-off across the storage capacity, access performance, and fault tolerance subject to different deployment requirements. It is shown that there exists an inherent trade-off in erasure coding between storage overhead and repair performance [13]. On the other hand, traditional erasure coding deployment often uses a single erasure code with fixed coding parameters for all data [5], [6], [11], [15], [35], [36]. Such a "one-size-fits-all" approach leads to inefficient adaptation to the dynamics in workload and reliability requirements. Thus, we argue that production storage systems should support *redundancy transitioning*, which adaptively adjusts the redundancy for erasure-coded stripes by renewing the coding parameters and re-encoding the currently stored stripes with the new coding parameters. Redundancy transitioning has recently received significant attention from the storage community in response to workload dynamics [49], changes in disk reliability [23]–[25], and the construction of wide-stripe codes [18], [26] (see §2.2 for detailed motivation).

- *This work was supported in part by National Natural Science Foundation of China (62302175, 62272185) and Research Grants Council of Hong Kong (AoE/P-404/18).*

- *Keyun Cheng, Huancheng Puyang, and Patrick P. C. Lee are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong (E-mails: {kycheng, hcpuyang22, pclee}@cse.cuhk.edu.hk).*
- *Xiaolu Li and Yuchong Hu are with the School of Computer Science at Huazhong University of Science and Technology, China (E-mails: {lixl666, yuchonghu}@hust.edu.cn).*
- *Jie Li and Ting-Yi Wu are with Huawei Technologies Co., Ltd., Hong Kong (E-mails: jieli873@gmail.com and wu.ting.yi@huawei.com).*
- *Corresponding author: Patrick P. C. Lee.*

Despite the significance, redundancy transitioning incurs substantial bandwidth overhead in a distributed environment, as it retrieves the currently stored stripes from different nodes for re-encoding and distributes the re-encoded data to a new set of nodes. Existing studies focus on reducing the bandwidth [19], [32], [33], [51] during the redundancy transitioning process. Unfortunately, even though the overall transitioning bandwidth is mitigated, we observe that the actual redundancy transitioning performance remains bottlenecked by the most loaded node (§2.3). Thus, in addition to mitigating the bandwidth, we should achieve *load balancing* across the nodes during redundancy transitioning, so as to reduce the redundancy transitioning time. However, designing a load-balanced redundancy transitioning solution is non-trivial, as redundancy transitioning is inherently a complex system operation with multiple steps that require careful scheduling (§2.2), and the scheduling complexity increases tremendously with the number of stripes and nodes in large-scale storage.

To this end, we propose BART, a load-balanced redundancy transitioning scheme for large-scale erasure-coded storage, with the objective of minimizing the overall redundancy transitioning time through carefully scheduled parallelization. In this work, we address the *merge regime* of the code conversion problem [34], also known as *stripe merging* [45], [51], to facilitate redundancy transitioning. In the merge regime, we aggregate multiple small-size stripes into a single large-size stripe, so as to reduce redundancy overhead while preserving the same number of tolerable block failures in the large-size stripe. We apply *parity merging* [34] in aggregation by retrieving the existing parity blocks of the small-size stripes to compute the new parity blocks for the large-size stripe, as it incurs less bandwidth compared to retrieving the data blocks of the small-size stripes for parity computations (§2.2).

Our contributions are summarized as follows:

- We formulate the redundancy transitioning problem based on parity merging [34] and show that finding an optimal load-balanced redundancy transitioning solution is non-trivial due to the significantly large solution space.
- We design BART, which decomposes the redundancy transitioning problem into three sub-problems and solves the sub-problems by efficient heuristics, so as to achieve load balancing, while keeping the transitioning bandwidth low. BART addresses both homogeneous (i.e., link capacities are identical) and heterogeneous (i.e., link capacities are different) environments.
- We evaluate BART in two aspects: simulations for large-scale storage, and prototype experiments on Alibaba Cloud [1]. Our simulations show that compared with the bandwidth-driven approach that aims to minimize the transitioning bandwidth, BART reduces the maximum load by up to 45.3% in a large-scale storage setting, while maintaining similar transitioning bandwidth. We also prototype BART on Hadoop 3.3.4 HDFS [3]. Our prototype experiments show that our BART prototype reduces the transitioning time by up to 25.9% compared with the bandwidth-driven approach.

We release the source code of BART (both the simulator and prototype) at **https://github.com/keyuncheng/BART**.

2 Background and Motivation

2.1 Basics of Erasure Coding

We consider a distributed storage system (e.g., GFS [16] and HDFS [41]) that stores data as fixed-size *blocks* across multiple *nodes*. To mitigate the I/O overhead, each block often has a large size (e.g., 64 MiB [16] or 256 MiB [35]). All read/write operations are performed in units of blocks. Erasure coding is performed on a collection of blocks for fault tolerance.

In this work, we focus on RS codes [39] as the erasure coding construction (§1). RS codes are constructed by two configurable parameters k and m. A (k, m) RS code encodes a set of k original uncoded blocks (called *data blocks*) into m coded blocks (called *parity blocks*), and the set of $k+m$ data/parity blocks forms a *stripe*. RS codes have the following practical properties: (i) *parameter generality*, i.e., RS codes support general parameters k and m; (ii) *storage optimality*, i.e., RS codes allow any k out of $k+m$ blocks of a stripe to reconstruct all original k data blocks (i.e., providing fault tolerance against any m lost blocks) and maintain the minimum storage redundancy (a.k.a. the *maximum distance separable (MDS) property*); and (iii) *systematic coding*, i.e., RS codes can keep the k data blocks within each stripe for direct access.

Mathematically, RS codes encode parity blocks as follows. Let $D_1, D_2, \cdots, D_k$ be the k data blocks and $P_1, P_2, \cdots, P_m$ be the m parity blocks of a stripe of a (k, m) RS code. Each parity block is computed from a linear combination of the k data blocks of the same stripe using Galois field arithmetic. In this work, we consider systematic Vandermonde-based RS codes [37], in which each parity block P_i $(1 \leq i \leq m)$ is computed as:

$$P_i = \sum\nolimits_{j=1}^{k} i^{j-1} D_j, \quad \text{where } 1 \leq i \leq m. \tag{1}$$

Systematic Vandermonde-based RS codes support efficient parity computation in redundancy transitioning (§2.2), yet they generally cannot maintain the MDS property for all parameters (k, m) under a finite Galois field size [19], [37]. In this work, we assume that the field size is sufficiently large, such that the MDS property of systematic Vandermonde-based RS codes still holds for a wide range of parameters (e.g., for any k and $m \leq 3$ under the field size 256 [4]).

2.2 Redundancy Transitioning

Redundancy transitioning refers to the process of changing the coding parameters (k, m) and hence adjusting the redundancy of erasure-coded stripes. We provide several motivating scenarios for which redundancy transitioning is suitable.

- **Workload changes.** Skewness is observed in the access patterns in production storage systems [10], [22], which contain small fractions of hot data that is frequently accessed and large fractions of cold data that is rarely accessed but needs to be persistently stored. Storage systems can adaptively switch between two erasure codes for different access patterns [49], where hot data is encoded with a high-redundancy code for high reconstruction performance, while cold data is encoded with a low-redundancy code for persistence with sub-par reconstruction performance.
- **Reliability changes.** Disk reliability in production changes over the disk lifetime, where a disk shows a higher failure rate in its early and wear-out stages and a lower failure rate in its middle stages [25]. Prior studies propose disk-adaptive redundancy [23]–[25] to re-encode the middle-stage data with low-redundancy codes and re-encode again the wear-out-stage data with high-redundancy codes, so as to achieve low storage overhead in the whole storage system while satisfying the minimum fault tolerance requirement.
- **Wide stripes.** To reduce operational costs, recent studies from both industry [9], [26] and academia [18] explore wide stripes for the highest possible storage savings, in which the stripe size $k+m$ is extremely large, while m is kept small. Wide stripes significantly increase the reconstruction cost due to the ultra-low redundancy [13], so a storage system can first encode newly written data that is likely to be frequently accessed into narrow stripes, followed by re-encoding the data into wide stripes as it ages [51].

There are two classes of redundancy transitioning, namely *scaling* [19], [21], [46]–[48] and *code conversion* [31]–[34], [45], [51]. Scaling re-encodes erasure-coded stripes with new coding parameters and re-distributes the blocks across all nodes, such that the blocks are *evenly* distributed across all nodes; in some cases, it also enforces the uniform distribution of data and parity blocks [19]. In contrast, code conversion also re-encodes and re-distributes blocks, but does not require that all blocks be evenly distributed. Compared with scaling, code conversion is more appealing to practical deployment for its simplicity (e.g., without the need for data rebalancing) [34]. In this work, we focus on code conversion.

Furthermore, we focus on the *merge regime* of code conversion [34], in which λ (k, m) *input stripes* are merged into a single $(\lambda k, m)$ *output stripe*, where $\lambda \geq 2$, while providing the same fault tolerance against any m lost blocks. The

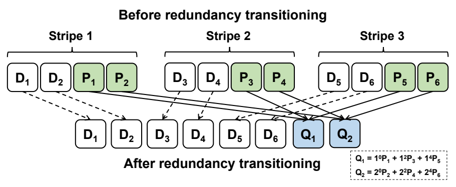

Before redundancy transitioning
Stripe 1
Stripe 2
Stripe 3
After redundancy transitioning
$Q_1 = 1^0P_1 + 1^2P_3 + 1^4P_5$
$Q_2 = 2^0P_2 + 2^2P_4 + 2^4P_6$

Figure 1: Example of $(2, 2, 3)$ parity merging.

merge regime addresses the scenario where storage-efficient persistence is much more critical than access performance (e.g., for cold data), by converting high-redundancy narrow stripes into low-redundancy wide stripes [9], [18], [26], [51].

There are two code conversion approaches for the merge regime, namely *re-encoding* and *parity merging*. To construct new parity blocks for the output stripe, re-encoding retrieves all λk data blocks from the λ input stripes to compute the new m parity blocks for the output stripe, while parity merging retrieves the λm parity blocks from the λ input stripes to compute the new parity blocks for the output stripe. If $k \geq m$ (which is commonly true in practice for low redundancy overhead), then parity merging always has lower bandwidth than re-encoding. Note that parity merging works only if the new parity blocks can be expressed as a function of the parity blocks of the stripes being merged, while Vandermonde-based RS codes satisfy this property (see below); other code constructions also exist and support parity merging [34]. In this work, we focus on parity merging under Vandermonde-based RS codes.

We elaborate on applying parity merging to the merge regime of code conversion as follows. We consider a (k, m, λ) parity merging problem, where λ (k, m) input stripes are merged into a $(\lambda k, m)$ output stripe, where the stripes before and after merging are still encoded under Vandermonde-based RS codes. We augment the notation in §2.1 for multiple stripes, and let $D_{(l-1)k+j}$ be the j^{th} data block and $P_{(l-1)m+i}$ be the i^{th} parity block in the l^{th} input stripe, where $1 \leq j \leq k$, $1 \leq i \leq m$, and $1 \leq l \leq \lambda$. For the l^{th} stripe, each parity block is computed as:

$$P_{(l-1)m+i} = \sum\nolimits_{j=1}^{k} i^{j-1} D_{(l-1)k+j}, \quad \text{where } 1 \leq i \leq m. \quad (2)$$

Let Q_i be the i^{th} new parity block of the output stripe, where $1 \leq i \leq m$. Each parity block of the output stripe is computed as:

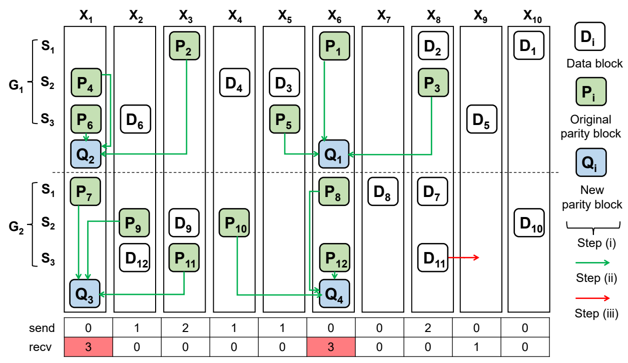
$$Q_i = \sum\nolimits_{j=1}^{\lambda k} i^{j-1} D_j, \quad \text{where } 1 \leq i \leq m. \quad (3)$$

We can relate Equations (2) and (3) as follows:

$$\begin{aligned} Q_i &= \sum\nolimits_{l=1}^{\lambda} i^{(l-1)k} \Big(\sum\nolimits_{j=1}^{k} i^{j-1} D_{(l-1)k+j}\Big) \\ &= \sum\nolimits_{l=1}^{\lambda} i^{(l-1)k} P_{(l-1)m+i}, \quad \text{where } 1 \leq i \leq m. \quad (4) \end{aligned}$$

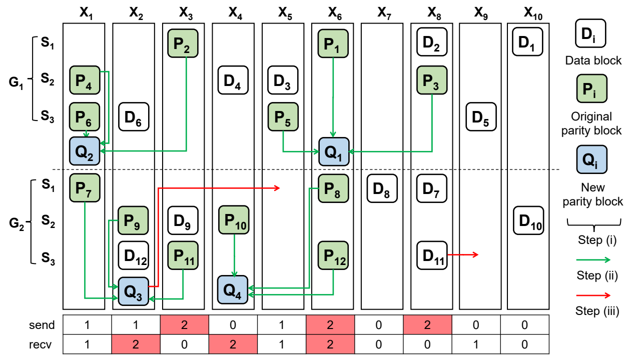
Thus, each new parity block Q_i can be computed from the i^{th} parity blocks of the λ input stripes. Figure 1 shows an example of $(2, 2, 3)$ parity merging, where three (2,2) input stripes are merged into one (6,2) output stripe.

Redundancy transitioning in large-scale storage systems involves a large number of stripes. We define a *stripe group* as the set of λ input stripes in a (k, m, λ) parity merging operation, and there are multiple stripe groups under parity merging. We divide the parity merging operation into three steps: (i) *stripe group construction*, which assigns currently stored stripes into different stripe groups; (ii) *parity block generation*, which merges parity blocks in each stripe group to generate new parity blocks; and (iii) *stripe re-distribution*, which re-distributes the blocks of all output stripes to ensure that the λk data blocks and the m new parity blocks of each output stripe are stored in distinct nodes for fault tolerance. Note that the three steps are essential and interdependent, working together to produce the output stripes while ensuring fault tolerance in distributed storage (§3).



	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
send	0	1	2	1	1	0	0	2	0	0
recv	3	0	0	0	0	3	0	0	1	0

(a) Bandwidth-driven approach



	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}
send	1	1	2	0	1	2	0	2	0	0
recv	1	2	0	2	1	2	0	0	1	0

(b) Load-balanced approach

Figure 2: Motivating examples of $(2, 2, 3)$ parity merging.

2.3 Motivation and Challenges

Redundancy transitioning triggers substantial bandwidth, so practical storage systems often run redundancy transitioning as background tasks with limited system resources [24]. Thus, it is critical to reduce the duration of redundancy transitioning to mitigate the interference to foreground operations. To motivate, we consider the *bandwidth-driven* redundancy transitioning scheme (e.g., [32], [33], [45], [51]) that aims to minimize the redundancy transitioning bandwidth. We consider a homogeneous environment, in which all link capacities are identical. We show via examples that although the bandwidth-driven approach reduces the transitioning bandwidth, it is not necessarily load-balanced. Here, we consider (2,2,3) parity merging for six input stripes stored in 10 nodes (denoted by $X_1, X_2, \cdots, X_{10}$).

Example of bandwidth-driven redundancy transitioning. Figure 2(a) shows bandwidth-driven redundancy transitioning, which merges the stripes with limited transitioning bandwidth (the design is based on bandwidth-driven stripe group construction, which is further explained in §4.1). For stripe group construction, we assign the six input stripes into two stripe groups, denoted by G_1 and G_2. For parity block generation, we generate the new parity blocks Q_1, Q_2, Q_3, and Q_4 via parity merging (denoted by Step (ii)). Note that a node needs to retrieve the original parity blocks from other nodes for parity block generation; for example, X_6 retrieves P_3 and P_5 from X_8 and X_5, respectively, and performs parity merging on P_1 (which is locally stored), P_3, and P_5 to form Q_1. For stripe re-distribution, some blocks are sent to other nodes for fault tolerance (denoted by Step (iii)); for example, X_8 sends D_{11} to X_9, so that all data and parity blocks of the output stripe in G_2 are stored in distinct nodes. Overall, the transitioning bandwidth is seven blocks (i.e., the total number of blocks being sent/received), while the most loaded nodes are X_1 and X_6, both of which retrieve three blocks.

Example of load-balanced redundancy transitioning. Figure 2(b) shows load-balanced redundancy transitioning, based on the same stripe groups as Figure 2(a). The process of G_1 remains unchanged. For G_2, X_2 retrieves P_7 and P_{11} from X_1 and X_3, respectively, and performs parity merging on P_7, P_9 (locally stored), and P_{11} to form Q_3. X_2 also sends Q_3 to X_5 for stripe re-distribution. Overall, the transitioning bandwidth is nine blocks (two more blocks than the bandwidth-driven approach), yet the maximum load reduces to two blocks (note that X_6 can send and receive two blocks simultaneously in full-duplex mode, so its load remains two blocks).

3 Problem Formulation and Analysis

We consider (k, m, λ) parity merging (§2.2) and pose the load-balancing problem as follows. Suppose that there are M input stripes that are randomly stored in N nodes, each of which can simultaneously send and receive data in full-duplex mode. We assume that M is divisible by λ, where the input stripes can be evenly divided into $\frac{M}{\lambda}$ stripe groups, such that all data are re-encoded in new coding parameters after redundancy transitioning. If M is not divisible by λ (i.e., the last stripe group has fewer than λ input stripes), we can add dummy zero-padded input stripes (i.e., all data and parity blocks are zeros) to the last stripe group to allow λ stripes for merging. Note that the dummy input stripes do not need to be physically stored, and hence they do not incur any storage overhead.

We focus on a distributed environment, where the redundancy transitioning performance is bottlenecked by network transmissions across nodes (as also justified in our Alibaba Cloud evaluation (§5.2)). In our problem formulation and system design, we assume a homogeneous setting where the link capacities across nodes are identical (i.e., all nodes can send or receive data at the same rate). We extend our system design to address heterogeneous environments with different link capacities in §4.4.

Our objective is to find a redundancy transitioning solution that minimizes the *maximum transitioning load (MTL)*, defined as the maximum number of blocks that a node sends or receives across all N nodes. For example, Figures 2(a) and 2(b) show two redundancy transitioning solutions, where their MTLs are three and two, respectively. By minimizing the MTL, we can minimize the redundancy transitioning time.

Finding an optimal redundancy transitioning solution is non-trivial. Recall that each solution needs to perform three steps: stripe group construction, parity block generation, and stripe re-distribution (§2.2). We show in the following that each step has numerous possible choices that enlarge the solution space.

First, we analyze the number of choices for stripe group construction. Given M input stripes, we first select λ out of the M stripes to form a group, followed by λ out of the $M-\lambda$ stripes to form the next group, and so on. In total, there are $\prod_{i=1}^{M/\lambda}\binom{M-\lambda(i-1)}{\lambda} = \frac{M!}{(\lambda!)^{M/\lambda}}$ possibilities of assigning M input stripes into $\frac{M}{\lambda}$ stripe groups.

Second, for parity block generation, we assign a node (called *encoding node*) out of the N nodes for each new parity block of an output stripe. Since there are m parity blocks for each of the $\frac{M}{\lambda}$ output stripes, there are a total of $N^{Mm/\lambda}$ choices of encoding nodes.

Finally, for stripe re-distribution, some nodes need to send the data/parity blocks to other nodes to maintain fault tolerance (i.e., all blocks of the same output stripe are stored in distinct nodes). Depending on the placement of the data blocks and the choices of encoding nodes, the number of blocks being re-distributed varies. In the ideal case, we do not need to re-distribute any block, if the data and parity blocks of each output stripe are already in distinct nodes after parity block generation.

Thus, the lower bound of the solution space for load-balanced (k, m, λ) parity merging is $\frac{M!}{(\lambda!)^{M/\lambda}} \times N^{Mm/\lambda}$. The number is huge, even for small N and M. For example, for $(2, 2, 3)$ parity merging with $M = 30$ input stripes and $N = 10$ nodes, there are at least 4.39×10^{44} possible solutions. It is infeasible to perform a simple brute-force search to find an optimal solution.

Note that the stripe group construction step can be further reduced to a k-means cluster problem [30], which is known to be NP-hard. Intuitively, we can treat each of M input stripes as a data point in some N-dimensional space corresponding to N nodes, and consider a stripe group as a cluster comprising λ data points. For load-balanced redundancy transitioning, our goal is to find $\frac{M}{\lambda}$ disjoint clusters to minimize the sum of a cost function, where the cost function can be defined based on the MTL. The subsequent parity block generation and stripe re-distribution steps can also affect the MTL of the resulting redundancy transitioning solution, thereby making the redundancy transitioning problem intractable.

4 BART Design

We present BART, a load-balanced redundancy transitioning scheme for erasure-coded storage. BART addresses (k, m, λ) parity merging to mitigate not only the MTL, but also the transitioning bandwidth as in prior bandwidth-driven approaches [32], [33], [45], [51], so as to reduce the redundancy transitioning time. Its core idea is to decompose the parity merging problem into the sub-problems of stripe group construction, parity block generation, and stripe re-distribution (which correspond to the three steps of a parity merging operation (§2.2)), and solve the subproblems with efficient heuristics. Specifically, BART first performs bandwidth-driven stripe group construction to mitigate the transitioning bandwidth and hence simplify load balancing in subsequent operations (§4.1). It then formulates bipartite-graph semi-matching problems for load balancing in both parity block generation (§4.2) and stripe re-distribution (§4.3). Our discussion mainly focuses on homogeneous environments, and we further extend BART to address heterogeneous environments (§4.4).

4.1 Stripe Group Construction

Overview. BART starts with bandwidth-driven stripe group construction to mitigate the transitioning bandwidth, and later load-balances the bandwidth across nodes in parity block generation (§4.2) and stripe re-distribution (§4.3). It adapts the idea of StripeMerge [51] to partition the input stripes into multiple stripe groups, such that each stripe group only transmits a limited number of blocks for parity merging. Note that StripeMerge only considers $\lambda = 2$, and BART extends its design for $\lambda \geq 2$.

Partial stripe groups. BART's core idea is to first form $\frac{M}{\lambda}$ *partial stripe groups (PSGs)* with two input stripes each, and then iteratively add an input stripe to each PSG until the PSG has λ input stripes. We define a γ*-PSG* as a PSG with γ input stripes (where $2 \leq \gamma \leq \lambda$), which can form a $(\gamma k, m)$ output stripe. We define the *transitioning cost* as the number of blocks being transferred in parity block generation and stripe re-distribution. In each iteration, BART aims to find the γ-PSGs whose transitioning costs are as small as possible.

We compute the transitioning cost for a γ-PSG as follows. For each new parity block to be generated, we choose the node that stores the most original parity blocks as the encoding node. After all new parity blocks are generated, we assume that the blocks are randomly re-distributed to other nodes for fault tolerance. We then obtain the transitioning cost. Here, we do not perform load balancing, and we defer the load-balancing steps to §4.2 and §4.3.

Note that the transitioning cost is an integer ranging from 0 to $(\gamma - 1)(k + m)$. In the ideal case, all parity blocks are *aligned* (i.e., the i^{th} parity blocks of all input stripes reside in the same node, which becomes an encoding node and generates the new i^{th} parity block of the output stripe) and all $k\gamma$ data blocks are stored in distinct nodes, while in the worst case, all $m\gamma$ parity blocks are in distinct nodes and all $k\gamma$ data blocks are stored in the same set of k nodes. For example, in Figure 2(a), the transitioning costs of G_1 and G_2 are three and four, respectively.

Heuristic. Algorithm 1 shows our heuristic that proceeds as follows.

- *Step 1 (Initialization):* Initially, we store all M input stripes into an input set $\mathcal{S}$ and set $\gamma = 2$. We consider all $\binom{M}{2} = \frac{M(M-1)}{2}$ choices of 2-PSGs. We sort all candidate 2-PSGs in ascending order of transitioning costs. We select the first $\frac{M}{\lambda}$ *non-overlapping* 2-PSGs in the sorted list into an output set $\mathcal{R}$, such that the input stripes of each selected 2-PSG will not be included in the subsequently selected 2-PSGs. We also remove the $\frac{2M}{\lambda}$ input stripes of the selected 2-PSGs from $\mathcal{S}$ (lines 1-6).

Algorithm 1 Stripe Group Construction

```
Input: $M$ input stripes stored in $\mathcal{S}$
Output: Constructed stripe groups stored in $\mathcal{R}$
1: // Step 1
2: Initialize $\gamma = 2$
3: Enumerate $\binom{M}{2}$ 2-PSGs
4: Sort the enumerated 2-PSGs in ascending order of transitioning costs
5: Initialize $\mathcal{R}$ as the first $\frac{M}{\lambda}$ non-overlapping 2-PSGs in the sorted list
6: Remove the $\frac{2M}{\lambda}$ input stripes of the selected 2-PSGs from $\mathcal{S}$
7: for $\gamma = 3$ to $\lambda$ do
8:     // Step 2
9:     Enumerate $\frac{M^2(\lambda-\gamma+1)}{\lambda^2}$ $\gamma$-PSGs by adding each input stripe in $\mathcal{S}$ to each $(\gamma-1)$-PSGs in $\mathcal{R}$
10:    // Step 3
11:    Sort the enumerated $\gamma$-PSGs in ascending order of transitioning costs
12:    Reset $\mathcal{R} = \emptyset$
13:    Set $\mathcal{R}$ as the first $\frac{M}{\lambda}$ non-overlapping $\gamma$-PSGs in the sorted list
14:    Remove the $\frac{M}{\lambda}$ input stripes of the selected $\gamma$-PSGs from $\mathcal{S}$
15: end for
```

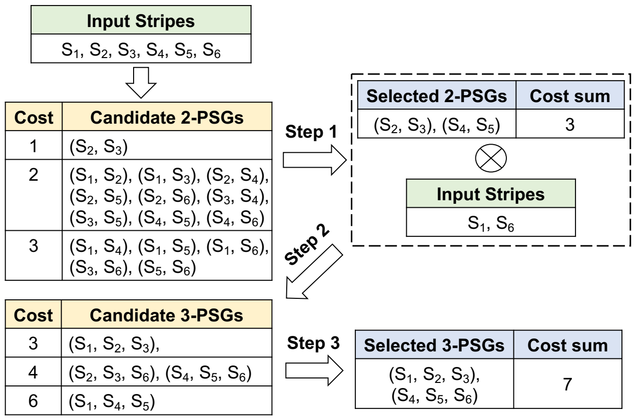


Figure 3: Stripe group construction with our heuristic.

- *Step 2 (Enumeration):* For $\gamma > 2$, we add each input stripe in $\mathcal{S}$ (where $|\mathcal{S}| = M - \frac{M(\gamma-1)}{\lambda}$) into the $(\gamma - 1)$-PSGs in $\mathcal{R}$ (where $|\mathcal{R}| = \frac{M}{\lambda}$) and form an enumerated set of $\frac{M^2(\lambda-\gamma+1)}{\lambda^2}$ of γ-PSGs (lines 8-9).
- *Step 3 (Selection):* We sort the enumerated set of γ-PSGs in ascending order of transitioning costs. We reset $\mathcal{R} = \emptyset$ and select the first $\frac{M}{\lambda}$ non-overlapping γ-PSGs in the sorted list into $\mathcal{R}$, such that the input stripes of each selected γ-PSG will not be included in the subsequently selected γ-PSGs. We also remove the $\frac{M}{\lambda}$ input stripes from $\mathcal{S}$ (lines 10-14).

We repeat Steps 2 and 3 for $2 < \gamma \leq \lambda$ (i.e., $\lambda - 2$ iterations). The final set of stripe groups (i.e., λ-PSGs) is stored in $\mathcal{R}$.

Example. Figure 3 depicts the design of stripe group construction in BART, based on the $(2, 2, 3)$ parity merging example in Figure 2. Let $S_1, S_2, \cdots, S_6$ be the $M = 6$ input stripes. We first examine $\binom{6}{2} = 15$ possible 2-PSGs. We then select $\frac{M}{\lambda} = 2$ 2-PSGs: (S_2, S_3) and (S_4, S_5), whose transitioning costs are one and two, respectively. We then enumerate four 3-PSGs by adding S_1 and S_6 into (S_2, S_3) and (S_4, S_5). Finally, we choose (S_1, S_2, S_3) and (S_4, S_5, S_6) as the stripe groups, whose total transitioning cost is seven.

Complexity analysis. We study the time complexity of our heuristic. Since we know the range of transitioning costs, we can perform counting sort on the set of γ-PSGs, so the sorting complexity is linear with the number of elements in the set (assuming that the range of transitioning costs is small). For Step 1, the complexity is $O(M^2)$ (i.e., $O(M^2)$ for counting sort plus $O(M^2)$ for selecting $\frac{M}{\lambda}$ 2-PSGs from the sorted list). For Step 2, the complexity is $O(\frac{M^2(\lambda-\gamma+1)}{\lambda^2})$ for enumerating γ-PSGs. For Step 3, the complexity is $O(\frac{M^2(\lambda-\gamma+1)}{\lambda^2})$ (for counting sort plus selecting $\frac{M}{\lambda}$ γ-PSGs). Since Steps 2 and 3 are repeated $\lambda - 2$ times, the total complexity of Steps 2 and 3 is $O(M^2)$. Thus, the overall complexity of the heuristic is $O(M^2)$.

Given the complexity of $O(M^2)$, when M is significantly large (e.g., with millions of stripes), performing bandwidth-driven stripe group construction directly over M stripes can significantly increase the time of stripe group construction for solution generation. Nevertheless, the solution generation process can be done offline based on the current data placement before redundancy transitioning is executed, so it does not affect the actual redundancy transitioning performance. Also, we can partition the input stripes into multiple batches and process the stripes on a per-batch basis. For example, considering $(6, 3, 3)$ parity merging with a batch of $M = 12000$ input stripes and a block size of 64 MiB [16] (equivalent to a total of 6.59 TiB of data and parity blocks), BART can generate the solution within 50.4 s in our simulations (§5.1).

4.2 Parity Block Generation

Overview. BART models the selection of encoding nodes for parity block generation as a bipartite-graph semi-matching problem that addresses the traffic sent and received by all nodes for load balancing. We start with a weighted bipartite graph that comprises a set of $\frac{Mm}{\lambda}$ *block vertices* that represent the new parity blocks of the $\frac{M}{\lambda}$ output stripes and a set of N *node vertices* that represent the N available nodes. Every block vertex is connected to every node vertex with an edge, whose weight represents the number of original parity blocks to retrieve if the node (associated with the node vertex) becomes the encoding node that generates the new parity block (associated with the block vertex).

The selection of encoding nodes is modeled as a *semi-matching* of the bipartite graph [17]. A semi-matching comprises a set of edges that match every block vertex to a node vertex, meaning that the corresponding node is the encoding node for the corresponding new parity block. Note that a node vertex can be matched with multiple block vertices (i.e., multiple new parity blocks can be generated independently at the same node). BART aims to find a semi-matching whose corresponding traffic is balanced across the nodes.

To help the selection of encoding nodes, we use a *traffic table* to record how many blocks each node sends or receives. There are three components that contribute to network traffic: (i) the number of original parity blocks for parity block generation (denoted by n_p), (ii) the number of data blocks for stripe re-distribution (denoted by n_d), and (iii) the number of new parity blocks for stripe re-distribution (denoted by n_q). For each node X_i ($1 \leq i \leq N$), we record s_i as the sum of n_p, n_d, and n_q being sent by X_i, and record r_i as n_p being received by X_i. Here, we do not consider n_d and n_q being received by X_i (i.e., the received traffic for parity block generation and stripe re-distribution), as they are determined by block placement after *all* encoding nodes are selected. We address them in our load-balancing heuristic in §4.3.

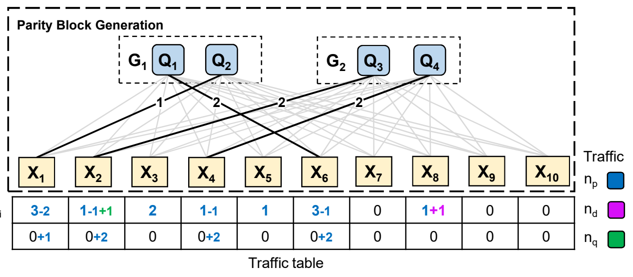
Algorithm 2 Parity Block Generation

Input: A weighted bipartite graph for parity block generation
Output: A semi-matching that represents the selection of encoding nodes

```
1: // Step 1
2: Initialize $s_i$ and $r_i$ (where $1 \leq i \leq N$) in the traffic table
3: // Step 2
4: Initialize an empty set of edges in the the semi-matching
5: for each new parity block $Q$ do
6:     Select the least-loaded node $X_i$ (where $1 \leq i \leq N$) as the encoding node of $Q$ based on the current traffic table
7:     Add the corresponding edge to semi-matching and update the traffic table accordingly
8: end for
9: // Step 3
10: while true do
11:     for each new parity block $Q$ whose current encoding node is $X_i$ do
12:         Substitute $X_j$ for $X_i$ (where $j \neq i$) as the encoding node of $Q$ if it improves the semi-matching
13:         Update the semi-matching and traffic table based on the substitution of encoding node
14:     end for
15:     Break if the semi-matching cannot be improved
16: end while
```

Heuristic. BART adopts a greedy heuristic for the selection of encoding nodes. It updates the traffic table on-the-fly and iterative adjusts the semi-matching based on the traffic table. Algorithm 2 shows our heuristic that proceeds as follows.

- *Step 1 (Initialization of the traffic table):* For $1 \leq i \leq N$, we initialize $r_i = 0$ and s_i as the sum of (i) the number of original parity blocks stored in X_i and (ii) the number of data blocks to re-distribute from X_i for all output stripes. The rationale for s_i is as follows: for (i), X_i needs to send its currently stored original parity blocks to other nodes for parity merging if it is not selected as an encoding node; for (ii), X_i sends all but one data block to other nodes for every output stripe to maintain the fault tolerance of the stripe (lines 1-2).
- *Step 2 (Initialization of the semi-matching):* We start with an empty set of edges in the semi-matching (line 4). For each block vertex, we select a node vertex among all N node vertices as follows. Suppose that we select node X_i as the encoding node for a new parity block (denoted by Q), and let w be the edge weight between the block vertex for Q and the node vertex for X_i in the bipartite graph. Based on the current traffic table, we update the traffic table as follows. First, we add r_i by w, meaning that X_i retrieves w original parity blocks to X_i from other nodes. Second, we subtract s_i by $\lambda - w$, meaning that X_i does not need to send its locally stored $\lambda - w$ original parity blocks to other nodes. Third, we add s_i by one if X_i needs to re-distribute Q to another node. We record the largest entry of the traffic table if X_i is chosen as the encoding node. We repeat the



s_i	3-2	1-1+1	2	1-1	1	3-1	0	1+1	0	0
r_i	0+1	0+2	0	0+2	0	0+2	0	0	0	0

Figure 4: Modeling of parity block generation. The dark edges are the semi-matching, which specifies the encoding nodes for generating the corresponding new parity blocks.

above update process on the current traffic table for every node X_i for $1 \leq i \leq N$, and select the node whose largest entry is the least among all nodes as the encoding node (i.e., the encoding node is currently the least-loaded node) (line 6). If there is a tie, we select the node that transfers the fewest blocks to generate and re-distribute Q. Given the selected encoding node, we add the edge that connects the node vertex and the block vertex to the semi-matching, and update the traffic table accordingly (line 7). We repeat the process for all block vertices and the semi-matching is initialized.

- *Step 3 (Optimization of the semi-matching):* We optimize the semi-matching in multiple iterations. In each iteration, we examine if each block vertex can be matched with another node vertex to "improve" the semi-matching; by "improve", we mean that if we can reduce the value of the largest entry in the traffic table or reduce the number of blocks transferred with the same value of the largest entry. We elaborate on how to check a block vertex as follows. Suppose that X_i is the currently selected encoding node for a new parity block Q. We examine if replacing X_i by another node X_j ($i \neq j$) improves the semi-matching. First, we reset the traffic table by unselecting X_i, where we subtract r_i by w, add s_i by $\lambda - w$, and subtract s_i by one if Q needs to be re-distributed. Next, we update the traffic table by selecting X_j. If substituting X_i with X_j improves the semi-matching, we update the semi-matching with the newly selected node vertex for X_j, and update the traffic table accordingly (lines 12-13). If we cannot improve the semi-matching in an iteration (i.e., we cannot change the current selection for all encoding nodes), the step is finished (line 15).

Example. Figure 4 depicts the selection of encoding nodes, based on the $(2, 2, 3)$ parity merging example in Figure 2. In Step 1, we initialize the traffic table. For example, we set $s_8 = 2$, as X_8 sends P_3 and D_{11} to other nodes. The largest entry of the initialized traffic table is three (i.e., s_1 and s_6). In Step 2, we initialize the semi-matching. For Q_1, we select X_6, which keeps the largest entry as three (i.e., s_1) and transfers two blocks. We add the edge connecting Q_1 and X_6 to the semi-matching, and update the traffic table by adding r_6 by two and subtracting s_6 by one. For Q_2, we select X_1, which reduces the largest entry to two and transfers one block. We proceed for Q_3 and Q_4 and finish the initialization of semi-matching, where the largest entry is two and a total of nine blocks (i.e., $\sum_{i=1}^{10} s_i$) are sent. In Step 3, we examine from Q_1 to Q_4 and find that the semi-matching cannot be improved by changing the current selection of encoding nodes. We finally set X_6, X_1, X_2, and X_4 as the encoding nodes for Q_1, Q_2, Q_3, and Q_4, respectively.

Remarks. BART may slightly increase the transitioning bandwidth over the bandwidth-driven approach (i.e., the total transitioning cost in §4.1). As BART aims for load balancing, some selected encoding nodes may no longer store the most original parity blocks, which incur extra transitioning bandwidth for retrieving the original parity blocks. Nevertheless, our evaluation shows that the transitioning bandwidth overhead of BART remains limited (§5).

Complexity analysis. We analyze the time complexity of our heuristic. For Step 1, the complexity is $O(M(k+m))$ for checking the stripe placements. For Step 2, the complexity is $O(MNm)$, in which for each of $\frac{Mm}{\lambda}$ block vertices, we check N node vertices by updating the traffic table in $O(\lambda)$ time each. For Step 3, the complexity is $O(\alpha MNm)$ (for resetting plus updating the traffic table), where α is the number of iterations. From our evaluation, α is typically small (within 10) and has limited performance overhead (§5). Thus, the overall time complexity is $O(\alpha MNm)$.

4.3 Stripe Re-distribution

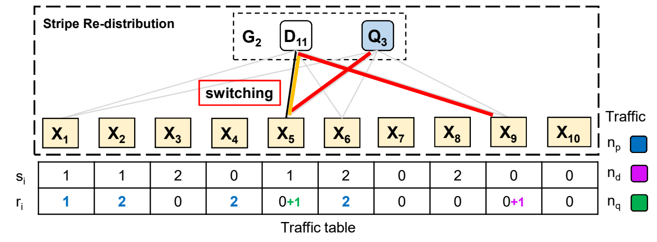
Overview. BART formulates another bipartite-graph semi-matching problem for load-balanced stripe re-distribution. We first create an unweighted bipartite graph based on the stripe placement. The bipartite graph consists of a set of block vertices representing the data and new parity blocks that need to be re-distributed after parity block generation, and a set of N node vertices that represent the N available nodes for stripe re-distribution. We connect a block vertex with a node vertex by an edge if the corresponding block can be re-distributed to the corresponding node, provided that the fault tolerance is maintained.

BART finds a semi-matching of the (unweighted) bipartite graph, such that each edge in the semi-matching means that the corresponding block is re-distributed to the corresponding node. It leverages the Hungarian algorithm [17], which provably finds a semi-matching that minimizes the maximum degree of a node vertex. It adapts the Hungarian algorithm to address the received traffic for parity block generation and stripe re-distribution (§4.2). BART aims to balance the received traffic for each node after all encoding nodes are selected.

Specifically, BART builds on finding *alternating paths* [17]. An alternating path, denoted by $(u_1, v_1, u_2, v_2, \cdots, u_n, v_n)$, is a sequence of distinct vertices in the bipartite graph, where u_i and v_i ($1 \leq i \leq n$) are a block vertex and a node vertex, respectively, and n ($n \leq N$) is the length of the path. It contains an alternate sequence of *unmatched* edges that are not in the semi-matching and *matched* edges that are in the semi-matching. To form an alternating path, we start with a block vertex u_1 that is not connected by any matched edge. We find a node vertex v_1 that is connected to u_1 by an unmatched edge, and add v_1 to the path. We then find a block vertex u_2 that is connected by a matched edge, and add u_2 to the path. We repeat the above process until we reach v_n that is not connected to any block vertex by a matched edge. Here, we assume that an alternating path always ends at a node vertex; in other words, for each block vertex u_i, we can

Algorithm 3 Stripe Re-distribution

```
Input: An unweighted bipartite graph for stripe re-distribution
Output: A semi-matching that represents the solution of stripe
    re-distribution
1: Initialize an empty set of edges in the semi-matching
2: for each block vertex $u$ do
3:     Form all alternating paths starting from $u$
4:     Select the first path whose last node vertex has the
       smallest $r_i$ (where $1 \leq i \leq N$)
5:     Perform switching on the selected alternating path
6:     Update the semi-matching and traffic table based on
       switching
7: end for
```



s_i	1	1	2	0	1	2	0	2	0	0
r_i	1	2	0	2	0+1	2	0	0	0+1	0

Figure 5: Modeling of stripe re-distribution. To re-distribute Q_3, we select the alternating path (Q_3, X_5, D_{11}, X_9), where $r_9 = 0$ is the smallest. We perform switching on the path and increase r_9 by one.

always find a node vertex v_i that is connected to u_i by an unmatched edge. This is true for large-scale storage systems, in which there are sufficient available nodes to which a block can be re-distributed.

A key operation is called *switching*, which converts all unmatched edges to matched edges, and converts all matched edges to unmatched edges, along the alternating path. Switching finds an available node to store the block for the block vertex u_1, while increasing the number of received blocks r_i for the last node vertex v_n by one (for other node vertices in the alternating path, the numbers of received blocks of the nodes remain unchanged).

Heuristic. Algorithm 3 shows our heuristic that proceeds as follows. We start with an empty semi-matching without any matched edge (line 1). For each block vertex u in the bipartite graph, we form all alternating paths starting from u (via breadth-first search), from which we select the first path whose last node vertex (corresponding to node X_i) has the smallest r_i (i.e., X_i is currently the least-loaded node) among all alternating paths (lines 3-4). We perform switching on the selected alternating path, so as to find the available node to store the block associated with the block vertex u. We update the semi-matching with the new set of matched edges from the alternating path after switching, and add r_i by one (lines 5-6). We repeat the above process for all block vertices and construct the semi-matching for stripe re-distribution.

Example. Figure 5 depicts our stripe re-distribution heuristic for the example in Figure 2(b). Suppose that D_{11} and Q_3 from the stripe group G_2 need to be re-distributed, where nodes X_1, X_5, X_6, and X_9 are the available nodes for re-distribution. We form the bipartite graph and start with an empty semi-matching.

First, we find a node to which D_{11} is re-distributed. We form all alternating paths starting from D_{11} (i.e., (D_{11}, X_1), (D_{11}, X_5), (D_{11}, X_6), and (D_{11}, X_9)). We check the current value of r_i corresponding to the last node vertex of each alternating path (i.e., $r_1 = 1$, $r_5 = 0$, $r_6 = 2$, and $r_9 = 0$, respectively). Among the paths, we select (D_{11}, X_5), the first path whose last node vertex has the smallest r_i (i.e., $r_5 = 0$). We perform switching on (D_{11}, X_5), after which we update the semi-matching with the only matched edge (i.e., (D_{11}, X_5)) and increase r_5 by one.

Second, we find a node to which Q_3 is re-distributed. We form all alternating paths starting from Q_3 (i.e., (Q_3, X_1), (Q_3, X_5, D_{11}, X_9), (Q_3, X_6), and (Q_3, X_9)), and check the current value of r_i corresponding to the last node vertex of each alternating path (i.e., $r_1 = 1$, $r_9 = 0$, $r_6 = 2$, and $r_9 = 0$, respectively). We select (Q_3, X_5, D_{11}, X_9), the first path whose last node vertex has the smallest r_i (i.e., $r_9 = 0$). We perform switching on (Q_3, X_5, D_{11}, X_9), after which we update the semi-matching with the new matched edges (i.e., (Q_3, X_5) and (D_{11}, X_9)), and increase r_9 by one. Finally, we re-distribute D_{11} and Q_3 to X_9 and X_5, respectively.

Complexity analysis. We analyze the time complexity of our heuristic. The number of blocks that need to be re-distributed (i.e., the number of block vertices in the bipartite graph) in the worst case is $\frac{M((\lambda-1)k+m)}{\lambda}$, in which each of $\frac{M}{\lambda}$ output stripes re-distributes $(\lambda-1)k$ data blocks when all λk data blocks are stored in the same k nodes and re-distributes all m new parity blocks that are generated on the same k nodes. The total number of edges in the bipartite graph is $\frac{M((\lambda-1)k+m)(N-k)}{\lambda}$, where each block can be re-distributed to $N-k$ available nodes.

For each block vertex, we form alternating paths via breadth-first search, such that each edge in the bipartite graph is visited at most once. The complexity for processing a single block vertex is $O(\frac{M((\lambda-1)k+m)(N-k)}{\lambda})$. Thus, the complexity of our heuristic for processing all block vertices is $O(\frac{M^2((\lambda-1)k+m)^2(N-k)}{\lambda^2})$.

Despite the high (worst-case) complexity, our evaluation shows that the number of blocks to be re-distributed is generally small (§5), as the prior steps of BART (§4.1 and §4.2) already reduce the transitioning bandwidth and hence the number of blocks to be re-distributed. Also, most alternating paths have short lengths, so the time to examine all alternating paths is limited. Overall, the actual performance overhead for stripe re-distribution remains limited (§5).

4.4 BART in Heterogeneous Environments

We further extend BART to address heterogeneous environments, in which case the redundancy transitioning performance is determined by not only the amount of data sent or received in each node, but also the link capacity of each node. Thus, minimizing the MTL in heterogeneous environments may not necessarily minimize the actual redundancy transitioning time.

We extend the heuristics by introducing a *transitioning time table* (in addition to the traffic table) into parity block generation (§4.2) and stripe re-distribution (§4.3). Given the available link capacities of all nodes, each entry of the transitioning time table records the upload and download times for the corresponding node to send and receive the blocks, respectively. BART initializes and updates the entries of the transitioning time table on-the-fly together with the entries of the traffic table. Specifically, let B be the block size, b_i and b'_i be the available upload and download link capacities of node X_i, respectively. The upload time (denoted by t_i) and the download time (denoted by t'_i) of X_i in the transitioning time table are computed as:

$$t_i = \frac{B\times s_i}{b_i}, \;\; t'_i = \frac{B\times r_i}{b'_i}, \;\; \text{where } 1 \leq i \leq M. \qquad (5)$$

where s_i and r_i are the numbers of blocks sent and received by X_i recorded in the traffic table, respectively. BART aims to find a redundancy transitioning solution that minimizes the *maximum data transfer time (MTT)*, defined as the maximum time to send or receive the data across all nodes. Note that in homogeneous environments (where $b_i = b'_i$ for all X_i), BART reduces both MTL and MTT simultaneously.

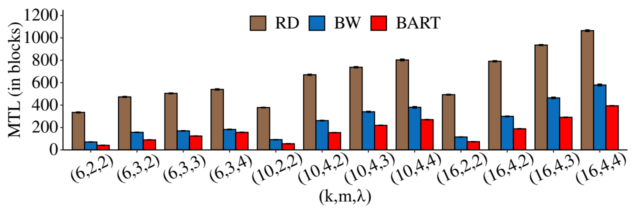
The extended heuristics work as follows. The heuristic for stripe group construction remains unchanged, as its primary objective is to mitigate the overall transitioning bandwidth instead of load balancing. We first focus on parity block generation. In Step 2, for each block vertex, suppose that BART chooses node X_i as the encoding node. After BART updates the traffic table, it also updates the transitioning time table based on Equation (5). It repeats the update process for both the traffic table and transitioning time table for all N nodes, and selects the node whose largest entry of the transitioning time table is the smallest among all nodes. Similarly, in Step 3, BART improves the semi-matching by reducing the value of the largest entry of the transitioning time table. We next focus on stripe re-distribution. For each block vertex, BART forms all alternating paths and selects the first alternating path whose last vertex has the smallest t'_i among all paths. It then performs switching and updates r_i and t'_i accordingly.

4.5 Discussion

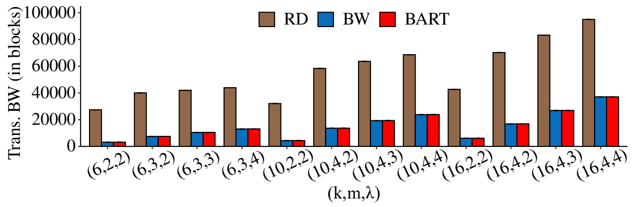
We discuss the performance trade-off between the time complexity of our heuristics and the reduction in both MTL and transitioning bandwidth. For comparison, we consider a randomized approach that performs stripe group construction, parity block generation, and stripe re-distribution randomly. Specifically, we first randomly assign the M stripes to $\frac{M}{\lambda}$ stripe groups, with a time complexity of $O(M)$. For each stripe group, we randomly assign m nodes to generate the new parity blocks, with a time complexity of $O(\frac{Mm}{\lambda})$. Finally, we randomly re-distribute the data and new parity blocks for fault tolerance (i.e., if multiple blocks of a stripe are stored in the same node, they will be randomly re-distributed to other distinct nodes, such that all blocks of a stripe are stored in distinct nodes), with a worst-case time complexity of $O(\frac{(M(\lambda-1)k+m)}{\lambda})$, where we re-distribute $(\lambda-1)k$ data blocks and m new parity blocks; note that random re-distribution cannot balance the received traffic of all nodes. Overall, BART has a larger time complexity than the randomized approach. Nevertheless, our evaluation shows that the randomized approach incurs a high MTL and transitioning bandwidth, while BART effectively reduces both MTL and transitioning bandwidth within a reasonable time (§5.1).

5 Evaluation

We evaluate BART via large-scale simulations (§5.1) and HDFS [3] experiments on Alibaba Cloud [1] (§5.2). We address the following questions: (i) What are the MTL (in homogeneous environments), MTT (in heterogeneous environments), and transitioning bandwidth of BART in large-scale settings? (ii) What is the actual performance of BART in terms of the time of redundancy transitioning in a real network environment?



(a) MTL



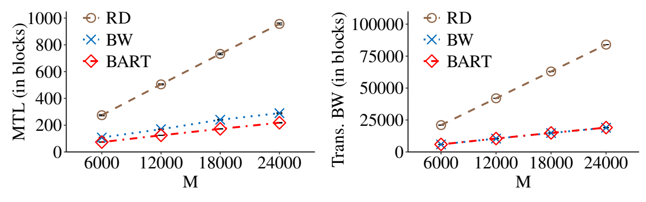
(b) Transitioning bandwidth

Figure 6: Experiment A1: Impact of (k, m, λ) (where (k, m, λ) is arranged in increasing lexicographic order).

5.1 Large-Scale Simulations

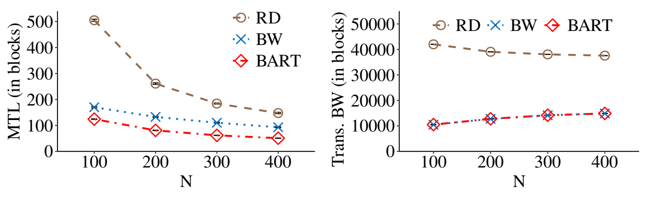
We implement a simulator in C++ with about 3.5 K LoC to evaluate BART in a large-scale storage cluster. We also implement two baselines for comparisons: (i) *RD*, the randomized approach which randomly performs stripe group construction, parity block generation, and stripe re-distribution (§4.5); and (ii) *BW*, the bandwidth-driven approach which performs bandwidth-driven stripe group construction, as well as the assumed steps for parity block generation and stripe re-distribution based on how we compute the transitioning cost without load balancing (§4.1). We conduct simulations on a Ubuntu 20.04 server equipped with an Intel i5-7500 3.4 GHz CPU, 16 GiB RAM, and a 7200 RPM 1 TiB SATA hard disk. We consider different (k, m, λ), M, and N, where some parameters have been studied in literature [34], [53]. Initially, the set of M stripes is randomly distributed across N nodes in a storage cluster. For homogeneous environments (Experiments A1-A4), we measure the MTL and transitioning bandwidth in units of blocks, so the link capacities (which are identical for all links) are not specified; for heterogeneous environments (Experiment A5), we measure the MTT by considering different settings of link capacities. We report the average results over 30 runs, with error bars showing the 95% confidence intervals based on the normal distribution.

Experiment A1: Impact of (k, m, λ). We first study the MTL and transitioning bandwidth of different schemes by varying (k, m, λ). We fix $M = 12000$ stripes (so that M is divisible by $\lambda = 2, 3, 4$) and $N = 100$ nodes. Figure 6 shows the results; note that some of the (k, m) settings are also used in production (e.g., $(6, 3)$ in Google Colossus [49] and $(10, 4)$ in Facebook's f4 [35]). Both BART and BW significantly reduce both MTL and transitioning bandwidth of RD, while BART



(a) MTL (b) Transitioning bandwidth

Figure 7: Experiment A2: Impact of M (where $N = 100$ and $(k, m, \lambda) = (6, 3, 3)$).



(a) MTL (b) Transitioning bandwidth

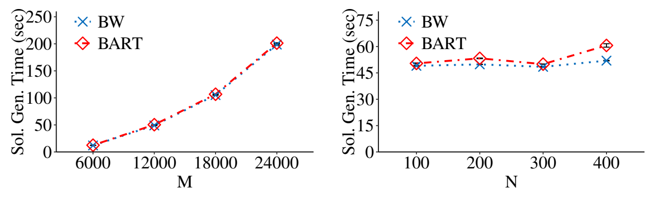
Figure 8: Experiment A3: Impact of N (where $M = 12000$ and $(k, m, \lambda) = (6, 3, 3)$).

further reduces the MTL of BW with only a slight increase in transitioning bandwidth via the load balancing steps (§4.2). For example, for $(6, 3, 3)$, BART reduces the MTL of RD and BW by 75.4% and 26.9%, respectively, while BART reduces the transitioning bandwidth of RD by 75.0% and slightly increases the transitioning bandwidth of BW by 0.56%.

Experiment A2: Impact of M. We study the impact of M on the MTL and transitioning bandwidth by varying M from 6000 to 24000 stripes. Here, we fix $N = 100$ nodes and $(k, m, \lambda) = (6, 3, 3)$. Figure 7 shows the results. While both MTL and transitioning bandwidth increase linearly with the number of input stripes, BART still effectively reduces the MTL over the baselines, while keeping the transitioning bandwidth low. For example, for $M = 24000$, BART reduces the MTLs of RD and BW by 77.2% and 24.7%, respectively, while BART reduces the transitioning bandwidth of RD by 77.3% and slightly increases the transitioning bandwidth of BW by 0.49%.

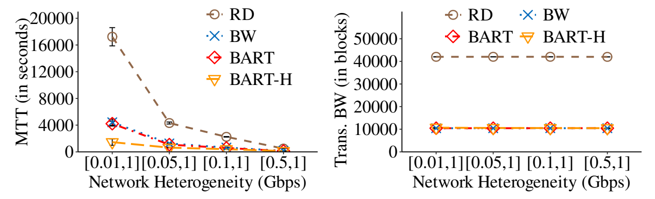
Experiment A3: Impact of N. We also study the impact of N on the MTL and transitioning bandwidth by varying N from 100 to 400 nodes. Here, we fix $M = 12000$ stripes and $(k, m, \lambda) = (6, 3, 3)$. Figure 8 shows the results. When N increases, the MTLs of all schemes decrease as the transitioning bandwidth is naturally spread across the nodes, while the transitioning bandwidth varies slightly across the number of nodes. Overall, BART can still effectively reduce the MTL. For example, when $N = 400$, BART reduces the MTLs of RD and BW by 65.6% and 45.3%, respectively, while BART reduces the transitioning bandwidth of RD by 60.4% and slightly increases the transitioning bandwidth of BW by 0.13%.

Experiment A4: Solution generation time. We measure the running times to generate a redundancy transitioning solution for BART and BW; we do not consider RD as it shows significantly poor transitioning performance. Here, we fix $(k, m, \lambda) = (6, 3, 3)$. We study the impact of M and N on the solution generation times by (i) varying M from 6000 to 24000 stripes with $N = 100$ nodes and



(a) Vary M (where $N = 100$) (b) Vary N (where $M = 12000$)

Figure 9: Experiment A4: Solution generation time (where $(k, m, \lambda) = (6, 3, 3)$).



(a) MTT (b) Transitioning bandwidth

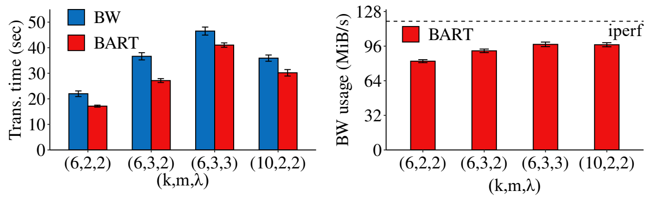
Figure 10: Experiment A5: Impact of network heterogeneity (where $B = 64\,\text{MiB}$, $M = 12000$, $N = 100$, and $(k, m, \lambda) = (6, 3, 3)$).

(ii) varying N from 100 to 400 nodes with $M = 12000$ stripes. Figure 9 shows the results. The solution generation times of both BART and BW increase significantly with M, while remaining similar even if N varies. The reason is that the solution generation times of both BART and BW are dominated by stripe group construction with a time complexity of $O(M^2)$, which is independent of N (§4.1). BART only shows slightly higher solution generation time than BW for performing the subsequent load-balancing steps. For example, for $M = 12000$ and $N = 100$, BART generates a redundancy transitioning solution within 50.4 s, where stripe group construction accounts for 98.9% of the overall solution generation time. BART can generate the redundancy transitioning solution offline, so the actual transitioning performance is not affected by the solution generation time overhead.

Experiment A5: Impact of network heterogeneity. We then study the impact of network heterogeneity, where we vary the available link capacity across the nodes. Here, we refer to the extension of BART for heterogeneous environments (§4.4) as BART-H for comparisons. We choose the available link capacities for nodes based on the uniform distribution [40], [55], where we set the largest link capacity as 1 Gbps, and vary the smallest link capacity from 0.01 Gbps to 0.5 Gbps. We set the default block size $B = 64\,\text{MiB}$ [16]. We fix $M = 12000$ stripes, $N = 100$ nodes, and $(k, m, \lambda) = (6, 3, 3)$. Figure 10 shows the results. BART-H effectively reduces the MTT of all baselines, especially when the smallest link bandwidth is lower, while keeping the transitioning bandwidth low. For example, for $U[0.01\,\text{Gbps}, 1\,\text{Gbps}]$, BART-H reduces the MTTs of RD, BW, and BART by 91.6%, 67.2% and 65.5%, respectively, while BART-H slightly increases the bandwidth of BW by 2.3%.

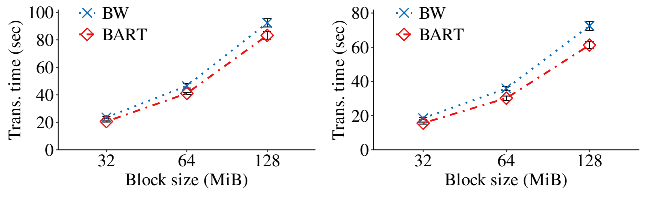
5.2 Testbed Experiments

Prototype implementation. To demonstrate the practicality of BART, we implement a BART prototype as a middleware



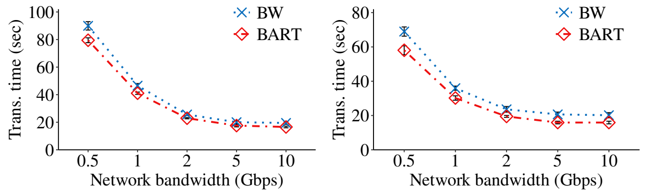
(a) Transitioning time (b) Network bandwidth usage

Figure 11: Experiment B1: Impact of (k, m, λ).



(a) $(6, 3, 3)$ (b) $(10, 2, 2)$

Figure 12: Experiment B2: Impact of block size.



(a) $(6, 3, 3)$ (b) $(10, 2, 2)$

Figure 13: Experiment B3: Impact of network bandwidth.

atop Hadoop 3.3.4 HDFS [3]. HDFS comprises a *NameNode* for storage management and multiple *DataNodes* for data storage. It stores data as fixed-sized blocks and supports systematic Vandermonde-based RS codes.

We implement BART with two components: a *Controller*, which resides in the NameNode, and multiple *Agents*, each of which resides in a DataNode. Suppose that HDFS stores erasure-coded stripes across the DataNodes before redundancy transitioning. Then, the Controller generates a redundancy transitioning solution based on the stripe metadata stored in the NameNode, and notifies the Agents about the actual operations. The Agents execute the transitioning operations to generate the output stripes. We implement the data transfer, I/O, and erasure coding functionalities in C++ with around 2.2 K LoC, in which we use ISA-L [7] to implement the erasure coding operations. We also support the data retrieval from HDFS with new coding parameters after redundancy transitioning, by adding around 1.4 K LoC written in Java to the HDFS codebase.

Evaluation methodology. We deploy the BART prototype on Alibaba Cloud [1]. We provision 31 `ecs.g7.xlarge` instances, including one instance serving the NameNode and 30 instances serving the DataNodes. Each instance has 4 vCPUs, 16 GiB RAM, and a 100 GiB enhanced SSD with PL1 performance [2]. It is installed with Ubuntu 20.04. All instances are connected via a 10 Gbps network, and we configure the network bandwidth in our evaluation via the Wondershaper tool [8]. By default, we set $M = 1200$ stripes, block size as 64 MiB [16], and network bandwidth as 1 Gbps. We evaluate the impact of (k, m, λ), block size and network bandwidth.

We measure the *transitioning time*, defined as the time from issuing a request for executing redundancy transitioning until all output stripes are generated. We report the average transitioning time over 10 runs, with error bars showing the 95% confidence intervals based on the student's t-distribution. We only plot the results of BART and BW, as RD shows significantly poor performance from our simulation results (§5.1).

Experiment B1: Impact of (k, m, λ). We first study the transitioning time of BART for different (k, m, λ). Figure 11(a) shows the results. By reducing the MTL, BART reduces the redundancy transitioning time of BW by 22.0%, 25.9%, 11.8%, and 16.0% for $(6, 2, 2)$, $(6, 3, 2)$, $(6, 3, 3)$, and $(10, 2, 2)$, respectively. We observe that the solution generation times of BART are within 0.5 s for different (k, m, λ) in our testbed, implying that the solution generation overhead is limited.

We also measure the bandwidth usage of BART as the ratio between the MTL multiplied by the block size (i.e., the amount of data transferred in the bottlenecked link) and the transitioning time. Figure 11(b) shows the bandwidth usage of BART; we also plot the available network bandwidth (measured by `iperf`) for comparisons. The bandwidth usage of BART is close to the available network bandwidth, implying that the transitioning performance is mainly bottlenecked by the most loaded node. This also justifies our assumption that network bandwidth is the main performance bottleneck (§3). For example, BART introduces 95.5 MiB/s and 95.4 MiB/s of bandwidth usage for $(6, 3, 3)$ and $(10, 2, 2)$, respectively.

Experiment B2: Impact of block size. We study the impact of block size on the transitioning time. We vary the block size from 32 MiB to 128 MiB, and set (k, m, λ) as $(6, 3, 3)$ and $(10, 2, 2)$. Figure 12 shows the results. BART maintains a fairly stable reduction of redundancy transitioning time compared with BW for different block sizes. For example, when the block size is 32 MiB, BART reduces the transitioning time of BW for $(6, 3, 3)$ and $(10, 2, 2)$ by 12.4% and 15.3%, respectively.

Experiment B3: Impact of network bandwidth. We also study the impact of network bandwidth. We vary the network bandwidth from 0.5 Gbps to 10 Gbps via the Wondershaper tool [8]. We again set (k, m, λ) as $(6, 3, 3)$ and $(10, 2, 2)$. Figure 13 shows the results. When the network bandwidth increases, both BART and BW further reduce the transitioning time, while BART still outperforms BW under different network bandwidth settings. For example, when the network bandwidth 10 Gbps, BART reduces the transitioning time of BW for $(6, 3, 3)$ and $(10, 2, 2)$ by 15.0% and 21.1%, respectively.

6 Related Work

Redundancy transitioning. Earlier studies consider the transitioning from replication to erasure coding for storage savings [14], [28], [43]. Recent studies consider the transitioning between erasure codes with different coding parameters. HACFS [49] dynamically changes between high-redundancy codes for hot data and low-redundancy codes for cold data. ERS [46] co-designs the encoding matrix and data placement to mitigate the I/Os in redundancy transitioning. HeART

[25], PACEMAKER [24], and Tiger [23] consider disk-adaptive redundancy to provide reliability guarantees with the lowest possible redundancy based on the prediction of disk failure rates. Note that HeART, PACEMAKER, and Tiger apply re-encoding (§2.2), while we focus on load-balanced parity merging.

Scaling. Scaling is a class of redundancy transitioning for erasure coding that not only changes the coding parameters, but also ensures the even storage distribution across all nodes. Existing studies on scaling focus on RAID [44], [54] and distributed storage [21], [48]. NCScale [19] shows that the scaling bandwidth can be minimized via network coding. Wu et al. [47] study the trade-off between the repair and scaling performance for locally repairable codes [20] from a data placement perspective.

Code conversion. Code conversion is another class of redundancy transitioning that we target in this work. Maturana and Rashmi [34] provide the first formal study for code conversion, with the objective of minimizing I/Os during code conversion. Follow-up studies include the analysis of the lower bound of conversion I/Os for all valid parameters [31] and conversion bandwidth [32], [33]. StripeMerge [51] addresses the special case of parity merging with $\lambda = 2$, and provides heuristics for implementing such parity merging in practical storage deployment. Wu et al. [45] study the optimal data placement for parity merging using locally repairable codes [20]. ClusterRT [52] addresses code conversion in rack-based data centers with limited cross-rack network bandwidth. It applies re-encoding to merge every λ input stripes to λ' stripes (where $1 \leq \lambda' < \lambda$), with the objective of minimizing the cross-rack transitioning bandwidth. In this work, we focus on load-balanced parity merging, where we also address heterogeneous network environments.

Load balancing for erasure coding. Some studies address load balancing in erasure-coded storage. EC-Cache [38] employs erasure coding to overcome the load imbalance from selective replication for object caching. EC-Scheduler [12] considers heterogeneous environments and dynamically adjusts the recovery workload for load balancing. Selective-EC [50] and ParaRC [29] improve the recovery performance via load-balanced recovery operations. Our work addresses load balancing in redundancy transitioning.

7 Conclusion and Future Work

We propose BART, a load-balanced redundancy transitioning scheme that reduces the redundancy transitioning time via carefully scheduled parallelization. BART builds on parity merging and decomposes the parity merging problem into three sub-problems that are solved by efficient heuristics, so as to achieve load balancing while keeping the transitioning bandwidth low. We evaluate BART via large-scale simulations and HDFS prototype experiments on Alibaba Cloud to demonstrate its effectiveness.

In this work, we focus on parity merging under Vandermonde-based RS codes. We pose the extension to general redundancy transitioning parameters and other erasure codes (e.g., locally repairable codes [20], [27]) as future work.

References

[1] Alibaba Cloud. https://www.alibabacloud.com/.
[2] Alibaba Cloud - ESSDs. https://www.alibabacloud.com/help/en/elastic-compute-service/latest/essds.
[3] Apache Hadoop 3.3.4. https://hadoop.apache.org/docs/r3.3.4/.
[4] Corrupted fragment on decode #10. https://github.com/intel/isa-l/issues/10.
[5] Erasure coding in Ceph. https://docs.ceph.com/en/latest/rados/operations/erasure-code/.
[6] Erasure coding in Hadoop 3.3.4. https://hadoop.apache.org/docs/r3.3.4/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html.
[7] ISA-L. https://github.com/intel/isa-l.
[8] Wondershaper. https://github.com/magnific0/wondershaper.
[9] VAST. https://vastdata.com/blog/providing-resilience-efficiently-part-ii/, 2019.
[10] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with skewed content popularity in MapReduce clusters. In *Proc. of ACM EuroSys*, 2011.
[11] B. Beach. Backblaze Vaults: Zettabyte-scale cloud storage architecture. https://www.backblaze.com/blog/vault-cloud-storage-architecture/, 2019.
[12] X. Cao, G. Yang, Y. Gu, C. Wu, J. Li, G. Xue, M. Guo, Y. Dong, and Y. Zhao. EC-Scheduler: A load-balanced scheduler to accelerate the straggler recovery for erasure coded storage systems. In *Proc. of IEEE/ACM IWQoS*, 2021.
[13] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, 2010.
[14] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson. DiskReduce: RAID for data-intensive scalable computing. In *Proc. of ACM PDSW*, 2009.
[15] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, 2010.
[16] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proc. of IEEE MSST*, 2003.
[17] N. J. A. Harvey, R. E. Ladner, L. Lovász, and T. Tamir. Semi-matchings for bipartite graphs and load balancing. *Elsevier Journal of Algorithms*, 59(1):53–78, 2006.
[18] Y. Hu, L. Cheng, Q. Yao, P. P. C. Lee, W. Wang, and W. Chen. Exploiting combined locality for wide-stripe erasure coding in distributed storage. In *Proc. of USENIX FAST*, 2021.
[19] Y. Hu, X. Zhang, P. P. C. Lee, and P. Zhou. NCScale: Toward optimal storage scaling via network coding. *IEEE/ACM Trans. on Networking*, 30(1):271–284, 2022.
[20] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure storage. In *Proc. of USENIX ATC*, 2012.
[21] J. Huang, X. Liang, X. Qin, P. Xie, and C. Xie. Scale-RS: An efficient scaling scheme for RS-coded storage clusters. *IEEE Trans. on Parallel and Distributed Systems*, 26(6):1704–1717, 2015.
[22] Q. Huang, K. Birman, R. Van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *Proc of ACM SOSP*, 2013.
[23] S. Kadekodi, F. Maturana, S. Athlur, A. Merchant, K. V. Rashmi, and G. R. Ganger. Tiger: Disk-adaptive redundancy without placement restrictions. In *Proc. of USENIX OSDI*, 2022.
[24] S. Kadekodi, F. Maturana, S. J. Subramanya, J. Yang, K. V. Rashmi, and G. R. Ganger. PACEMAKER: Avoiding HeART attacks in storage clusters with disk-adaptive redundancy. In *Proc. of USENIX OSDI*, 2020.
[25] S. Kadekodi, K. V. Rashmi, and G. R. Ganger. Cluster storage systems gotta have HeART: Improving storage efficiency by exploiting disk-reliability heterogeneity. In *Proc. of USENIX FAST*, 2019.
[26] S. Kadekodi, S. Silas, D. Clausen, and A. Merchant. Practical design considerations for wide locally recoverable codes (LRCs). In *Proc. of USENIX FAST*, 2023.
[27] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg. On fault tolerance, locality, and optimality in locally repairable codes. In *Proc. of USENIX ATC*, 2018.

[28] R. Li, Y. Hu, and P. P. Lee. Enabling efficient and reliable transition from replication to erasure coding for clustered file systems. *IEEE Trans. on Parallel and Distributed Systems*, 28(9):2500–2513, 2017.

[29] X. Li, K. Cheng, K. Tang, P. P. Lee, Y. Hu, D. Feng, J. Li, and T.-Y. Wu. ParaRC: Embracing sub-packetization for repair parallelization in MSR-coded storage. In *Proc. of USENIX FAST*, 2023.

[30] M. Mahajan, P. Nimbhorkar, and K. Varadarajan. The planar k-means problem is np-hard. In *Lecture Notes in Computer Science*, 2009.

[31] F. Maturana, V. S. C. Mukka, and K. V. Rashmi. Access-optimal linear MDS convertible codes for all parameters. In *Proc. of IEEE ISIT*, 2020.

[32] F. Maturana and K. V. Rashmi. Bandwidth cost of code conversions in distributed storage: Fundamental limits and optimal constructions. In *Proc. of IEEE ISIT*, 2021.

[33] F. Maturana and K. V. Rashmi. Bandwidth cost of code conversions in the split regime. In *Proc. of IEEE ISIT*, 2022.

[34] F. Maturana and K. V. Rashmi. Convertible codes: Enabling efficient conversion of coded data in distributed storage. *IEEE Trans. on Information Theory*, 68(7):4392–4407, 2022.

[35] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's warm BLOB storage system. In *Proc. of USENIX OSDI*, 2014.

[36] A.-J. Peters, M. K. Simon, and E. A. Sindrilaru. Erasure coding for production in the EOS open storage system. In *Proc. of CHEP*, 2019.

[37] J. S. Plank and C. Huang. Tutorial: Erasure coding for storage applications. Slides presented at FAST-2013: 11th Usenix Conference on File and Storage Technologies, Feb 2013.

[38] K. V. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proc. of USENIX OSDI*, 2016.

[39] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[40] Z. Shen, P. P. C. Lee, and J. Shu. Efficient routing for cooperative data regeneration in heterogeneous storage networks. In *Proc. of IEEE/ACM IWQoS*, 2016.

[41] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of IEEE MSST*, 2010.

[42] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication:a quantitative comparison. In *Proc. of IPTPS*, 2002.

[43] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Trans. on Computer Systems*, 14(1):108–136, 1996.

[44] C. Wu and X. He. GSR: A global stripe-based redistribution approach to accelerate RAID-5 scaling. In *Proc. of IEEE ICPP*, 2012.

[45] S. Wu, Q. Du, P. P. Lee, Y. Li, and Y. Xu. Optimal data placement for stripe merging in locally repairable codes. In *IEEE INFOCOM*, 2022.

[46] S. Wu, Z. Shen, and P. P. C. Lee. Enabling I/O-efficient redundancy transitioning in erasure-coded KV stores via elastic reed-solomon codes. In *Proc. of IEEE SRDS*, 2020.

[47] S. Wu, Z. Shen, and P. P. C. Lee. On the optimal repair-scaling trade-off in locally repairable codes. In *Proc. of IEEE INFOCOM*, 2020.

[48] S. Wu, Y. Xu, Y. Li, and Z. Yang. I/O-efficient scaling schemes for distributed storage systems with CRS codes. *IEEE Trans. on Parallel and Distributed Systems*, 27(9):2639–2652, 2016.

[49] M. Xia, M. Saxena, M. Blaum, and D. A. Pease. A tale of two erasure codes in HDFS. 2015.

[50] L. Xu, M. Lyu, Q. Li, L. Xie, C. Li, and Y. Xu. SelectiveEC: Towards balanced recovery load on erasure-coded storage systems. *IEEE Trans. on Parallel and Distributed Systems*, 33(10):2386–2400, 2022.

[51] Q. Yao, Y. Hu, L. Cheng, P. P. C. Lee, D. Feng, W. Wang, and W. Chen. StripeMerge: Efficient wide-stripe generation for large-scale erasure-coded storage. In *Proc. of the IEEE ICDCS*, 2021.

[52] F. Zhang, F. Nan, B. Xu, Z. Shen, J. Zhai, D. Kalplun, and J. Shu. Achieving tunable erasure coding with cluster-aware redundancy transitioning. *ACM Trans. on Architecture and Code Optimization*, 21(3):1–24, 2024.

[53] X. Zhang, Y. Hu, P. P. Lee, and P. Zhou. Toward optimal storage scaling via network coding: From theory to practice. In *Proc. of IEEE INFOCOM*, 2018.

[54] W. Zheng and G. Zhang. FastScale: Accelerate RAID scaling by minimizing data migration. In *Proc. of USENIX FAST*, 2011.

[55] Y. Zhu, J. Lin, P. P. C. Lee, and Y. Xu. Boosting degraded reads in heterogeneous erasure-coded storage systems. *IEEE Trans. on Computers*, 64(8):2145–2157, 2015.

Keyun Cheng received the B.Eng. degree in Software Engineering from Sun Yat-Sen University in 2018, and the M.Sc. degree in Computer Science from The Chinese University of Hong Kong in 2019. He is now a Ph.D. student in Computer Science and Engineering at The Chinese University of Hong Kong. His research interests include distributed storage systems and erasure coding.

Huancheng Puyang received the B.Eng. degree in Computer Science from The Hong Kong University of Science and Technology in 2021. He is currently pursuing a Ph.D. degree in Computer Science and Engineering at The Chinese University of Hong Kong. His research interests include storage systems, cloud computing and stream processing systems.

Xiaolu Li received the BEng degree from the University of Science and Technology of China, in 2016, and the PhD degree in computer science and engineering from the Chinese University of Hong Kong, in 2020. She is now a lecturer with the School of Computer Science and Technology, Huazhong University of Science and Technology. Her current research interests include distributed storage system, erasure-coding, and container storage.

Patrick P. C. Lee (Senior Member, IEEE) received the BEng (first-class honors) degree in information engineering from the Chinese University of Hong Kong, in 2001, the MPhil degree in computer science and engineering from the Chinese University of Hong Kong, in 2003, and the PhD degree in computer science from Columbia University, in 2008. He is now a professor with the Department of Computer Science and Engineering, Chinese University of Hong Kong. His research interests include various applied/systems topics including storage systems, distributed systems and networks, and cloud computing.

Yuchong Hu (Member, IEEE) received the BS degree in computer science and technology from the School of the Gifted Young, University of Science and Technology of China, Anhui, China, in 2005, and the PhD degree in computer science and technology from the School of Computer Science, University of Science and Technology of China, in 2010. He is currently a professor with the School of Computer Science and Technology, Huazhong University of Science and Technology. His research interests include improving data reliability (e.g., erasure coding) and Big Data storage systems.

Jie Li (Member, IEEE) received the B.S. and M.S. degrees in mathematics from Hubei University, Wuhan, China, in 2009 and 2012, respectively, and the Ph.D. degree from the Department of Communication Engineering, Southwest Jiaotong University, Chengdu, China, in 2017. He is currently a Senior Researcher with the Theory Laboratory, Huawei Technologies Company Ltd., Hong Kong. His research interests include distributed computing, private information retrieval, coding for distributed storage, and sequence design. He received the IEEE Jack Keil Wolf ISIT Student Paper Award in 2017.

Ting-Yi Wu received the Ph.D. degree in communication engineering from National Chiao Tung University, Hsinchu, Taiwan, in 2013. He is currently a Principal Engineer with the Theory Laboratory, Central Research Institute, 2012 Labs, Huawei Technology Company Ltd. His research interests mainly focus on erasure coding for storage, networks, and communication.