A Rack-Aware Pipeline Repair Scheme for Erasure-Coded Distributed Storage Systems

Tong Liu
tongliu@temple.edu
Temple University
Philadelphia, USA

Shakeel Alibhai
shakeel.alibhai@temple.edu
Temple University
Philadelphia, USA

Xubin He
xubin.he@temple.edu
Temple University
Philadelphia, USA

ABSTRACT

Nowadays, modern industry data centers have employed erasure codes to provide reliability for large amounts of data at a low cost. Although erasure codes provide optimal storage efficiency, they suffer from high repair costs compared to traditional three-way replication: when a data miss occurs in a data center, erasure codes would require high disk usage and network bandwidth consumption across nodes and racks to repair the failed data. In this paper, we propose RPR, a rack-aware pipeline repair scheme for erasure-coded distributed storage systems. RPR for the first time investigates the insights of the racks, and explores the connection between the node level and rack level to help improve the repair performance when a single failure or multiple failures occur in a data center. The evaluation results on several common RS code configurations show that, for single-block failures, our RPR scheme reduces the total repair time by up to 81.5% compared to the traditional RS code repair method and 50.2% compared to the state-of-the-art CAR algorithm. For multi-block failures, RPR reduces the total repair time and cross-rack data transfer traffic by up to 64.5% and 50%, respectively, over the traditional repair.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**; **Reliability**; **Distributed architectures**.

KEYWORDS

data reliability, erasure coding, distributed storage system

ACM Reference Format:
Tong Liu, Shakeel Alibhai, and Xubin He. 2020. A Rack-Aware Pipeline Repair Scheme for Erasure-Coded Distributed Storage Systems. In *49th International Conference on Parallel Processing - ICPP (ICPP '20), August 17–20, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3404397.3404444

1 INTRODUCTION

To handle the recent explosive data growth, data centers today typically deploy thousands of commodity storage nodes or servers in multiple geographic locations to provide large-scale storage services [7], [8]. Failures are universal to hardware devices, and when the number of devices is very large—such as in a distributed storage system—failures would occur much more frequently [22, 29]. To maintain data reliability and availability when a failure occurs, redundancy techniques such as three-way replication [10, 33] are commonly applied by traditional distributed storage systems. The redundancy method is straightforward and efficient when the data volume is small; however, in large-scale data centers, the 200% storage overhead brought by replication is dramatic and unacceptable. As a compromise, erasure coding, a storage-efficient fault-tolerant technique, can provide the same or higher level of reliability while requiring much less data redundancy [34]. As a result, erasure coding has become increasingly popular and widely adopted by enterprises including Microsoft [15], Facebook [26], and Google [7].

Among multiple erasure code families, the Maximum Distance Seperatable (MDS) codes are the most popular ones which can provide the optimal storage efficiency. Among the MDS codes, the Reed-Solomon (RS) code [27] is the most widely used code in practice. An RS code is usually configured with two parameters, n and k. An RS (n, k) code has n original data chunks and k parity chunks, which are the coded results from the original n data chunks. With the $n + k$ dependent chunks, which are referred to as a *stripe*, any amount of failed chunks less than or equal to k can be recovered by decoding any n of the remaining available chunks in the stripe. When a chunk failure occurs, the traditional replication method would only require one chunk transfer to recover (copying the data from one of the remaining available replicas), while a (n, k) erasure code needs to retrieve n available chunks of the same stripe.

Recent research [21, 25] has found that this scenario is even more severe in large data centers. In modern data centers, storage nodes are normally partitioned into different racks to provide rack-level fault tolerance. Nodes within the same rack are connected via a top-of-rack (TOR) switch, and multiple racks are connected by the aggregation switch [5, 13, 19]. To recover a data node inside one rack, multiple data nodes need to be transferred, either within the same rack or across multiple racks. According to Facebook [26], a median of over 180 terabytes (TB) of data are transferred through top-of-rack switches every day for the purpose of recovery. In addition, the in-production cross-rack bandwidth is usually 1Gb/s, while the inner-rack bandwidth is 10Gb/s [28]. This means that cross-rack bandwidth is a scarcer resource, a fact also confirmed by previous researchers [5, 6]. As a result, although RS code improves storage efficiency, it causes a significant increase in the disk and network traffic for data centers; specifically, it dramatically consumes precious cross-rack bandwidth. Further, traditional RS code

ICPP '20, August 17–20, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-8816-0/20/08…$15.00
https://doi.org/10.1145/3404397.3404444

repair would transfer a very large volume of data to the recovery node and rack, which makes the data center load-imbalanced and further hurts the cross-rack data transfer performance.

In this paper, we propose RPR, a rack-aware pipeline repair scheme that can significantly reduce the cross-rack data transfer and the total repair time when a single failure or multiple failures occur in an erasure-coded distributed storage system. RPR is also beneficial for the data center from the load-balance aspect. The RPR mechanism consists of three techniques: (i) **data-parity placement:** when the parity chunks are generated, they are placed on the racks by certain rules designed to increase the decoding speed; (ii) **inner-rack partial decoding:** before the data chunks are transferred to the recovery node/rack, partial decoding is first conducted, which reduces the cross-rack data transfer and improves the load balance; and (iii) **cross-rack pipeline repair:** after partial decoding, a repair pipeline algorithm schedules the sequence of inner-rack and cross-rack transfers to achieve the optimal total repair time.

Our contributions are summarized as follows:

- We propose RPR, a novel repair scheme that can tolerate multiple failures and significantly reduce the amount of cross-rack data transfers and the total repair time when failures occur in an erasure-coded distributed storage system.
- We present a pipeline-based greedy algorithm which can schedule an optimal repair pipeline for inner-rack decoding/transfer and cross-rack decoding/transfer for a general RS (n, k) code.
- We perform mathematical analysis, simulator evaluation, and real-world evaluation of the RPR scheme. The results indicate that, for single-block failures, our RPR scheme reduces the total repair time by up to 81.5% compared to the traditional repair method and 50.2% compared to CAR [32]. For multi-block failures, RPR reduces the total repair time by up to 64.5% and the cross-rack data transfer traffic by up to 50% over the traditional RS code repair.

2 BACKGROUND

2.1 Erasure Coding

2.1.1 Basics. As mentioned in Section I, the RS (Reed-Solomon) code is the most widely deployed code in today's production. In this paper, we focus on this code as well.

RS code is a matrix-based coding scheme [23], and Figure 1 presents the details of the matrix coding process of a (5, 3) code. The encoding matrix has eight rows. The top five rows are an identity matrix so that, after the encoding process, the original data is kept the same. The bottom three rows, which are called the coding matrix, consist of the encoding coefficients $\{e_{0,0}, e_{0,1}, ..., e_{2,4}\}$ constructed from the *Vandermonde* matrix [4]. They are then multiplied with the five original data chunks $\{D_0, D_1, D_2, D_3, D_4\}$ to generate the parity chunks $\{P_0, P_1, P_2\}$. For more detail, take P_0, generated by the following equation, as an example:

$$e_{0,0} * D_0 \oplus e_{0,1} * D_1 \oplus e_{0,2} * D_2 \oplus e_{0,3} * D_3 \oplus e_{0,4} * D_4 = P_0 \quad (1)$$

In the Vandermonde matrix, all the values in the first row of the coding matrix are 1, so equation (1) can be simplified as:

$$D_0 \oplus D_1 \oplus D_2 \oplus D_3 \oplus D_4 = P_0 \quad (2)$$

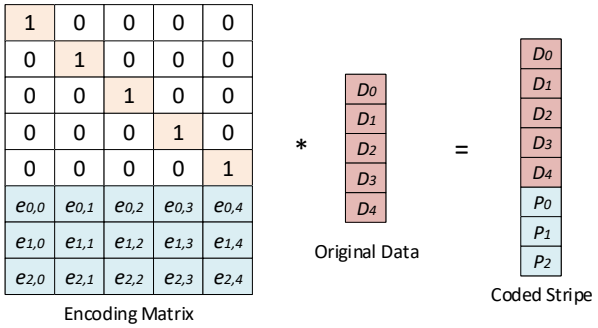


Figure 1: Matrix-vector encoding process of an RS (5, 3) code.

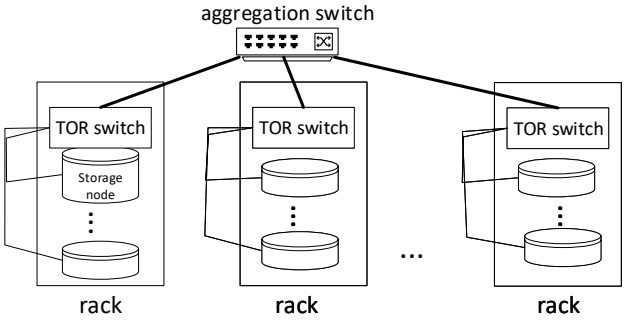


Figure 2: Typical top-of-rack (TOR) network connectivity architecture.

When l failures occur in the coded stripe ($0 < l \leq k$), we pick n rows in the encoding matrix that correspond to any n remaining available chunks in the coded stripe; we denote the matrix formed from these n rows as M'. We then invert this matrix to M'^{-1}, which serves as the decoding matrix. After multiplying the decoding matrix with the remaining available chunks, the lost data is recovered. This traditional RS recovery requires n chunks to be transferred to the recovery node. (In this work, we consider data blocks, data chunks and data nodes (which save data chunks) as the same concept. We also consider parity blocks, parity nodes and parity chunks as the same.) This can result in a data transfer bottleneck at the recovery node when the data volume is huge, as well as cause load imbalance.

2.1.2 Partial decoding. In the matrix coding/decoding process, all the operations are calculated according to Galois Field (GF) arithmetic [27], which is a finite field that has a limited number of elements. In the Galois Field, the results of any operation still lie in the field. One important property of GF arithmetic is that addition is equivalent to XOR.

As an example, consider an RS (4, 2) code with four data chunks $\{D_0, D_1, D_2, D_3\}$ and two parity chunks $\{P_0, P_1\}$. When D_2 fails, assume that D_0, D_1, D_3, and P_0 are selected to recover the failed chunk in the recovery node. Then the default recovery equation of D_2 would be:

$$D_0 \oplus D_1 \oplus D_3 \oplus P_0 = D_2 \quad (3)$$

With partial decoding deployed, the default recovery can be divided into two parts: decode D_0 and D_1 to produce an intermediate recovery chunk I_0, and decode D_3 and P_0 to produce an intermediate recovery chunk I_1. After that, D_2 can be recovered by decoding I_0 and I_1. The recovery process would be:

$$D_0 \oplus D_1 = I_0, D_3 \oplus P_0 = I_1, I_0 \oplus I_1 = D_2 \tag{4}$$

With partial decoding, the lost data can be decoded partially and in parallel, thus mitigating the transfer bottleneck and load imbalance issues.

2.2 Data Center Architecture and Chunk Placement

Modern large-scale data centers hold data in thousands of commodity storage nodes (servers) organized across multiple racks. As shown in Figure 2, each rack has a top-of-rack (TOR) switch that connects the nodes within the same rack. Outside the rack, there is an aggregation switch which connects all the TOR switches so that the nodes across different racks can communicate [5, 12, 32].

In a large data center, not only may disks and servers fail, but rack failures may also occur multiple times per year [21]. To ensure rack-level fault tolerance, previous data centers distribute each coded stripe evenly across multiple racks [7, 14, 21]. However, the one-node-per-rack placement brings significant cross-rack traffic to the data center when a failure or update occurs. To address this issue, recent research [13, 31, 32] proposes placing n nodes across q racks (where $q < n$) so that the repair/update performance can be improved. As a trade-off, when multiple nodes are placed in one rack, the data center may not be able to provide multi-rack fault tolerance. In this paper, we focus on the single-rack fault tolerance scenario, which is considered a reasonable and common configuration by previous work.

2.3 Traditional Repair Process

Given an RS (n, k) code, to ensure single-rack fault tolerance, each rack can contain at most k nodes from the same stripe. To simplify the discussion, assume each rack contains the same number of k nodes, and the $n + k$ nodes in the same stripe are distributed across q racks. Figure 3 shows an example of a traditional RS $(4, 2)$ code repair process. The four original data blocks $\{d_0, d_1, d_2, d_3\}$ and two parity blocks $\{p_0, p_1\}$ are evenly distributed across the three racks $\{r_0, r_1, r_2\}$. Assume that node d_1 fails and nodes $\{d_0, d_2, d_3, p_0\}$ are selected to recover it. In the traditional decoding process [16, 31], the four selected available nodes are transferred to the recovery node/rack. Assuming that one cross-rack transfer takes t_c time, the time for the recovery node to receive all the necessary information and start the decoding process is $4 * t_c$. In Figure 3, timesteps ① to ④ indicate the steps for the total data transfer of this single-node failure recovery.

Assume that the cross-rack bandwidth is ω, the data in each node to be transferred is B, and the decoding speed is δ. Then the total repair time for the traditional repair process, t_{total}, is the sum of the data transfer time and the decoding time:

$$t_{total} = 4 * \frac{B}{\omega} + 1 * \frac{B}{\delta} \tag{5}$$

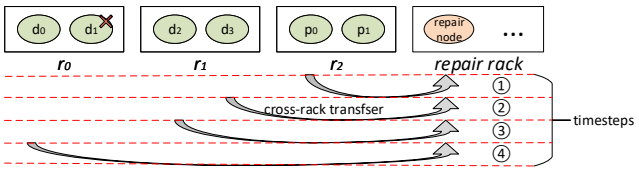


Figure 3: Example of traditional RS $(4, 2)$ code repair process when a single-block failure occurs.

In practice, the cross-rack bandwidth is approximately 1Gb/s (128MB/s) [28], and the decoding speed for the RS code is approximately 1000MB/s [23]. Clearly, the cross-rack bandwidth of the repair node/rack is the bottleneck of the whole repair process. To repair one failed node, four nodes need to be transferred across racks to the repair node/rack; this consumes a large amount of the precious cross-rack bandwidth and takes a long time due to the low bandwidth. In addition, all the data uploads happen in the recovery node/rack, which makes the system load-imbalanced.

3 DESIGN

3.1 Inner-rack Partial Decoding

As discussed, the high cross-rack network traffic and load imbalance is caused by the fact that the recovery node can only start the decoding process after it receives all the information from the other nodes/racks. To mitigate the high traffic and balance the load, it would be intuitively better if the data transfer and decoding processes could be distributed to other nodes/racks. In addition, according to previous research [28, 30], the cross-rack bandwidth in production is 1Gb/s, while the inner-rack bandwidth is 10Gb/s. This means that it would also be more beneficial if some of the cross-rack transfers could be replaced by inner-rack transfers. Thus, inner-rack partial decoding is a promising technique that can be applied here.

Partial decoding, as described in Section II.A, utilizes a property of the RS code through which the data is encoded and decoded via linear combinations. Traditionally, to repair a data block, all related data blocks need to be transferred to the recovery node/rack. With partial decoding, nodes in the same rack can decode locally first and then generate an intermediate coding block (with the same size as an original data block). In this case, the cross-rack data transfer for each rack would be at most one block, and since the inner-rack bandwidth is around 10 times as the cross-rack bandwidth, the data transfer time can also be reduced significantly.

Figure 4 shows the repair process of the example used in Figure 3, but with inner-rack partial decoding. Instead of transferring four blocks to the repair node/rack, racks r_1 and r_2 first conduct inner partial decoding in parallel. This time, timestep ① only takes one inner-rack transfer timestep, t_i. Compared to the traditional repair process, which has a transfer time of $4 * t_c$, the partial decoding has a transfer time of only $2 * t_c + t_i$, where $t_c \approx 10 * t_i$.

Following is the recursive inner-rack partial decoding algorithm for each rack involved in the repair process:

The algorithm ***Inner*** runs for each rack when the repair process starts and takes the nodes involved in the repair as the input. ***Inner*** divides the nodes into several pairs and performs partial decoding recursively until the final intermediate block I is generated. In the

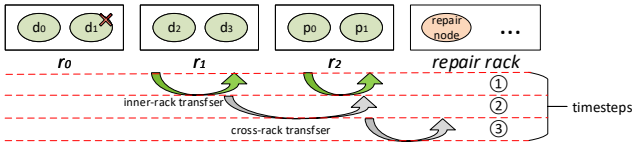


Figure 4: Example of an RS $(4, 2)$ code repair process with inner-rack partial decoding.

Algorithm 1: ***Inner*** $(d_0, d_1, ..., d_{n-1})$

input : Selected data blocks in the rack involved in the repair process, $\{d_0, d_1, ..., d_{n-1}\}$.
output: An intermediate block I that will be cross-rack transferred for further decoding.

if $n == 0$ **then**
  return I
if $n\%2 == 0$ **then**
  for $a \leftarrow 0$ **to** $\frac{n-2}{2}$ **do**
    $i_a = d_{2a} \oplus d_{2a+1}$;
  Inner $(i_0, i_1, ..., i_{\frac{n-2}{2}})$
else
  for $a \leftarrow 0$ **to** $\frac{n-3}{2}$ **do**
    $i_a = d_{2a} \oplus d_{2a+1}$;
  Inner $(i_0, i_1, ..., i_{\frac{n-3}{2}} \oplus d_{n-1})$

example shown in Figure 4, after each intermediate block is generated, the cross-rack transfer schedule would be straightforward. However, when multiple failures occur, or when there are more nodes involved in the repair process, the inner-rack and cross-rack transfer schedules would be much more complicated. The reason for this is that when multiple failures occur in multiple nodes/racks, the finish times of the inner-rack partial decodings may vary for different nodes/racks. To start cross-rack partial decoding, some intermediate data blocks must first be generated by certain inner-rack partial decodings. In order to maintain the data consistency, as well as make full use of the parallelism of the inner-rack and cross-rack transfers, a scheduling algorithm is essential.

3.2 Cross-rack Pipeline Scheduling

To address this issue, we propose a pipeline-based greedy algorithm which aims to maintain the data consistency during the entire repair process as well as achieve the optimal total repair time. (The proof is in Section IV.B.)

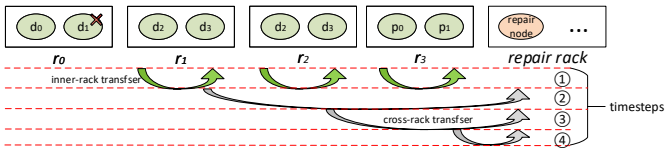
In the example in Figure 4, the repair schedule for RS $(4, 2)$ code looks straightforward. However, when the code is more complex, various repair schedules can be selected with respect to the inner-rack and cross-rack data transfers. Consider another simple example, RS $(6, 2)$ code, as shown in Figure 5. Figures 5a and 5b display two possible inner-rack and cross-rack transfer schedules. To minimize the cross-rack transfer volume, we select the nodes in racks r_1, r_2, and r_3 to conduct the inner-rack partial decodings and cross-rack transfers. In schedule 1, after timestep ①, all three racks finish the partial decoding and prepare to send the intermediate data to the repair node/rack. Without loss of generality, r_1 starts its cross-rack transfer first. Then, since the repair rack is busy receiving the data from r_1, racks r_2 and r_3 will have to wait until the transfer is finished. Likewise, after r_1 finishes its cross-rack transfer, r_3 will also need to wait until r_2 finishes. For schedule 2, similarly, after timestep ①, all three racks finish the partial decoding and prepare to send the intermediate data. The difference is that this time, only r_3 will first be scheduled to send the intermediate data I_3 to the repair rack. In the meantime, r_1 will send its intermediate data I_1 to r_2. There, I_1 and I_2 will be further partially decoded, creating the intermediate data block I_{1+2}. These two cross-rack transfers will be conducted simultaneously in timestep ②. After timestep ②, the intermediate block I_{1+2} will be cross-rack transferred to the repair rack, conduct the final partial decoding with I_3, and obtain the reconstructed data block d_1.

These two schedules both have three cross-rack transfers and use three timesteps; however, their total repair times have a big difference. This is due to the high bandwidth difference between inner-rack and cross-rack transfers. As previously assumed, the time for one cross-rack transfer is $t_c \approx 10 * t_i$, where t_i is the time for one inner-rack transfer. Then for schedule 1, the total repair time would be approximately $3 * t_c + t_i$, or $31 * t_i$. (The decoding time is neglected here because it is small compared to the data transfer time in this approximate calculation.) For schedule 2, after the inner-rack transfer, r_1 and r_2 do not wait for r_3 to finish the cross-rack transfer; instead, r_1 immediately sends the intermediate block to r_2, and r_3 immediately sends the intermediate block to the recovery rack. In this case, the two cross-rack transfers in timestep ② can be conducted in parallel, thereby avoiding the time wasted by waiting. Therefore, the total repair time can be estimated to be around $1 * t_i + 2 * t_c$, or $21 * t_i$.

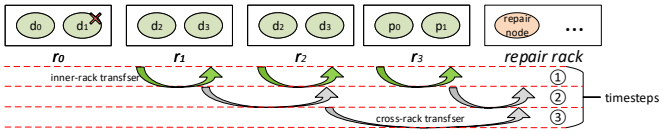
Algorithm 2: ***Cross*** (nodes and racks)

input : Failed block and its corresponding rack ID; nodes in each rack that are in the same stripe and their corresponding rack IDs.
output: The reconstructed data block d_r.

if *recovery node gets all intermediate data* **then**
  do last partial decoding;
  return d_r;
for *each rack* **do**
  if *inner decoding is possible* **then**
    Inner (nodes in current rack);
  else
    start cross-rack transfer with any other rack which has no inner-rack transfer;
  if *cross-rack transfer with any other rack which has no cross-rack transfer is possible* **then**
    start cross-rack transfer with selected rack;
  else
    wait until the rack which finishes the cross-rack first, then start transfer;
Cross (racks that conduct inner-rack partial decodings in the next timestep)



(a) Schedule 1.



(b) Schedule 2.

Figure 5: Example of an RS $(6, 2)$ code repair process with two different schedules.

From this example, the cross-rack pipeline scheduling algorithm for a general RS code (n, k) with j failures can be described in Algorithm 2, the ***Cross*** algorithm:

- For each rack, if an inner-rack decoding can be conducted, start the inner-rack transfer and partial decoding. Since inner-rack transfers are faster than cross-rack transfers and can reduce the cross-rack transfer data volume, they should always have priority over cross-rack transfers.
- If an inner-rack transfer cannot be scheduled, then start a cross-rack transfer with any other rack which currently has no inner-rack transfer. (Either its inner-rack transfer would have finished or it does not need to conduct one). The recovery rack can also be selected, but only when every non-recovery rack is busy.
- When the inner-rack transfer finishes, start a cross-transfer with any other rack that does not currently have a cross-rack transfer.
- If every other rack has a cross-rack transfer, then wait until the one that finishes first, and then start the transfer.
- Repeat recursively until all the racks finish their cross-rack transfers or the recovery node receives all the intermediate data.

The ***Cross*** algorithm works together with the ***Inner*** algorithm: whenever ***Inner*** finishes the inner-rack data transfer and produces the intermediate data, ***Cross*** will give the optimal cross-rack transfer schedule based on the current node and rack transfer status.

3.3 Data Pre-placement Optimization for Single-block Failure

Previous work has found that single-block failures are the most common and dominant failures in distributed storage systems [9, 18, 24, 29]. To optimize for this case, we propose the pre-placement scheme, which utilizes the previously mentioned properties of the RS code.

As described in Section II.A, a typical decoding process requires building the decoding matrix M'^{-1}. The multiplication of M'^{-1} and the remaining available chunks will recover all the failed chunks. According to our observation, generating the decoding matrix may take up to 75% of the total decoding time. However, in the case of a single-block failure, this time-consuming process can sometimes be neglected if the data and parity blocks are placed in a certain pattern.

According to equations (2) and (3), when the Vandermonde matrix—the standard matrix employed in the Jerasure [23] library—is used, any failed block D_f in the original data blocks $\{D_0, D_1, D_2, ..., D_{n-1}\}$ can be recovered by a single calculation with the help of the first parity block P_0:

$$D_f = D_0 \oplus D_1 \oplus ... \oplus D_{f-1} \oplus D_{f+1} \oplus ... \oplus D_{n-1} \oplus P_0 \quad (6)$$

Intuitively, to reduce the cross-rack data transfer, it would be better to transfer as much information as possible in each cross-rack transfer. Thus, to reduce the amount of cross-rack transfers and avoid building the decoding matrix, the best way is to put P_0 with data blocks instead of parity blocks in the same rack. For example, in Figure 4, both p_0 and p_1 are involved in the repair process. In this case, the decoding process will construct the decoding matrix and generate the failed block based on the matrix calculation, which is time-consuming. A simple way of avoiding this would be to switch the placement of d_0 and p_1. Then according to equation (6), the failed block d_1 can be calculated by one XOR equation without creating the decoding matrix.

For an RS (n, k) code, if the first parity block P_0 is placed with all data blocks in the same rack, then, assuming the block failure rate is same for all blocks, there is a $\frac{1}{n}$ chance that there is no need to build the decoding matrix when a single-block failure occurs. It should be noted that this pre-placement has no negative effect on other performance metrics, such as I/O or load balance. In addition, while this does not benefit the multi-block failure scenario (as the decoding matrix must be built in that case), the pre-placement scheme does not negatively impact it either.

3.4 Extension to Multi-block Failures

When one failure occurs, the failed block can be reconstructed by using equations (3) and (4). However, when multiple failures occur, the procedure becomes much more complicated since multiple decoding equations are required. In this subsection, we discuss the extended multi-block repair scheme of RPR, which can reduce the total repair time when multiple failures occur.

Assume that there is an (n, k) erasure coded system with original data blocks $\{d_0, d_1, ..., d_{n-1}\}$ and parity blocks $\{p_0, p_1, ..., p_{k-1}\}$. Then the encoded functions for this system are:

$$\begin{aligned} &e_{0,0} * d_0 \oplus e_{0,1} * d_1 \oplus ... \oplus e_{0,n-1} * d_{n-1} = p_0 \\ &... \\ &e_{k-1,0} * d_0 \oplus e_{k-1,1} * d_1 \oplus ... \oplus e_{k-1,n-1} * d_{n-1} = p_{k-1} \end{aligned} \quad (7)$$

where $e_{i,j}$ is the encoding coefficient.

Further, assume that there are k failed blocks, $\{d_{f,0}, d_{f,1}, ..., d_{f,k-1}\} \in \{d_0, d_1, ..., d_{n-1}\}$. Then from equation (7), we have:

$$\begin{aligned} &e'_{0,0} * d_0 \oplus e'_{0,1} * d_1 \oplus ... \oplus e'_{0,n-1} * p_{k-1} = d_{f,0} \\ &... \\ &e'_{k-1,1} * d_0 \oplus e'_{k,1} * d_1 \oplus ... \oplus e'_{k-1,n-1} * p_{k-1} = d_{f,k-1} \end{aligned} \quad (8)$$

where $e'_{i,j}$ is the coefficient after the conversion. Note that the blocks on the left side of each sub-equation in equation (8) do not include the failed block on the right side of each sub-equation.

To recover each failed block, the data on the left side of each equation is needed. Once again, we can use partial decoding and cross-rack scheduling to reduce the total repair time. Assume that the stripe is distributed across q racks. Then, for each sub-equation in (8), every rack can only transfer an intermediate data block $I_{i,j}$ to the recovery node/rack, which is the partial decoding result of the available blocks in the current rack. Then equation (8) can be further written as:

$$\begin{aligned} &I_{0,0} \oplus I_{0,1} \oplus \ldots \oplus I_{0,q-1} = d_{f,0} \\ &\ldots \\ &I_{k-1,0} \oplus I_{k-1,1} \oplus \ldots \oplus I_{k-1,q-1} = d_{f,k-1} \end{aligned} \tag{9}$$

In the multi-block failure scenario, the partial decoding in each rack needs to be conducted multiple times to obtain all the required intermediate data blocks. After the intermediate data blocks are generated, each sub-equation in (9) can be considered as a single-block failure scenario as previously analyzed. The difference is that there will be multiple cross-rack transfers for each rack. Then the ***Inner-multi*** and ***Cross-multi*** algorithms, corresponding to the multi-block failure scenario, can be summarized as in Algorithms 3 and 4. (Due to the space limitation, the details of Algorithms 3 and 4 are shown in the external links.)

4 ANALYSIS

4.1 Total Repair Time with Partial Decoding

To provide data reliability, assume that each erasure-coded stripe in a data center is distributed across $n + k$ nodes and q racks with an RS (n, k) code. For rack R_i, the number of nodes involved in the repair process is r_i. To ensure single-rack fault tolerance, r_i should be in the range $[1, k]$. As described in Section III, the cross-rack bandwidth is usually $\frac{1}{10}$ of the inner-rack bandwidth; therefore, assume that $t_c = 10 * t_i$, where t_c is the time cost for one cross-rack transfer for one data block and t_i is the time cost for one inner-rack transfer for one data block. In addition, assume that the decoding time with the decoding matrix built is t_{wd} and the decoding time without building the decoding matrix is t_{nd}. According to our observation, we assume that $t_{wd} = 4 * t_{nd}$; the actual time is shown in the evaluation section. Since the decoding time is small compared to the data transfer time, it is neglected in the total repair time analysis.

For traditional repair, when a data failure occurs, n remaining available blocks will be chosen to be cross-transferred to the repair node/rack. Then for traditional repair, the total repair time t_{total_t} is:

$$t_{total_t} = n * t_c \tag{10}$$

With the RPR scheme, the total repair process for a data transfer can be divided into two parts, a cross-rack transfer and an inner-rack transfer. Then the total inner-rack transfer time T_{inner} is:

$$T_{inner} = ((Max\lfloor log_2 r_i \rfloor) + 1) * t_i \tag{11}$$

The total cross-rack transfer time T_{cross} in the worst case is:

$$T_{cross} = (\lfloor log_2 q \rfloor + 1) * t_c \tag{12}$$

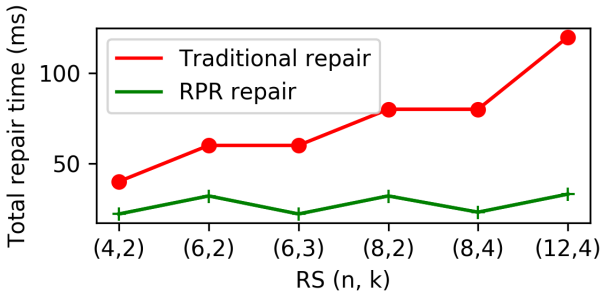


Figure 6: Theoretical total repair time for traditional repair and RPR repair with different RS codes.

To simplify the analysis, assume that each rack has the same r_i, which is k. Then in the worst case, where no pipeline is used at all, the total repair time for the RPR scheme t_{total_r} is:

$$\begin{aligned} &t_{total_r} = T_{inner} + T_{cross} = \\ &(\lfloor log_2 k \rfloor + 1) * t_i + (\lfloor log_2 q \rfloor + 1) * t_c \end{aligned} \tag{13}$$

Figure 6 shows the repair time trend for different RS codes with traditional repair and RPR repair (worst case). In the figure, t_i and t_c are assumed to be 1ms and 10ms, respectively. The figure indicates that, with n increasing in the RS code, the traditional repair time increases linearly, while the RPR repair time increases steadily and with a much smaller scale.

4.2 Greedy Pipeline Cross-rack Schedule

The pipeline cross-rack schedule proposed in Section III.B aims to achieve the minimum total repair time during the whole repair process. The rest of this sub-section proves that this schedule is an optimal greedy algorithm.

Suppose S is the schedule produced by RPR and O is the optimal solution, which has the minimum repair time. The Algorithm 2 can be simplified as follows:

(1) If an inner-rack transfer can be conducted, then conduct it.
(2) If an inner-rack transfer cannot be conducted, then start a cross-rack transfer with any rack which does not currently have an inner-rack transfer.
(3) When the inner-rack transfer finishes, start a cross-rack transfer with any rack which does not have a cross-rack transfer.
(4) If every other rack has a cross-rack transfer, then wait until the first one finishes and then start the transfer.
(5) If the recovery rack does not receive all the intermediate data, go back to step 1.

Let T_s be the total repair time produced by schedule S from Algorithm 2 and T_o be the optimal (minimal) repair time from O, and assume that S is not optimal. Then $T_s > T_o$. A repair process consists of multiple inner-rack and cross-rack transfers, some of which can be conducted in parallel in a single inner-rack or cross-rack timestep. These cost one t_i and one t_c, respectively. t_c and t_i are the same for T_s and T_o, so T_s must have more timesteps than T_o. The extra timesteps either include inner-rack or cross-rack transfer timesteps. If we assume that the extra timesteps include any inner-rack timesteps, then we would contradict steps 1 and

2 in the algorithm, because any inner-rack transfer is prioritized to be conducted in parallel for each rack at the beginning of the repair process unless an inner-rack transfer cannot be conducted. However, if we assume that the extra timesteps include any cross-rack transfers, then we would contradict steps 3 and 4, because according to the algorithm, no rack will be idle unless there is a data consistency issue (step 4). This means that if T_o has fewer cross-rack transfer timesteps than T_s, then T_o will not able to maintain data consistency and thus cannot produce the correct repair result. This contradicts the assumption that O is the optimal solution.

This concludes the proof that the RPR schedule is the optimal solution which can achieve the minimum repair time while also maintaining data consistency throughout the entire data transfer and data decoding processes.

4.3 Multi-block Failure Recovery Limitations

In this section, we discuss the performance limitations of RPR in the worst case and we also describe how these limitations only apply to RS (n, k) codes in certain scenarios.

4.3.1 Code configuration. Based on the design of the multi-failure recovery in Section III.D, we know that when multiple failures occur, we have a series of steps to recover the failed data. First, each rack needs to finish the multiple inner-rack data transfer and partial decoding operations, which, in the worst case, would take $k * t_i$ time (with partial decoding time neglected). Next, the racks will begin the cross-rack data transfer to get the intermediate data blocks and then conduct further partial decoding. For an RS (n, k) code, the best case for the failures is that all k failures occur in the same rack, because there are not any extra cross-rack transfers for inner partial decoding. In this case, each rack only needs to transfer k intermediate blocks from the inner-partial decoding to the recovery node/rack. It is worth noting that, in the multi-block failure scenario, the partial decoding rules are different from the single-block failure scenario. In the single-block failure scenario, there will only be one sub-equation in equation (9), which means the available data/parity blocks can be decoded with any combination. However, in the multi-block failure scenario, the available data/parity blocks need to be decoded multiple times with the corresponding decoding coefficients from the decoding matrix in each sub-equation. Assume an RS (n, k) code with $\frac{n+k}{k} \leq 3$ (or $\frac{n}{k} \leq 2$). In the best case, which is when only one rack has failures, there will be at most two remaining racks and one recovery rack; therefore, there are at most three racks in total for the cross-rack transfer. Note that here $\frac{n+k}{k}$ equals q, the number of racks that are used to place the $n + k$ data/parity blocks. Then for each sub-equation in equation (9), the time needed for the cross-rack transfer is $\lceil log_2 3 \rceil * t_c$. Since there are k sub-equations, the total repair time needed is $\lceil log_2 3 \rceil * k = \frac{n}{k} * k = n$ cross-rack timesteps, which is the same as the traditional RS code repair time. On the other hand, for codes with $\frac{n+k}{k} > 3$, for each sub-equation, the repair time with partial decoding would be $\lceil log_2 q \rceil * t_c$ and the total repair time would be $\lceil log_2 q \rceil * t_c * k$. Compared to the traditional repair time $n * t_c$, the improvement would be $1 - (\frac{\lceil log_2 q \rceil * k}{n})$.

An RS (n, k) code with $\frac{n+k}{k} \leq 3$ means that the storage overhead is equal to or greater than 50%. However, the major motivation for using erasure coding to replace the traditional three-way replication is its high storage efficiency. Therefore, erasure codes with $\frac{n+k}{k} > 3$ tend to be used in the industry. For example, Facebook's HDFS-RAID uses a default code configuration of $(10, 4)$, [25, 35], and Windows Azure System uses $(12, 2, 2)$ [14].

4.3.2 Cross-rack traffic. Equation (9) shows that, in each sub-equation, the number of intermediate blocks I generated is $\frac{n}{k}$. In the worst case (where k failures occur), there are k sub-equations in equation (9); therefore, the total number of intermediate data blocks that must be generated is $\frac{n}{k} * k = n$. This implies that RPR does not reduce the cross-rack data transfer traffic when the worst case occurs. However, RPR does not increase the traffic either.

4.3.3 Multi-block failure with $2 \sim (k-1)$ failures (non-worst case). Assume that for an RS (n, k) code where $\frac{n+k}{k} \leq 3$ there are l-block failures, where $2 \leq l \leq (k-1)$. Then the total repair time would be $\lceil log_2 3 \rceil * l = 2 * l$ cross-rack timesteps. Since $n \leq 2k$, the total repair time is highly likely to be lower than the traditional repair time, n cross-rack timesteps. Similarly, the cross-rack traffic in this case would be $\frac{n}{k} * l$ blocks, which is less than the traditional repair's cross-rack data transfer with n block transfers. Therefore, when the worst case does not occur, which means that the number of failures is $2 \sim (k-1)$, RPR is able to deliver better improvement to both the total repair time and the cross-rack data transfer traffic for any code configuration.

5 EVALUATION

We evaluate the RPR scheme from two aspects: i) simulations on the Simics [2] simulator, in which multiple nodes and racks are created and run the RPR repair scheme, and ii) experiments on Amazon EC2, in which real-world machines in geo-distributed areas are selected to imitate the rack-level scenario.

5.1 Simics Simulation

Simics [20] is a platform for full-system simulation that is sufficiently generic to model embedded systems, desktops, clusters, and more. In our experiments, Simics generates a cluster of multiple nodes each time, as the experiment requires. Each node/server has a single-core Intel(R) Core(TM)-i7 CPU @ 2.00 GHz with 4GB RAM running on Ubuntu 16.04.4 LTS. Each node has an upload/download network bandwidth as 1Gb/s. The erasure-coded RPR prototype is built based on the Jerasure Library v2.0 [17].

Simics itself cannot simulate the rack granularity in a cluster. To achieve the different inner-rack transfer and cross-rack transfer bandwidth, *wondershaper* [3] is applied to our simulated cluster. Wondershaper is a script that allows the user to limit the bandwidth of one or more network adapters. Since the upload/download bandwidth in the Simics cluster is 1Gb/s, we assume that to be the inner-rack transfer bandwidth. For each two nodes in the same rack, the bandwidth between them is the default 1Gb/s. On the other hand, for nodes that do not reside in the same rack, the bandwidth is set to 0.1Gb/s, which is $\frac{1}{10}$ of the default bandwidth.

5.1.1 Single-block failure. We first focus on the single-block failure scenario, in which a random data block in the encoded stripe is assumed to have failed. In our experiments, each data/parity block is configured to be 256MB. In this part, we compare the repair performance in two aspects, the cross-rack traffic and the total repair time. The comparisons are conducted among RPR, traditional

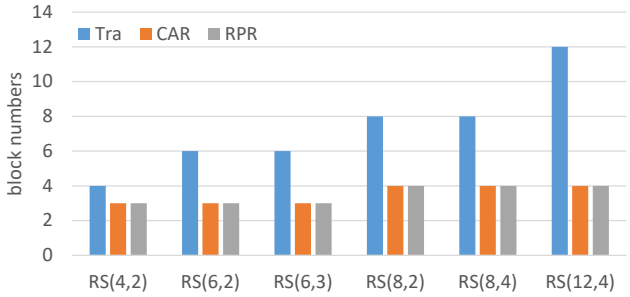


Figure 7: Cross-rack traffic for traditional repair (Tra), CAR, and RPR repair of single-block failures with different RS codes on the Simics simulator.

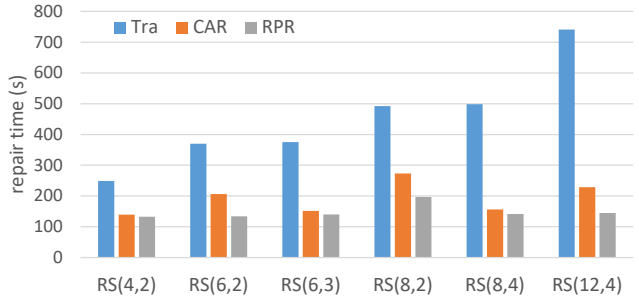


Figure 8: Total repair time for traditional repair (Tra), CAR, and RPR repair of single-block failures with different RS codes on the Simics simulator.

repair, and CAR [32], which, to our knowledge, is the state-of-the-art cross-rack single-failure repair scheme. Again, we use the code configuration in Section IV.A.

Figure 7 shows that both RPR and CAR can reduce the cross-rack traffic compared to the traditional repair scheme. In the case of a single-block failure, the cross-rack traffic is the same for CAR and RPR because they both apply the partial decoding technique to reduce the cross-rack traffic.

Figure 8 shows the total repair time difference among the three repair schemes. Because of the greedy pipeline repair scheme, RPR can choose the fastest repair schedule in all code configurations. However, for CAR, since its main focus is on load balance and there is no repair schedule, in certain code configurations, after the inner partial decoding, the intermediate data will wait for the other cross-rack transfers to finish, thereby resulting in extra cross-rack transfer timesteps. In the Simics simulation, based on the six common RS code configurations, RPR can reduce the total repair time by an average of 67% and up to 81.5% compared to the traditional repair. Compared to CAR, RPR can reduce the total repair time by an average of 24% and up to 37%.

5.1.2 *Multi-block failure.*

- ***Non-worst case.*** As discussed in Section IV, RPR also supports the multi-block failure scenario for any RS (n, k) code (when the worst case does not occur), while CAR only focuses on the single failure scenario. In this section, we compare the total repair time and the cross-rack transfer traffic between RPR and the traditional repair with different code configurations. Different block locations in the stripe may result in different optimal repair times, because the repair schedule is determined by the remaining available blocks after a failure occurs. Among the six code configurations we use in the single-block scenario, three of them—$(4, 2)$, $(6, 2)$, and $(8, 2)$—are not chosen in this experiment because the only multi-block failure for them is the worst case. For codes $(6, 3)$, $(8, 4)$, and $(12, 4)$, there may exist different failure scenarios with respect to the number of failures. For example, code $(8, 4)$ may have a two-block failure or a three-block failure. To represent this, we use a third parameter z in the code configuration: (n, k, z) means that a z-block failure occurs for the (n, k) code, where $2 \leq z < k$. For Figures 9 and 10, the value shown in RPR's bar is the average number for all possible block locations for each code configuration, and the upper and lower caps show the highest and lowest values within all scenarios. The results in Figure 9 show that RPR reduces the total repair time by an average of 40.75% and up to 64.5% compared to the traditional repair scheme; with respect to the cross-rack transfer traffic, RPR uses an average of 29.35% and up to 50% fewer cross-rack data transfers compared to the traditional repair scheme.

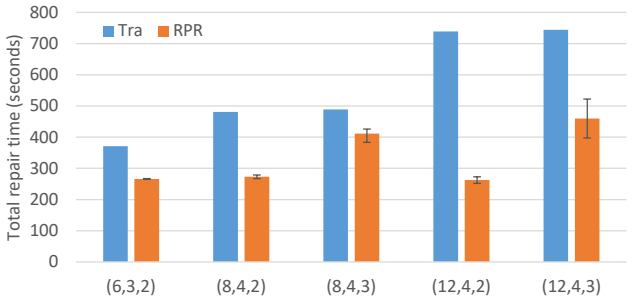


Figure 9: Total repair time for traditional repair (Tra) and RPR repair of multiple failures ($2 \sim k - 1$ failures) with different RS codes on the Simics simulator. The third argument of each code represents the number of block failures.

- ***Worst case.*** When the worst case (where k blocks fail) occurs, RPR can provide better performance for an RS (n, k) code with $\frac{n+k}{k} > 3$. Similarly, we choose the codes that satisfy the code configuration among the six codes we use in the single-block failure scenario: codes $(6, 2)$, $(8, 2)$, and $(12, 4)$. In Figure 11, the value represented by RPR's bar is the average total repair time for all possible block locations for each code configuration, while the upper and lower caps display the highest and lowest values among all scenarios. The results in Figure 11 show that, in the multi-block failure scenario's worst case, RPR reduces the total repair time by an average of 18.3% and up to 29.8% compared to the traditional repair scheme.

5.2 Amazon EC2 Evaluation

To validate that RPR works in real-world systems, we further evaluate RPR in AWS EC2 [1]. To simulate the rack-level data transfer, we launch instances (virtual machines) in five different continents: Ohio in North America, Tokyo in Asia, Paris in Europe, São Paulo

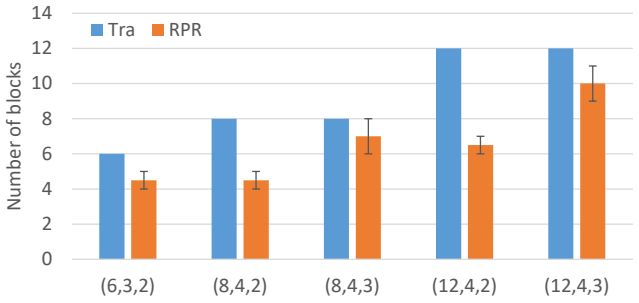


Figure 10: Cross-rack data transfer traffic for traditional repair (Tra) and RPR repair of multiple failures ($2 \sim k-1$ failures) with different RS codes on the Simics simulator. The third argument of each code represents the number of block failures.

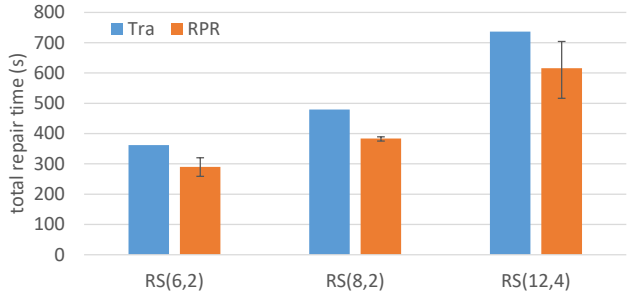


Figure 11: Total repair time for traditional repair (Tra) and RPR repair of multiple failures with different RS codes on the Simics simulator in the worst case (k failures).

Table 1: Inter- and intra-bandwidths (Mbps) across regions

	Ohio	Tokyo	Paris	São Paulo	Sydney
Ohio	583.39	51.798	59.281	67.613	41.4
Tokyo		583.26	45.56	41.605	91.21
Paris			641.403	56.57	40.79
São Paulo				631.416	34.44
Sydney					565.39

in South America, and Sydney in Australia. We consider machines within the same continent to be machines within same rack, and machines across continents to be machines across racks. Table 1 shows the inter- and intra-bandwidth across or within each region. The average cross-region bandwidth is 53.03Mbps, and the average inner-region bandwidth is 600.97Mbps. The ratio of cross-region and inner-region bandwidth is 11.32, which approximately matches our previous assumption (10:1). In this part, each virtual machine is created based on a *t2.micro* type Linux Kernel 4.14 Amazon Linux 2 AMI with 1 vCPU, 1 GB RAM, and 8 GB SSD.

5.2.1 Single-block failure. In this evaluation, we use the same code configuration as in Section V.A to compare the total repair times of the traditional repair, CAR, and RPR. The cross-rack traffic is the same as the Simics simulation. Figure 12 shows the results, which indicate that the repair time difference of CAR and RPR is more

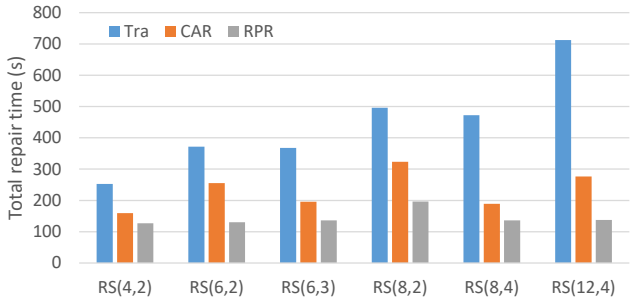


Figure 12: Total repair time for traditional repair (Tra), CAR, and RPR repair of single-block failures with different RS codes on AWS EC2 machines.

significant than the Simics simulation results. This is because in EC2 machines the decoding time with the traditional decoding function of a 256MB file is around 20 seconds, while our optimized decoding function only costs around 2.5 seconds. In the EC2 experiments, RPR can reduce the total repair time by an average of 67.6% and up to 80.8% compared to the traditional repair. Compared to CAR, RPR can reduce the total repair time by an average of 37.2% and up to 50.3%.

5.2.2 Multi-block failure. In this evaluation, we use the same code configurations as in the Simics multi-failure simulation.

- ***Non-worst case.*** For the cross-rack data transfer traffic, the result of the AWS EC2 evaluation is the same as the Simics result since the scheduling is the same. In Figure 13, the value shown in RPR's bar is the average total repair time for all possible block locations for each code configuration, and the upper and lower caps represent the highest and lowest values for all scenarios. Figure 13 shows the results: when the worst case does not occur, RPR can reduce the total repair time by an average of 39.93% and up to 61.96% compared to the traditional repair method.
- ***Worst case.*** Figure 14 shows the results, which indicate that, in the worst multi-block failure case, RPR can reduce the total repair time by an average of 20.6% and up to 32.8% compared to the traditional repair method.

6 RELATED WORK

Recently, multiple works focusing on different aspects of the rack-level storage system have been proposed. Their motivations can be divided into the following categories:

- *Reduce the data transfer for* ***encoding the replicas to stripes when hot data become warm/cold.*** Xie et al. find that the conventional sequential striping causes risky blocks and expensive network consumption when encoding the replicas. Thus, a non-sequential striping according to the data layout is proposed [36], which results in low transfer cost across racks. EAR [19] observes that when data blocks are first stored with replication in clustered file systems, the replica placement plays a critical role in determining the system's encoding performance. By solving a maximum matching

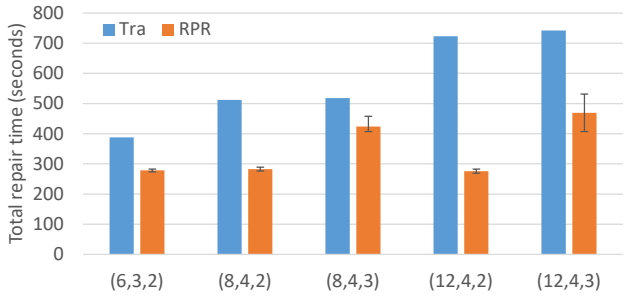


Figure 13: Total repair time for traditional repair (Tra) and RPR repair of multi-block failures ($2 \sim k-1$ failures) with different RS codes on AWS EC2 machines. The third argument of each code represents the number of block failures.

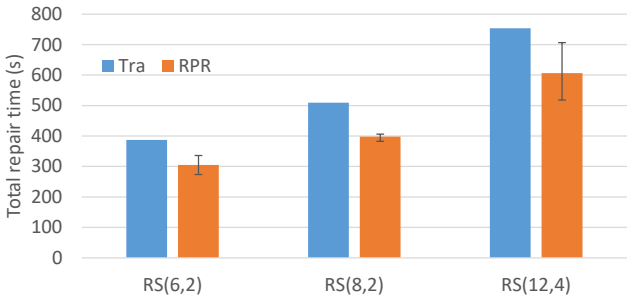


Figure 14: Total repair time for traditional repair (Tra) and RPR repair of multi-block failures with different RS codes on AWS EC2 machines when the worst case (k failures) occurs.

problem, EAR avoids downloading data blocks from other racks during the encoding operation.

- *Reduce the data transfer for **data updates in intensive-updating erasure-coded storage systems**.* To mitigate frequent data updates in erasure-coded storage systems, CAU [30] selectively chooses the parity based on the update pattern and the data layout to reduce the cross-rack update traffic when an update occurs.
- *Reduce the data transfer for **reconstructing the failed data when failure occurs**.* Gong et al. [11] considers the minimization of the regeneration time by selecting the participating nodes in heterogeneous networks. Nevertheless, it only works well when the nodes' bandwidth vary significantly. CAR [32] examines the per-stripe recovery solutions across multiple stripes and constructs a multi-stripe recovery solution that balances the amount of cross-rack repair traffic across multiple racks. However, it only focuses on the single-block failure scenario, which is not enough in practical data center where multiple failures are possible. In contrast, RPR is capable of giving an optimal reconstruction solution for both single-block and multi-block failures.

7 CONCLUSION

In this paper, we focus on the cross-rack data transfer overhead issue with respect to both traffic and repair time when failures occur in a data center. We propose RPR, a rack-aware pipeline repair scheme comprised of three techniques: data-parity placement, inner-rack partial decoding, and cross-rack pipeline scheduling. RPR supports both single-block and multi-block failures. We conduct experiments on both Simics and AWS EC2; results from both platforms show that, in the single-block failure scenario, RPR significantly improves the repair performance compared to the traditional RS code as well as CAR, the state-of-the-art rack-aware single-block recovery scheme, with total repair time reductions of up to 81.5% and 50.2%, respectively. When multiple blocks fail, RPR also improves the repair performance compared to the traditional RS code repair with total repair time reductions of up to 64.5% and cross-rack data transfer reductions of up to 50%.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable comments and feedback. This work was supported by the National Science Foundation (CCF-1717660, CCF-1813081 and CNS-1828363).

REFERENCES

[1] [n.d.]. Amazon Elastic Compute Cloud (Amazon EC2). https://aws.amazon.com/ec2/.
[2] [n.d.]. Simics Full System Simulator. https://www.windriver.com/products/simics/.
[3] [n.d.]. wondershaper - A network traffic management tool in Linux. https://github.com/magnific0/wondershaper.
[4] AA Bjrrck and V Pereyra. 1970. Solution of Vandermonde system of equations. *Math Comput* 24 (1970), 893–903.
[5] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. 2013. Leveraging endpoint flexibility in data-intensive clusters. In *ACM SIGCOMM Computer Communication Review*, Vol. 43. ACM, 231–242.
[6] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.
[7] Daniel Ford, François Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in globally distributed storage systems. (2010).
[8] Svend Frolund, Arif Merchant, Yasushi Saito, Susan Spence, and Alistair Veitch. 2004. A decentralized algorithm for erasure-coded virtual disks. In *International Conference on Dependable Systems and Networks, 2004.* IEEE, 125–134.
[9] Yingxun Fu, Jiwu Shu, and Xianghong Luo. 2014. A stack-based single disk failure recovery scheme for erasure coded storage systems. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems.* IEEE, 136–145.
[10] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. (2003).
[11] Qingyuan Gong, Jiaqi Wang, Dongsheng Wei, Jin Wang, and Xin Wang. 2015. Optimal node selection for data regeneration in heterogeneous distributed storage systems. In *2015 44th international conference on parallel processing.* IEEE, 390–399.
[12] Yuchong Hu, Patrick PC Lee, and Xiaoyang Zhang. 2016. Double regenerating codes for hierarchical data centers. In *2016 IEEE International Symposium on Information Theory (ISIT).* IEEE, 245–249.
[13] Yuchong Hu, Xiaolu Li, Mi Zhang, Patrick PC Lee, Xiaoyang Zhang, Pan Zhou, and Dan Feng. 2017. Optimal repair layering for erasure-coded data centers: From theory to practice. *ACM Transactions on Storage (TOS)* 13, 4 (2017), 33.
[14] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in windows azure storage. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12).* 15–26.
[15] Cheng Huang and Lihao Xu. 2008. STAR: An efficient coding scheme for correcting triple storage node failures. *IEEE Trans. Comput.* 57, 7 (2008), 889–901.
[16] Jianzhong Huang, Xianhai Liang, Xiao Qin, Qiang Cao, and Changsheng Xie. 2014. Push: A pipelined reconstruction i/of or erasure-coded storage clusters. *IEEE Transactions on Parallel and Distributed Systems* 26, 2 (2014), 516–526.
[17] Kevin M. Greenan James S. Plank. [n.d.]. Jerasure: A Library in C Facilitating Erasure Coding for Storage Applications Version 2.0. https://github.com/magnific0/wondershaper.
[18] Osama Khan, Randal C Burns, James S Plank, William Pierce, and Cheng Huang. 2012. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads.. In *FAST.* 20.

[19] Runhui Li, Yuchong Hu, and Patrick PC Lee. 2017. Enabling efficient and reliable transition from replication to erasure coding for clustered file systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2500–2513.

[20] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.

[21] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. 2014. f4: Facebook's Warm {BLOB} Storage System. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*. 383–398.

[22] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. 2007. Failure trends in a large disk drive population. (2007).

[23] James S. Plank. [n.d.]. Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications. http://web.eecs.utk.edu/~jplank/plank/papers/CS-07-603.pdf.

[24] KV Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B Shah, and Kannan Ramchandran. 2015. Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*. 81–94.

[25] KV Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2013. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Presented as part of the 5th {USENIX} Workshop on Hot Topics in Storage and File Systems*.

[26] KV Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2014. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 331–342.

[27] Irving S Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics* 8, 2 (1960), 300–304.

[28] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, Vol. 6. VLDB Endowment, 325–336.

[29] Bianca Schroeder and Garth A Gibson. 2007. Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you?. In *FAST*, Vol. 7. 1–16.

[30] Zhirong Shen and Patrick PC Lee. 2018. Cross-Rack-Aware Updates in Erasure-Coded Data Centers. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, 80.

[31] Zhirong Shen, Patrick PC Lee, Jiwu Shu, and Wenzhong Guo. 2017. Cross-rack-aware single failure recovery for clustered file systems. *IEEE Transactions on Dependable and Secure Computing* (2017).

[32] Zhirong Shen, Jiwu Shu, and Patrick PC Lee. 2016. Reconsidering single failure recovery in clustered file systems. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 323–334.

[33] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. 2010. The hadoop distributed file system.. In *MSST*, Vol. 10. 1–10.

[34] Hakim Weatherspoon and John D Kubiatowicz. 2002. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems*. Springer, 328–337.

[35] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A Pease. 2015. A Tale of Two Erasure Codes in {HDFS}. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*. 213–226.

[36] Yanwen Xie, Dan Feng, and Fang Wang. 2017. Non-sequential striping for distributed storage systems with different redundancy schemes. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 231–240.