Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage

Subrata Mitra[†], Rajesh Panta[‡], Moo-Ryong Ra[‡], Saurabh Bagchi[†]

{mitra4, sbagchi}@purdue.edu {rpanta, mra}@research.att.com

[†]Purdue University [‡]AT&T Labs Research

Abstract

With the explosion of data in applications all around us, erasure coded storage has emerged as an attractive alternative to replication because even with significantly lower storage overhead, they provide better reliability against data loss. Reed-Solomon code is the most widely used erasure code because it provides maximum reliability for a given storage overhead and is flexible in the choice of coding parameters that determine the achievable reliability. However, reconstruction time for unavailable data becomes prohibitively long mainly because of network bottlenecks. Some proposed solutions either use additional storage or limit the coding parameters that can be used. In this paper, we propose a novel distributed reconstruction technique, called Partial Parallel Repair (PPR), which divides the reconstruction operation to small partial operations and schedules them on multiple nodes already involved in the data reconstruction. Then a distributed protocol progressively combines these partial results to reconstruct the unavailable data blocks and this technique reduces the network pressure. Theoretically, our technique can complete the network transfer in $\lceil(log_2(k+1))\rceil$ time, compared to k time needed for a (k, m) Reed-Solomon code. Our experiments show that PPR reduces repair time and degraded read time significantly. Moreover, our technique is compatible with existing erasure codes and does not require any additional storage overhead. We demonstrate this by overlaying PPR on top of two prior schemes, Local Reconstruction Code and Rotated Reed-Solomon code, to gain additional savings in reconstruction time.

Keywords Erasure code, Distributed storage, Network transfer, Repair, Reconstruction, Utilization

EuroSys '16, April 18-21, 2016, London, United Kingdom

ACM ISBN 978-1-4503-4240-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2901318.2901328

1. Introduction

Tremendous amount of data has been created in the past few years. Some studies show that 90% of world's data was created in the last two years [7]. Not only are we generating huge amounts of data, but the pace at which the data is being created is also increasing rapidly. Along with this increase, there is also the user expectation of high availability of the data, in the face of occurrence of failures of disks or disk blocks. Replication is a commonly used technique to provide reliability of the stored data. However, replication makes data storage even more expensive because it increases the cost of raw storage by a factor equal to the replication count. For example, many practical storage systems (e.g., HDFS [43], Ceph [1], Swift [6], etc.) maintain three copies of the data, which increases the raw storage cost by a factor of three.

In recent years, erasure codes (EC) have gained favor and increasing adoption as an alternative to data replication because they incur significantly less storage overhead, while maintaining equal (or better) reliability. In a (k, m) Reed-Solomon (RS) code, the most widely used EC scheme, a given set of k data blocks, called *chunks*, are encoded into $(k + m)$ chunks. The total set of chunks comprises a *stripe*. The coding is done such that any k out of $(k + m)$ chunks are sufficient to recreate the original data. For example, in RS (4, 2) code, 4MB of user data is divided into four 1MB blocks. Then, two additional 1MB parity blocks are created to provide redundancy. In case of a triple replicated system, all four 1MB blocks are replicated three times. Thus, an RS (4, 2) coded system requires $1.5x$ bytes of raw storage to store x bytes of data and it can tolerate up to two data block failures. On the other hand, a triple replication system needs $3x$ bytes of raw storage and can tolerate the same number of simultaneous failures.

Although attractive in terms of reliability and storage overhead, a major drawback of erasure codes is the expensive repair or reconstruction process — when an encoded

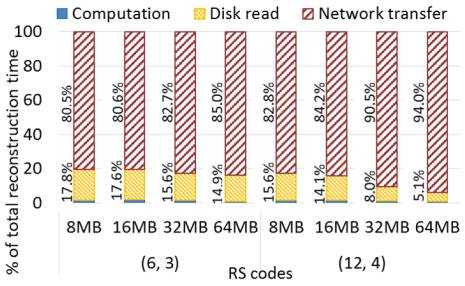


Figure 1: Percentage of time taken by different phases during a degraded read using traditional RS reconstruction technique.

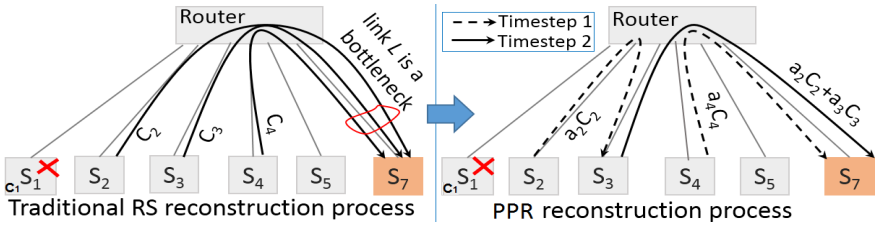


Figure 2: Comparison of data transfer pattern between traditional and PPR reconstruction for RS (3, 2) code. C_2, C_3, etc. are the chunks hosted by the servers. When Server S_1 fails, Server S_7 becomes the repair destination. Network link L to S_7 is congested during traditional repair.

chunk (say c bytes) is lost because of a disk or server[1] failure, in a (k, m) code system, $k \times c$ bytes of data need to be retrieved from k servers to recover the lost data. In the triple replicated system, on the other hand, since each chunk of c bytes is replicated three times, the loss of a chunk can be recovered by copying only c bytes of data from any one of the remaining replicas. This k-factor increase in network traffic causes reconstruction to be very slow, which is a critical concern for any production data center of reasonable size, where disk, server or network failures happen quite regularly, thereby necessitating frequent data reconstructions. In addition, long reconstruction time degrades performance for normal read operations that attempts to read the erasured[2] data. During the long reconstruction time window, the probability of further data loss increases, thereby increasing the susceptibility to a permanent data loss.

It should be noted that, while it is important to reduce repair traffic, practical storage systems also need to maintain a given level of data reliability and storage overhead. Using erasure codes that incur low repair traffic at the expense of increased storage overhead and inferior data reliability is therefore a non-starter. However, reducing repair traffic without negatively impacting storage overhead and data reliability is a challenging task. It has been shown theoretically that there exists a fundamental tradeoff among data reliability, storage overhead, volume of repair traffic, and repair degree. Dimakis *et al.* [17] provide a mathematical formulation of an optimal tradeoff curve that answers the following question—for a given level of data reliability (i.e. a given (k, m) erasure coding scheme), what is the minimum repair traffic that is feasible while maintaining a given level of storage overhead? At one end of this optimal curve lies a family of erasure codes called *Minimum Storage Codes* that require minimum storage overhead, but incur high repair bandwidth. At another end of the spectrum lies a set of erasure codes called *Minimum Bandwidth Codes* that require optimal repair traffic, but incur high storage overhead and repair degree. Existing works fall at different points of this optimal tradeoff curve. For example, RS codes, popular in many practical storage systems [14, 36], require minimum storage space, but create large repair traffic. Locally repairable codes [22, 31, 42] require less repair traffic, but add extra parity chunks, thereby increasing the storage overhead.

In this paper, we design a practical EC repair technique called PPR, which reduces repair time without negatively affecting data reliability, storage overhead, and repair degree. Note that our technique reduces repair time, but not the total repair traffic aggregated over the links. Further, our approach is complementary to existing repair-friendly codes since PPR can be trivially overlaid on top of *any* existing EC scheme.

Code params	Users	Possible reduction in network transfer	Possible reduction in maximum BW usage/server
(6,3)	QFS[30], Google ColossusFS[2]	50%	50%
(8,3)	Yahoo Object Store[8]	50%	62.5%
(10,4)	Facebook HDFS [35]	60%	60%
(12,4)	Microsoft Azure [22]	66.6%	66.6%

Table 1: Advantages of PPR: Potential improvements in network transfer time and maximum bandwidth requirement per server

Key insight A key reason why reconstruction is slow in EC systems is the poor utilization of network resources during reconstruction. A reconstruction of the failed chunk requires a *repair server* to fetch k chunks (belonging to the same stripe as the failed chunk) from k different servers. This causes the network link into the repair server to become congested, increasing the network transfer time. The measurements in our clusters show that network transfer time constitutes up to 94% of the entire reconstruction time, as illustrated in Fig. 1. Other researchers have also reported similar results [26, 36, 42, 44]. Fig. 2 shows an example of a reconstruction of a failed chunk in a (3, 2) EC system. The network link into the repair server (server S_7) is three times more congested than network links to other servers. PPR attempts to solve this problem by redistributing the reconstruction traffic more uniformly across the existing network links, thereby improving the utilization of network resources and decreasing reconstruction time. In order to redistribute

[1] We use the term "server" to refer to the machine that stores the replicated or erasure-encoded data or parity chunks.

[2] An erasure refers to loss, corruption, unavailability of data or parity chunks.

the reconstruction traffic, PPR takes a novel approach for performing reconstruction — instead of centralizing reconstruction in a single reconstruction server, PPR divides reconstruction into several *partial parallel repair* operations that are performed simultaneously at multiple servers, as shown in Fig. 2. Then these results from partial computation are collected using a tree-like overlay network. By splitting the repair operation among multiple servers, PPR removes the congestion in the network link of the repair server and redistributes the reconstruction traffic more evenly across the existing network links. Theoretically, PPR can complete the network transfer for a single chunk reconstruction in $O(log_2(k))$ time, compared to $O(k)$ time needed for a (k, m) RS code. Table 1 shows expected reduction in network transfer time during reconstruction for typical erasure coding parameters used in practical systems. Although PPR does not reduce the total amount of data transferred during reconstruction, it reduces reconstruction time significantly by distributing data transfers more evenly across the network links.

One of the benefits of PPR is that it can be overlaid on top of almost all published erasure coding schemes. The list includes, but is not limited to, the most widely used RS code, LRC code (Locally Repairable Code or Local Reconstruction Code [22, 31, 42]), PM-MSR code [36], RS-Hitchhiker code [38], Rotated RS [24] code. This is because the distribution of PPR is orthogonal to the coding and placement techniques that distinguish these prior works.

In considering the effect of any scheme on reconstruction of missing chunks in an EC system, we need to consider two different kinds of reconstruction. The first is called *regular repair* or *proactive repair*, in which a monitoring daemon proactively detects that a chunk is missing or erroneous and triggers reconstruction. The second is called *degraded read*, in which a client tries to read a lost data chunk that has not been repaired yet and then has to perform reconstruction in the critical path. PPR achieves a significant reduction in times for both these kinds of reconstruction. Degraded reads are an important concern for practical storage systems because degraded read operations happen quite often, more frequently than regular repairs. Transient errors amount to 90% of data center failures [19], because of issues like rolling software updates, OS issues, and non-disk system failures [22, 24]. In these cases, actual repairs are not necessary, but degraded reads are inevitable since client requests can happen during the transient failure period. Furthermore, many practical systems delay the repair operation to avoid initiating costly repair of transient errors [44].

PPR introduces a load-balancing approach to further reduce the reconstruction time when multiple concurrent requests are in progress. We call this variant m-PPR. When selecting k servers out of $(k + m)$ available servers for reconstruction, PPR chooses those servers that have already cached the data in memory, thereby avoiding the time-consuming disk IO on such servers. The m-PPR protocol tries to schedule the simultaneous reconstruction of multiple stripes in such a way that the network traffic is evenly distributed among existing servers. We present further details of m-PPR in Sec. 6.

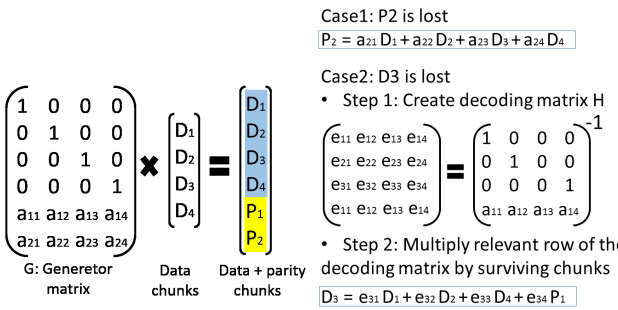
We implemented PPR on top of the Quantcast File System (QFS) [30], which supports RS-based erasure coded storage. For typical erasure coding parameters depicted in Table 1, our prototype achieves up to a 59% reduction in repair time out of which 57% is from reduction in network transfer time alone. Such significant reduction in reconstruction time is achieved without degrading data reliability or increasing storage overhead.

This paper makes the following contributions:

- We introduce PPR, a novel distributed reconstruction technique that significantly reduces network transfer time and thus reduces overall reconstruction time for erasure coded storage systems by up to 59%.
- We present additional optimization methods to further reduce reconstruction time: a) a caching scheme for reducing IO read time and b) a scheduling scheme targeted for multiple simultaneous reconstruction operations.
- We demonstrate our technique can be easily overlaid on previous sophisticated codes beyond Reed-Solomon, such as LRC and Rotated RS, which were targeted to reduce repair time. PPR provides additional 19% and 35% reduction in reconstruction time, respectively, over and above these codes.

The rest of the paper is organized as follows. In Sec. 2 we give a primer of the mathematics behind RS coding. Sec. 3 provides motivation and Sec. 4 describes the main PPR technique in detail. In Sec. 5 we talk about handling multiple reconstructions using PPR. Sec. 6 provides design and implementation details. In Sec. 7 we evaluate PPR w.r.t traditional repair and other proposed solutions. In Sec. 8 we discuss the related works and finally in Sec. 9 we conclude.

2. Primer on Reed-Solomon Coding



(a) RS (4, 2) encoding (b) RS (4, 2) reconstruction

Figure 3: Encoding and Reconstruction in Reed-Solomon coding

Erasure coded storage is attractive mainly because it requires less storage overhead for a given level of reliability. Out of many available erasure coding techniques, ***Reed-Solomon (RS) coding*** [39] is the most widely used. RS code belongs to the class of Maximum Distance Separable (MDS) codes [27], which offers the maximum reliability for a given storage overhead. For a (k, m) RS code, the available data item of size N is divided into k equal ***data chunks*** each of size N/k. Then m additional ***parity chunks*** are calculated from the original k data chunks. The term ***stripe*** refers to this set of $(k + m)$ chunks that is created from the original data. The mathematical property, based on which the parity chunks are created, ensures that *any* missing chunk (data or parity) can be reconstructed using any k of the remaining chunks. After the reconstruction process, the server where the reconstructed data is hosted is referred to as the ***repair site***. Thus, the repair site is a server for a regular repair while for degraded read, it is the client component which has issued the read request.

RS Encoding: An RS encoding operation can be represented as a matrix-vector multiplication where the vector of k data chunks is multiplied by a particular matrix G of size $(k + m) \times k$, as illustrated in Fig. 3a for a (4, 2) RS code. This matrix G is called the *generator matrix* and is constructed from the *Vandermonde* matrix [13] and the elements a_{ij} etc. are calculated according to Galois Field (GF) arithmetic [39]. In GF arithmetic, addition is equivalent to XOR; thus, adding chunk A with chunk B would involve bit-wise XOR operations. Multiplying chunks by a scalar constant (such as the elements of G) is equivalent to multiplying each GF word component by the constant.

RS Reconstruction: In Fig. 3a, when a chunk is lost, it can be reconstructed using some linear algebraic operations with G and a remaining chunk set from the stripe. For example, in Case-1 in Fig. 3b, if a parity chunk (*e.g.*, P_2) is lost, it can be recalculated by multiplying the corresponding *row* (*i.e.* the last row in the example) of G by the data chunk vector. On the other hand, if a data chunk (*e.g.*, D_3) is lost, the reconstruction involves two steps: the first step calculates a *decoding matrix* H, by taking the inverse of a matrix created using any k (*i.e.*, four in our example) surviving rows of G. We refer to the elements of H as decoding coefficients. The second step multiplies the previously selected k surviving chunks (a combination of data and parity) by the row of the decoding matrix corresponding to the lost chunk (*i.e.* the 3^{rd} row in the figure). Thus the decoding process is to solve a set of independent linear equations.

3. The Achilles' Heel of EC Storage: Reconstruction Time

Both for regular repair and degraded read, the reconstruction path consists of three major steps: multiple servers read the relevant chunks from their own disks (usually done in parallel at each server), each server sends the read chunk to the repair site over the network and finally some computation is performed at the repair site to reconstruct the erasured chunk. For regular repairs, the reconstructed chunk is finally written back to the disk while for degraded reads, the data is directly used by the user request. Thus, the reconstruction time for (k, m) RS coding can be approximated as follows

$$T_{reconst} = \frac{C}{B_I} + \frac{kC}{B_N} + T_{comp}(kC) \quad (1)$$

Where C is chunk size, B_I and B_N denote the IO and network bandwidth, respectively. T_{comp} is the computation time, which is a function of a total data size (kC).

As we see from Fig. 1, network transfer and IO read are the two most time consuming steps, while the computation time is relatively insignificant. Among these, network transfer time is the most dominant factor because k chunk transfers are required per reconstruction. Often such huge data transfer creates a network bottleneck near the repair site. For example, Facebook [42] uses RS(10, 4) code with a data chunk size of 256MB. In this case, for repairing a single chunk, more than 20Gbits need to be funneled into one server. This volume of data has been found to overwhelm network resources in many practical cases leading to extremely long reconstruction time. In spite of recent advances in network technology, with the rapid growth of network heavy applications, the network still remains the most scarce resource in data centers and we anticipate network transfer time will continue to remain a bottleneck for reconstruction operations in EC storage.

Such long reconstruction time would still have been a non-issue if reconstructions were infrequent enough. However, traces of failures from large data centers [35, 42] indicate, that is not the case. Analyzing failures in Facebook data centers, Rashmi *et al.* [35] report on average 50 machine unavailability events (where the machine fails for more than 15 minutes) per day, in a data center with a few thousand machines, each of which has a storage capacity of 24-36TB. To maintain data reliability, these events ultimately lead to reconstruction operations. Moreover, Sathiamoorthy *et al.* [42] report that transient errors with no permanent data loss correspond to 90% of data center failure events. These cases often lead to degraded reads where the reconstruction operation happens in the critical path of the user read request.

Thus, long reconstruction time is the main hindrance toward wide scale adoption of erasure coded storage for distributed storage and network transfer time is expected to remain the primary cause for this for the foreseeable future.

This observation has also been made by many prior researchers [38, 50]. Their solutions have taken two forms. In the first form, several solutions design new coding schemes that reduce reconstruction traffic, but incur a higher storage overhead [22, 50]. In the second form, the proposed solutions place erasure encoded data in such a way that the

amount of data that needs to be read for the common failure cases is kept small [24, 38].

In this work, we observe that there is a third way of reducing the network bottleneck during recovery in erasure coded storage: determining intelligently *where* the repair takes place. In all existing repair schemes, the repair operation happens in a centralized location — the repair site — which is either the server where the recovered chunk will be placed, or the client that initiates the read request for the lost data. We propose a distributed repair technique where partial results are computed locally at the server hosting the chunks. Then these results are aggregated to reconstruct the missing chunk. This distributed technique may not appear to be significant because the computational burden of repair in erasure codes is minimal. However, the process of conveying *all* the chunks to a single point in itself creates a bottleneck and load imbalance on some network links. The process of distributing the repair burden among multiple servers has the benefit of removing such a bottleneck and load imbalance. This forms the key innovation in our proposed system PPR. It distributes the task of decoding among multiple servers, in a fashion reminiscent of binomial reduction trees from the High Performance Computing (HPC) world [45].

Because of a mathematical property of the repair operation, this distribution means that the amount of traffic coming out of any aggregator server is exactly half of the sum of the traffics coming in from the two inputs, into the aggregator server. The final destination of the repair traffic, where the complete reconstructed data is finally available, is not overloaded with network traffic in its incoming link. Rather, with PPR, even the incoming link to that destination server gets approximately as much traffic as the first aggregator server. This mathematical property has the desired effect of reducing the network transfer time during repair from erasure coded storage.

4. Design: Partial Parallel Repair (PPR)

We present an efficient reconstruction technique that focuses on reducing network transfer time during reconstruction. PPR divides the entire repair operation into a set of partial operations that are then scheduled to execute in parallel on multiple servers. PPR reduces the pressure on the two primary constrained resources, network capacity and disk reads.

We address the *reconstruction latency problem* in two steps; first, using the main PPR algorithm (Sec. 4.1), we make *single chunk* reconstruction highly efficient. Second, we speed up simultaneous reconstructions resulting from multiple chunk failures[3] by evenly apportioning the load of these multiple reconstructions. The multiple reconstruction scenario arises most commonly because of a hard drive failure. We discuss this aspect of the solution, which we call multiple-PPR (*m*-PPR), in Sec. 5.

4.1 Efficient single chunk reconstruction: Main PPR

As discussed before, to reconstruct an erasured chunk, the EC storage system needs to gather k other chunks and perform the required computation. This step often incurs high latency because of the large volume of data transfer over a particular link, namely, the one leading to the final destination, which becomes the bottleneck.

Based on the repair operation of RS code, we make the following two observations that fundamentally drive the design of PPR:

- The actual reconstruction equation used for computing the missing chunks (either data or parity), as shown in Fig. 3b, is linear and the XOR operations (*i.e.*, the additions) over the terms are *associative*.
- The multiplication by the scalar decoding coefficients or a XOR between two terms do not increase the size of the data. Thus, the size of all the terms that would be XORed, as well as the size of the final reconstructed chunk, is the same as the size of the original chunks that were retrieved from different servers. For instance, let $R = a_1C_1 + a_2C_2$ be the equation for reconstruction. Here a_1, a_2 are the decoding coefficients and R denotes a missing chunk that will be reconstructed from the existing chunks C_1 and C_2. All individual terms in the above equation, *e.g.*, C_1, C_2, a_1C_1, and a_2C_2, will have the same volume of data which is equal to the chunk size (*e.g.* 64MB).

These two observations lead to the fundamental design principle of PPR: *distribute* the repair operation over a number of servers that only computes a *partial* result locally and in parallel, and then forward the intermediate result to the next designated server en route to the final destination. The servers involved in the distributed operations are the ones that host the surviving chunks of that stripe. This design ensures that the part of the data needed for reconstruction is already available locally.

PPR takes a few *logical timesteps* to complete the reconstruction operation, where in each timestep a set of servers perform some partial repair operations to generate intermediate results in parallel. These partial operations constitute either a scalar multiplication of the local chunk data by the corresponding decoding coefficient [4] (this operation happens only during the first logical timestep) or an aggregate XOR operation between the received intermediate results from the earlier servers. For example, in Fig. 2, chunk C_1 is lost because of a failure in server S_1. Server S_7 is chosen as a new host to repair and store C_1. Now C_1 can be reconstructed using the equation: $C_1 = a_2C_2 + a_3C_3 + a_4C_4$, where a_2, a_3, and a_4 are the decoding coefficients corresponding to

[3] Each individual chunk failure is still the only failure in its corresponding stripe. Such single chunk failure in a stripe captures almost 99% of the failure cases (Sec. 1).

[4] We use the term *decoding coefficient* in a generic way. During reconstruction of a parity chunk for RS codes, an encoding operation may be performed. In that case, such coefficients will be 1.

chunks C_2, C_3, and C_4. In `timestep 1`, S_2 sends its partial result a_2C_2 to S_3. In *parallel*, S_4 sends its partial result a_4C_4 to S_7, while at the same time S_3 also computes its own partial result a_3S_3. In `timestep 2`, S_3 sends its aggregated (*i.e.* XORed) results to S_7 reducing the overall network transfer time by a factor of 1/3 or 33%. This behavior can be explained as follows. Let the chunk size be C MB and the available network bandwidth be B_N MB/s. In traditional reconstruction, $3C$ MB of data goes through a particular link, resulting in a network transfer time of approximately $3C/B_N$ sec. In PPR, in each timestep, only one chunk is transferred over *any particular* link (since parallel transfers have different source and destination servers). Thus, the network transfer time in each timestep is C/B_N sec, and since there are *two* timesteps involved in this example, the total network transfer time is $2C/B_N$. The number of timesteps required in PPR can be generalized as $\lceil log_2(k+1) \rceil$, as we will elaborate next.

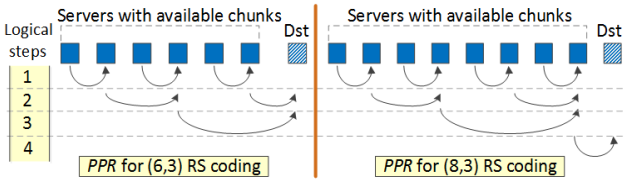


Figure 4: Data transfer pattern during traditional reconstruction for (6, 3) and (8, 3) RS coding

4.2 Reduction in network transfer time

Even though PPR takes a few logical *timesteps* to complete the reconstruction process, in reality, it significantly reduces the total reconstruction time. Essentially, PPR overlays a *tree-like* reduction structure (also referred to as a Binomial Reduction Tree in HPC [28, 45]) over the servers that hold the relevant chunks for reconstruction. Fig. 4 shows more examples of PPR-based reconstruction techniques for RS codes (6,3) and (8,3) where network transfers are completed in only three and four logical timesteps, respectively. Each timestep takes C/B_N amount of time where, C is the chunk size and B_N is the available bandwidth, which results in a total network transfer time of $3C/B_N$ and $4C/B_N$, respectively. In comparison, traditional RS reconstruction for RS (6,3) and (8,3) would bring six and eight chunks to a particular server with a network transfer time of $6C/B_N$ and $8C/B_N$ respectively. Thus PPR can reduce network transfer time by 50% in both cases. We introduce the following theorem to generalize the observation.

THEOREM 1. *For (k, m) RS coding, network transfer time for* PPR*-based reconstruction is $\lceil (log_2(k+1)) \rceil \times C/B_N$ as compared to $k \times C/B_N$ for the original reconstruction technique. Thus* PPR *reduces the network transfer time by a factor of $\frac{k}{\lceil (log_2(k+1)) \rceil}$.*

Proof: PPR *reconstruction*: During reconstruction, in total $(k+1)$ servers are involved, out of which k servers host the relevant chunks and the remaining one is the repair site. PPR performs a binary tree-like reduction where $(k+1)$ servers are the leaf nodes of the tree. Completion of each logical timestep in PPR is equivalent to moving one level up towards the root in a binary tree, while reaching the root marks the completion of PPR. Since the height of a binary tree with $(k+1)$ leaves is $log_2(k+1)$, PPR requires exactly $log_2(k+1)$ logical steps to complete when $(k+1)$ is a power of two; the `ceil` function is used if that is not the case. During each step, the network transfer time is C/B_N since the same amount C is being transferred on each link and each link has bandwidth B_N. Thus, the total network transfer time is $\lceil (log_2(k+1)) \rceil \times C/B_N$.

Baseline EC reconstruction: A total of k chunks, each of size C, will be simultaneously retrieved from k servers. Thus the ingress link to the repair server becomes the bottleneck. If B_N is the bandwidth of that ingress link, the total network transfer time becomes $k \times C/B_N$.

Thus, PPR reduces the network transfer time by a factor of $\frac{k}{\lceil (log_2(k+1)) \rceil}$.

If $k = 2^n - 1$, where $n \in \mathbb{Z}^+$, then the network transfer time is reduced by a factor of $\Omega(\frac{2^n}{n})$. This reduction in network transfer time becomes larger for increasing values of n, *i.e.*, for larger values of k. Since larger values of k (for a fixed m) can reduce the storage overhead of erasure coded storage even further, coding with high values of k is independently beneficial for storing large amounts of data. However, it has not been adopted in practice mainly because of the *lengthy reconstruction time problem.* ■

Moreover, as an additional benefit, the *maximum* data transfer over *any* link during reconstruction is also reduced by a factor of approximately $\lceil (log_2(k+1)) \rceil / k$. In PPR, the cumulative data transfer across all logical timesteps and including both ingress and egress links is $C \times \lceil log_2(k+1) \rceil$. This behavior can be observed in Fig. 4 which illustrates PPR-based reconstruction technique for RS codes (6,3) and (8,3). Such a reduction facilitates a more uniform utilization of the links in the data center, which is a desirable property.

Network architecture: We assume the network to have either a *fat-tree* [10] like topology, where each level gets approximately full bisection bandwidth with similar network capacity between any two servers in the data center, or a VL2-like [20] architecture, which gives the illusion of all servers connected to a monolithic giant virtual switch. These architectures are the most popular choices in practice [9]. If servers have non-homogeneous network capacity, PPR can be extended to use servers with higher network capacity as aggregators, since these servers often handle multiple flows during reconstruction, as depicted in Fig. 4.

When is* PPR *most useful? The benefits of PPR become prominent when network transfer is the bottleneck. Moreover, the effectiveness of PPR increases with higher values of k as discussed before. Interestingly, we found that PPR also becomes more attractive for larger chunk sizes. For

a given k, larger chunks tend to create higher contention in the network. Nevertheless, for other circumstances, PPR should be at least as good as traditional reconstruction since it introduces negligible overhead.

PPR ***vs staggered data transfer:*** Since the reconstruction process causes network congestion at the server acting as the repair site, a straightforward approach to avoid congestion could be to stagger data transfer, with the repair server issuing requests for chunks one-by-one from other servers. However, staggering data transfer adds unnecessary serialization to the reconstruction process and increases the network transfer time. The main problem with this approach is that it avoids congestion in the network link to the repair server by under-utilizing the available bandwidth of network links. Thus, although simple and easy to implement, staggered data transfer may not be suitable for scenarios where reconstructions need to be fast, e.g., in case of degraded reads. PPR decreases network congestion and simultaneously increases parallelism in the repair operation.

Compatibility with other ECs: Since the majority of the practical erasure codes are linear and associative, PPR-based reconstruction can be readily applied on top of them. PPR can also be easily extended to handle even non-linear codes, as long as the overall reconstruction equation can be decomposed into a few independent and partial associative operations.

4.3 Computation speed-up and reduced memory footprint

PPR provides two additional benefits.

Parallel computations: PPR distributes the reconstruction job among multiple servers that perform partial reconstruction functions in parallel. For example, scalar multiplication with decoding coefficients[5] and some aggregating XOR operations are done in parallel, as opposed to traditional serial computation at the repair site. For RS(k, m) code Table 2 highlights the difference between PPR and traditional RS reconstruction, in terms of the computation on the critical path.

Reduced memory footprint: In traditional RS reconstruction, the repair site collects all the k necessary chunks and performs the repair operation on those chunks. Since the processor actively performs multiplication or bitwise XOR operations on these k chunks residing in memory, the memory footprint of such reconstruction operation is on the order of kC, where C is the chunk size. In PPR, the maximum bound on memory footprint in any of the involved servers is $C \times \lceil log_2(k+1) \rceil$, because a server deals with only $\lceil log_2(k+1) \rceil$ chunks at most.

[5] In Cauchy-Reed Solomon coding, multiplications are replaced by XOR operations [32]

PPR reconstruction computation	Traditional RS reconstruction computation
Creation of the decoding matrix + One Galois-field multiplication with coefficients (since parallel at multiple servers) + $ceil(log_2(k+1))$ number of XOR operations (done by aggregating servers)	Creation of the decoding matrix + k Galois-field multiplications with coefficients + k number of XOR operations

Table 2: Faster reconstruction: Less computation per server because of parallelism in PPR technique

4.4 Reducing disk IO with in-memory chunk caching

To reduce the reconstruction time as much as possible, in addition to optimizing network transfer, we also try to reduce disk IO time. Although read operations from multiple servers can be done in parallel, disk read still contributes a non-trivial amount of time to reconstruction, up to 17.8% in the experiment, as shown in Fig. 1. We design an in-memory least recently used (LRU) cache that keeps most frequently used chunks in each server. As a result, the chunk required for reconstruction can be obtained from memory, without incurring the cost of reading it from disk. In addition, PPR maintains a usage profile for chunks that are present in the cache using the associated timestamp. The usage profile can influence the decision regarding which chunk failures should be handled urgently. A chunk that is frequently used, and hence in the cache, should be repaired urgently. Even though caching helps reducing the total reconstruction time, the technique itself is orthogonal to the main PPR technique. Caching can also be used with traditional repair techniques to reduce IO time.

5. Multiple Concurrent Repairs: m-PPR

In any reasonably sized data center, there can be multiple chunk failures at any given time because of either scattered transient failures, machine maintenance, software upgrade, or hard disk failures. Although proactive repairs for such failures are often delayed (*e.g.*, by 15 minutes by Google [19]) in anticipation that the failure was transient, multiple simultaneous reconstructions can still happen at any point in time. A naive attempt to perform multiple overlapping reconstructions may put pressure on shared resources, such as network and disk IO, leading to poor reconstruction performance. We design *m*-PPR, an algorithm that schedules multiple reconstruction-jobs in parallel while trying to minimize the competition for shared resources between multiple reconstruction operations. At the core, each repair job uses the PPR-based reconstruction technique described earlier. Scheduling of multiple reconstructions through *m*-PPR is handled by a *Repair-Manager* (RM), which runs within a centralized entity (*e.g.* the *Meta-Server* in our *Quantcast File System* based implementation).

The RM keeps track of various information for all the servers, such as whether a chunk is available in its in-memory cache, the number of ongoing repair operations scheduled on the server, and the load that users impose

Algorithm 1 m-PPR: Scheduling algorithm for multiple reconstructions

```
1: for all missingChunk ∈ missingChunkList do
2:     hosts ← GETAVAILABLEHOSTS(missingChunk);
3:     reconstSrc ← SELECTSOURCES(hosts); //Choose best sources
4:     reconstDst ← SELECTDESTINATION(hosts, allServers); //Choose
   the best destination
5:     // Schedule a PPR-based single reconstruction
6:     SCHEDULERECONSTRUCTION(reconstSrc, reconstDst);
7:     // Update state to capture the impact of scheduled reconstruction
8:     UPDATESERVERWEIGHTS( );
9: end for
10: // Choose k out of k + m − 1 available sources
11: procedure SELECTSOURCES(hosts)
12:     sortedHosts ← SORTSOURCES(hosts);
13:     selectedSources ← [];
14:     while selectedSources.size ≤ k do
15:         anotherSourceServer ← sortedHosts.pop();
16:         selectedSources.add(anotherSourceServer);
17:     end while
18:     return selectedSources;
19: end procedure
20: //Find a destination server as repair site
21: procedure SELECTDESTINATION(hosts, allServers)
22:     ifdegraded read return Client; //Degraded read:client is destination
23:     // For reliability, exclude existing hosts
24:     possibleDsts ← FINDPOSSIBLEDESTINATIONS(hosts,allServers);
25:     sortedDsts ← SORTDESTINATIONS(possibleDsts);
26:     chosenDst ← sortedDsts.pop(); //Choose the best repair site
27:     return chosenDst
28: end procedure
```

on the servers. Based on these information, the RM uses *greedy heuristics* to choose the best source and destination servers for each reconstruction job. For the source servers, m-PPR selects the k best servers out of the remaining $k+m-1$ servers. For the destination server, it chooses one out of the available $N-(k+m)$ servers, where N is the total number of available servers. In practice, the number of possible destination servers is further constrained by various factors. For example, some applications might require the chunks corresponding to one data stripe to be in close network proximity. Others might want affinity of some data to specific storage types, such as SSD. Some applications might want to avoid servers with identical failure and upgrade domains [14]. The RM calculates, for each potential server, a *source weight* and a *destination weight* as follows:

$$w_{src} = a_1(hasCache) - a_2(\#reconstructions) - a_3(userLoad) \quad (2)$$

$$w_{dst} = -[b_1(\#repairDsts) + b_2(userLoad)] \quad (3)$$

Here a_is, b_is in Eq.(2) and Eq.(3) are the coefficients denoting the importance of various parameters in the source and destination weight equations. The `hasCache` is a binary variable denoting whether the relevant chunk is already present in the in-memory cache of that particular server. The number of reconstructions (`#reconstructions`) in Eq.(2) represents how many reconstruction jobs are currently being handled by the server. `userLoad` measures the network load handled by that server as part of regular user requests. The value of `#reconstructions` gives an indication of the maximum possible network bandwidth utilization by reconstruction operation at that server. In Eq.(3), the number of repair destinations (`#repairDsts`) represents how many repair jobs are using this server as their final destination. Intuitively, these source and destination weights represent the goodness of a server as a source or destination candidate for scheduling the next repair job.

Choosing the coefficients: We calculate the ratio of a_1 and a_2 as $\alpha(ceil(log_2(k+1)))/\beta$. Here α represents the percentage reduction in the total reconstruction time, if a chunk is found in the in-memory cache of a source server. β denotes the ratio of network transfer time to the total reconstruction time in PPR. Intuitively, we compare how many simultaneous reconstructions would be onerous enough to offset benefit of having a chunk in the cache. We calculate the ratio a_2 and a_3 as $C \times \lceil log_2(k) \rceil$. Essentially, from `userLoad` we calculate an equivalent number of PPR-based reconstruction operations that would generate similar traffic. The ratio of b_1 and b_2 is identical to this. For simplicity, we set a_2 and b_1 to one and calculate other coefficients. For example, for RS(6, 3) code, 64MB chunk size, and cluster with 1Gbps network we calculate values of a_1, a_2, and a_3 to be `0.36`, `1.0`, and `0.005` when `userLoad` is measured in MB.

Scheduling: The RM maintains a queue with missing chunk identifiers. To schedule reconstruction of multiple chunks in a batch using m-PPR algorithm, it pops up items one by one from the head of the queue and greedily schedules reconstruction jobs for each of those missing chunks. The RM uses Eq.(2) to calculate goodness score for servers holding relevant chunks of the missing data item and iteratively selects the best k source servers to schedule a PPR job. If fewer than k source servers are available, the RM skips that reconstruction and puts it back at the end of the queue for re-trial. The RM also needs to find a suitable destination server to schedule a repair job. However, not all available servers in the data center are good candidates for the destination server because of reliability reasons. The servers already holding the corresponding data or parity chunks from the same stripe and the ones in the same *failure domain*[6] or *upgrade domain*[7] should be avoided for reliability reasons. For the remaining destination candidates, the RM calculates a weight to capture the current load on that server using Eq.(3). Finally, the most lightly loaded server is selected as the final destination for that repair job. After scheduling a job, all the weights are updated to reconsider the impact on the shared resources. This entire process is illustrated in Algo. 1.

[6] Servers that can fail together *e.g.*, within a rack.

[7] Servers that are likely to be down at the same time because of the software or hardware upgrades.

The overall complexity of m-PPR for scheduling a reconstruction is $O\{N\log(N)\}$. Again, N is the number of possible destination servers and also $N \gg k, m$.

Staleness of information: Some of the parameters used in Eq.(2) and Eq.(3), such as `hasCache` and `userLoad` can be slightly stale as RM collects these metrics through *heartbeats* from the servers. Such staleness is limited by the frequency of heartbeats (5 sec in our setup). Thus, such minor staleness does not affect the usability of m-PPR. Moreover, RM monitors all the scheduled reconstructions and, if a job does not finish within a threshold time, RM reschedules it for choosing a new set of servers. Essentially, m-PPR is a greedy algorithm because for each reconstruction it chooses the best server combination possible at that point.

Beyond a centrally managed server: The scheduling load in m-PPR can be easily distributed over multiple RMs. Each one of these RMs would be responsible for coordinating repairs of a subset of chunk failures. In a more distributed architecture, one of the source servers can also take the responsibility of choosing a new destination server and distribute a repair plan to coordinate the repair with other peers.

6. Design and Implementation

6.1 Background: QFS architecture

Quantcast File System (QFS) is a popular high-performance distributed file system that provides stable RS-based erasure coding for lower storage overhead and higher reliability [30]. QFS evolved from the Kosmos File System originally developed at Microsoft. The QFS architecture has three major components, as illustrated in Fig. 5a. A centralized ***Meta-Server*** manages the file system's directory structure and how RS chunks are mapped to physical storage locations. A ***Chunk Server*** runs on each machine where the data is hosted and manages disk IO. A ***Client*** refers to an entity that interfaces with the user requests. During read (or write) operation, a Client communicates with the Meta-Server to identify which Chunk Server holds (or will hold, in the case of write) the data, then directly interacts with a Chunk Server to transfer the data. Chunk Servers periodically send *heartbeat* messages to the Meta-Server and the Meta-Server periodically checks the availability of the chunks (*monitoring*). If the Meta-Server detects a disk or server failure (through heartbeat), or a corrupted or a missing chunk (through monitoring), it starts the repair process, first designating one Chunk Server as a repair site and then performing the traditional repair process. In case of degraded read, where the client identifies a missing chunk while trying to read, the reconstruction happens in the critical path initiated by the client, which again first gathers k chunks before executing a decoding operation.

6.2 PPR protocol

In this section, we elaborate on the relevant implementation details to enable PPR in QFS.

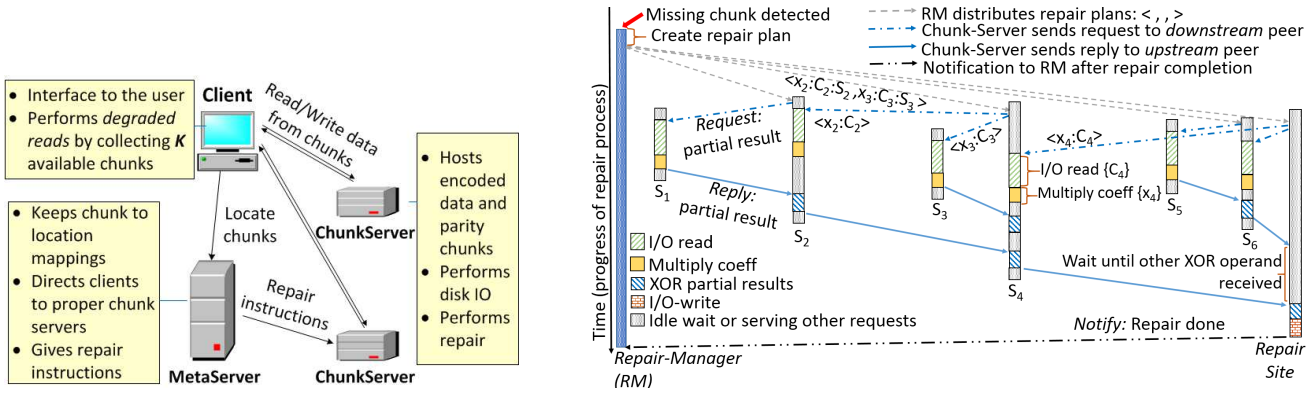
Reconstruction during regular repairs: For a regular repair, the Meta-Server invokes a ***Repair-Manager*** (RM). The RM selects k out of the remaining $k+m-1$ chunks for the reconstruction of the missing chunk. This selection is done by the m-PPR algorithm (Algo. 1). The RM first analyzes which chunks are available for repair and computes the decoding matrix accordingly. From the decoding matrix, the RM calculates decoding coefficients corresponding to each participating chunk. The RM distributes these coefficients along with a ***repair plan*** to only $k/2$ Chunk Servers (*e.g.*, S_2, S_4, S_6 in Fig. 5b) and also to the repair site.

In Fig. 5b, a Chunk Server S_4 receives a plan command $< x_2{:}C_2{:}S_2,\ x_3{:}C_3{:}S_3 >$ from the RM, where x_i's are the decoding coefficients, C_i's are the chunk identifiers (chunkId), and S_i's are the corresponding Chunk Servers. This plan indicates S_4 would aggregate partial results from downstream peers S_2 and S_3. Therefore, S_4 sends requests $< x_2{:}C_2 >$ and $< x_3{:}C_3 >$ to these servers indicating S_2 and S_3 would return results after reading their local chunks C_2 and C_3. Before returning the results, servers S_2 and S_3 also multiply chunks C_2 and C_3 by their corresponding coefficients x_2 and x_3, respectively. As part of the *same* repair plan, S_4 also receives a request $< x_4{:}C_4 >$ from its upstream peer (in this case the repair site). Thus S_4 schedules a local disk read for chunk C_4, which is then multiplied by the coefficient x_4. S_4 waits for results from S_2 and S_3 and performs incremental XORs before replying to its upstream peer with an aggregated result.

The repair site aggregates the results by XORing all the results coming from the downstream Chunk Servers to reconstruct the missing chunk and writes back to the disk at the end of the operation. Finally, this destination Chunk Server sends a message to the RM indicating a successful completion of the repair operation.

Reconstruction during degraded reads: If a degraded read operation triggers the PPR-based reconstruction, a Client acts as the repair site and informs the RM about a missing chunk. Then the RM distributes a repair plan with the *highest priority*.

Tail completion: The number of flows, as well as the number of nodes involved in PPR, is exactly the same as in traditional repair. It is equal to k. Since k is small in practice (between 6 and 12), the probability of encountering a relatively slow node is small in both traditional repair and PPR. Nevertheless, the RM uses node usage statistics (CPU and I/O counters collected with the Heartbeat messages) to de-prioritize the slow nodes before creating the repair plan. If reconstruction does not complete within a certain time threshold (because of unpredictable congestion or failures),



(a) Three major components in the QFS architecture

(b) Message exchanges among Chunk Servers and the RM for RS(6, 3) reconstruction using PPR

Figure 5: (a) QFS architecture and (b) PPR protocol timeline

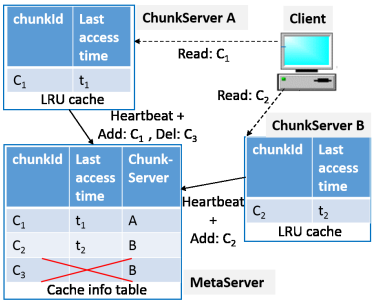


Figure 6: Protocol for LRU cache. Updates are piggybacked with heartbeat messages

the RM reschedules the reconstruction process with a new repair plan.

6.3 IO pipelining, caching, and efficient use of memory

Overlapping disk IO with network transfer: Disk IO (read/write) time is another dominant component in the overall reconstruction time. Aggregation Chunk Servers that had posted downstream requests (*e.g.*, S_2, S_4, S_6), read *different* chunks from disk and wait[8] for data transfer from their downstream peer Chunk Servers to complete. Then they apply the aggregating XOR operation and send the result to further upstream servers in the tree. To increase parallelism, aggregation Chunk Servers schedule IO-reads in parallel with data transfer from network.

Caching: We attempt to further reduce the impact of IO-read time by introducing an *in-memory* caching mechanism in Chunk Servers, as described in Section 4.4. When choosing k out of the remaining $k + m - 1$ Chunk Servers for a reconstruction operation in m-PPR protocol, RM gives higher priority to hot chunks but tries to avoid *very-hot* chunks in order to minimize the impact on application performance. However, for multiple simultaneous reconstructions, we found that making sure that these reconstructions use disjoint servers has a greater benefit than cache-aware server assignment, since in general data centers are constrained by network resources.

[8] Because network transfer of a chunk usually takes longer than IO time.

6.4 Implementation details

The choice of a codebase: We implemented our technique with QFS [5] written in C++. Among several alternatives, we chose QFS because of its simpler architecture and reasonable popularity in the community. However, our PPR technique is general enough to be applicable to other widely used erasure coded storage systems. Specifically the architecture of HDFS with erasure coding [3] is almost identical to that of QFS, and therefore PPR is directly applicable. In addition, our technique can also be applied to Ceph [1], another popular distributed storage system that supports erasure coding. In Ceph, clients use a pseudo-random mapping function called CRUSH [49] to place and access data chunks, rather than relying on a centralized meta server. Nonetheless, it does have a centralized entity, called *ceph monitor* (`ceph-mon`) that knows the layout of Object Storage Devices (OSDs) (equivalent to Chunk Servers in QFS). `ceph-mon` is responsible for checking the health of each OSD, letting the newly joined OSDs know the topology, etc. Thus, we can augment such an entity with RM to enable PPR. Moreover, we can also augment *any* OSD with RM function, since all OSDs know where a given chunk is (or will be) located based on the pseudo-random mapping function.

Changes made to the codebase: To implement PPR, we have made the following changes to the QFS codebase. First, we extended the QFS code to make the chunk size configurable; QFS uses a fixed chunk size of 64MB. Second, we implemented PPR decoding operations using Jerasure and GF-Complete [33] libraries, which were not the defaults in QFS. Jerasure allows a configurable set of coding parameters, while the default in QFS only supports the (6, 3) code. Third, we augmented the Meta-Server with the RM to calculate decoding coefficients, create a repair plan, and distribute it to Chunk Servers. The RM also keeps track of the cached chunks at Chunk Servers. Fourth, the Chunk Server's state machine was modified to incorporate the PPR

protocol to communicate with the peers and the RM, and search for a chunk in its memory cache before attempting to perform disk IO. Lastly, it is worthwhile to note that our implementation of PPR-based reconstruction is fully transparent to the end user.

7. Evaluation

In this section we evaluate our implementation of PPR on top of QFS and compare the repair performance with QFS's traditional Reed-Solomon-based reconstruction technique. Our primary metric is the reduction in repair time. We also layer PPR on top of two other popular and practical erasure codes, namely LRC [22] and Rotated RS [24], and evaluate the effectiveness of PPR when used with these codes.

Experimental setup: We use two OpenStack [4] clusters—a 16 host lab cluster (SMALLSITE) and an 85 host production cluster (BIGSITE), to demonstrate the scalability advantages of PPR. In SMALLSITE, each machine belongs to one rack and has 16 physical CPU cores with 24GB RAM. Each core operates at 2.67GHz. They are connected to a 1Gbps network. Each VM instance runs Ubuntu 14.04.3 with four vcpus, 8GB memory, and 80GB of storage space. In BIGSITE, the machines have dual 10-core 2.8GHz CPUs and are connected by two bonded 10G NICs, with each NIC going to an independent ToR (Top-of-Rack) switch. However, an `iperf` test showed an average bandwidth of about 1.4Gbps between any two VMs (such lower than expected bandwidth is due to the well-know VxLAN issues, which we do not discuss here for brevity). For both proactive repair and degraded read experiments on the SMALLSITE, we kill a single Chunk Server, which affects a small number of chunks. For each experiment, we report the mean values computed from 20 runs. We measure repair time on the RM as the difference between the time when it starts a repair process and the time when it is notified by a completion message sent from the repair site. For degraded reads, we measure the latency as the time elapsed from the time instant when a client posts a read request to the time instant when it finishes reconstructing the lost chunk(s).

7.1 Performance improvement with main PPR

7.1.1 Improving regular repair performance

Fig. 7a illustrates the percentage reduction in the repair time achieved by PPR compared to the baseline traditional RS repair technique, for four different codes: (6, 3), (8, 3), (10, 4), and (12, 4), each with chunk sizes of 8MB, 16MB, 32MB, and 64MB. PPR reduces the repair time quite significantly. For a higher value of k the reduction is even higher and reaches up to 59%. This is mainly because in PPR the network transfer time increases with $log(k)$, as opposed to increasing linearly in k as in the traditional RS repair (Sec. 4.2). Another interesting observation is that PPR becomes more attractive for higher chunk sizes. To investigate this further, we performed an experiment by varying the chunk size from 8MB to 96MB for the (12, 4) RS code. Fig. 7b illustrates that the benefit of PPR is higher at higher chunk sizes, *e.g.*, 53% at 8MB while 57% at 96MB. This is because as the chunk size grows, it increases the network pressure on the link connected to the repair site, leading to a higher delay. PPR can alleviate such a situation through its partial and parallel reconstruction mechanism. It should be noted that many practical storage systems use big chunks so that relevant objects (e.g., profile photos in a social networking applications) can be contained within a single chunk, thereby avoiding the need to fetch multiple chunks during user interaction.

7.1.2 Improving degraded read latency

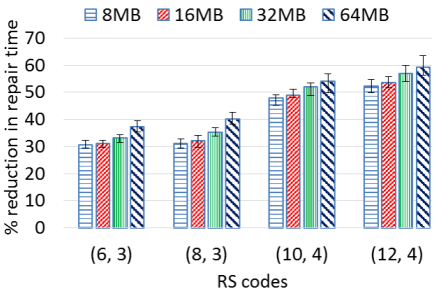
Recall that a degraded read happens when a user submits a read request for some data that is currently unavailable. As a result, the requested chunk must be reconstructed on the fly at the client before the system replies to the user request. Fig. 7c illustrates how PPR can drastically reduce the degraded read latency for four common RS coding parameters: (6, 3), (8, 3), (10, 4), and (12, 4), and for two different chunk sizes: 8MB and 64MB. Fig. 7c shows that the reduction in the degraded read latency becomes more prominent for the codes with higher values of k. Moreover, it is also noticeable that at a higher chunk size PPR provides even more benefits because of the reason discussed in Section 7.1.1.

7.2 Improving degraded reads under constrained bandwidth

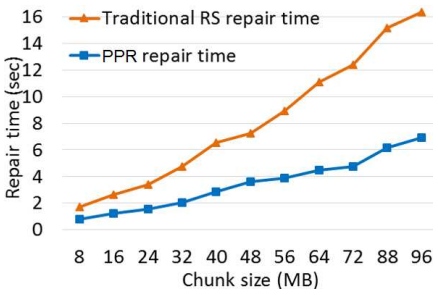
PPR not only reduces the reconstruction time but also reduces the *maximum amount of data* transferred to any Chunk Server or a Client involved in the reconstruction process. In a PPR-based reconstruction process, a participating Chunk Server needs to transfer only $\lceil(log_2(k+1))\rceil$ number of chunks over its network link, as opposed to k number of chunks in a traditional repair. This becomes a desirable property when the network is heavily loaded or under-provisioned. In the next experiment, we use the Linux traffic control implementation (*tc*) to control the network bandwidth available to all the servers and measure the degraded read throughput. As shown in Fig. 7d, as we decrease the available bandwidth from 1Gbps to 200Mbps, the degraded read throughput with the traditional RS reconstruction rapidly drops to 1.2MB/s and 0.8MB/s for RS(6, 3) and RS(12, 4), respectively. Since, network transfers are distributed in PPR, it achieves higher throughput—8.5MB/s and 6.6MB/s for RS(6, 3) and RS(12, 4), respectively. With a relatively well-provisioned network (1Gbps), the gains of PPR are $1.8X$ and $2.5X$, while with the constrained bandwidth (200Mbps), the gains become even more significant, almost $7X$ and $8.25X$.

7.3 Benefit from caching

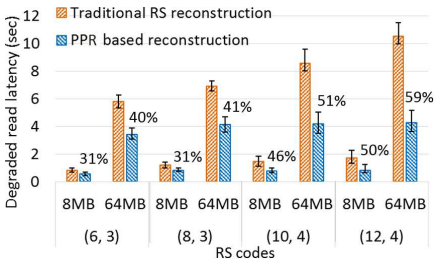
In this section we evaluate the individual contribution of the distributed reconstruction technique and caching mecha-



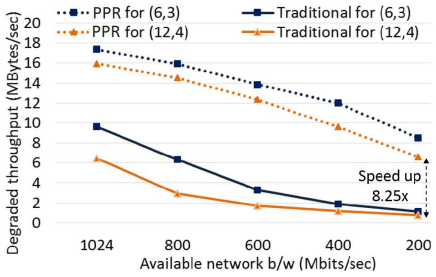
(a) Percentage reduction in repair time with PPR over baseline Reed-Solomon code for different chunk sizes and coding parameters



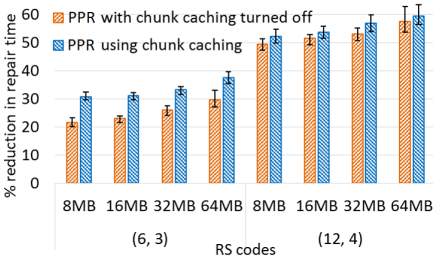
(b) Traditional repair vs. PPR using RS (12, 4) code. PPR's benefit becomes more obvious as we increase the chunk size



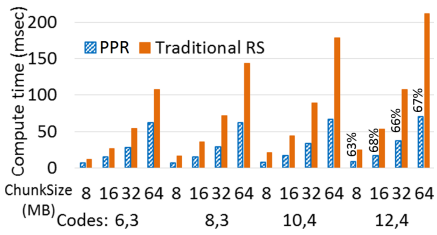
(c) Improvement in degraded read latency



(d) Degraded read throughput under constrained bandwidth



(e) Percentage reduction: PPR without chunk caching vs. PPR with chunk caching. The baseline is standard RS code.



(f) Improved computation time during reconstruction

Figure 7: Performance evaluation on SMALLSITE with a small number of chunk failures

nism to the overall benefit of PPR. The former reduces the network transfer time, while the latter reduces the disk IO time. Fig. 7e shows that chunk caching is more useful for lower values of k (*e.g.*, (6, 3) code). For higher values of k or for higher chunk sizes, the benefit of caching becomes marginal because the improvement in the network transfer time dominates that of the disk IO time. For instance, for $k = 12$ and 64MB chunk size, the caching mechanism provides only 2% additional savings in the total repair time. However, the caching mechanism can reduce the demand on disk IO, making it available for other workloads. Knowing the exact access patterns of data chunks will help us identify better caching strategies and choose the right cache size. We leave such exploration in realistic settings for future work.

7.4 Improvement in computation time

Now we compare PPR's computation to the serial computation in a traditional RS reconstruction, *i.e.*, a default QFS implementation with the Jerasure 2.0 library [33]. Note that during reconstruction, either *decoding* (when a data chunk is lost) or *encoding* (when a parity chunk is lost) can happen (Fig. 3b). The amounts of computation required by RS encoding and decoding are almost identical [34]. The only difference is the extra matrix inversion involved in decoding. During our experiments we randomly killed a Chunk Server to create an erasure. Since, for the codes we used, there are more data chunks than parity chunks ($k > m$), decoding happens with higher probability than encoding. We report the average numbers and do not explicitly distinguish based on the type of the lost chunk. Fig. 7f shows that PPR can significantly speed up the computation time using its parallelism. These gains are consistent across different chunk sizes. Moreover, the gain is higher for higher values of k because the critical path in PPR needs fewer multiplications and XOR operations compared to traditional decoding. Existing techniques to reduce computation time for erasure codes using GPUs [15] or hardware acceleration techniques [11, 23] are complementary to PPR. They can serve as drop-in replacements to the current Jerasure library used by PPR. However, it should be noted, repair in erasure-coded storage is not a compute-bound task, but a network-bound task. Nevertheless, PPR helps to reduce the overall computation time.

7.5 Evaluation with simultaneous failures (*m*-PPR)

In this section we evaluate the effectiveness of m-PPR in scheduling multiple repairs caused by simultaneous chunk failures. We control the number of simultaneous chunk failures by killing the appropriate number of Chunk Servers. We performed this experiment in the BIGSITE, where we placed the Meta-Server and the Client on two hosts and ran 83 Chunk Servers on the rest. The coding scheme was RS(12, 4) with 64MB chunks. Fig. 8 shows that our technique provides a significant reduction (31%–47%) in total repair time com-

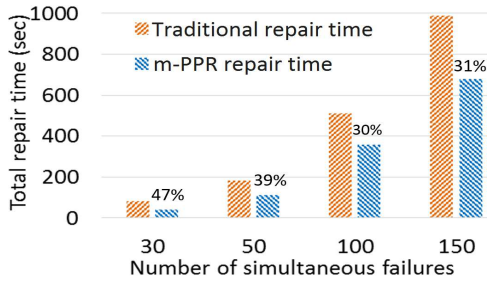


Figure 8: Comparison of total repair time for simultaneous failures triggered by Chunk Server crash

pared to the traditional RS repair. However, the benefit seems to decrease with a higher number of simultaneous failures. This is because, in our testbed configuration, the network links to the host servers that are shared between multiple repairs tend to get congested for large number of failures. Consequently *m*-PPR has less flexibility in choosing the repair servers. If the testbed has more resources (more hosts, higher network capacity, etc.), *m*-PPR will perform much better for the scale of simultaneous failures considered in our experiments. However, it should be noted that the main PPR technique does not reduce the total amount of data transferred over the network during repair. Rather it distributes the network traffic more uniformly across network links. For a large number of simultaneous failures, if the repair sites are spread across the data center, *m*-PPR would provide reduced benefit compared to the single failure case. This is because the simultaneous repair processes on multiple nodes already spread the network traffic more evenly compared to the case of a single failure. Overall, the result validates that *m*-PPR can effectively handle multiple repairs and minimizes the competition for shared resources (*e.g.*, network and disk) for a moderate number of simultaneous failures.

7.6 Scalability of the Repair-Manager

The Repair-Manager (RM) creates and distributes a repair plan to a few Chunk Servers that are selected as the aggregators. We investigate if the RM can become the bottleneck at large scale. As detailed in Sec. 5, the *m*-PPR scheduling algorithm has a time complexity of $O(Nlog(N))$ for scheduling each repair, where N is the number of possible destination servers. N is usually a small fraction of the total number of machines in the data center. Additionally, to handle a data chunk failure, RM computes the decoding coefficients, which involves a matrix inversion. Following this, RM sends $(1 + \frac{k}{2})$ messages to distribute the plan to the aggregation Chunk Servers. Not surprisingly, we observe that the time for coefficient calculation is negligible. Specifically for RS (6, 3) and (12, 4) codes, we measured the time period between the instant when the plan is created to the instant when the RM finishes distributing the plan for a single repair. It took on average 5.3ms and 8.7ms respectively. Thus for the two coding schemes, one instance of the RM is capable of handling 189 repairs/sec and 115 repairs/sec, respectively. Further, as discussed in Sec. 5, the planning capability can be easily parallelized by using multiple RM instances, each of which can handle disjoint sets of repairs.

7.7 Compatibility with other repair-friendly codes

PPR is compatible with most of the existing erasure coding techniques. Its applicability is not limited to only RS codes. We demonstrate its compatibility by applying it on top of two popular erasure coding techniques—Local Reconstruction Code (LRC) [22] and Rotated RS code [24]. These are the state-of-the-art codes targeted for reducing the repair time.

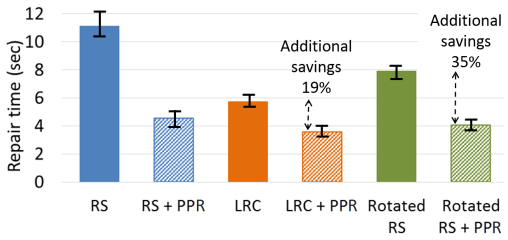


Figure 9: PPR repair technique can work with LRC and Rotated RS and can provide additional improvement in repair time

Improvements over LRC code: Huang *et al.* introduced Local Reconstruction Code (LRC) in Windows Azure Storage to reduce the network traffic and disk IO during the reconstruction process [22]. LRC stores additional local parities for subgroups of chunks, thereby increasing the storage overhead for comparable reliability. For example, a (12, 2, 2) LRC code uses two global parities and two local parities, one each for a subgroup of six chunks. If one chunk in a subgroup fails, LRC needs only six other chunks to reconstruct the original data compared to 12 in RS (12, 4) code. Papailiopoulos *et al.* [31] and Sathiamoorthy *et al.* [42] also proposed Locally Repairable Codes that are conceptually similar. For our experiments, we emulated a (12, 2, 2) LRC code that transfers six chunks over the network, in the best case, to one Chunk Server in order to reconstruct a missing chunk. Then we applied PPR-based reconstruction technique for LRC to create `LRC+PPR`.

In `LRC+PPR` only three chunks are transferred over any particular network link. In Fig. 9, for a 64MB chunk size, PPR-based reconstruction on (12, 4) RS code was faster than a (12, 2, 2) LRC code reconstruction because the maximum number of chunks that must go through any particular network link is only $4C$ for PPR as opposed to $6C$ in case of LRC, where C is the chunk size. More interestingly, `LRC+PPR` version performs even better resulting in 19% additional reduction, compared to using LRC alone. Even in the worst case, for the `LRC+PPR` only three chunks are transferred over any particular network link.

Improvements over Rotated RS code: Khan *et al.* [24] proposed Rotated RS code that modifies the classic RS code in two ways: a) each chunk belonging to a single stripe is further divided into r sub-chunks and b) XOR on the encoded data fragments are not performed within a row but across adjacent rows. For Rotated RS code, the repair of r failed chunks (called "fragments" in [24]), requires exactly $\frac{r}{2}(k + \lceil(\frac{k}{m})\rceil)$ other symbols when r is even, compared to $r \times k$ data fragments in the RS code. On an average, for a RS(12, 4) code and $r = 4$ (as used by the authors [24]), the reconstruction of a single chunk requires approximately nine other chunks, as opposed to 12 chunks in traditional RS codes. However, the reconstruction is still performed after gathering *all* the necessary data on a single Chunk Server. As can be observed from Fig. 9, PPR with RS code outperforms Rotated RS. Moreover, the combined version `Rotated RS`+PPR performs even better and results in 35% additional reduction compared to the traditional RS repair.

7.8 Discussion

It is worthwhile to discuss whether emerging technologies, such as the zero-copy-based high throughput networks (e.g., Remote Direct Memory Access (RDMA)), would remove the network bottleneck. However, it should be noted that other system components are also getting better in performance. For example, Non-Volatile Memory Express (NVMe) and hardware-accelerator-based EC computation have the potential to make the non-network components to be even faster. Moreover, application data is likely to grow exponentially putting even more pressure on the future data center network. Thus, techniques like PPR that attempt to reduce the network bottleneck would still be relevant.

8. Related Work

Quantitative comparison studies have shown that EC has lower storage overhead than replication while providing better or similar reliability [41, 48]. TotalRecall [12] dynamically predicts the availability level of different files and applies EC or replication accordingly. Publications from Facebook [29] and Microsoft [22] discuss the performance optimizations and fault tolerance of their EC storage systems.

A rich body of work targets the reconstruction problem in EC storage. Many new codes have been proposed to reduce the amount of data needed during reconstruction. They achieve this either by increasing the storage overheads [18, 22, 31, 37], or under restricted scope [21, 24, 47, 51]. We have already covered the idea behind Local Reconstruction Codes [22] and the conceptually identical Locally Repairable Codes [31, 42] when presenting our evaluation of PPR coupled with these codes. The main advantage of our technique compared to these is that PPR neither requires additional storage overhead nor mandates a specific coding scheme. Moreover, our technique is fully compatible with these codes and can provide additional benefits if used together with them, as shown in our evaluation. Another body of work suggests new coding schemes to reduce the amount of repair and IO traffic, but comes with restricted settings. Examples are Rotated RS [24] and Hitchhiker [38]. Yet another class of optimized recovery algorithms are EVENODD [51] and RDP codes [47]. However, they support only two parities, making them less useful for many systems [38]. In contrast, PPR can work with any EC code.

In a different context, Silberstein *et al.* [44] proposed that delaying repairs can lead to bandwidth conservation and marginally increases the performance of degraded reads as well. However, such a policy decision will not be applicable to many scenarios because it puts the reliability of the data at risk. Xia *et al.* [50] proposed a hybrid technique using two different codes in the same system, *i.e.*, a fast code and a compact code. They attempted to achieve faster recovery for frequently accessed files using the fast code, and to get lower storage overhead for the less frequently accessed files using the compact code. This technique is orthogonal to our work, and PPR can again be used for *both* fast and compact codes to make reconstruction faster. In the context of reliability in replicated systems, Chain Replication [46] discusses how the number of possible replica sets affects the data durability. Carbonite [16] explores how to improve reliability while minimizing replica maintenance under transient failures. These are orthogonal to PPR. Lastly, several papers evaluate advantages of deploying EC in distributed storage systems. OceanStore [25, 40] combines replication and erasure coding for WAN storage to provide highly scalable and durable storage composed of untrusted servers.

9. Conclusion

In this paper we present a distributed reconstruction technique called PPR, for erasure coded storage. This achieves reduction in the time needed to reconstruct missing or corrupted data chunks, without increasing the storage requirement or lowering data reliability. Our technique divides the reconstruction into a set of partial operations and schedules them in parallel using a distributed protocol that overlays a reduction tree to aggregate the results. We introduce a scheduling algorithm called *m*-PPR for handling concurrent failures that coordinates multiple reconstructions in parallel while minimizing the conflict for shared resources. Our experimental results show PPR can reduce the reconstruction time by up to 59% for a (12, 4) Reed-Solomon code and can improve the degraded read throughput by 8.25X, which can be directly perceived by the users. Our technique is compatible with many existing codes and we demonstrate how PPR can provide additional savings on latency when used with other repair-friendly codes.

Acknowledgments

We thank all the reviewers and our shepherd Prof. Lorenzo Alvisi for their constructive feedback and suggestions.

References

[1] Ceph http://ceph.com/.

[2] Google Colossus File System: http://static.googleusercontent.com/media/research.google.com/en/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf.

[3] Erasure Coding Support inside HDFS: https://issues.apache.org/jira/browse/HDFS-7285.

[4] OpenStack: Open source software for creating private and public clouds: http://www.openstack.org/.

[5] Quantcast File System http://quantcast.github.io/qfs/.

[6] OpenStack Object Storage (Swift): http://swift.openstack.org.

[7] Big Data and What it Means: http://www.uschamberfoundation.org/bhq/big-data-and-what-it-means.

[8] Yahoo Cloud Object Store: http://yahooeng.tumblr.com/post/116391291701/yahoo-cloud-object-store-object-storage-at.

[9] A. Akella, T. Benson, B. Chandrasekaran, C. Huang, B. Maggs, and D. Maltz. A universal approach to data center network design. In *ICDCN*, 2015.

[10] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. 2008.

[11] L. Atieno, J. Allen, D. Goeckel, and R. Tessier. An adaptive reed-solomon errors-and-erasures decoder. In *ACM/SIGDA FPGA*, 2006.

[12] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker. Total recall: System support for automated availability management. In *NSDI*, 2004.

[13] k. Björck and V. Pereyra. Solution of vandermonde systems of equations. *Mathematics of Computation*, 1970.

[14] B. Calder, J. Wang, A. Ogus, N. Nilakantan, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[15] X. Chu, C. Liu, K. Ouyang, L. S. Yung, H. Liu, and Y.-W. Leung. Perasure: a parallel cauchy reed-solomon coding library for gpus. In *IEEE ICC*, 2015.

[16] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *NSDI*, 2006.

[17] A. G. Dimakis, P. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 2010.

[18] K. S. Esmaili, L. Pamies-Juarez, and A. Datta. Core: Cross-object redundancy for efficient data repair in storage systems. In *IEEE International Conference on Big Data*, 2013.

[19] 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 *OSDI*, 2010.

[20] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *SIGCOMM*, 2009.

[21] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang. Nccloud: applying network coding for the storage repair in a cloud-of-clouds. In *FAST*, 2012.

[22] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In *ATC*, 2012.

[23] H. M. Ji. An optimized processor for fast reed-solomon encoding and decoding. In *IEEE ICASSP*, 2002.

[24] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads. In *FAST*, 2012.

[25] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[26] J. Li and B. Li. Beehive: erasure codes for fixing multiple failures in distributed storage systems. In *HotStorage*, 2015.

[27] F. J. MacWilliams and N. J. A. Sloane. The theory of error correcting codes. *North-Holland*, 1977.

[28] S. Mitra, I. Laguna, D. H. Ahn, S. Bagchi, M. Schulz, and T. Gamblin. Accurate application progress analysis for large-scale parallel debugging. In *PLDI*, 2014.

[29] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. F4: Facebooks warm blob storage system. In *OSDI*, 2014.

[30] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The quantcast file system. *VLDB*, 2013.

[31] D. S. Papailiopoulos and A. G. Dimakis. Locally repairable codes. *IEEE Transactions on Information Theory*, 2014.

[32] J. S. Plank and L. Xu. Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications. In *IEEE International Symposium on Network Computing and Applications (NCA)*, 2006.

[33] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2. Technical report, Technical Report CS-08-627, University of Tennessee, 2008.

[34] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, Z. Wilcox-O'Hearn, et al. A performance evaluation and examination of open-source erasure coding libraries for storage. In *FAST*, 2009.

[35] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *HotStorage*, 2013.

[36] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth. In *FAST*, 2015.

[37] K. V. Rashmi, N. B. Shah, and P. V. Kumar. Optimal exact-regenerating codes for distributed storage at the msr and mbr points via a product-matrix construction. *IEEE Transactions on Information Theory*, 2011.

[38] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *SIGCOMM*, 2014.

[39] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *SIAM*, 1960.

[40] S. C. Rhea, P. R. Eaton, D. Geels, H. Weatherspoon, B. Y. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *FAST*, 2003.

[41] R. Rodrigues and B. Liskov. High availability in dhts: Erasure coding vs. replication. In *Peer-to-Peer Systems*. Springer, 2005.

[42] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. In *VLDB*, 2013.

[43] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST)*, 2010.

[44] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *SYSTOR*, 2014.

[45] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *International Journal of High Performance Computing Applications*, 2005.

[46] R. Van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[47] Z. Wang, A. G. Dimakis, and J. Bruck. Rebuilding for array codes in distributed storage systems. In *GLOBECOM Workshop*, 2010.

[48] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Peer-to-Peer Systems*. Springer, 2002.

[49] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. Crush: Controlled, scalable, decentralized placement of replicated data. In *ACM/IEEE conference on Supercomputing*, 2006.

[50] M. Xia, M. Saxena, M. Blaum, and D. A. Pease. A tale of two erasure codes in hdfs. In *FAST*, 2015.

[51] L. Xiang, Y. Xu, J. Lui, and Q. Chang. Optimal recovery of single disk failure in rdp code storage systems. In *SIGMETRICS*, 2010.