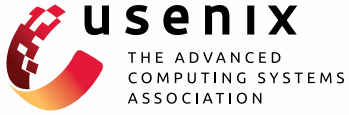


Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth

KV Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan Ramchandran, ***University of California, Berkeley***

https://www.usenix.org/conference/fast15/technical-sessions/presentation/rashmi

This paper is included in the Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15).

February 16–19, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-201

Open access to the Proceedings of the 13th USENIX Conference on File and Storage Technologies is sponsored by USENIX

Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage and Network-bandwidth

K. V. Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, Kannan Ramchandran
University of California, Berkeley

Abstract

Erasure codes, such as Reed-Solomon (RS) codes, are increasingly being deployed as an alternative to data-replication for fault tolerance in distributed storage systems. While RS codes provide significant savings in storage space, they can impose a huge burden on the I/O and network resources when reconstructing failed or otherwise unavailable data. A recent class of erasure codes, called minimum-storage-regeneration (MSR) codes, has emerged as a superior alternative to the popular RS codes, in that it minimizes network transfers during reconstruction while also being optimal with respect to storage and reliability. However, existing practical MSR codes do not address the increasingly important problem of I/O overhead incurred during reconstructions, and are, in general, inferior to RS codes in this regard. In this paper, we design erasure codes that are *simultaneously optimal in terms of I/O, storage, and network bandwidth*. Our design builds on top of a class of powerful practical codes, called the product-matrix-MSR codes. Evaluations show that our proposed design results in a significant reduction the number of I/Os consumed during reconstructions (a $5\times$ reduction for typical parameters), while retaining optimality with respect to storage, reliability, and network bandwidth.

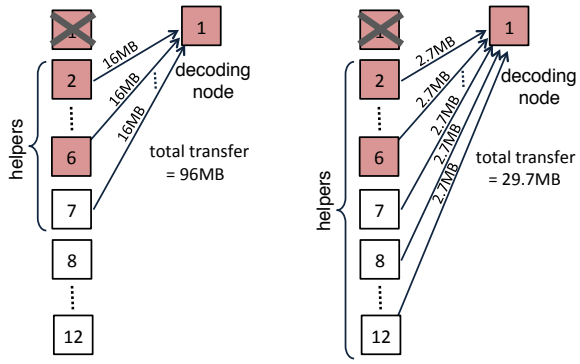
1 Introduction

The amount of data stored in large-scale distributed storage architectures such as the ones employed in data centers is increasing exponentially. These storage systems are expected to store the data in a reliable and available fashion in the face of multitude of temporary and permanent failures that occur in the day-to-day operations of such systems. It has been observed in a number of studies that failure events that render data unavailable occur quite frequently in data centers (for example, see [32, 14, 28] and references therein). Hence, it is imperative that the data is stored in a redundant fashion.

Traditionally, data centers have been employing triple replication in order to ensure that the data is reliable and that it is available to the applications that wish to consume it [15, 35, 10]. However, more recently, the enormous amount of data to be stored has made replication an expensive option. Erasure coding offers an alternative means of introducing redundancy, providing higher levels of reliability as compared to replication while requiring much lower storage overheads [5, 39, 37]. Data centers and cloud storage providers are increasingly turning towards this option [14, 9, 10, 4], with Reed-Solomon (RS) codes [31] being the most popular choice. RS codes make optimal use of storage resources in the system for providing reliability. This property makes RS codes appealing for large-scale, distributed storage systems where storage capacity is one of the critical resources [1]. It has been reported that Facebook has saved multiple Petabytes of storage space by employing RS codes instead of replication in their data warehouse cluster [3].

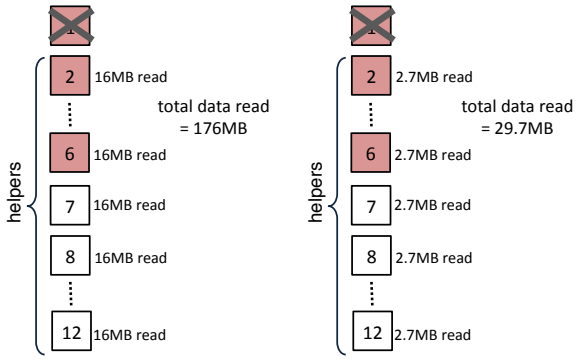
Under RS codes, redundancy is introduced in the following manner: a file to be stored is divided into equal-sized units, which we will call *blocks*. Blocks are grouped into sets of k each, and for each such set of k blocks, r *parity* blocks are computed. The set of these $(k+r)$ blocks consisting of both the data and the parity blocks constitute a *stripe*. The parity blocks possess the property that any k blocks out the $(k+r)$ blocks in a stripe suffice to recover the entire data of the stripe. It follows that the failure of any r blocks in a stripe can be tolerated without any data loss. The data and parity blocks belonging to a stripe are placed on different nodes in the storage network, and these nodes are typically chosen from different racks.

Due to the frequent temporary and permanent failures that occur in data centers, blocks are rendered unavailable from time-to-time. These blocks need to be replaced in order to maintain the desired level of reliability. Under RS codes, since there are no replicas, a missing block



(a) RS (b) MSR (e.g., PM-MSR)

Figure 1: Amount of data transfer involved in reconstruction of block 1 for an RS code with $k = 6$, $r = 6$ and an MSR code with $k = 6$, $r = 6$, $d = 11$, with blocks of size 16MB. The data blocks are shaded.



(a) PM-MSR (b) PM-RBT (this paper)

Figure 2: Amount of data read from disks during reconstruction of block 1 for $k = 6$, $r = 6$, $d = 11$ with blocks of size 16MB.

is replaced by downloading *all* data from (any) k other blocks in the stripe and decoding the desired data from it. We will refer to this operation as a *reconstruction* operation. A reconstruction operation may also be called upon to serve read requests for data that is currently unavailable. Such read requests are called *degraded reads*. Degraded reads are served by reconstructing the requisite data on-the-fly, i.e., immediately as opposed to as a background job.

Let us look at an example. Consider an RS code with $k = 6$ and $r = 6$. While this code has a storage overhead of $2x$, it offers orders of magnitude higher reliability than $3x$ replication. Figure 1a depicts a stripe of this code, and also illustrates the reconstruction of block 1 using the data from blocks 2 to 7. We call the blocks that are called upon during the reconstruction process as *helpers*. In Figure 1a, blocks 2 to 7 are the helpers. In this example, in order to reconstruct a 16MB block, $6 \times 16MB = 96$MB of data is read from disk at the helpers and transferred across the network to the node performing the decoding computations. In general, the disk read and the network transfer overheads during a reconstruction operation is *k times* that under replication. Consequently, under RS codes, reconstruction operations result in a large amount of disk I/O and network transfers, putting a huge burden on these system resources.

There has been considerable interest in the recent past in designing a new class of (network-coding based) codes called *minimum-storage-regenerating (MSR) codes*, which were originally formulated in [12]. Under an MSR code, an unavailable block is reconstructed by downloading a small fraction of the data from any d ($> k$) blocks in the stripe, in a manner that the total amount of data transferred during reconstruction is lower than that in RS codes. Let us consider an example with $k = 6$, $r = 6$ and $d = 11$. For these parameters, reconstruction of block 1 under an MSR code is illustrated in Figure 1b. In this example, the total network transfer is only 29.7MB as opposed to 96MB under RS. MSR codes are optimal with respect to storage and network transfers: the storage capacity and reliability is identical to that under RS codes, while the network transfer is significantly lower than that under RS and in fact, is the minimum possible under any code. However, MSR codes do not optimize with respect to I/Os. The I/O overhead during a reconstruction operation in a system employing an MSR code is, in general, higher than that in a system employing RS code. This is illustrated in Figure 2a which depicts the amount data read from disks for reconstruction of block 1 under a practical construction of MSR codes called product-matrix-MSR (PM-MSR) codes. This entails reading 16MB at each of the 11 helpers, totaling 176MB of data read.

I/Os are a valuable resource in storage systems. With the increasing speeds of newer generation network interconnects and the increasing storage capacities of individual storage devices, I/O is becoming the primary bottleneck in the performance of storage systems. Moreover, many applications that the storage systems serve today are I/O bound, for example, applications that serve a large number of user requests [7] or perform data-intensive computations such as analytics [2]. Motivated by the increasing importance of I/O, in this paper, we investigate practical erasure codes for storage systems that are I/O optimal.

In this paper, we design erasure codes that are *simultaneously optimal in terms of I/O, storage, and network bandwidth* during reconstructions. We first identify two properties that aid in transforming MSR codes to be disk-read optimal during reconstruction while retaining their storage and network optimality. We show that a class

of powerful practical constructions for MSR codes, the product-matrix-MSR codes (PM-MSR) [29], indeed satisfy these desired properties. We then present an algorithm to transform *any* MSR code satisfying these properties into a code that is optimal in terms of the amount of data read from disks. We apply our transformation to PM-MSR codes and call the resulting I/O optimal codes as PM-RBT codes. Figure 2b depicts the amount of data read for reconstruction of block 1: PM-RBT entails reading only 2.7MB at each of the 11 helpers, totaling 29.7MB of data read as opposed to 176MB under PM-MSR. We note that the PM-MSR codes operate in the regime $r \geq k-1$, and consequently the PM-RBT codes also operate in this regime.

We implement PM-RBT codes and show through experiments on Amazon EC2 instances that our approach results in 5× reduction in I/Os consumed during reconstruction as compared to the original product-matrix codes, for a typical set of parameters. For general parameters, the number of I/Os consumed would reduce approximately by a factor of $(d-k+1)$. For typical values of d and k, this can result in substantial gains. We then show that if the relative frequencies of reconstruction of blocks are different, then a more holistic system-level design of helper assignments is needed to optimize I/Os across the entire system. Such situations are common: for instance, in a system that deals with degraded reads as well as node failures, the data blocks will be reconstructed more frequently than the parity blocks. We pose this problem as an optimization problem and present an algorithm to obtain the optimal solution to this problem. We evaluate our helper assignment algorithm through simulations using data from experiments on Amazon EC2 instances.

2 Background

2.1 Notation and Terminology

The computations for encoding, decoding, and reconstruction in a code are performed using what is called *finite-field* arithmetic. For simplicity, however, throughout the paper the reader may choose to consider usual arithmetic without any loss in comprehension. We will refer the smallest granularity of the data in the system as a *symbol*. The actual size of a symbol is dependent on the finite-field arithmetic that is employed. For simplicity, the reader may consider a symbol to be a single byte.

A vector will be column vector by default. We will use boldface to denote column vectors, and use T to denote a matrix or vector transpose operation.

We will now introduce some more terminology in addition to that introduced in Section 1. This terminology is illustrated in Figure 3. Let n $(=k+r)$ denote the total number of blocks in a stripe. In order to encode the k data blocks in a stripe, each of the k data blocks are first divided into smaller units consisting of w symbols each. A set of w symbols from each of the k blocks is encoded to obtain the corresponding set of w symbols in the parity blocks. We call the set of data and parities at the granularity of w symbols as a *byte-level* stripe. Thus in a byte-level stripe, $B = kw$ original data symbols (w symbols from each of the k original data blocks) are encoded to generate nw symbols (w symbols for each of the n encoded blocks). The data symbols in different byte-level stripes are encoded independently in an identical fashion. Hence in the rest of the paper, for simplicity of exposition, we will assume that each block consists of a single byte-level stripe. We denote the B original data symbols as $\{m_1,\ldots,m_B\}$. For $i \in \{1,\ldots,n\}$, we denote the w symbols stored in block i by $\{s_{i1},\ldots,s_{iw}\}$. The value of w is called the *stripe-width*.

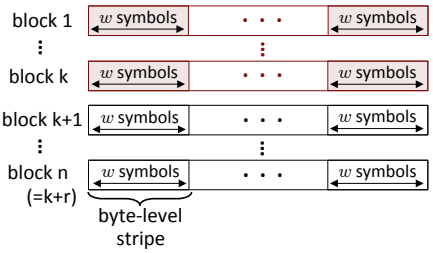


Figure 3: Illustration of notation: hierarchy of symbols, byte-level stripes and block-level stripe. The first k blocks shown shaded are systematic.

During reconstruction of a block, the other blocks from which data is accessed are termed the *helpers* for that reconstruction operation (see Figure 1). The accessed data is transferred to a node that performs the decoding operation on this data in order to recover the desired data. This node is termed the *decoding node*.

2.2 Linear and Systematic Codes

A code is said to be *linear*, if all operations including encoding, decoding and reconstruction can be performed using linear operations over the finite field. Any linear code can be represented using a $(nw \times B)$ matrix G, called its *generator matrix*. The nw encoded symbols can be obtained by multiplying the generator matrix with a vector consisting of the B message symbols:

$$G \left[\begin{array}{cccc} m_1 & m_2 & \cdots & m_B \end{array} \right]^T . \qquad (1)$$

We will consider only linear codes in this paper.

In most systems, the codes employed have the property that the original data is available in unencoded (i.e., raw) form in some k of the n blocks. This property is appealing since it allows for read requests to be served

directly without having to perform any decoding operations. Codes that possess this property are called *systematic* codes. We will assume without loss of generality that the first k blocks out of the n encoded blocks store the unencoded data. These blocks are called *systematic* blocks. The remaining $(r = n-k)$ blocks store the encoded data and are called *parity* blocks. For a linear systematic code, the generator matrix can be written as

$$\begin{bmatrix} I \\ \hat{G} \end{bmatrix}, \tag{2}$$

where I is a $(B \times B)$ identity matrix, and $\hat{G}$ is a $((nw - B) \times B)$ matrix. The codes presented in this paper are systematic.

2.3 Optimality of Storage Codes

A storage code is said to be optimal with respect to the storage-reliability tradeoff if it offers maximum fault tolerance for the storage overhead consumed. A (k,r) RS code adds r parity blocks, each of the same size as that of the k data blocks. These r parity blocks have the property that any k out of these $(k+r)$ blocks are sufficient to recover all the original data symbols in the stripe. Thus failure of any r arbitrary blocks among the $(k+r)$ blocks in a stripe can be tolerated without any data loss. It is well known from analytical results [22] that this is the maximum possible fault tolerance that can be achieved for the storage overhead used. Hence, RS codes are optimal with respect to the storage-reliability tradeoff.

In [12], the authors introduced another dimension of optimality for storage codes, that of reconstruction bandwidth, by providing a lower bound on the amount of data that needs to be transferred during a reconstruction operation. Codes that meet this lower bound are termed optimal with respect to storage-bandwidth tradeoff.

2.4 Minimum Storage Regenerating Codes

A minimum-storage-regenrating (MSR) code [12] is associated with, in addition to the parameters k and r introduced Section 1, a parameter $d\,(>k)$ that refers to the number of helpers used during a reconstruction operation. For an MSR code, the stripe-width w is dependent on the parameters k and d, and is given by $w = d-k+1$. Thus, each byte-level stripe stores $B = kw = k(d-k+1)$ original data symbols.

We now describe the reconstruction process under the framework of MSR codes. A block to be reconstructed can choose *any* d other blocks as helpers from the remaining $(n-1)$ blocks in the stripe. Each of these d helpers compute some function of the w symbols stored whose resultant is a single symbol (for each byte-level stripe). Note that the symbol computed and transferred by a helper may, in general, depend on the choice of $(d-1)$ other helpers.

Consider the reconstruction of block f. Let D denote that set of d helpers participating in this reconstruction operation. We denote the symbol that a helper block h transfers to aid in the reconstruction of block f when the set of helpers is denoted by D as t_{hfD}. This resulting symbol is transferred to the decoding node, and the d symbols received from the helpers are used to reconstruct block f.

Like RS codes, MSR codes are optimal with respect to the storage-reliability tradeoff. Furthermore, they meet the lower bound on the amount of data transfer for reconstruction, and hence are optimal with respect to storage-bandwidth tradeoff as well.

2.5 Product-Matrix-MSR Codes

Product-matrix-MSR codes are a class of practical constructions for MSR codes that were proposed in [29]. These codes are linear. We consider the systematic version of these codes where the first k blocks store the data in an unencoded form. We will refer to these codes as *PM-vanilla* codes.

PM-vanilla codes exist for all values of the system parameters k and d satisfying $d \geq (2k-2)$. In order to ensure that, there are atleast d blocks that can act as helpers during reconstruction of any block, we need atleast $(d+1)$ blocks in a stripe, i.e., we need $n \geq (d+1)$. It follows that PM-vanilla codes need a storage overhead of

$$\frac{n}{k} \geq \left(\frac{2k-1}{k}\right) = 2 - \frac{1}{k}. \tag{3}$$

We now briefly describe the reconstruction operation in PM-vanilla codes to the extent that is required for the exposition of this paper. We refer the interested reader to [29] for more details on how encoding and decoding operations are performed. Every block i $(1 \leq i \leq n)$ is assigned a vector $\mathbf{g_i}$ of length w, which we will call the *reconstruction vector* for block i. Let $\mathbf{g_i} = [g_{i1}, \ldots, g_{iw}]^T$. The n $(= k+r)$ vectors, $\{\mathbf{g_1}, \ldots, \mathbf{g_n}\}$, are designed such that any w of these n vectors are linearly independent.

During reconstruction of a block, say block f, each of the chosen helpers, take a linear combination of their w stored symbols with the reconstruction vector of the failed block, $\mathbf{g_f}$, and transfer the result to the decoding node. That is, for reconstruction of block f, helper block h computes and transfers the symbol

$$t_{hfD} = \sum_{j=1}^{w} s_{hj} g_{fj}\,, \tag{4}$$

where $\{s_{h1}, \ldots, s_{hw}\}$ are the w symbols stored in block h, and D denotes the set of helpers.

PM-vanilla codes are optimal with respect to the storage-reliability tradeoff and storage-bandwidth tradeoff. However, PM-vanilla codes are not optimal with respect to the amount of data read during reconstruction: The values of most coefficients g_{fj} in the reconstruction vector are non-zero. Since the corresponding *symbol* s_{hj} must be read for every g_{fj} that is non-zero, the absence of sparsity in g_{fj} results in a large I/O overhead during the rebuilding process, as illustrated in Figure 2a (and experimentally evaluated in Figure 7).

3 Optimizing I/O during reconstruction

We will now employ the PM-vanilla codes to construct codes that optimize I/Os during reconstruction, while retaining optimality with respect to storage, reliability and network-bandwidth. In this section, we will optimize the I/Os *locally* in individual blocks, and Section 4 will build on these results to design an algorithm to optimize the I/Os *globally* across the entire system. The resulting codes are termed the *PM-RBT* codes. We note that the methods described here are more broadly applicable to other MSR codes as discussed subsequently.

3.1 Reconstruct-by-transfer

Under an MSR code, during a reconstruction operation, a helper is said to perform *reconstruct-by-transfer* (RBT) if it does not perform any computation and merely transfers one its stored symbols (per byte-level stripe) to the decoding node.[1] In the notation introduced in Section 2, this implies that $\mathbf{g_f}$ in (4) is a unit vector, and

$$t_{hfD} \in \{s_{h1}, \ldots, s_{hw}\} .$$

We call such a helper as an *RBT-helper*. At an RBT-helper, the amount of data read from the disks is *equal* to the amount transferred through the network.

During a reconstruction operation, a helper reads requisite data from the disks, computes (if required) the desired function, and transfers the result to the decoding node. It follows that the amount of network transfer performed during reconstruction forms a lower bound on the amount of data read from the disk at the helpers. Thus, a lower bound on the network transfers is also a lower bound on the amount of data read. On the other hand, MSR codes are optimal with respect to network transfers during reconstruction since they meet the associated lower bound [12]. It follows that, under an MSR code, an RBT-helper is optimal with respect to the amount of data read from the disk.

[1] This property was originally titled 'repair-by-transfer' in [34] since the focus of that paper was primarily on node failures. In this paper, we consider more general reconstruction operations that include node-repair, degraded reads etc., and hence the slight change in nomenclature.

We now present our technique for achieving the reconstruct-by-transfer property in MSR codes.

3.2 Achieving Reconstruct-by-transfer

Towards the goal of designing reconstruct-by-transfer codes, we first identify two properties that we would like a helper to satisfy. We will then provide an algorithm to convert any (linear) MSR code satisfying these two properties into one that can perform reconstruct-by-transfer at such a helper.

Property 1*:* The function computed at a helper is independent of the choice of the remaining $(d-1)$ helpers. In other words, for any choice of h and f, t_{hfD} is independent of D (recall the notation t_{hfD} from Section 2.4).

This allows us to simplify the notation by dropping the dependence on D and referring to t_{hfD} simply as t_{hf}.

Property 2*:* Assume Property 1 is satisfied. Then the helper would take $(n-1)$ linear combinations of its own data to transmit for the reconstruction of the other $(n-1)$ blocks in the stripe. We want every w of these $(n-1)$ linear combinations to be linearly independent.

We now show that under the product-matrix-MSR (PM-vanilla) codes, every helper satisfies the two properties enumerated above. Recall from Equation (4), the computation performed at the helpers during reconstruction in PM-vanilla codes. Observe that the right hand side of Equation (4) is independent of 'D', and therefore the data that a helper transfers during a reconstruction operation is dependent only on the identity of the helper and the block being reconstructed. The helper, therefore, does not need to know the identity of the other helpers. It follows that PM-vanilla codes satisfy Property 1.

Let us now investigate Property 2. Recall from Equation (4), the set of $(n-1)$ symbols, $\{t_{h1}, \ldots, t_{h(h-1)}, t_{h(h+1)}, \ldots, t_{hn}\}$, that a helper block h transfers to aid in reconstruction of each the other $(n-1)$ blocks in the stripe. Also, recall that the reconstruction vectors $\{\mathbf{g}_1, \ldots, \mathbf{g}_n\}$ assigned to the n blocks are chosen such that every w of these vectors are linearly independent. It follows that for every block, the $(n-1)$ linear combinations that it computes and transfers for the reconstruction of the other $(n-1)$ blocks in the stripe have the property of any w being independent. PM-vanilla codes thus satisfy Property 2 as well.

PM-vanilla codes are optimal with respect to the storage-bandwidth tradeoff (Section 2.3). However, these codes are not optimized in terms of I/O. As we will show through experiments on the Amazon EC2 instances (Section 5.3), PM-vanilla codes, in fact, have a higher I/O overhead as compared to RS codes. In this section, we will make use of the two properties listed above to transform the PM-vanilla codes into being I/O optimal for reconstruction, while retaining its properties of being storage and network optimal. While we focus on the PM-

vanilla codes for concreteness, we remark that the technique described is generic and can be applied to any (linear) MSR code satisfying the two properties listed above.

Under our algorithm, each block will function as an RBT-helper for some w other blocks in the stripe. For the time being, let us assume that for each helper block, the choice of these w blocks is given to us. Under this assumption, Algorithm 1 outlines the procedure to convert the PM-vanilla code (or in general any linear MSR code satisfying the two aforementioned properties) into one in which every block can function as an RBT-helper for w other blocks. Section 4 will subsequently provide an algorithm to make the choice of RBT-helpers for each block to optimize the I/O cost across the entire system.

Let us now analyze Algorithm 1. Observe that each block still stores w symbols and hence Algorithm 1 does not increase the storage requirements. Further, recall from Section 2.5 that the reconstruction vectors $\left\{\mathbf{g}_{i_{h1}}, \cdots, \mathbf{g}_{i_{hw}}\right\}$ are linearly independent. Hence the transformation performed in Algorithm 1 is an invertible transformation within each block. Thus the property of being able to recover all the data from any k blocks continues to hold as under PM-vanilla codes, and the transformed code retains the storage-reliability optimality.

Let us now look at the reconstruction process in the transformed code given by Algorithm 1. The symbol transferred by any helper block h for the reconstruction of any block f remains identical to that under the PM-vanilla code, i.e., is as given by the right hand side of Equation (4). Since the transformation performed in Algorithm 1 is invertible within each block, such a reconstruction is always possible and entails the minimum network transfers. Thus, the code retains the storage-bandwidth optimality as well. Observe that for a block f in the set of the w blocks for which a block h intends to function as an RBT-helper, block h now directly stores the symbol $t_{hf} = [s_{h1} \ \cdots \ s_{hw}]\mathbf{g}_f$. As a result, whenever called upon to help block f, block h can directly read and transfer this symbol, thus performing a reconstruct-by-transfer operation. As discussed in Section 3.1, by virtue of its storage-bandwidth optimality and reconstruct-by-transfer, the transformed code from Algorithm 1 (locally) optimizes the amount of data read from disks at the helpers. We will consider optimizing I/O across the entire system in Section 4.

3.3 Making the Code Systematic

Transforming the PM-vanilla code using Algorithm 1 may result in a loss of the systematic property. A further transformation of the code, termed 'symbol remapping', is required to make the transformed code systematic. Symbol remapping [29, Theorem 1] involves transforming the original data symbols $\{m_1, \ldots, m_B\}$ using a bijective transformation before applying Algorithm 1, as described below.

Since every step of Algorithm 1 is linear, the encoding under Algorithm 1 can be represented by a generator matrix, say G_{Alg1}, of dimension $(nw \times B)$ and the encoding can be performed by the matrix-vector multiplication: $G_{Alg1} \left[\ m_1 \ m_2 \ \cdots \ m_B \ \right]^T$. Partition G_{Alg1} as

$$G_{Alg1} = \left[\begin{array}{c} G_1 \\ G_2 \end{array} \right], \tag{5}$$

where G_1 is a $(B \times B)$ matrix corresponding to the encoded symbols in the first k systematic blocks, and G_2 is an $((nw - B) \times B$ matrix. The symbol remapping step to make the transformed code systematic involves multiplication by G_1^{-1}. The invertibility of G_1 follows from the fact that G_1 corresponds to the encoded symbols in the first k blocks and all the encoded symbols in any set of k blocks are linearly independent. Thus the entire encoding process becomes

$$\left[\begin{array}{c} G_1 \\ G_2 \end{array} \right] G_1^{-1} \left[\begin{array}{c} m_1 \\ m_2 \\ \vdots \\ m_B \end{array} \right] = \left[\begin{array}{c} I \\ G_2 G_1^{-1} \end{array} \right] \left[\begin{array}{c} m_1 \\ m_2 \\ \vdots \\ m_B \end{array} \right], \tag{6}$$

where I is the $(B \times B)$ identity matrix. We can see that the symbol remapping step followed by Algorithm 1 makes the first k blocks systematic.

Since the transformation involved in the symbol remapping step is invertible and is applied to the data symbols before the encoding process, this step does not affect the performance with respect to storage, reliability, and network and I/O consumption during reconstruction.

3.4 Making Reads Sequential

Optimizing the amount of data read from disks might not directly correspond to optimized I/Os, unless the data read is *sequential*. In the code obtained from Algorithm 1, an RBT-helper reads one symbol per byte-level stripe during a reconstruction operation. Thus, if one chooses the w symbols belonging to a byte-level stripe within a block in a contiguous manner, as in Figure 3, the data read at the helpers during reconstruction operations will be fragmented. In order to the read sequential, we employ the hop-and-couple technique introduced in [27]. The basic idea behind this technique is to choose symbols that are farther apart within a block to form the byte-level stripes. If the stripe-width of the code is w, then choosing symbols that are a $\frac{1}{w}$ fraction of the block size away will make the read at the helpers during reconstruction operations sequential. Note that this technique does not affect the natural sequence of the raw data in the data blocks, so the normal read operations can be served directly without any sorting. In this manner, we ensure

Algorithm 1 Algorithm to achieve reconstruct-by-transfer at helpers

Encode the data in k data blocks using PM-vanilla code to obtain the n encoded blocks
for every block h in the set of n blocks
  Let $\{i_{h1}, \ldots, i_{hw}\}$ denote the set of w blocks that block h will help to reconstruct by transfer
  Let $\{s_{h1}, \ldots, s_{hw}\}$ denote the set of w symbols that block h stores under $PM - vanilla$
  Compute $[s_{h1} \ \cdots \ s_{hw}] \left[\mathbf{g}_{i_{h1}} \ \cdots \ \mathbf{g}_{i_{hw}}\right]$ and store the resulting w symbols in block h instead of the original w symbols

that reconstruct-by-transfer optimizes I/O at the helpers along with the amount of data read from the disks.

4 Optimizing RBT-Helper Assignment

In Algorithm 1 presented in Section 3, we assumed the choice of the RBT-helpers for each block to be given to us. Under any such given choice, we saw how to perform a *local* transformation at each block such that reconstruction-by-transfer could be realized under that assignment. In this section, we present an algorithm to make this choice such that the I/Os consumed during reconstruction operations is optimized *globally* across the entire system. Before going into any details, we first make an observation which will motivate our approach.

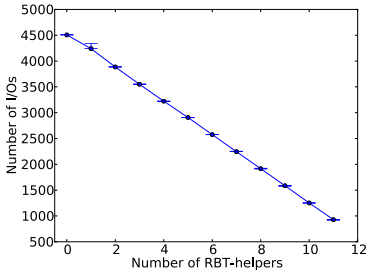
A reconstruction operation may be instantiated in any one of the following two scenarios:

- *Failures:* When a node fails, the reconstruction operation restores the contents of that node in another storage nodes in order to maintain the system reliability and availability. Failures may entail reconstruction operations of either systematic or parity blocks.

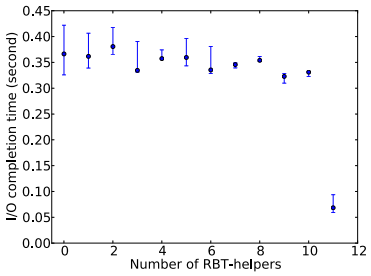
- *Degraded reads:* When a read request arrives, the systematic block storing the requested data may be busy or unavailable. The request is then served by calling upon a reconstruction operation to recover the desired data from the remaining blocks. This is called a degraded read. Degraded reads entail reconstruction of only the systematic blocks.

A system may be required to support either one or both of these scenarios, and as a result, the importance associated to the reconstruction of a systematic block may often be higher than the importance associated to the reconstruction of a parity block.

We now present a simple yet general model that we will use to optimize I/Os holistically across the system. The model has two parameters, δ and p. The relative importance between systematic and parity blocks is captured by the first parameter δ. The parameter δ takes a value between (and including) 0 and 1, and the cost associated with the reconstruction of any parity block is assumed to be δ times the cost associated to the reconstruction of any systematic block. The "cost" can be used to capture the relative difference in the frequency



(a) Total number of I/Os consumed



(b) Maximum of the I/O completion times at helpers

Figure 4: Reconstruction under different number of RBT-helpers for $k = 6$, $d = 11$, and a block size of 16MB.

of reconstruction operations between the parity and systematic blocks or the preferential treatment that one may wish to confer to the reconstruction of systematic blocks in order to serve degraded reads faster.

When reconstruction of any block is to be carried out, either for repairing a possible failure or for a degraded read, not all remaining blocks may be available to help in the reconstruction process. The parameter p $(0 \leq p \leq 1)$ aims to capture this fact: when the reconstruction of a block is to be performed, every other block may individually be unavailable with a probability p independent of all other blocks. Our intention here is to capture the fact that if a block has certain number of helpers that can function as RBT-helpers, not all of them may be available when reconstruction is to be performed.

We performed experiments on Amazon EC2 measuring the number of I/Os performed for reconstruction when precisely j $(0 \leq j \leq d)$ of the available helpers are RBT-helpers and the remaining $(d - j)$ helpers are non-RBT-helpers. The non-RBT-helpers do not perform reconstruct-by-transfer, and are hence optimal with re-

Algorithm 2 Algorithm for optimizing RBT-helper assignment

```
//To compute number of RBT-helpers for each block
Set num_rbt_helpers[block] = 0 for every block
for total_rbt_help = nw to 1
  for block in all blocks
    if num_rbt_helpers[block] < n-1
      Set improvement[block] = Cost(num_rbt_helpers[block]) - Cost(num_rbt_helpers[block]+1)
    else
      Set improvement[block] = -1
  Let max_improvement be the set of blocks with the maximum value of improvement
  Let this_block be a block in max_improvement with the largest value of num_rbt_helpers
  Set num_rbt_helpers[this_block] = num_rbt_helpers[this_block]+1

//To select the RBT-helpers for each block
Call the Kleitman-Wang algorithm [20] to generate a digraph on n vertices with incoming degrees num_rbt_helpers
and all outgoing degrees equal to w
for every edge i → j in the digraph
  Set block i as an RBT-helper to block j
```

spect to network transfers but not the I/Os. The result of this experiment aggregated from 20 runs is shown in Figure 4a, where we see that the number of I/Os consumed reduces linearly with an increase in j. We also measured the maximum of the time taken by the d helper blocks to complete the requisite I/O, which is shown Figure 4b. Observe that as long as $j < d$, this time decays very slowly upon increase in j, but reduces by a large value when j crosses d. The reason for this behavior is that the time taken by the non-RBT-helpers to complete the required I/O is similar, but is much larger than the time taken by the RBT-helpers.

Algorithm 2 takes the two parameters δ and p as inputs and assigns RBT-helpers to all the blocks in the stripe. The algorithm optimizes the expected cost of I/O for reconstruction across the system, and furthermore subject to this minimum cost, minimizes the expected time for reconstruction. The algorithm takes a greedy approach in deciding the *number* of RBT-helpers for each block. Observing that, under the code obtained from Algorithm 1, each block can function as an RBT-helper for at most w other blocks, the total RBT-helping capacity in the system is nw. This total capacity is partitioned among the n blocks as follows. The allocation of each unit of RBT-helping capacity is made to the block whose expected reconstruction cost will reduce the most with the help of this additional RBT-helper. The expected reconstruction cost for any block, under a given number of RBT-helper blocks, can be easily computed using the parameter p; the cost of a parity block is further multiplied by δ. Once the number of RBT-helpers for each block is obtained as above, all that remains is to make the choice of the RBT-helpers for each block. In making this choice, the only constraints to be satisfied are the number of RBT-helpers for each block as determined above and that no block can help itself. The Kleitman-Wang algorithm [20] facilitates such a construction.

The following theorem provides rigorous guarantees on the performance of Algorithm 2.

Theorem 1 *For any given (δ, p), Algorithm 2 minimizes the expected amount of disk reads for reconstruction operations in the system. Moreover, among all options resulting in this minimum disk read, the algorithm further chooses the option which minimizes the expected time of reconstruction.*

The proof proceeds by first showing that the expected reconstruction cost of any particular block is convex in the number of RBT-helpers assigned to it. It then employs this convexity, along with the fact that the expected cost must be non-increasing in the number of assigned RBT-helpers, to show that no other assignment algorithm can yield a lower expected cost. We omit the complete proof of the theorem due to space constraints.

The output of Algorithm 2 for $n = 15$, $k = 6$, $d = 11$ and $(\delta = 0.25, p = 0.03)$ is illustrated in Fig 5. Blocks $1, \ldots, 6$ are systematic and the rest are parity blocks. Here, Algorithm 2 assigns 12 RBT-helpers to each of the systematic blocks, and 11 and 7 RBT-helpers to the first and second parity blocks respectively.

The two 'extremities' of the output of Algorithm 2 form two interesting special cases:

1. Systematic (SYS): All blocks function as RBT-helpers for the k systematic blocks.

2. Cyclic (CYC): Block $i \in \{1, \ldots, n\}$ functions as an RBT-helper for blocks $\{i+1, \ldots, i+w\}$ mod n.

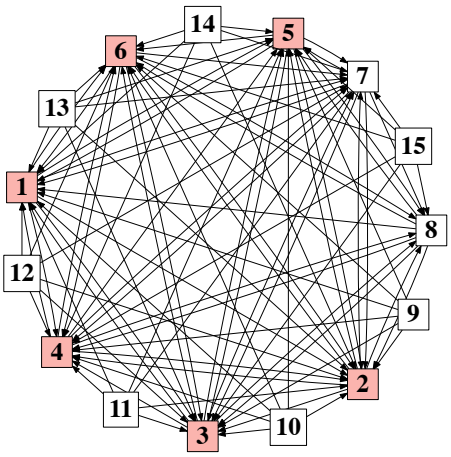


Figure 5: The output of Algorithm 2 for the parameters $n = 15$, $k = 6$, $d = 11$, and $(\delta = 0.25, p = 0.03)$ depicting the assignment of RBT-helpers. The directed edges from a block indicate the set of blocks that it helps to reconstruct-by-transfer. Systematic blocks are shaded.

Algorithm 2 will output the SYS pattern if, for example, reconstruction of parity blocks incur negligible cost (δ is close to 0) or if $\delta < 1$ and p is large. Algorithm 2 will output the CYC pattern if, for instance, the systematic and the parity blocks are treated on equal footing (δ is close to 1), or in low churn systems where p is close to 0.

While Theorem 1 provides mathematical guarantees on the performance of Algorithm 2, Section 5.7 will present an evaluation of its performance via simulations using data from Amazon EC2 experiments.

5 Implementation and Evaluation

5.1 Implementation and Evaluation Setting

We have implemented the PM-vanilla codes [29] and the PM-RBT codes (Section 3 and Section 4) in C/C++. In our implementation, we make use of the fact that the PM-vanilla and the PM-RBT codes are both linear. That is, we compute the matrices that represent the encoding and decoding operations under these codes and execute these operations with a single matrix-vector multiplication. We employ the Jerasure2 [25] and GF-Complete [26] libraries for finite-field arithmetic operations also for the RS encoding and decoding operations.

We performed all the evaluations, except those in Section 5.5 and Section 5.6, on Amazon EC2 instances of type m1.medium (with 1 CPU, 3.75 GB memory, and attached to 410 GB of hard disk storage). We chose m1.medium type since these instances have hard-disk storage. We evaluated the encoding and decoding performance on instances of type m3.medium which run on an Intel Xeon E5-2670v2 processor with a 2.5GHz clock speed. All evaluations are single-threaded.

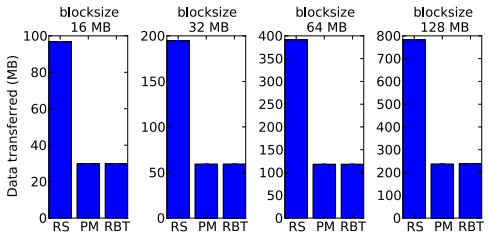


Figure 6: Total amount of data transferred across the network from the helpers during reconstruction. Y-axes scales vary across plots.

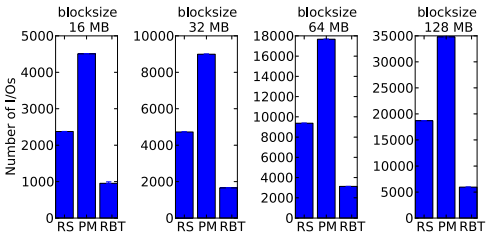
All the plots are from results aggregated over 20 independent runs showing the median values with 25th and 75th percentiles. In the plots, PM refers to PM-vanilla codes and RBT refers to PM-RBT codes. Unless otherwise mentioned, all evaluations on reconstruction are for $(n = 12,\ k = 6,\ d = 11)$, considering reconstruction of block 1 (i.e., the first systematic block) with all $d = 11$ RBT-helpers. We note that all evaluations except the one on decoding performance (Section 5.5) are independent of the identity of the block being reconstructed.

5.2 Data Transfers Across the Network

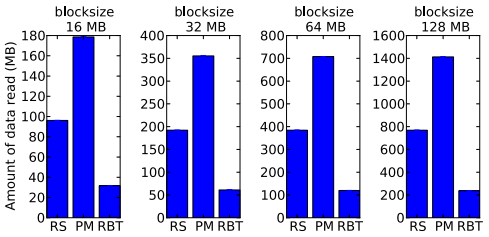
Figure 6 compares the total amount of data transferred from helper blocks to the decoding node during reconstruction of a block. We can see that, both PM-vanilla and PM-RBT codes have identical and significantly lower amount of data transferred across the network as compared to RS codes: the network transfers during the reconstruction for PM-vanilla and PM-RBT are about $4x$ lower than that under RS codes.

5.3 Data Read and Number of I/Os

A comparison of the total number of disk I/Os and the total amount of data read from disks at helpers during reconstruction are shown in Figure 7a and Figure 7b respectively. We observe that the amount of data read from the disks is as given by the theory across all block sizes that we experimented with. We can see that while PM-vanilla codes provide significant savings in network transfers during reconstruction as compared to RS as seen in Figure 6, they result in an increased number of I/Os. Furthermore, the PM-RBT code leads to a significant reduction in the number of I/Os consumed and the amount of data read from disks during reconstruction.



(a) Total number of disk I/Os consumed



(b) Total amount of data read from disks

Figure 7: Total number of disk I/Os and total amount of data read from disks at the helpers during reconstruction. Y-axes scales vary across plots.

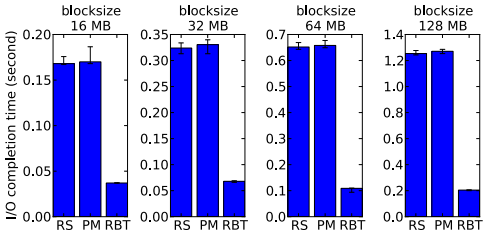


Figure 8: Maximum of the I/O completion times at helpers. Y-axes scales vary across plots.

For all the block sizes considered, we observed approximately a $5x$ reduction in the number of I/Os consumed under the PM-RBT as compared to PM-vanilla (and approximately $3\times$ reduction as compared to RS).

5.4 I/O Completion Time

The I/O completion times during reconstruction are shown in Figure 8. During a reconstruction operation, the I/O requests are issued in parallel to the helpers. Hence we plot the the maximum of the I/O completion times from the $k = 6$ helpers for RS coded blocks and the maximum of the I/O completion times from $d = 11$ helpers for PM-vanilla and PM-RBT coded blocks. We can see that PM-RBT code results in approximately $5\times$ to $6\times$ reduction I/O completion time.

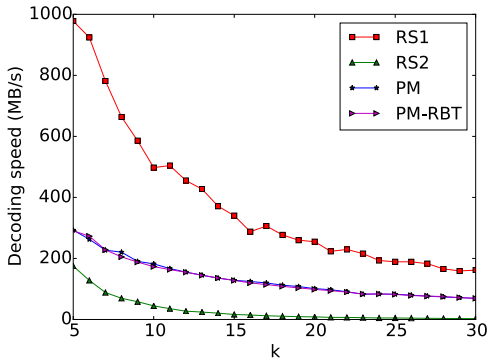


Figure 9: Comparison of decoding speed during reconstruction for various values of k with $n = 2k$, and $d = 2k-1$ for PM-vanilla and PM-RBT.

5.5 Decoding Performance

We measure the decoding performance during reconstruction in terms of the amount of data of the failed/unavailable block that is decoded per unit time. We compare the decoding speed for various values of k, and fix $n = 2k$ and $d = 2k-1$ for both PM-vanilla and PM-RBT. For reconstruction under RS codes, we observed that a higher number of systematic helpers results in a faster decoding process, as expected. In the plots discussed below, we will show two extremes of this spectrum: (RS1) the best case of helper blocks comprising all the existing $(k-1) = 5$ systematic blocks and one parity block, and (RS2) the worst case of helper blocks comprising all the $r = 6$ parity blocks.

Figure 9 shows a comparison of the decoding speed during reconstruction of block 0. We see that the best case (RS1) for RS is the fastest since the operation involves only substitution and solving a small number of linear equations. On the other extreme, the worst case (RS2) for RS is much slower than PM-vanilla and PM-RBT. The actual decoding speed for RS would depend on the number of systematic helpers involved and the performance would lie between the RS1 and RS2 curves. We can also see that the transformations introduced in this paper to optimize I/Os does not affect the decoding performance: PM-vanilla and PM-RBT have roughly the same decoding speed. In our experiments, we also observed that in both PM-vanilla and PM-RBT, the decoding speeds were identical for the n blocks.

5.6 Encoding Performance

We measure the encoding performance in terms of the amount of data encoded per unit time. A comparison of the encoding speed for varying values of k is shown in Figure 10. Here we fix $d = 2k-1$ and $n = 2k$ and vary the values of k. The lower encoding speeds for PM-

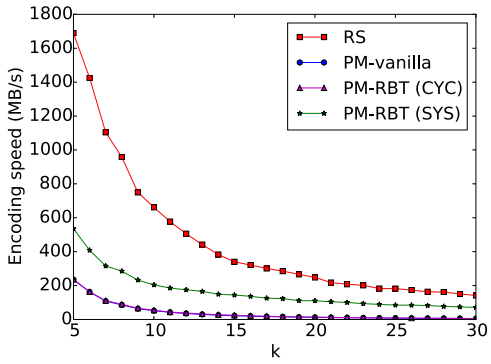


Figure 10: Encoding speed for various values of k with $n = 2k$, and $d = 2k-1$ for PM-vanilla and PM-RBT.

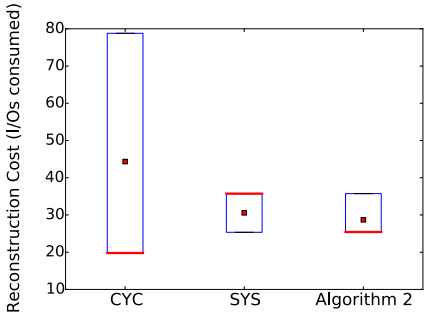


Figure 11: A box plot of the reconstruction cost for different RBT-helper assignments, for $\delta = 0.25$, $p = 0.03$, $n = 15$, $k = 6$, $d = 11$, and block size of 16MB. In each box, the mean is shown by the small (red) square and the median is shown by the thick (red) line.

vanilla and PM-RBT as compared to RS are expected since the encoding complexity of these codes is higher. In RS codes, computing each encoded symbol involves a linear combination of only k data symbols, which incurs a complexity of $O(k)$, whereas in PM-vanilla and PM-RBT codes each encoded symbol is a linear combination of kw symbols which incurs a complexity of $O(k^2)$.

Interestingly, we observe that encoding under PM-RBT with the SYS RBT-helper pattern (Section 4) is significantly faster than that under the PM-vanilla code. This is because the generator matrix of the code under the SYS RBT-helper pattern is sparse (i.e., has many zero-valued entries); this reduces the number of finite-field multiplication operations, that are otherwise computationally heavy. Thus, PM-RBT with SYS RBT-helper pattern results in faster encoding as compared to PM-vanilla codes, in addition to minimizing the disk I/O during reconstruction. Such sparsity does not arise under the CYC RBT-helper pattern, and hence its encoding speed is almost identical to PM-vanilla.

We believe that the significant savings in disk I/O offered by PM-RBT codes outweigh the cost of decreased encoding speed. This is especially true for systems storing immutable data (where encoding is a one-time overhead) and where encoding is performed as a background operation without falling along any critical path. This is true in many cloud storage systems such as Windows Azure and the Hadoop Distributed File System where data is first stored in a triple replicated fashion and then encoded in the background.

Remark: The reader may observe that the speed (MB/s) of encoding in Figure 10 is faster than that of decoding during reconstruction in Figure 9. This is because encoding addresses k blocks at a time while the decoding operation addresses only a single block.

5.7 RBT-helper Assignment Algorithm

As discussed earlier in Section 4, we conducted experiments on EC2 performing reconstruction using different number of RBT-helpers (see Figure 4). We will now evaluate the performance of the helper assignment algorithm, Algorithm 2, via simulations employing the measurements obtained from these experiments. The plots of the simulation results presented here are aggregated from one million runs of the simulation. In each run, we failed one of the n blocks chosen uniformly at random. For its reconstruction operation, the remaining $(n-1)$ blocks were made unavailable (busy) with a probability p each, thereby also making some of the RBT-helpers assigned to this block unavailable. In the situation when only j RBT-helpers are available (for any j in $\{0,\ldots,d\}$), we obtained the cost of reconstruction (in terms of number of I/Os used) by sampling from the experimental values obtained from our EC2 experiments with j RBT-helpers and $(d-j)$ non-RBT-helpers (Figure 4a). The reconstruction cost for parity blocks is weighted by δ.

Figure 11 shows the performance of the RBT-helper assignment algorithm for the parameter values $\delta = 0.25$ and $p = 0.03$. The plot compares the performance of three possible choices of helper assignments: the assignment obtained by Algorithm 2 for the chosen parameters (shown in Figure 5), and the two extremities of Algorithm 2, namely SYS and CYC. We make the following observations from the simulations. In the CYC case, the unweighted costs for reconstruction are homogeneous across systematic and parity blocks due to the homogenity of the CYC pattern, but upon reweighting by δ, the distribution of costs become (highly) bi-modal. In Figure 11, the performance of SYS and the solution obtained from Algorithm 2 are comparable, with the output of Algorithm 2 slightly outperforming SYS. This is as expected since for the given choice of parameter val-

ues $\delta = 0.25$ and $p = 0.03$, the output of Algorithm 2 (see Figure 5) is close to SYS pattern.

6 Related Literature

In this section, we review related literature on optimizing erasure-coded storage systems with respect to network transfers and the amount of data read from disks during reconstruction operations.

In [17], the authors build a file system based on the minimum-bandwidth-regenerating (MBR) code constructions of [34]. While system minimizes network transfers and the amount of data read during reconstruction, it mandates additional storage capacity to achieve the same. That is, the system is not optimal with respect to storage-reliability tradeoff (recall from Section 2). The storage systems proposed in [18, 23, 13] employ a class of codes called local-repair codes which optimize the number of blocks accessed during reconstruction. This, in turn, also reduces the amount of disk reads and network transfers. However, these systems also necessitate an increase in storage-space requirements in the form of at least 25% to 50% additional parities. In [21], authors present a system which combines local-repair codes with the graph-based MBR codes presented in [34]. This work also necessitates additional storage space. The goal of the present paper is to optimize I/Os consumed during reconstruction *without* losing the optimality with respect to storage-reliability tradeoff.

[16] and [6], the authors present storage systems based on random network-coding that optimize resources consumed during reconstruction. Here the data that is reconstructed is not identical and is only "functionally equivalent" to the failed data. As a consequence, the system is not systematic, and needs to execute the decoding procedure for serving every read request. The present paper designs codes that are systematic, allowing read requests during the normal mode of operation to be served directly without executing the decoding procedure.

In [27], the authors present a storage system based on a class of codes called Piggybacked-RS codes [30] that also reduces the amount of data read during reconstruction. However, PM-RBT codes provide higher savings as compared to these codes. On the other hand, Piggybacked-RS codes have the advantage of being applicable for all values of k and r, whereas PM-RBT codes are only applicable for $d \geq (2k-2)$ and thereby necessitate a storage overhead of atleast $(2-\frac{1}{k})$. In [19], authors present Rotated-RS codes which also reduce the amount of data read during rebuilding. However, the reduction achieved is significantly lower than that in PM-RBT.

In [33], the authors consider the theory behind reconstruction-by-transfer for MBR codes, which as discussed earlier are not optimal with respect to storage-reliability tradeoff. Some of the techniques employed in the current paper are inspired by the techniques introduced in [33]. In [36] and [38], the authors present optimizations to reduce the amount of data read for reconstruction in array codes with two parities. [19] presents a search-based approach to find reconstruction symbols that optimize I/O for arbitrary binary erasure codes, but this search problem is shown to be NP-hard.

Several works (e.g., [8, 24, 11]) have proposed system-level solutions to reduce network and I/O consumption for reconstruction, such as caching the data read during reconstruction, batching multiple reconstruction operations, and delaying the reconstruction operations. While these solutions consider the erasure code as a black-box, our work optimizes this black-box and can be used in conjunction with these system-level solutions.

7 Conclusion

With rapid increases in the network-interconnect speeds and the advent of high-capacity storage devices, I/O is increasingly becoming the bottleneck in many large-scale distributed storage systems. A family of erasure-codes called minimum-storage-regeneration (MSR) codes has recently been proposed as a superior alternative to the popular Reed-Solomon codes in terms of storage, fault-tolerance and network-bandwidth consumed. However, existing practical MSR codes do not address the critically growing problem of optimizing for I/Os. In this work, we show that it is possible to have your cake and eat it too, in the sense that *we can minimize disk I/O consumed, while simultaneously retaining optimality in terms of both storage, reliability and network-bandwidth*.

Our solution is based on the identification of two key properties of existing MSR codes that can be exploited to make them I/O optimal. We presented an algorithm to transform Product-Matrix-MSR codes into I/O optimal codes (which we term the PM-RBT codes), while retaining their storage and network optimality. Through an extensive set of experiments on Amazon EC2, we have shown that our proposed PM-RBT codes result in significant reduction in the I/O consumed. Additionally, we have presented an optimization framework for helper assignment to attain a system-wide globally optimal solution, and established its performance through simulations based on EC2 experimentation data.

8 Acknowledgements

We thank Ankush Gupta and Diivanand Ramalingam for their contributions to the initial version of the PM-vanilla code implementation. We also thank our shepherd Randal Burns and the anonymous reviewers for their valuable comments.

References

[1] Facebook's Approach to Big Data Storage Challenge. http://www.slideshare.net/Hadoop_Summit/facebooks-approach-to-big-data-storage-challenge.

[2] Hadoop. http://hadoop.apache.org.

[3] HDFS RAID. http://www.slideshare.net/ydn/hdfs-raid-facebook.

[4] Seamless reliability. http://www.cleversafe.com/overview/reliable, Feb. 2014.

[5] ABD-EL-MALEK, M., COURTRIGHT II, W. V., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M. P., PRASAD, M., SALMON, B., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Ursa minor: Versatile cluster-based storage. In *FAST* (2005), vol. 5, pp. 5–5.

[6] ANDRÉ, F., KERMARREC, A.-M., LE MERRER, E., LE SCOUARNEC, N., STRAUB, G., AND VAN KEMPEN, A. Archiving cold data in warehouses with clustered network coding. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), ACM, p. 21.

[7] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., VAJGEL, P., ET AL. Finding a needle in haystack: Facebook's photo storage. In *OSDI* (2010), vol. 10, pp. 1–8.

[8] BHAGWAN, R., TATI, K., CHENG, Y. C., SAVAGE, S., AND VOELKER, G. M. Total recall: System support for automated availability management. In *Proc. 1st conference on Symposium on Networked Systems Design and Implementation (NSDI)* (2004).

[9] BORTHAKUR, D. Hdfs and erasure codes (HDFS-RAID). http://hadoopblog.blogspot.com/2009/08/hdfs-and-erasure-codes-hdfs-raid.html, 2009.

[10] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., ET AL. Windows Azure storage: a highly available cloud storage service with strong consistency. In *Proc. ACM Symposium on Operating Systems Principles* (2011), pp. 143–157.

[11] CHUN, B.-G., DABEK, F., HAEBERLEN, A., SIT, E., WEATHERSPOON, H., KAASHOEK, M. F., KUBIATOWICZ, J., AND MORRIS, R. Efficient replica maintenance for distributed storage systems. In *NSDI* (2006), vol. 6, pp. 225–264.

[12] DIMAKIS, A. G., GODFREY, P. B., WU, Y., WAINWRIGHT, M., AND RAMCHANDRAN, K. Network coding for distributed storage systems. *IEEE Transactions on Information Theory 56*, 9 (Sept. 2010), 4539–4551.

[13] ESMAILI, K. S., PAMIES-JUAREZ, L., AND DATTA, A. CORE: Cross-object redundancy for efficient data repair in storage systems. In *IEEE International Conference on Big data* (2013), pp. 246–254.

[14] FORD, D., LABELLE, F., POPOVICI, F., STOKELY, M., TRUONG, V., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in globally distributed storage systems. In *9th USENIX Symposium on Operating Systems Design and Implementation* (Oct. 2010).

[15] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S. The Google file system. In *ACM SIGOPS Operating Systems Review* (2003), vol. 37, ACM, pp. 29–43.

[16] HU, Y., CHEN, H. C., LEE, P. P., AND TANG, Y. Nccloud: Applying network coding for the storage repair in a cloud-of-clouds. In *USENIX FAST* (2012).

[17] HU, Y., YU, C., LI, Y., LEE, P., AND LUI, J. NCFS: On the practicality and extensibility of a network-coding-based distributed file system. In *International Symposium on Network Coding (NetCod)* (Beijing, July 2011).

[18] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in Windows Azure Storage. In *USENIX Annual Technical Conference (ATC)* (June 2012).

[19] KHAN, O., BURNS, R., PLANK, J., PIERCE, W., AND HUANG, C. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. In *Proc. Usenix Conference on File and Storage Technologies (FAST)* (2012).

[20] KLEITMAN, D. J., AND WANG, D.-L. Algorithms for constructing graphs and digraphs with given valences and factors. *Discrete Mathematics 6*, 1 (1973), 79–88.

[21] KRISHNAN, M. N., PRAKASH, N., LALITHA, V., SASIDHARAN, B., KUMAR, P. V., NARAYANAMURTHY, S., KUMAR, R., AND NANDI, S. Evaluation of codes with inherent double replication for hadoop. In *Proc. USENIX HotStorage* (2014).

[22] MACWILLIAMS, F., AND SLOANE, N. *The Theory of Error-Correcting Codes, Part I*. North-Holland Publishing Company, 1977.

[23] MAHESH, S., ASTERIS, M., PAPAILIOPOULOS, D., DIMAKIS, A. G., VADALI, R., CHEN, S., AND BORTHAKUR, D. Xoring elephants: Novel erasure codes for big data. In *VLDB Endowment* (2013).

[24] MICKENS, J., AND NOBLE, B. Exploiting availability prediction in distributed systems. In *NSDI* (2006).

[25] PLANK, J. S., AND GREENAN, K. M. Jerasure: A library in c facilitating erasure coding for storage applications–version 2.0. Tech. rep., Technical Report UT-EECS-14-721, University of Tennessee, 2014.

[26] PLANK, J. S., MILLER, E. L., GREENAN, K. M., ARNOLD, B. A., BURNUM, J. A., DISNEY, A. W., AND MCBRIDE, A. C. Gf-complete: A comprehensive open source library for galois field arithmetic, version 1.0. Tech. Rep. CS-13-716, University of Tennessee, October 2013.

[27] RASHMI, K., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of the 2014 ACM conference on SIGCOMM* (2014), ACM, pp. 331–342.

[28] RASHMI, K. V., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Proc. USENIX HotStorage* (June 2013).

[29] RASHMI, K. V., SHAH, N. B., AND KUMAR, P. V. Optimal exact-regenerating codes for the MSR and MBR points via a product-matrix construction. *IEEE Transactions on Information Theory 57*, 8 (Aug. 2011), 5227–5239.

[30] RASHMI, K. V., SHAH, N. B., AND RAMCHANDRAN, K. A piggybacking design framework for read-and download-efficient distributed storage codes. In *IEEE International Symposium on Information Theory* (July 2013).

[31] REED, I., AND SOLOMON, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics 8*, 2 (1960), 300–304.

[32] SCHROEDER, B., AND GIBSON, G. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proc. 5th USENIX conference on File and Storage Technologies (FAST)* (2007).

[33] SHAH, N. B. On minimizing data-read and download for storage-node recovery. *IEEE Communications Letters* (2013).

[34] SHAH, N. B., RASHMI, K. V., KUMAR, P. V., AND RAMCHANDRAN, K. Distributed storage codes with repair-by-transfer and non-achievability of interior points on the storage-bandwidth tradeoff. *IEEE Transactions on Information Theory 58*, 3 (Mar. 2012), 1837–1852.

[35] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The Hadoop distributed file system. In *IEEE Symp. on Mass Storage Systems and Technologies* (2010).

[36] WANG, Z., DIMAKIS, A., AND BRUCK, J. Rebuilding for array codes in distributed storage systems. In *Workshop on the Application of Communication Theory to Emerging Memory Technologies (ACTEMT)* (Dec. 2010).

[37] WEATHERSPOON, H., AND KUBIATOWICZ, J. D. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems* (2002), Springer, pp. 328–337.

[38] XIANG, L., XU, Y., LUI, J., AND CHANG, Q. Optimal recovery of single disk failure in RDP code storage systems. In *ACM SIGMETRICS* (2010), vol. 38, pp. 119–130.

[39] ZHANG, Z., DESHPANDE, A., MA, X., THERESKA, E., AND NARAYANAN, D. Does erasure coding have a role to play in my data center. *Microsoft research MSR-TR-2010 52* (2010).