Erasure Coding in Windows Azure Storage

Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin
Microsoft Corporation

Abstract

Windows Azure Storage (WAS) is a cloud storage system that provides customers the ability to store seemingly limitless amounts of data for any duration of time. WAS customers have access to their data from anywhere, at any time, and only pay for what they use and store. To provide durability for that data and to keep the cost of storage low, WAS uses erasure coding.

In this paper we introduce a new set of codes for erasure coding called Local Reconstruction Codes (LRC). LRC reduces the number of erasure coding fragments that need to be read when reconstructing data fragments that are offline, while still keeping the storage overhead low. The important benefits of LRC are that it reduces the bandwidth and I/Os required for repair reads over prior codes, while still allowing a significant reduction in storage overhead. We describe how LRC is used in WAS to provide low overhead durable storage with consistently low read latencies.

1 Introduction

Windows Azure Storage (WAS) [1] is a scalable cloud storage system that has been in production since November 2008. It is used inside Microsoft for applications such as social networking search, serving video, music and game content, managing medical records, and more. In addition, there are thousands of customers outside Microsoft using WAS, and anyone can sign up over the Internet to use the system. WAS provides cloud storage in the form of Blobs (user files), Tables (structured storage), Queues (message delivery), and Drives (network mounted VHDs). These data abstractions provide the overall storage and work flow for applications running in the cloud.

WAS stores all of its data into an append-only distributed file system called the stream layer [1]. Data is appended to the end of active *extents*, which are replicated three times by the underlying stream layer. The data is originally written to 3 full copies to keep the data durable. Once reaching a certain size (e.g., 1 GB), extents are sealed. These sealed extents can no longer be modified and thus make perfect candidates for erasure coding. WAS then erasure codes a sealed extent lazily in the background, and once the extent is erasure-coded the original 3 full copies of the extent are deleted.

The motivation for using erasure coding in WAS comes from the need to reduce the cost of storage. Erasure coding can reduce the cost of storage over 50%, which is a tremendous cost saving as we will soon surpass an Exabyte of storage. There are the obvious cost savings from purchasing less hardware to store that much data, but there are significant savings from the fact that this also reduces our data center footprint by 1/2, the power savings from running 1/2 the hardware, along with other savings.

The trade-off for using erasure coding instead of keeping 3 full copies is performance. The performance hit comes when dealing with i) a lost or offline data fragment and ii) hot storage nodes. When an extent is erasure-coded, it is broken up into k data fragments, and a set of parity fragments. In WAS, a data fragment may be lost due to a disk, node or rack failure. In addition, cloud services are *perpetually in beta* [2] due to frequent upgrades. A data fragment may be offline for seconds to a few minutes due to an upgrade where the storage node process may be restarted or the OS for the storage node may be rebooted. During this time, if there is an on-demand read from a client to a fragment on the storage node being upgraded, WAS reads from enough fragments in order to dynamically reconstruct the data being asked for to return the data to the client. This reconstruction needs to be optimized to be as fast as possible and use as little networking bandwidth and I/Os as possible, with the goal to have the reconstruction time consistently low to meet customer SLAs.

When using erasure coding, the data fragment the client's request is asking for is stored on a specific storage node, which can greatly increase the risk of a storage node becoming hot, which could affect latency. One way that WAS can deal with hot storage nodes is to recognize the fragments that are hot and then replicate them to cooler storage nodes to balance out the load, or cache the data and serve it directly from DRAM or SSDs. But, the read performance can suffer for the potential set of reads going to that storage node as it gets hot, until the data is cached or load balanced. Therefore, one optimization WAS has is if it looks like the read to a data fragment is going to take too long, WAS in parallel tries to perform a reconstruction of the data fragment (effectively treating the storage node with the original data fragment as if it was offline) and return to the client whichever of the two results is faster.

For both of the above cases the time to reconstruct a data fragment for on-demand client requests is crucial. The problem is that the reconstruction operation is only as fast as the slowest storage node to respond to reading

of the data fragments. In addition, we want to reduce storage costs down to 1.33x of the original data using erasure coding. This could be accomplished using the standard approach of Reed-Solomon codes [13] where we would have (12, 4), which is 12 data fragments and 4 code fragments. This means that to do the reconstruction we would need to read from a set of 12 fragments. This i) greatly increases the chance of hitting a hot storage node, and ii) increases the network costs and I/Os and adds latency to read that many fragments to do the reconstruction. Therefore, we want to design a new family of codes to use for WAS that provides the following characteristics:

1. Reduce the minimal number of fragments that need to be read from to reconstruct a data fragment. This provides the following benefits: i) reduces the network overhead and number of I/Os to perform a reconstruction; ii) reduces the time it takes to perform the reconstruction since fewer fragments need to be read. We have found the time to perform the reconstruction is often dominated by the slowest fragments (the stragglers) to be read from.
2. Provide significant reduction in storage overhead to 1.33x while maintaining higher durability than a system that keeps 3 replicas for the data.

In this paper, we introduce Local Reconstruction Codes (LRC) that provide the above properties. In addition, we describe our erasure coding implementation and important design decisions.

2 Local Reconstruction Codes

In this section, we illustrate LRC and its properties through small examples, which are shorter codes (thus higher overhead) than what we use in production, in order to simplify the description of LRC .

2.1 Definition

We start with a Reed-Solomon code example to explain the concept of *reconstruction cost*. A (6, 3) Reed-Solomon code contains 6 data fragments and 3 parity fragments, where each parity is computed from all the 6 data fragments. When any data fragment becomes unavailable, no matter which data and parity fragments are used for reconstruction, 6 fragments are always required. We define *reconstruction cost* as the number of fragments required to reconstruct an unavailable *data fragment*. Here, the reconstruction cost equals to 6.

The goal of LRC is to reduce the reconstruction cost. It achieves this by computing some of the parities from a subset of the data fragments. Continuing the example with 6 data fragments, LRC generates 4 (instead of 3) parities. The first two parities (denoted as p_0 and p_1) are *global parities* and are computed from *all* the data fragments. But, for the other two parities, LRC divides the

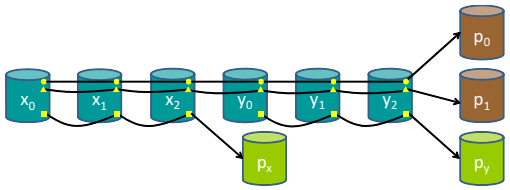


Figure 1: **A (6, 2, 2) LRC Example.** ($k = 6$ data fragments, $l = 2$ local parities and $r = 2$ global parities.)

data fragments into two equal size groups and computes one *local parity* for each group. For convenience, we name the 6 data fragments (x_0, x_1 and x_2) and (y_0, y_1 and y_2). Then, local parity p_x is computed from the 3 data fragments in one group (x_0, x_1 and x_2), and local parity p_y from the 3 data fragments in another group (y_0, y_1 and y_2).

Now, let's walk through reconstructing x_0. Instead of reading p_0 (or p_1) and the other 5 data fragments (x_1, x_2, y_0, y_1 and y_2), it is more efficient to read p_x and two data fragments (x_1 and x_2) to compute x_0. It is easy to verify that the reconstruction of *any* single data fragment requires only 3 fragments, half the number required by the Reed-Solomon code.

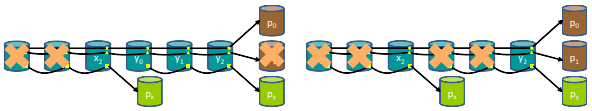
This LRC example adds one more parity than the Reed-Solomon one, so it might appear that LRC reduces reconstruction cost at the expense of higher storage overhead. In practice, however, these two examples achieve completely different trade-off points in the design space, as described in Section 3.3. In addition, LRC provides more options than Reed-Solomon code, in terms of trading off storage overhead and reconstruction cost.

We now formally define Local Reconstruction Codes. A (k, l, r) LRC divides k data fragments into l groups, with k/l data fragments in each group. It computes one local parity within each group. In addition, it computes r global parities from all the data fragments. Let n be the total number of fragments (data + parity). Then $n = k + l + r$. Hence, the normalized storage overhead is $n/k = 1 + (l + r)/k$. The LRC in our example is a $(6, 2, 2)$ LRC with storage cost of $1 + 4/6 = 1.67$x, as illustrated in Figure 1.

2.2 Fault Tolerance

Thus far, we have only defined which data fragments are used to compute each parity in LRC. To complete the code definition, we also need to determine *coding equations*, that is, how the parities are computed from the data fragments. We choose the coding equations such that LRC can achieve the *Maximally Recoverable* (MR) property [14], which means it can decode any failure pattern which is information-theoretically decodable.

Let's first explain the Maximally Recoverable property. Given the (6, 2, 2) LRC example, it contains 4 parity fragments and can tolerate *up to* 4 failures. However,



(a) 3 Failures (b) 4 Failures

Figure 2: Decoding 3 and 4 Failures in LRC.

LRC is not Maximum Distance Separable [12] and therefore cannot tolerate *arbitrary* 4 failures. For instance, say the 4 failures are x_1, x_2, x_3 and p_x. This failure pattern is non-decodable because there are only two parities - the global parities - that can help to decode the 3 missing data fragments. The other local parity p_y is useless in this example. It is *impossible* to decode 3 data fragments from merely 2 parity fragments, regardless of coding equations. These types of failure patterns are called *information-theoretically* non-decodable.

Failure patterns that are possible to reconstruct are called *information-theoretically* decodable. For instance, the 3-failure pattern in Figure 2(a) and the 4-failure pattern in Figure 2(b) are both information-theoretically decodable. For these two failure patterns, it is possible to construct coding equations such that it is equivalent to solving 3 unknowns using 3 linearly independent equations in Figure 2(a) and 4 unknowns using 4 linearly independent equations in Figure 2(b).

Conceivably, it is not difficult to construct a set of coding equations that can decode a specific failure pattern. However, the real challenge is to construct a *single* set of coding equations that achieves the *Maximally Recoverable* (MR) property [14], or being able to decode all the information-theoretically decodable failure patterns – the exact goal of LRC.

2.2.1 Constructing Coding Equations

It turns out that the LRC can tolerate arbitrary 3 failures by choosing the following two sets of coding coefficients (α's and β's) for group x and group y, respectively. We skip the proofs due to space limitation. Let

$$q_{x,0} = \alpha_0 x_0 + \alpha_1 x_1 + \alpha_2 x_2 \quad (1)$$

$$q_{x,1} = \alpha_0^2 x_0 + \alpha_1^2 x_1 + \alpha_2^2 x_2 \quad (2)$$

$$q_{x,2} = x_0 + x_1 + x_2 \quad (3)$$

and

$$q_{y,0} = \beta_0 y_0 + \beta_1 y_1 + \beta_2 y_2 \quad (4)$$

$$q_{y,1} = \beta_0^2 y_0 + \beta_1^2 y_1 + \beta_2^2 y_2 \quad (5)$$

$$q_{y,2} = y_0 + y_1 + y_2. \quad (6)$$

Then, the LRC coding equations are as follows:

$$p_0 = q_{x,0} + q_{y,0}, \quad p_1 = q_{x,1} + q_{y,1}, \quad (7)$$

$$p_x = q_{x,2}, \quad p_y = q_{y,2}. \quad (8)$$

Next, we determine the values of α's and β's so that the LRC can decode all information-theoretically decodable 4 failures. We focus on non-trivial cases as follows:

1. **None of the four parities fails.** The four failures are equally divided between group x and group y. Hence, we have four equations whose coefficients are given by the matrix, which result in the following determinant:

$$G = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ \alpha_i & \alpha_j & \beta_s & \beta_t \\ \alpha_i^2 & \alpha_j^2 & \beta_s^2 & \beta_t^2 \end{pmatrix}$$

$$Det(G) = (\alpha_j - \alpha_i)(\beta_t - \beta_s)(\alpha_i + \alpha_j - \beta_s - \beta_t).$$

2. **Only one of p_x and p_y fails.** Assume p_y fails. For the remaining three failures, two are in group x and the third one in group y. We now have three equations with coefficients given by

$$G' = \begin{pmatrix} 1 & 1 & 0 \\ \alpha_i & \alpha_j & \beta_s \\ \alpha_i^2 & \alpha_j^2 & \beta_s^2 \end{pmatrix}$$

$$Det(G') = \beta_s(\alpha_j - \alpha_i)(\beta_s - \alpha_j - \alpha_i).$$

3. **Both p_x and p_y fail.** In addition, the remaining two failures are divided between group x and group y. We have two equations with coefficients given by

$$G'' = \begin{pmatrix} \alpha_i & \beta_s \\ \alpha_i^2 & \beta_s^2 \end{pmatrix}$$

$$Det(G'') = \alpha_i \beta_s (\beta_s - \alpha_i).$$

To ensure all the cases are decodable, all the matrices G, G' and G'' should be non-singular, which leads to the following conditions:

$$\alpha_i,\ \alpha_j,\ \beta_s,\ \beta_t \neq 0 \quad (9)$$

$$\alpha_i,\ \alpha_j \neq \beta_s,\ \beta_t \quad (10)$$

$$\alpha_i + \alpha_j \neq \beta_s + \beta_t \quad (11)$$

One way to fulfill these conditions is to assign to α's and β's the elements from a finite field GF(2^4) [12], where every element in the field is represented by 4 bits. α's are chosen among the elements whose lower order 2 bits are zero. Similarly, β's are chosen among the elements whose higher order 2 bits are zero. That way, the lower order 2 bits of α's (and the sum of α's) are always zero, and the higher order 2 bits of β's (and the sum of β's) are always zero. Hence, they will never be equal and all the above conditions are satisfied.

This way of constructing coding equations requires a very small finite field and makes implementation practical. It is a critical improvement over our own Pyramid

codes [14]. In Pyramid codes, coding equation coefficients are discovered through a search algorithm, whose complexity grows exponentially with the length of the code. For parameters considered in Windows Azure Storage, the search algorithm approach would have resulted in a large finite field, which makes encoding and decoding complexity high.

2.2.2 Putting Things Together

To summarize, the $(6, 2, 2)$ LRC is capable of decoding arbitrary three failures. It can also decode all the information-theoretically decodable four failure patterns, which accounts for 86% of all the four failures. In short, the $(6, 2, 2)$ LRC achieves the Maximally Recoverable property [14].

2.2.3 Checking Decodability

Given a failure pattern, how can we easily check whether it is information-theoretically decodable? Here is an efficient algorithm. For each local group, if the local parity is available, while at least one data fragment is erased, we *swap* the parity with one erased data fragment. The swap operation marks the data fragment as available and the parity as erased. Once we complete all the local groups, we examine the data fragments and the global parities. If the total number of erased fragments (data and parity) is no more than the number of global parities, the algorithm declares the failure pattern information-theoretically decodable. Otherwise, it is non-decodable. This algorithm can be used to verify that the examples in Figure 2 are indeed decodable.

2.3 Optimizing Storage Cost, Reliability and Performance

Throughout the entire section, we have been focusing on the (6, 2, 2) LRC example. It is important to note that all the properties demonstrated by the example generalize to arbitrary coding parameters.

In general, the key properties of a (k, l, r) LRC are: $i)$ single data fragment failure can be decoded from k/l fragments; $ii)$ arbitrary failures up to $r + 1$ can be decoded. Based on the following theorem, these properties impose a lower bound on the number of parities [21].

Theorem 1. *For any (n, k) linear code (with k data symbols and $n - k$ parity symbols) to have the property:*

1. *arbitrary $r + 1$ symbol failures can be decoded;*
2. *single data symbol failure can be recovered from $\lceil k/l \rceil$ symbols,*

the following condition is necessary:

$$n - k \geq l + r. \tag{12}$$

Since the number of parities of LRC meets the lower bound *exactly*, LRC achieves its properties with the minimal number of parities.

2.4 Summary

Now, we summarize Local Reconstruction Codes. A (k, l, r) LRC divides k data fragments into l local groups. It encodes l local parities, one for each local group, and r global parities. Any single data fragment failure can be decoded from k/l fragments within its local group.

In addition, LRC achieves Maximally Recoverable property. It tolerates up to $r + 1$ arbitrary fragment failures. It also tolerates failures more than $r+1$ (up to $l+r$), provided those are information-theoretically decodable.

Finally, LRC provides low storage overhead. Among all the codes that can decode single data fragment failure from k/l fragments and tolerate $r + 1$ failures, LRC requires the minimum number of parities.

3 Reliability Model and Code Selection

There are many choices of parameters k, l and r for LRC. The question is: what parameters should we choose in practice? To answer this, we first need to understand the reliability achieved by each set of parameters. Since 3-replication is an accepted industry standard, we use the reliability of 3-replication as a reference. Only those sets of parameters that achieve equal or higher reliability than 3-replication are considered.

3.1 Reliability Model

Reliability has long been a key focus in distributed storage systems [28, 29, 30, 31]. Markov models are commonly used to capture the reliability of distributed storage systems. The model is flexible to consider both independent or correlated failures [10, 32, 33, 34]. We add a simple extension to generalize the Markov model, in order to capture unique state transitions in LRC. Those transitions are introduced because the failure mode depends on not only the size of failure, but also which subset of nodes fails. In our study, we focus on independent failures, but the study can be readily generalized to correlated failures [10, 34].

3.1.1 Modeling (6, 2, 2) LRC

We start with the standard Markov model to analyze reliability. Each state in the Markov process represents the number of available fragments (data and parity). For example, Figure 3 plots the Markov model diagram for the $(6, 2, 2)$ LRC.

Let λ denote the failure rate of a single fragment. Then, the transition rate from all fragments healthy `State 10` to one fragment failure `State 9` is 10λ. The extension from the standard Markov model lies in `State 7`, which can transition into two states with 6 healthy fragments. `State 6` represents a state where there are four decodable failures. On the other hand, `State 6F` represents a state with four non-decodable failures. Let p_d denote the percentage of decodable four failure cases. Then the transition rate from `State 7` to

`State 6` is $7\lambda p_d$ and the transition to `State 6F` is $7\lambda(1-p_d)$.

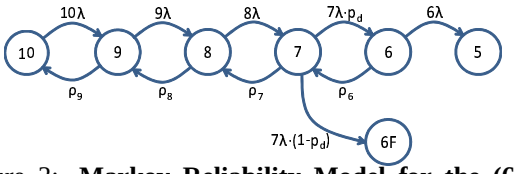


Figure 3: **Markov Reliability Model for the (6, 2, 2) LRC.** (State represents the number of healthy fragments.)

In the reverse direction, let ρ_9 denote the transition rate from one fragment failure `state 9` back to all fragments healthy `state 10`, which equals to the average repair rate of single-fragment failures.

Assume there are M storage nodes in the system, each with S storage space and B network bandwidth. When single storage node fails, assume the repair load is evenly distributed among the remaining $(M-1)$ nodes. Further, assume repair traffic is throttled so that it only uses ϵ of the network bandwidth on each machine. When erasure coding is applied, repairing one failure fragment requires more than one fragment, denoted as repair cost C. Then, the average repair rate of single-fragment failures is $\rho_9 = \epsilon(M-1)B/(SC)$.

Other transition rates ρ_8 through ρ_6 can be calculated similarly. In practice, however, the repair rates beyond single failure are dominated by the time taken to detect failure and trigger repair.[1] Let T denote the detection and triggering time. It suffices to set $\rho_8 = \rho_7 = \rho_6 = 1/T$.

Any single storage node stores both data and parity fragments from different extents. So, when it fails, both data and parity fragments need to be repaired. The repair cost can be calculated by averaging across all the fragments.

For the $(6, 2, 2)$ LRC, it takes 3 fragments to repair any of the 6 data fragments and the 2 local parities. Further, it takes 6 fragments to repair the 2 global parities. Hence, the average repair cost $C = (3 \times 8 + 6 \times 2)/10 = 3.6$. Also, enumerating all four failure patterns, we obtain the decodability ratio as $p_d = 86\%$.

3.1.2 Reliability of (6, 2, 2) LRC

Now, we use a set of typical parameters ($M = 400$, $S = 16$TB, $B = 1$Gbps, $\epsilon = 0.1$ and $T = 30$ minutes) to calculate the reliability of the LRC, which is also compared to 3-replication and Reed-Solomon code in Table 1. Since the Reed-Solomon code tolerates three failures, while 3-replication tolerates only two failures, it should be no surprise that the Reed-Solomon code offers higher reliability than 3-replication. Similarly, the LRC tolerates not only three failures, but also 86% of the four-failure cases, so it naturally achieves the highest reliability.

[1] When multiple failures happen, most affected coding groups only have a single fragment loss. *Unlucky* coding groups with two or more fragment losses are relatively few. Therefore, not many fragments enter multi-failure repair stages. In addition, multi-failure repairs are prioritized over single-failure ones. As a result, multi-failure repairs are fast and they take very little time, compared to detecting the failures and triggering the repairs.

	MTTF (years)
3-replication	3.5×10^9
(6, 3) Reed-Solomon	6.1×10^{11}
(6, 2, 2) LRC	2.6×10^{12}

Table 1: Reliability of 3-Replication, RS and LRC.

3.2 Cost and Performance Trade-offs

Each set of LRC parameters (k, l and r) yields one set of values of reliability, reconstruction cost and storage overhead. For the (6, 2, 2) LRC, the reliability (MTTF in years) is 2.6×10^{12}, the reconstruction cost is 3 and the storage overhead is 1.67x.

We obtain many sets of values by varying the parameters. Since each fragment has to place on a different fault domain, the number of fault domains in a cluster limits the total number of fragments in the code. We use 20 as the limit here, since our storage stamps (clusters) have up to 20 fault domains. Using the reliability (MTTF) of 3-replication as the threshold, we keep those sets of parameters that yield equal or higher reliability than 3-replication. We then plot the storage overhead and the reconstruction cost of the remaining sets in Figure 4. Again, each individual point represents one set of coding parameters. Each parameter set represents certain trade-offs between storage cost and reconstruction performance.

Different coding parameters can result in the same storage overhead (such as 1.5x), but vastly different reconstruction cost. In practice, it only makes sense to choose the one with the lower reconstruction cost. Therefore, we outline the lower bound of all the trade-off points. The lower bound curve characterizes the fundamental trade-off between storage cost and reconstruction performance for LRC.

3.3 Code Parameter Selection

Similarly, for (k, r) Reed-Solomon code, we vary the parameters k and r (so long as $k + r \leq 20$) and also obtain a lower bound cost and performance trade-off curve. We compare Reed-Solomon to LRC in Figure 5. On the Reed-Solomon curve, we mark two special points, which are specific parameters chosen by existing planet-scale cloud storage systems. In particular, RS (10, 4) is used in HDFS-RAID in Facebook [8] and RS (6, 3) in GFS II in Google [9, 10]. Again, note that all the trade-off points achieve higher reliability than 3-replication, so they are

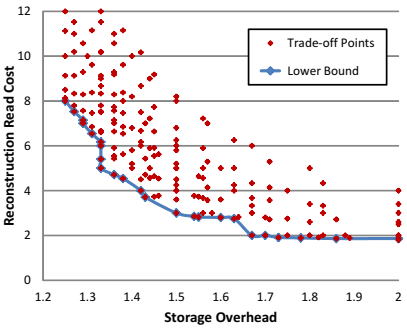


Figure 4: Overhead vs. Recon. Cost.

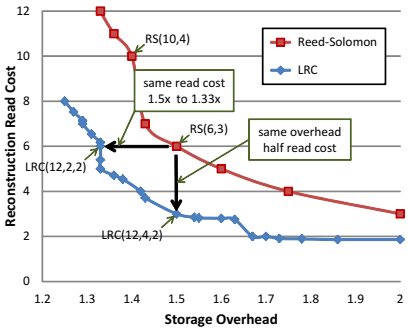


Figure 5: LRC vs. RS Code.

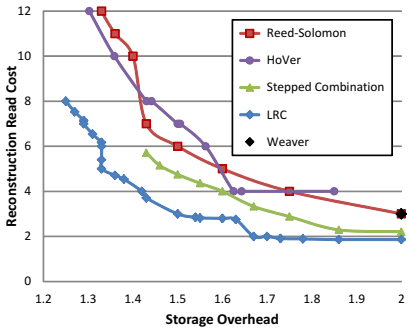


Figure 6: LRC vs. Modern Codes.

candidates for WAS.

The comparison shows that LRC achieves better cost and performance trade-off than Reed-Solomon across the range of parameters. If we keep the storage overhead the same, reconstruction in LRC is much more efficient than that in Reed-Solomon. On the other hand, if we keep reconstruction cost the same, LRC can greatly reduce storage overhead, compared to Reed-Solomon. In general, we can choose a point along the cost and performance trade-off curve, which can reduce storage overhead and reconstruction cost at the same time.

Compared to (6, 3) Reed-Solomon, we could keep storage overhead the same (at 1.5x) and replace the Reed-Solomon code with a (12, 4, 2) LRC. Now, the reconstruction cost is reduced from 6 to 3 for single-fragment failures, a reduction of 50%. This is shown by the vertical move in Figure 5.

Alternatively, as shown by the horizontal move in Figure 5, we could keep reconstruction cost the same (at 6) and replace the Reed-Solomon code with a (12, 2, 2) LRC. Now, the storage overhead is reduced from 1.5x to 1.33x. For the scale of WAS, such reduction translates into significant savings.

3.4 Comparison - Modern Storage Codes

We apply the same reliability analysis to state-of-the-art erasure codes designed specifically for storage systems, such as Weaver codes [18], HoVer codes [19] and Stepped Combination codes [20]. We examine the trade-off between reconstruction cost and storage overhead when these codes are at least as reliable as 3-replication.

The results are plotted in Figure 6. The storage overhead of Weaver codes is 2x or higher, so only a single point (storage overhead = 2x, reconstruction cost = 3) is shown in the Figure. We observe that the trade-off curve of Stepped Combination codes is strictly below that of Reed-Solomon. Therefore, both codes can achieve more efficient reconstruction than Reed-Solomon when storage overhead is fixed. Similarly, they require less storage overhead when reconstruction cost is fixed. HoVer codes offer many trade-off points better than Reed-Solomon.

Compared to these modern storage codes, LRC is superior in terms of the trade-off between reconstruction cost and storage overhead. The primary reason is that LRC separates parity fragments into local ones and global ones. In this way, local parities only involve minimum data fragments and can thus be most efficient for providing reconstruction reads when there is a hot spot, for reconstructing single failures, and for reconstructing a single fragment that is offline due to upgrade. On the other hand, global parities involve all the data fragments and can thus be most useful to provide fault tolerance when there are multiple failures. In contrast, in Weaver codes, HoVer codes and Stepped Combination codes, all parities carry both the duty of reconstruction and fault tolerance. Therefore, they cannot achieve the same trade-off as LRC.

LRC is optimized for reconstructing data fragments, but not parities, in order to quickly reconstruct on-demand based reads from clients. In terms of parity reconstruction, Weaver codes, HoVer codes and Stepped Combination codes can be more efficient. For instance, let's compare Stepped Combination code to LRC at the storage overhead of 1.5x, where both codes consist of 12 data fragments and 6 parities. For the Stepped Combination code, every parity can be reconstructed from 3 fragments, while for LRC the reconstruction of global parities requires as many as 12 fragments. It turns out that there is fundamental contention between parity reconstruction and data fragment reconstruction, which we studied in detail separately [21]. In WAS, since parity reconstruction happens only when a parity is lost (e.g., disk, node or rack failure), it is off the critical path of serving client reads. Therefore, it is desirable to trade the efficiency of parity reconstruction in order to improve the performance of data fragment reconstruction, as is done in LRC.

3.5 Correlated Failures

The Markov reliability model described in Section 3.1 assumes failures are independent. In practice, correlated failures do happen [10, 34]. One common correlated fail-

ure source is all of the servers under the same fault domain. WAS avoids these correlated failures by always placing fragments belonging to the same coding group in different fault domains.

To account for additional correlated failures, the Markov reliability model can be readily extended by adding transition arcs between non-adjacent states [10].

4 Erasure Coding Implementation in WAS

The WAS architecture has three layers within a storage stamp (cluster) - front-end layer, partitioned object layer and stream replication layer [1].

Erasure coding in WAS is implemented in the stream layer as a complementary technique to full data replication. It is also possible to implement erasure coding across multiple storage stamps on several data centers [38]. Our choice to implement erasure coding inside the stream layer is based on the fact that it fits the overall WAS architecture where the stream layer is responsible for keeping the data durable within a stamp and the partition layer is responsible for geo-replicating the data between data centers (see [1] for more details).

4.1 Stream Layer Architecture

The main components of the stream layer are the Stream Managers (SM), which is a Paxos [37] replicated server, and Extent Nodes (EN) (see Figure 7).

Streams used by the partition layer are saved as a list of extents in the stream layer. Each extent consists of a list of append blocks. Each block is CRC'd and this block is the level of granularity the partitioned object layer uses for appending data to the stream, as well as reading data (the whole block is read to get any bytes out of the block, since the CRC is checked on every read). Each extent is replicated on multiple (usually three) ENs. Each write operation is committed to all nodes in a replica set in a daisy chain, before an acknowledgment is sent back to the client. Write operations for a stream keep appending to an extent until the extent reaches its maximum size (in the range of 1GB-3GB) or until there is a failure in the replica set. In either case, a new extent on a new replica set is created and the previous extent is sealed. When an extent becomes sealed, its data is immutable, and it becomes a candidate for erasure coding.

4.2 Erasure Coding in the Stream Layer

The erasure coding process is completely asynchronous and off the critical path of client writes. The SM periodically scans all sealed extents and schedules a subset of them for erasure coding based on stream policies and system load. We configure the system to automatically erasure code extents storing Blob data, but also have the option to erasure code Table extents too.

As a first step of erasure coding of an extent, the SM creates fragments on a set of ENs whose number depends on the erasure coding parameters, for example 16 fragments for LRC (12, 2, 2).

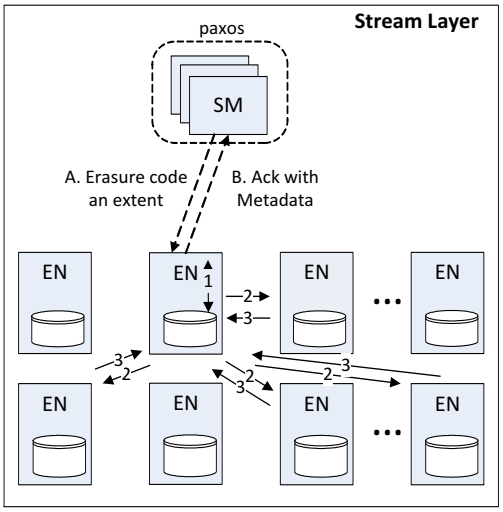


Figure 7: **Erasure Coding of an Extent** (not all target ENs are shown).

The SM designates one of the ENs in the extent's replica set as the coordinator of erasure coding (see Figure 7) and sends it the metadata for the replica set. From then on, the coordinator EN has the responsibility of completing the erasure coding. The coordinator EN has, locally on its node, the full extent that is to be erasure-coded. The EN prepares the extent for erasure coding by deciding where the boundaries for all of the fragments will be in the extent. It chooses to break the extent into fragments at append block boundaries and not at arbitrary offsets. This ensures that reading a block will not cross multiple fragments.

After the coordinator EN decides what the fragment offsets are, it communicates those to the target ENs that will hold each of the data and parity fragments. Then the coordinator EN starts the encoding process and keeps sending the encoded fragments to their designated ENs. The coordinator EN, as well as each target EN, keeps track of the progress made and persists that information into each new fragment. If a failure occurs at any moment in this process, the rest of the work can be picked up by another EN based on the progress information persisted in each fragment. After an entire extent is coded, the coordinator EN notifies the SM, which updates the metadata of the extent with fragment boundaries and completion flags. Finally, the SM schedules full replicas of the extent for deletion as they are no longer needed.

The fragments of an extent can be read directly by a client (i.e., the partition or front-end layer in WAS) by contacting the EN that has the fragment. However, if that target EN is not available or is a hot spot, the client can contact any of the ENs that has any of the fragments of the extent, and perform a reconstruction read (see Fig-

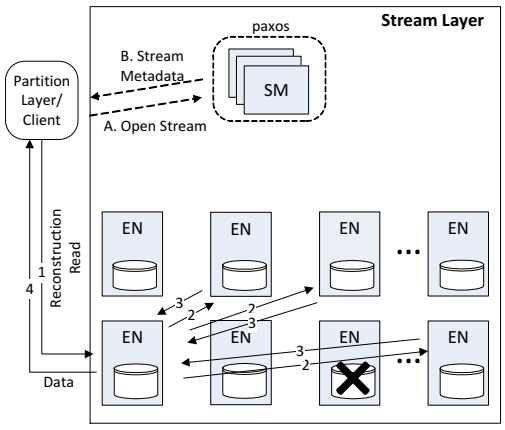


Figure 8: **Reconstruction for On-Demand Read** (not all target ENs are shown).

ure 8). That EN would read the other needed fragments from other ENs, then reconstruct that fragment, cache the reconstructed fragment locally in case there are other reads to it, and return the results to the client. LRC reduces the cost of this reconstruction operation considerably by reducing the number of source ENs that need to be accessed.

If the EN or the disk drive that hosts the extent fragment is unavailable for an extended period of time, the SM initiates the reconstruction of the fragment on a different EN. This operation is almost identical to the reconstruction steps shown in Figure 8 except for the fact that the operation is initiated by the SM instead of the client, and the data is written to disk rather than being sent back to the client.

4.3 Using Local Reconstruction Codes in Windows Azure Storage

When LRC is used as an erasure coding algorithm, each extent is divided into k equal-size *data* fragments. Then l local and r global *parity* fragments are created.

The placement of the fragments takes into account two factors: i) load, which favors less occupied and less loaded extent nodes; ii) reliability, which avoids placing two fragments (belonging to the same erasure coding group) into the same correlated domain. There are two primary correlated domains: fault domain and upgrade domain. A fault domain, such as rack, categorizes a group of nodes which can fail together due to common hardware failure. An upgrade domain categorizes a group of nodes which are taken offline and upgraded at the same time during each upgrade cycle. Upgrade domains are typically orthogonal to fault domains.

Let's now use LRC (12, 2, 2) as an example and illustrate an actual fragment placement in WAS. A WAS stamp consists of 20 racks. For maximum reliability, each of the total 16 fragments for an extent is placed in a different rack. If each fragment were similarly placed on a different upgrade domain, then at least 16 upgrade domains are required. In practice, however, too many upgrade domains can slow down upgrades and it is desirable to keep the number of upgrade domains low. In WAS, we currently use 10 upgrade domains. This means that we have at most 10% of the storage stamps (cluster) resources offline at any point in time during a rolling upgrade.

Given our desire to have fewer upgrade domains than fragments, we need an approach for placing the fragments across the upgrade domains to still allow LRC to perform its fast reconstruction for the fragments that are offline. To that end, we exploit the local group property of LRC and group fragments belonging to different local groups into same upgrade domains. In particular, we place the two local groups x and y so that their data fragments x_i and y_i are in the same upgrade domain i (i.e., x_0 and y_0 are placed in the same upgrade domain, but different fault domains). Similarly, we place the local parities p_x and p_y in one upgrade domain as well. The two global parities are placed in two separate upgrade domains from all other fragments.

Take for example LRC (12, 2, 2). In total, we use 9 upgrade domains for placing the fragments for an extent. There are two local groups, each with 6 data fragments, plus 2 local parities (1 per group), and then 2 global parities. When using 9 upgrade domains, we put the 2 global parities into two upgrade domains with no other fragments in them, we then put the 2 local parities into the same upgrade domain with no other fragments in them, and then the remaining 6 upgrade domains hold the 12 data fragments for the 2 local groups.

During an upgrade period, when one upgrade domain is taken offline, every single data fragment can still be accessed efficiently - either reading directly from the fragment or reconstructing from other fragments within its local group.

4.4 Designing for Erasure Coding

Scheduling of Various I/O Types. The stream layer handles a large mix of I/O types at a given time: on-demand open/close, read, and append operations from clients, create, delete, replicate, reconstruct, scrub, and move operations generated by the system itself, and more. Letting all these I/Os happen at their own pace can quickly render the system unusable. To make the system fair and responsive, operations are subject to throttling and scheduling at all levels of the storage system. Every EN keeps track of its load at the network ports and on individual disks to decide to accept, reject, or delay I/O requests. Similarly, the SM keeps track of data replication load on individual ENs and the system as a whole to make decisions on when to initiate replication, erasure

coding, deletion, and various other system maintenance operations. Because both erasure coding and decoding requires accessing multiple ENs, efficiently scheduling and throttling these operations is crucial to have fair performance for other I/O types. In addition, it is also important to make sure erasure coding is keeping up with the incoming data rate from customers as well as internal system functions such as garbage collection. We have a Petabyte of new data being stored to WAS every couple of days, and the built out capacity expects a certain fraction of this data to be erasure-coded. Therefore, the erasure coding needs to be scheduled such that it keeps up with the incoming rate of data, even when there are critical re-replications that also need to be scheduled due to a lost disk, node or rack.

Reconstruction Read-ahead and Caching. Reconstruction of unavailable fragments is done in unit sizes greater than the individual append blocks (up to 5MB) to reduce the number of disk and network I/Os. This read-ahead data is cached in memory (up to 256MB) of the EN that has done the reconstruction. Further sequential reads are satisfied directly from memory.

Consistency of Coded Data. Data corruption can happen throughout the storage stack for numerous reasons [36]. In a large-scale distributed storage system, data can become corrupted while at rest, while being read or written in memory, and while passing through several data paths. Therefore, it is essential to check the consistency of the data in every step of the storage system operations in addition to periodically scrubbing the data at rest.

Checksum and parity are the two primary mechanisms to protect against data corruption [35]. In WAS, we employ various CRC (Cyclic Redundancy Check) fields to detect data and metadata corruptions. For example, each append block contains a header with CRC of the data block, which is checked when the data is written and every time data is read. When a particular data read or reconstruction operation fails due to CRC checks, the operation is retried using other combinations of erasure-coded fragments. Also, the fragment with the corrupted block is scheduled for regeneration on the next available EN.

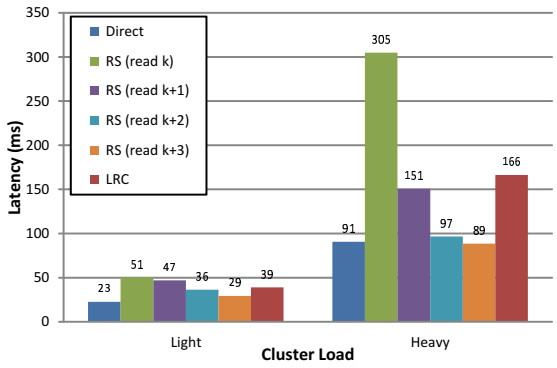
After each erasure encoding operation, several decoding combinations are tried from memory on the coordinator EN to check for successful restorations. This step is to ensure that the erasure coding algorithm itself does not introduce data inconsistency. For LRC (12, 2, 2), we perform the following decoding validations before allowing the EC to complete: $i)$ randomly choose one data fragment in each local group and reconstruct it using its local group; $ii)$ randomly choose one data fragment and reconstruct it using one global parity and the remaining data fragments; $iii)$ randomly choose one data fragment and reconstruct it using the other global parity and the remaining data fragments; $iv)$ randomly choose two data fragments and reconstruct them; $v)$ randomly choose three data fragments and reconstruct them; and $vi)$ randomly choose four data fragments (at least one in each group) and reconstruct them. After each decoding combination above, the CRC of the decoded fragment is checked against the CRC of the data fragment for successful reconstruction of data.

Finally, the coordinator EN performs a CRC of all of the final data fragments and checks that CRC against the original CRC of the full extent that needed to be erasure-coded. This last step ensures we have not used data that might become corrupted in memory during coding operations. If all these checks pass, the resulting coded fragments are persisted on storage disks. If any failure is detected during this process, the erasure coding operation is aborted, leaving the full extent copies intact, and the SM schedules erasure coding again on another EN later.

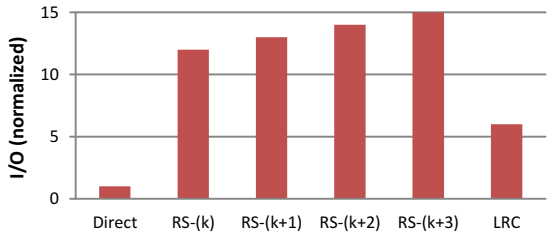
Arithmetic for Erasure Coding. Directly using Galois Field arithmetic for the erasure coding implementation is expensive because of all the emulation operations required on top of the integer arithmetic. Therefore, it is general practice to optimize Galois Field arithmetic operations by pre-computing and using addition and multiplication tables, which improves coding speed. In addition, Reed-Solomon codes can be further optimized by a transformation that enables the use of XOR operations exclusively [22]. To get the best possible performance for this transformation, the XOR operations can be ordered specifically based on the patterns in the coding and decoding matrices [23]. This scheduling removes most of the redundant operations and eliminates checking the coding matrix again and again during the actual coding pass. WAS uses all of the above optimizations to streamline in-memory encode and decode operations. Since modern CPUs perform XOR operations extremely fast, in-memory encode and decode can be performed at the speed which the input and output buffers can be accessed.

5 Performance

WAS provides cloud storage in the form of Blobs (user files), Tables (structured storage), Queues (message delivery), and Drives (network mounted VHDs). Applications have different workloads, which access each type of storage differently. The size of I/O can be polarized: small I/O is typically in the 4KB to 64KB range, predominantly assessing Tables and Queues; large I/Os (mostly 4MB), primarily accessing Blobs; and Drives can see a mixture of both. In this section, we characterize the performance of LRC and compare it to Reed-Solomon for small and large I/Os, respectively.

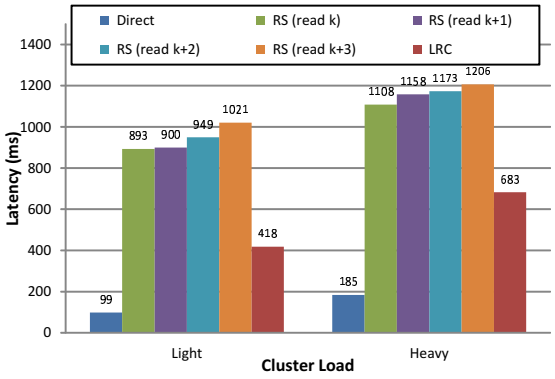


(a) Latency

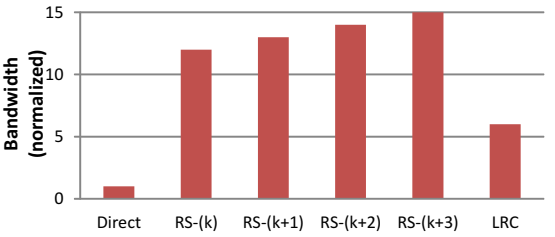


(b) Reconstruction I/O

Figure 9: **Small (4KB) I/O Reconstruction** - (12, 4) Reed-Solomon vs. (12, 2, 2) LRC.



(a) Latency



(b) Reconstruction Bandwidth

Figure 10: **Large (4MB) I/O Reconstruction** - (12, 4) Reed-Solomon vs. (12, 2, 2) LRC.

We compare LRC (12, 2, 2) to Reed-Solomon (12, 4), both of which yield storage cost at 1.33x. Note that (12, 3) Reed-Solomon is *not* an option, because its reliability is lower than 3-replication. Results are obtained on our production cluster with significant load variation over time. We separate the results based on the cluster load and contrast the gain of LRC when the cluster is lightly loaded to when it is heavily loaded. The production cluster, where these results were gathered, has one 1Gbps NIC for each storage node.

5.1 Small I/Os

The key metric for small I/O is latency and the number of I/Os taken by the requests. We run experiments with a mixture of direct read (reading a single fragment), reconstruction read with Reed-Solomon and LRC. The average latency results are summarized in Figure 9.

When the cluster load is light, all the latencies appear very low and comparable. There is not much difference between direct read and reconstruction with either Reed-Solomon or LRC. However, when the cluster load is heavy, there are definitely differences.

When a data fragment is unavailable, the reconstruction read with Reed-Solomon can be served by randomly selecting $k = 12$ fragments out of the remaining 15 fragments to perform erasure decoding. Unfortunately, the latency turns out much larger than that of direct read – 305ms vs. 91ms – because it is determined by the slowest fragment among the entire selection. Given the high latency, we exploit a simple technique - selecting more (denoted as k') fragments and decoding from the first k arrivals (represented as `RS (read k') or RS+(k')` in Figure 9). This technique appears very effective in weeding out slow fragments and reduces the latency dramatically.

In comparison, reconstruction with LRC is fast. It requires only 6 fragments to be read and achieves the latency of 166ms, which is comparable to 151ms with Reed-Solomon reading 13 fragments. Note that extremely aggressive reads (reading all 15) with Reed-Solomon achieves even lower latency, but it comes at a cost of many more I/Os. The relative number of I/Os, normalized by that of direct read, is shown in Figure 9(b). The fast reconstruction and the I/O cost savings are the reasons why we chose LRC over Reed-Solomon for Windows Azure Storage's erasure coding.

5.2 Large I/Os

We now examine the reconstruction performance of 4MB large I/Os. The key metric is latency and bandwidth consumption. We compare direct read to reconstruction with Reed-Solomon and LRC. The results are presented in Figure 10.

The results are very different from the small I/O case. Even when the cluster load is light, the reconstruction with erasure coding is already much slower than direct read, given the amount of bandwidth it consumes. Compared to direct read taking 99ms, Reed-Solomon with 12

fragments takes 893ms – 9 times slower.

In large I/Os, the latency is mostly bottlenecked by network and disk bandwidth, and the bottleneck for these results was the 1Gbps network card on the storage nodes. Since LRC reduces the number of fragments by half, its latency is 418ms and significantly reduced from that of Reed-Solomon. Note that, different from the small I/O case, aggressive reads with Reed-Solomon using more fragments does not help, but rather hurts latency. The observation is similar when the cluster load is heavy.

Because of the significantly reduced latency and the bandwidth savings, which are particularly important when the system is under heavy load or has to recover from a rack failure, we chose LRC for Windows Azure Storage.

5.3 Decoding Latency

To conclude the results, we also wanted to compare the latency spent on decoding fragments between Reed-Solomon and LRC. The average latency of decoding 4KB fragments is 13.2us for Reed-Solomon and 7.12us for LRC. Decoding is faster in LRC than Reed-Solomon because only half the number of fragments are involved. Even so, the decoding latencies are typically in microseconds and *several orders of magnitude smaller* than the overall latency to transfer the fragments to perform the reconstruction. Therefore, from the latency of decoding standpoint, LRC and Reed-Solomon are comparable. Note that, pure XOR-based codes, such as Weaver codes [18], HoVer codes [19] and Stepped Combination codes [20], can be decoded even faster. The gain of faster decoding, however, would not matter in WAS, as the decoding time is orders of magnitude smaller than the transfer time.

6 Related Work

Erasure Coding in Storage Systems: Erasure coding has been applied in many large-scale distributed storage systems, including storage systems at Facebook and Google [3, 4, 5, 8, 9, 10]. The advantage of erasure coding over simple replication is that it can achieve much higher reliability with the same storage, or it requires much lower storage for the same reliability [7]. The existing systems, however, do not explore alternative erasure coding designs other than Reed-Solomon codes [13]. In this work, we show that, under the same reliability requirement, LRC allows a much more efficient cost and performance trade-off than Reed-Solomon.

Performance: In erasure-coded storage systems, node failures trigger rebuilding process, which in turn results in degraded latency performance on reconstruction reads [6]. Moreover, experience shows that transient errors in which no data are lost account for more than 90% of data center failures [10]. During these periods as well as upgrades, even though there is no background data rebuilding, reads trying to access unavailable nodes are still served through reconstruction.

Complementary to system techniques, such as load balancing and prioritization [11], LRC explores whether the erasure coding scheme itself can be optimized to reduce repair traffic and improve user I/Os.

Erasure Code Design: LRC is a critical improvement over our own Pyramid codes [14]. LRC exploits non-uniform parity degrees, where some parities connect to fewer data nodes than others. Intuitively, the parities with fewer degrees facilitate efficient reconstruction. This direction was originally pioneered for communication by landmark papers on Low Density Parity Check (LDPC) codes [15, 16]. LDPC were recently explored in the area of storage [17, 20]. In particular, Plank et al. [17] applied enumeration and heuristic methods to search for parity-check erasure codes of small length. Due to exponential search space, the exploration was limited to 3, 4 and 5 parities. The codes discovered cannot tolerate arbitrary three failures, which is the minimum requirement in WAS. Stepped Combination codes [20] are LDPC codes with very small length, offering fault tolerance guarantee and efficient reconstruction, but do not provide the same trade-offs that LRC can achieve..

Reed-Solomon Codes are Maximum Distance Separable (MDS) codes [12], which require minimum storage overhead for given fault tolerance. LRC is not MDS and thus requires higher storage overhead for the same fault tolerance. The additional storage overhead from the local parities are exploited for efficient reconstruction. This direction of trading storage overhead for reconstruction efficiency is also explored by other state-of-the-art erasure codes designed specifically for storage systems, such as Weaver codes [18], HoVer codes [19] and Stepped Combination codes [20]. We show that LRC achieves better trade-offs than these modern storage codes for WAS' erasure coding design goals .

To improve reconstruction performance, instead of reading from fewer fragments as in LRC, a promising alternative is to read instead from more fragments, but less data from each [24, 25, 26, 27]. However, practical solutions known so far [26, 27] achieve only around 20%-30% savings in terms of I/O and bandwidth, much less than LRC.

7 Summary

Erasure coding is critical to reduce the cost of cloud storage, where our target storage overhead is 1.33x of the original data. When using erasure coding, fast reconstruction of offline data fragments is important for performance. In Windows Azure Storage, these data fragments can be offline due to disk, node, rack and switch failures, as well as during upgrades.

We introduced Local Reconstruction Codes as a way to reduce the number of fragments that need to be read from to perform this reconstruction, and compared LRC to Reed-Solomon. We showed that LRC (12, 2, 2), which has a storage overhead of 1.33x, saves significant I/Os and bandwidth during reconstruction when compared to Reed-Solomon (12, 4). In terms of latency, LRC has comparable latency for small I/Os and better latency for large I/Os.

We chose LRC (12, 2, 2) since it achieves our 1.33x storage overhead target and has the above latency, I/O and bandwidth advantages over Reed-Solomon. In addition, we needed to maintain durability at the same or higher level than traditional 3 replicas, and LRC (12, 2, 2) provides better durability than the traditional approach of keeping 3 copies. Finally, we explained how erasure coding is implemented, some of the design considerations, and how we can efficiently lay out LRC (12, 2, 2) across the 20 fault domains and 10 upgrade domains used in Windows Azure Storage.

8 Acknowledgements

We would like to thank Andreas Haeberlen, Geoff Voelker, and anonymous reviewers for providing valuable feedback on this paper. We would also like to thank all of the members of the Windows Azure Storage team.

References

[1] B. Calder et al., "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," *ACM SOSP*, 2011.

[2] D. T. Meyer et al., "Fast and Cautious Evolution of Cloud Storage," *HotStorage*, 2010.

[3] J. Kubiatowicz et al., "OceanStore: An Architecture for Global-Scale Persistent Storage," *ACM ASPLOS*, Nov. 2000.

[4] A. Haeberlen, A. Mislove, and P. Druschel, "Glacier: Highly durable, decentralized storage despite massive correlated failures," *USENIX NSDI*, 2005.

[5] M. Abd-El-Malek et al., "Ursa Minor: Versatile Cluster-based Storage," *USENIX FAST*, 2005.

[6] C. Ungureanu et al., "HydraFS: A High-Throughput File System for the HYDRAstor Content-Addressable Storage System," *USENIX FAST*, 2010.

[7] H. Weatherspoon, and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," *In Proc. IPTPS*, 2001.

[8] D. Borthakur et al., "HDFS RAID," *Hadoop User Group Meeting*, Nov. 2010.

[9] A. Fikes, "Storage Architecture and Challenges," *Google Faculty Summit*, 2010.

[10] D. Ford et al., "Availability in Globally Distributed Storage Systems,", *USENIX OSDI*, 2010.

[11] L. Tian et al., "PRO: A Popularity-based Multi-threaded Reconstruction Optimization for RAID-Structured Storage Systems," *USENIX FAST*, 2007

[12] F. J. MacWilliams and N. J. A. Sloane, "The Theory of Error Correcting Codes," *Amsterdam: North-Holland*, 1977.

[13] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields", *J. SIAM*, 8(10), 300-304, 1960.

[14] C. Huang, M. Chen, and J. Li, "Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems," *Proc. of IEEE NCA*, Cambridge, MA, Jul. 2007.

[15] R. G. Gallager, "Low-Density Parity-Check Codes," *MIT Press*, Cambridge, MA, 1963.

[16] M. G. Luby et al., "Efficient Erasure Correcting Codes," *IEEE Transactions on Information Theory*, 2011.

[17] J. S. Plank, R. L. Collins, A. L. Buchsbaum, and M. G. Thomason, "Small Parity-Check Erasure Codes - Exploration and Observations," *Proc. DSN*, 2005.

[18] J. L. Hafner, "Weaver codes: Highly fault tolerant erasure codes for storage systems," *USENIX FAST*, 2005.

[19] J. L. Hafner, "HoVer Erasure Codes for Disk Arrays," *Proc. of DSN*, 2006.

[20] K. M. Greenan, X. Li, and J. J. Wylie, "Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs," *IEEE Mass Storage Systems and Technologies*, 2010.

[21] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin, "On the Locality of Codeword Symbols," *Allerton*, 2011.

[22] J. Blomer et al., "An XOR-Based Erasure-Resilient Coding Scheme," Technical Report No. TR-95-048, ICSI, Berkeley, California, Aug. 1995.

[23] J. Luo, L. Xu, and J. S. Plank, "An Efficient XOR-Scheduling Algorithm for Erasure Codes Encoding," *Proc. DSN*, Lisbon, Portugal, June, 2009.

[24] A. G. Dimakis et al., "Network Coding for Distributed Storage Systems," *IEEE Transactions on Information Theory*, Vol. 56, Issue 9, Sept. 2010.

[25] L. Xiang et al., "Optimal recovery of single disk failure in RDP code storage systems," *ACM SIGMETRICS*, 2010.

[26] O. Khan et al., "Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads," *USENIX FAST*, San Jose, Feb. 2012.

[27] Y. Hu et al., "NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds," *USENIX FAST*, San Jose, 2012.

[28] J. H. Howard et al., "Scale and Performance in a Distributed File System," *ACM ToCS*, Feb. 1988.

[29] M. Satyanarayanan et al., "CODA: A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers*, Apr. 1990.

[30] B. Liskov et al., "Replication in the Harp File System," *ACM SOSP*, Pacific Grove, CA, Oct. 1991.

[31] F. Dabek et al., "Wide-Area Cooperative Storage with CFS," *ACM SOSP*, 2001.

[32] B. G. Chun et al., "Efficient Replica Maintenance for Distributed Storage Systems," *USENIX NSDI*, 2006.

[33] Q. Xin et al., "Reliability mechanisms for very large storage systems," *Proc. of IEEE Conference on Mass Storage Systems and Technologies*, 2003.

[34] S. Nath et al., "Subtleties in Tolerating Correlated Failures in Wide-Area Storage Systems," *USENIX NSDI*, 2006.

[35] A. Krioukov et al., "Parity Lost and Parity Regained," *USENIX FAST*, Feb. 2008.

[36] L. N. Bairavasundaram et al., "An Analysis of Data Corruption in the Storage Stack," *USENIX FAST*, Feb. 2008.

[37] L. Lamport, "The Part-Time Parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133-169, May 1998.

[38] J. K. Resch, and J. S. Plank, "AONT-RS: Blending Security and Performance in Dispersed Storage Systems," *USENIX FAST*, Feb. 2011.