On Fault Tolerance, Locality, and Optimality in Locally Repairable Codes

Oleg Kolosov, *School of Electrical Engineering, Tel Aviv University;*
Gala Yadgar, *Computer Science Department, Technion, and School of Electrical Engineering, Tel Aviv University;* Matan Liram, *Computer Science Department, Technion;*
Itzhak Tamo, *School of Electrical Engineering, Tel Aviv University;*
Alexander Barg, *Department of ECE/ISR, University of Maryland*

https://www.usenix.org/conference/atc18/presentation/kolosov

This paper is included in the Proceedings of the
2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-939133-02-1

On Fault Tolerance, Locality, and Optimality in Locally Repairable Codes

Oleg Kolosov[†], Gala Yadgar[*†], Matan Liram[*], Itzhak Tamo[†], and Alexander Barg[§]
[†]*School of Electrical Engineering, Tel Aviv University*
[*]*Computer Science Department, Technion*
[§]*Department of ECE/ISR, University of Maryland*

Abstract

Erasure codes are used in large-scale storage systems to allow recovery of data from a failed node. A recently developed class of erasure codes, termed *locally repairable codes (LRCs)*, offers tradeoffs between storage overhead and repair cost. LRCs facilitate more efficient recovery scenarios by storing additional parity blocks in the system, but these additional blocks may eventually increase the number of blocks that must be reconstructed. Existing codes differ in their use of the additional parity blocks, but also in their locality semantics and in the parameters for which they are defined. As a result, existing theoretical models cannot be used to directly compare different LRCs to determine which code will offer the best recovery performance, and at what cost.

In this study, we perform the first systematic comparison of existing LRC approaches. We analyze Xorbas, Azure's LRCs, and the recently proposed Optimal-LRCs in light of two new metrics: the *average degraded read cost*, and the *normalized repair cost*. We show the tradeoff between these costs and the code's fault tolerance, and that different approaches offer different choices in this tradeoff. Our experimental evaluation on a Ceph cluster deployed on Amazon EC2 further demonstrates the different effects of realistic network and storage bottlenecks on the benefit from each examined LRC approach. Despite these differences, the normalized repair cost metric can reliably identify the LRC approach that would achieve the lowest repair cost in each setup.

1 Introduction

In large-scale storage systems consisting of hundreds of thousands of servers, node failures are the norm rather than the exception. For this reason, redundancy is added to ensure availability of the data despite the failures. Typically, the redundancy of hot data is achieved by *replication* of each data object, ensuring the availability of the data as long as one replica is available. This also allows efficient reconstruction of data that was stored on a failed node from the surviving replicas.

Due to the high overhead of replication, most of the data is stored redundantly by erasure coding. With an (n,k) *erasure code*, the data is split into k *data blocks* that are used to generate $n-k$ *parity blocks*. The blocks are distributed across n different nodes, so that the original data can be reconstructed as long as at least k blocks are available. The *storage overhead* of erasure coding is $\frac{n}{k}$ —considerably lower than that of replication. However, reconstruction of one data block requires reading k surviving blocks—an overhead considerably higher than that of replication.

Storage systems distinguish between two types of node failures. *Transient failures* may be caused by system restarts or updates, after which the node is available again. During this time, read operations of data stored on the failed node are served as *degraded reads*—only the required data blocks are reconstructed from the surviving blocks. *Permanent failures* occur when the node malfunctions and is no longer accessible. Typically, a failure is considered permanent after 15 minutes of unavailability, which triggers full recovery of its data. Recent studies indicate that transient failures comprise 90% of failure events [28], and only the remaining 10% trigger full node recovery. Nevertheless, recovery traffic incurs significant load on the data center's servers and network—up to 180TB of data transfer between racks each day, according to a recent study on Facebook's data centers [24].

The vast majority of failures (up to 98% [24]) constitute exactly one unavailable node. Thus, several approaches have been used to design erasure codes that can withstand several concurrent failures, but optimize the recovery cost of a single node. These include preprocessing the surviving data to minimize repair network bandwidth [6,22,25], and reducing the amount of data read from each surviving node [7, 9, 12, 14, 23, 38]. These codes can reduce the amount of data read by up to 50%, but in many realistic settings, this reduction is no more than 25%, or not applicable due to the required I/O granularity [18].

A different approach increases the storage overhead and utilizes the added redundancy to optimize the recovery of a single data node. An (n,k,r) *locally repairable code (LRC)* supports the *local* recovery of an unavailable block by reading at most r surviving blocks. These codes were originally designed to reduce the cost of degraded reads, and thus most of them optimize only the recovery of data blocks [10,11]. Others further optimize the recov-

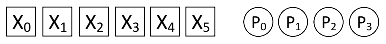

Figure 1: (10,6) Reed-Solomon

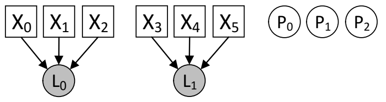

Figure 2: (11,6,3) Azure-LRC

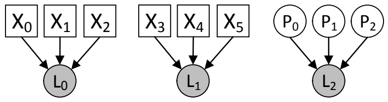

Figure 3: (12,6,3) Optimal-LRC

ery of all of the parity blocks, but do so for a limited set of system parameters [28]. In a recent work [30], new codes were constructed that support local recovery of both data and parity blocks, with the same storage overhead as previously known constructions.

LRCs present an inherent tradeoff. On the one hand, they considerably reduce the amount of data that must be read for degraded reads and recovery. On the other hand, in order to store the additional parity, the system must store more blocks on each of its nodes, or allocate more nodes for the same amount of data. In the first case, more data must be reconstructed whenever a node fails, while the latter increases the probability of failure in the system. As a result, LRCs not only increase the system's storage overhead, but might also increase its overall recovery costs. Different codes offer different tradeoffs between storage overhead and recovery cost, and between recovery cost and the cost of degraded reads. Furthermore, they are defined for different (n,k,r) combinations and differ in their locality semantics. Thus, directly comparing their costs and benefits is a nontrivial task, which makes it hard to choose the optimal code and configuration for a given system.

In this study, we perform the first comprehensive analysis of the different LRC approaches. We take into account the overall cost of recovery, including data and parity blocks, as well as the maximum number of failed blocks the code can recover. Our analysis includes *Xorbas* [28], *Azure-LRC*—the LRC codes used by Microsoft Azure [11], and *Optimal-LRC*—a recently proposed theoretically optimal code [30]. We also define a new code, *Azure-LRC+1*, which is based on Azure-LRC and supports efficient recovery of all parity blocks.

We conduct a theoretical comparison between the different LRC approaches. Our analysis demonstrates the limitations of existing measures, such as locality and average repair cost. Thus, we define new metrics that model each code's overhead, full-node repair cost, degraded read cost, and fault tolerance. Our results demonstrate the tradeoff between the objectives measured by these costs, and how different codes optimize different objectives.

We follow the theoretical analysis with an evaluation of these codes in a Ceph cluster deployed in AWS EC2. Our experimental evaluation shows that we can accurately predict the amount of data required by each code for reconstructing an entire storage node. This prediction also provides a good estimate of the time required for reconstruction, for most combinations of storage type, network configurations, and foreground traffic.

The rest of this paper is organized as follows. Section 2 presents LRCs and motivates our analysis. We describe our new LRC and metrics in Section 3, and our theoretical analysis in Section 4. Our system-level setup is described in Section 5, with the evaluation in Section 6. We survery related work in Section 7, and conclude in Section 8.

2 Preliminaries

The storage overhead of an erasure code is defined as $\frac{n}{k}$. Its *minimal distance*, d, is defined as the smallest number of concurrent node failures that may cause data loss. In other words, there is at least one combination of d node failures from which the code will not be able to recover the data. An important class of codes, termed *maximum distance separable (MDS)* codes, is characterized by the relation $d = n - k + 1$, and provides the largest possible d for given n and k. In MDS codes, k surviving blocks are required to recover a failed block. Reed-Solomon codes [26] are the most commonly used MDS codes owing to their parameter flexibility and efficient implementation.

An (n,k,r) locally repairable code (LRC) consists of k data blocks and $n-k$ parity blocks. The data blocks are grouped into *local groups* no larger than r. A *local parity* is computed from each local group of blocks and can be used for the recovery of any block in this group. In total, each local group of LRC contains at most $r+1$ blocks. In case of an arbitrary failure of one block in a local group, r surviving blocks are required for its recovery. A *global parity* is a function of all data blocks, and can thus be used to recover any lost block. Pyramid codes [10], which are based on (n,k) Reed-Solomon codes were the first suggested family of LRCs. Another family, Azure-LRC, is a variation of Pyramid codes and is used in Windows Azure [11].

Figure 1 depicts a (10,6) Reed-Solomon code, and Figure 2 shows the (11,6,3) Azure-LRC which results from replacing one of its global parities with two local parities. In this example, P_3 was replaced with L_0 and L_1, which can be used in the recovery of groups (X_0,X_1,X_2) and (X_3,X_4,X_5), respectively. In the new code, any of the data blocks can be repaired by reading the remaining three blocks in its local group. Thus, the recovery cost of a data block is reduced by 50%. However, the overhead increases by 10%, from $\frac{10}{6}$ to $\frac{11}{6}$. Note also that the new code is non-MDS: it can repair any four missing blocks but not any five, therefor $d = 5$, but $n-k+1 = 6$.

Azure-LRC successfully reduces the repair cost of data blocks and local parities, and, as a result, the degraded

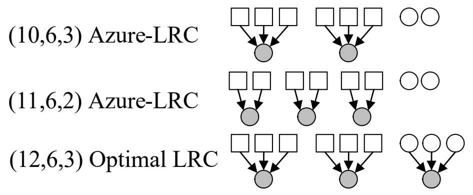

Figure 4: Three LRCs with the same k, demonstrating different tradeoffs between locality and overhead.

read cost. However, due to the allocation of blocks to nodes, when an entire node must be reconstructed, this node will include a significant number of global parities, which will require k surviving blocks for recovery. For example, in (11,6,3) Azure-LRC, which contains $m = 3$ global parities, an average of $\frac{3}{11} = 27.2\%$ of the blocks stored on each node will be global parities.

Coding theory distinguishes between two types of (n,k,r) LRCs. In codes with *information-symbol locality*, only the data blocks can be repaired in a local fashion by r surviving blocks, while the global parities require k blocks for recovery. We refer to these codes as *data-LRCs*. Pyramid and Azure-LRC are data-LRCs. In contrast, in codes with *all-symbol locality*, all the blocks, including the global parities, can be repaired locally from r surviving blocks. We refer to such codes as *full-LRCs*.

Optimal-LRC is a recently proposed full-LRC [30]. In this code, k data blocks and m global parities are divided into groups of size r, and a local parity is added to each group, allowing repair of *any* lost block by the r surviving blocks in its group. r does not necessarily divide $m+k$, but Optimal-LRC requires that $n \bmod (r+1) \neq 1$. Figure 3 shows a (12,6,3) Optimal-LRC. Each of the global parities, P_0, P_1, and P_2, can be reconstructed from the other global parities and the local parity L_2. The overhead of this code is higher than that of the (11,6,3) Azure-LRC in Figure 2, but its minimum distance is also higher ($d = 6$).

Full-LRCs introduce a new point in the tradeoff between fault tolerance and performance, which previously consisted only of MDS codes and data-LRCs. Gopalan et al. [8] proved that an upper bound on the minimal distance for an (n,k,r) LRC is $d \leq n-k-\lceil \frac{k}{r} \rceil + 2$. Codes that achieve this bound are regarded as optimal; in particular, Optimal-LRC has been shown to achieve this bound. Specifically, the minimum distance of Optimal-LRC was shown to be [30]:

$$d = n-k-\left\lceil \frac{k}{r} \right\rceil + 2, \quad \text{if } (r+1)|n$$

$$d \geq n-k-\left\lceil \frac{k}{r} \right\rceil + 1, \quad \text{if } (r+1) \not| n, \quad r|(k+1).$$

Challenges. Azure-LRC provides an appealing tradeoff when compared to Reed-Solomon codes: a 10% increase in storage overhead can halve the cost of all degraded reads and most block repairs. Unfortunately, the comparison between data-LRCs and full-LRCs is not similarly straightforward. Consider, for example, the three codes in Figure 4. The (11,6,2) Azure-LRC has three local parities, one more than the (10,6,3) Azure-LRC, which reduces its r from 3 to 2, but increases its overhead by 10%. The (12,6,3) Optimal-LRC also has three local parities. However, rather than reducing r, the additional local parity enables local repair of the global parities. Thus, r represents different locality semantics in each of these models. In addition, each model represents a different tradeoff between the cost of degraded read and the cost of full node repair, and between these costs and the overhead.

It is not entirely clear which of these codes will have the lowest repair cost. Clearly, r alone cannot serve as a metric for comparing data-LRCs to full-LRCs. The *average repair cost (ARC)* used in previous analyses [11] fails to capture the effect the code's overhead has on its repair cost. In the next section, we introduce three composite metrics that facilitate a systematic comparison of LRCs.

The task of comparing different codes is further complicated by the fact that existing codes are not all defined for the same range of parameters. Our new metrics alleviate this problem to some extent. To eliminate the problem completely, we adopt a somewhat 'flexible' interpretation of Azure-LRC. We also use a new construction of Optimal-LRC, which is optimal for parameters for which an explicit construction has not been given before.

Finally, theoretically proven benefits are not always achievable in real systems. The repair-cost benefit of different codes may be determined by factors such as storage and network bandwidth, the nature and priority of the foreground load, and the system-level implementation. Thus, we complement our theoretical analysis with an evaluation on a distributed cluster in Amazon EC2, where we verify our metrics and identify additional factors that should be taken into account when designing an erasure coded storage system.

3 Methodology

Metrics. The starting point of our theoretical analysis consists of the existing measures described above: r is the maximal number of blocks required for the recovery of any block or a data block, in full-LRCs and data-LRCs, respectively. The *overhead* of the code is $\frac{n}{k}$, and its minimal distance is d. We use d to represent the code's fault tolerance, despite its inherent limitation—two codes with the same d may be considered equally fault tolerant, although one may prevent data loss in more combinations of correlated failures than the other [11]. The *mean time to data loss (MTTDL)* is considered a more accurate measure for fault tolerance. However, to calculate the MTTDL of a code, one must construct a Markov chain for every specific set of n,k,r parameters. In addition, this model does not always yield an analytic closed-form equation. Thus, d is more appropriate for our large-scale analysis. For a limited comparison of a small set of constructions, d can be replaced with MTTDL.

The average repair cost (ARC) has been used in previous studies [11], and is based on the assumption that the probability of repair due to failure is the same for all blocks. It is defined as

$$ARC = \frac{\sum_{i=1}^{n} cost(b_i)}{n},$$

where b_i is the ith block in the code, and $cost(b_i)$ is the number of blocks required for the repair of b_i. For example, the ARC of the (10,6,3) Azure-LRC in Figure 4 is $\frac{(8\times3)+(2\times6)}{10} = 3.6$. Similarly, in the same figure, the ARC of the (11,6,2) Azure-LRC is 2.73 and that of the (12,6,3) Optimal-LRC is 3.

ARC does not take into account the higher overhead of some of these codes, which implies that more blocks will have to be repaired in the event of a node failure. We address this by defining a new composite metric for the cost of full-node repair. The *normalized repair cost (NRC)* of a code is the product of its ARC and overhead:

$$NRC = ARC \times \frac{n}{k} = \frac{\sum_{i=1}^{n} cost(b_i)}{k}.$$

NRC can also be viewed as the average cost of repairing a failed data block, where the cost of repairing the parity blocks is amortized over the k data blocks. For example, the NRC of the (10,6,3) Azure-LRC in Figure 4 is $\frac{(8\times3)+(2\times6)}{6} = 6$. Similarly, the NRC of the (11,6,2) Azure-LRC is 5 and that of the (12,6,3) Optimal-LRC is 6.

ARC is also inappropriate for modeling the cost of degraded reads. By definition, degraded reads refer to data blocks only, while ARC averages the repair cost of all blocks—data and parity alike. We define the *average degraded read cost* (*'degraded cost'*, in short) as the average cost of repairing data blocks only:

$$Degraded\ cost = \frac{\sum_{i=1}^{k} cost(b_i)}{k},$$

where blocks $b_1, ..., b_k$ are the object's data blocks. For example, the degraded cost of the (10,6,3) Azure-LRC and the (12,6,3) Optimal-LRC in Figure 4 is $\frac{6\times3}{6} = 3$. Similarly, the degraded cost of the (11,6,2) Azure-LRC is 2. Note that in the general case, the degraded cost is not always equal to r.

We base our analysis on three existing LRCs: Xorbas, Azure-LRC, and Optimal-LRC. We use Reed-Solomon codes as a baseline for some of our comparisons. Below, we describe how we extended the definitions of these codes for our analysis and evaluation.

Xorbas. Xorbas [28] is a full-LRC, in which the global parities can be recovered from the local parities. Figure 5 shows a (16,10,5) Xorbas code. Each local parity belongs

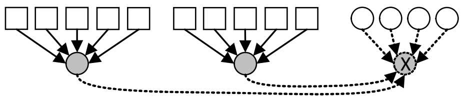

Figure 5: (16,10,5) Xorbas. $\otimes$ marks a function computed by the local parities, not a real block.

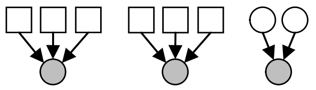

Figure 6: (11,6,3) Azure-LRC+1

to a group containing five data blocks. The special construction of Xorbas ensures that any of the global parities can be reconstructed by the remaining global parities and the two local parities. Thus, $r = 5$ for *all* the blocks in the code. This special property can be maintained if we remove the same number of blocks from each group. For example, a (13,8,4) Xorbas code can be obtained by removing two data blocks and one global parity from the original construction. The number of global parities can be further reduced to achieve a lower overhead, without reducing r. For example, a (12,8,4) Xorbas code has the same r as the (13,8,4) code, but a smaller d.

Azure-LRC. We use Azure-LRC as the data-LRC in our evaluation. It is explicitly defined in its original paper only for (n,k,r) where r divides k, and the number of local parities is $l = \frac{k}{r}$ [11]. For the sake of analysis, we extend this code to the general case as follows. In an (n,k,r) Azure-LRC where r does not divide k, the number of local parities is $l = \lceil \frac{k}{r} \rceil$. $l-1$ groups contain r data blocks and one local parity. The remaining group contains $k \mod r$ data blocks and one local parity. For the code to have at least one global parity, we must only ensure that $k + l < n$. Although this extension results in asymmetric allocation of data blocks to groups, it allows us to consider Azure-LRC in most (n,k,r) combinations.

Azure-LRC+1. For the sake of analysis, we define a new full-LRC which is based on Azure-LRC. An (n,k,r) *Azure-LRC+1* code is constructed by adding one local parity to the group of global parities of an $(n-1,k,r)$ Azure-LRC. This local parity is computed as the XOR of all the global parities. Figure 6 shows an (11,6,3) Azure-LRC+1 constructed from the (10,6,3) Azure-LRC in Figure 4. When one global parity block is missing, it can be repaired from the remaining global parities and the additional local parity. Thus, Azure-LRC+1 will have $l+1$ local parities, $l = \lceil \frac{k}{r} \rceil$, and can be constructed as long as $k + l + 1 < n$. This naïve definition implies that an Azure-LRC+1 construction may result in a local parity added to a 'group' of one global parity. Nevertheless, it has the added value of being directly and easily applicable to any system that uses Azure-LRC or Pyramid codes, and is thus an important aspect of our analysis.

Optimal-LRC. The original Optimal-LRC construction [30] was shown to be optimal for the cases described in Section 2. However, extending this construction to all admissible (n,k,r) combinations results in a code with lower d, which is suboptimal. To address this issue, we devised a new construction of codes in the spirit of the original construction. The advantage of the new construction is that it applies to all parameters n,k,r such that $n \bmod (r+1) \neq 1$. Furthermore, it can be shown that our new construction attains the largest possible minimum distance even when the upper bound $n-k-\lceil \frac{k}{r} \rceil + 2$ is not attainable. In summary, the new construction is the first optimal construction for *all* admissible parameters. The construction and the proof of its optimality can be found in [13].

Evaluation parameters. We computed the ARC, NRC, degraded cost, overhead, and d for each of the codes described above, for all (n,k,r) combinations for which they are defined, where $9 \leq n \leq 19$ and $\frac{n}{k} \leq 2$. These combinations include specific sets of parameters that appear in the literature and in documented deployments: (18,12,3) Azure-LRC [11], (16,10,5) Xorbas, (14,10) Reed-Solomon [24], and (9,6) Reed-Solomon [37]. Due to space constraints, and for clarity of presentation, we show only results for $12 \leq n \leq 18$ and $\frac{n}{k} \leq 1.6$, which include the more common combinations. This range of parameters suffices for demonstrating our observations, which we verified on the complete range.

4 Theoretical Analysis

Figure 7 shows NRC and the degraded read cost of the different codes. For the same n, k, and r, the degraded cost is usually the same for all codes. It is different when r does not divide k, where the codes differ in their allocation of blocks to groups. As we expected, for the same n and k, increasing r increases each code's degraded read cost and NRC. However, when comparing different codes, neither r nor the degraded read cost can indicate which code will have the lowest full-node repair cost. For example, the NRC of (14,10,6) Azure-LRC+1 is lower than that of (14,10,5) Azure-LRC, although its degraded cost is higher.

Figure 8 shows the minimum distance, d, of the different codes. Figures 7 and 8 together demonstrate a clear tradeoff between repair costs and fault tolerance. In general, for given n, k, and r, to increase d one must either increase n or increase r, thus increasing both the degraded cost and NRC. Nevertheless, different codes offer different points in this tradeoff.

Data-LRC vs. full-LRC. For the same (n,k,r), there is always one full-LRC with a lower NRC than that of Azure-LRC. However, in most settings, the reduction in NRC is coupled with a reduction in d. In the settings in which it is defined, Xorbas achieves the same d but a higher NRC than Azure-LRC+1 and Optimal-LRC. Optimal-LRC and Azure-LRC+1 achieve the same d and NRC in many settings. In the settings where the NRC of Azure-LRC+1 is lower than that of Optimal-LRC, its d is also lower (except for a few corner cases discussed below).

In Figure 9, we compare the NRC of (n,k,r) Azure-LRC to that of the $(n+1,k,r)$ full-LRCs with the same d. The full-LRCs use an additional local parity to allow fast repair of the global parities. This addition always reduces the repair cost, despite the increase in storage overhead.

Optimality of Optimal-LRC. Despite its optimal properties, our analysis reveals that for a given (n,k,r), Optimal-LRC does not always achieve the lowest NRC. Optimal-LRC is designed to accommodate the global parities with the data blocks in the same group. However, when the number of global parities is much smaller than r, this results in increasing the size of one of the groups, thus increasing the NRC. For example, Figure 10 shows a (12,8,5) Azure-LRC whose NRC is lower than that of (12,8,5) Optimal-LRC, and a (16,10,6) Azure-LRC+1 whose NRC is lower than that of (16,10,6) Optimal-LRC. In both cases, Optimal-LRC can achieve a lower NRC with a smaller r, possibly at the cost of reducing d.

NRC vs. d. Our results demonstrate a subtle tradeoff between repair cost (NRC) and d. Codes with the same (n,k,r) may or may not have the same d, and are thus not directly comparable: one may satisfy fault tolerance requirements that the other does not. To facilitate a more systematic comparison, we defined another composite metric, *repair-distance ratio (rd-ratio)*, $\frac{NRC}{d}$. This can be viewed as a measure of the efficiency with which a code allocates its local parities, with the conflicting objectives of maximizing d and minimizing *NRC*.

Figure 11 shows the *rd*-ratio of all LRCs. It shows that the code with the lowest *rd*-ratio is different for different (n,k,r) combinations, and is not necessarily a full-LRC. For example, when (n,k,r) is (14,10,5), Azure-LRC has the lowest *rd*-ratio. Another interesting observation is that when fixing n and k, different codes achieve their minimal *rd*-ratio with different values of r. For example, the *rd*-ratio of (17,12,5) Optimal-LRC is lower than that of (17,12,4) Optimal LRC. When we fix (n,k) and consider the "best" r for each code, we observe that Optimal-LRC achieves the lowest *rd*-ratio. This demonstrates that this code is optimal in its allocation of local parity blocks—it efficiently reduces the repair cost with minimal reduction in d. The *rd*-ratio can be generalized to reflect different weights of NRC and d, e.g., by defining it as $\frac{NRC}{d^x}$.

Target fault tolerance. The required fault tolerance in a distributed storage system is determined by many factors, including the number of nodes, their organization into racks and clusters, and the anticipated causes of failure. Nevertheless, once the required level of fault tolerance is determined, the goal is to select a code which will provide this level at the lowest cost. In this context,

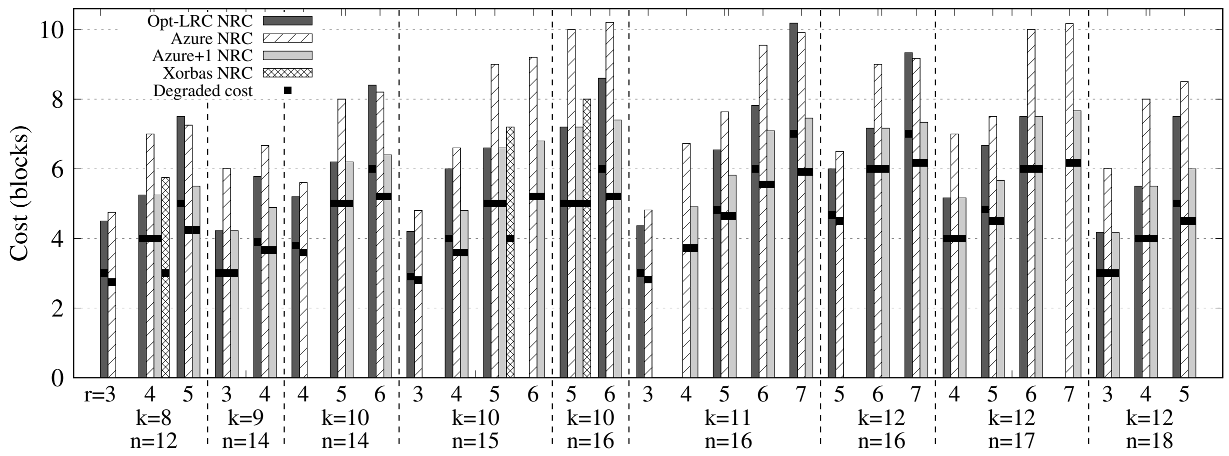

Figure 7: NRC and the degraded cost for all the codes in our evaluation. The repair cost of the full-LRCs is always lower than that of Azure-LRC.

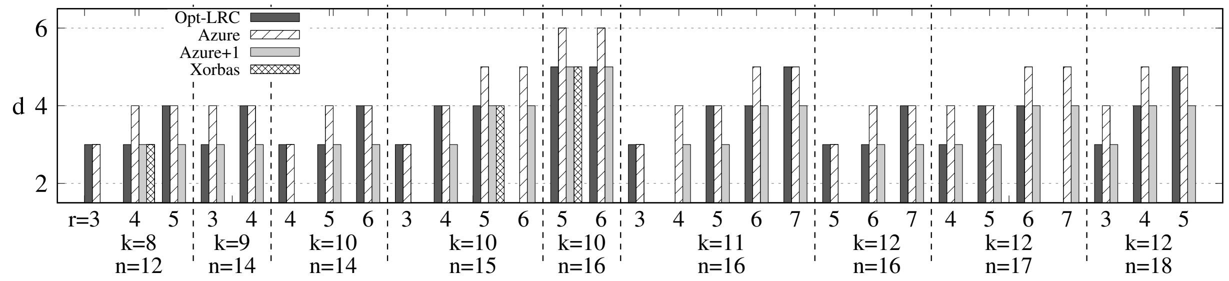

Figure 8: d for all the codes in our evaluation.

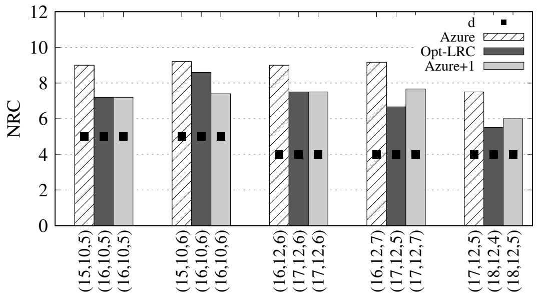

Figure 9: NRC for (n,k,r) Azure-LRC and $(n+1,k,r)$ Azure-LRC+1 and Optimal-LRC. Adding a local parity always reduces repair cost, despite the increase in overhead.

the code with the lowest rd-ratio may not be the optimal choice—a different code may have a higher ratio but provide the required level of fault tolerance at a lower overhead or repair cost.

We defined a threshold value of d, d_{th}, and compared the NRC of all the codes for which $d \geq d_{th}$. We considered $d_{th} \in \{3,4,5\}$, corresponding to the minimum distance of commonly deployed configurations. Figure 12 shows the NRC of all the LRCs whose $d \geq 4$. Many constructions do not provide the required fault tolerance, and are thus absent from this figure. Different codes achieved the lowest NRC for different k,n combinations. However, we note that a construction of Azure-LRC and Optimal-LRC with the required d was defined for every k,n combination. This demonstrates the flexibility of both codes. We observed similar results when setting the threshold d_{th} to 3 or 5, where increasing the threshold removed more codes from the comparison, and vice versa.

Our theoretical evaluation results demonstrate the challenges in comparing different LRC codes and approaches. Our metrics, NRC, degraded cost, and rd-ratio, provide a framework for directly comparing all codes in all parameter combinations. Our comparison demonstrates the benefit of full-LRCs, the flexibility of Optimal-LRC, and the realistic settings in which they may reduce the amount of data read and thus the system repair cost. In the following, we extend our notion of 'repair cost' to additional performance measures.

5 System-Level Evaluation Setup

The goal of our system-level evaluation was threefold: to validate the accuracy of NRC when predicting the amount of data read for node reconstruction, to evaluate its ability to estimate repair time and bandwidth, and to compare the recovery efficiency of the different LRCs in a real system. We omitted the minimum distance, d, from this part of our analysis, because it is not measured empirically. We focused on four representative (n,k) combinations, and compared Reed-Solomon codes, Azure-LRC, Azure-LRC+1, and Optimal-LRC in these setups. We excluded Xorbas from this part of our analysis due to design limi-

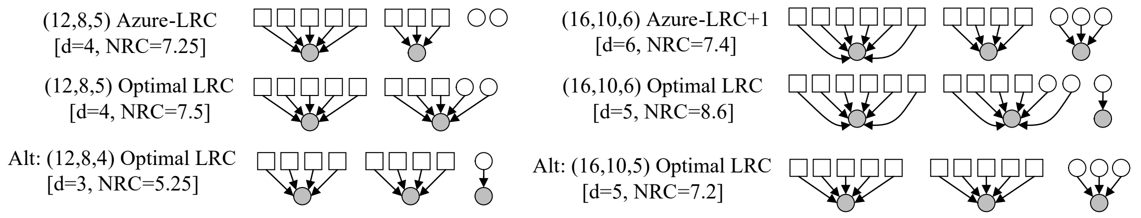

Figure 10: Examples where (n,k,r) Optimal-LRC does not achieve the lowest NRC. In both cases, an alternative $(n,k,r-1)$ Optimal-LRC achieves a lower NRC, possibly at the cost of reducing d.

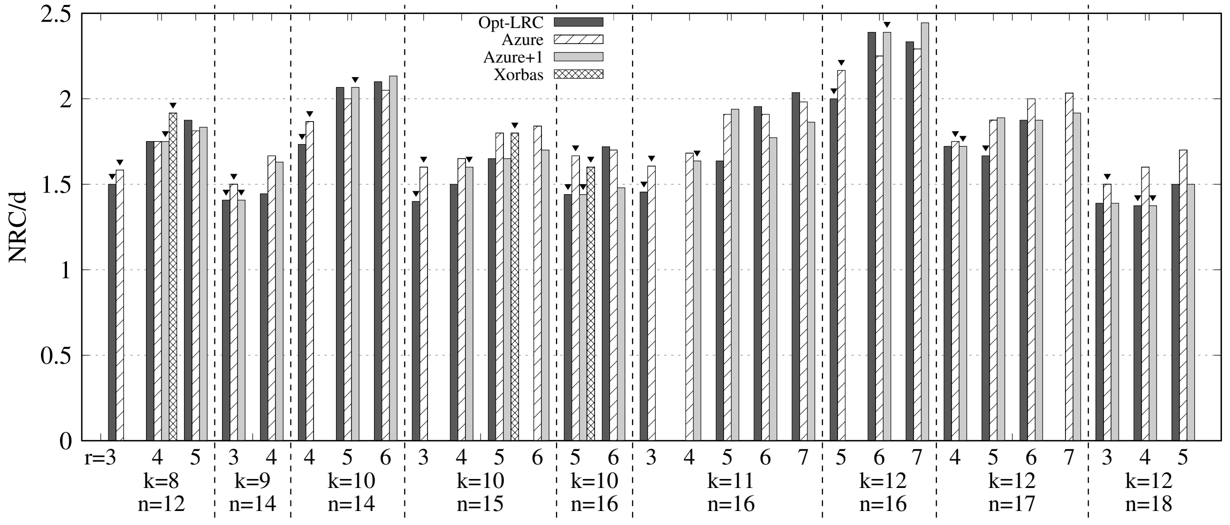

Figure 11: Repair-distance ratio ($\frac{NRC}{d}$). For each (n,k), different codes achieve their minimal rd-ratio (marked by the small triangle) with different values of r.

tations described below.

We performed our evaluation in Ceph—a distributed open-source storage system [34]. Ceph's object storage service, RADOS [36], is responsible for object placement, failure detection and failure recovery. Ceph's nodes are called *object storage devices (OSDs)*. Objects in Ceph are assigned to *placement groups*, which define the allocation of blocks to OSDs. The mapping of placement groups to OSDs is implemented by a pseudo-random mechanism, CRUSH, to ensure load balancing [35].

The *primary OSD* in each placement group is responsible for encoding the data and distributing the data and parity blocks to the remaining, *secondary*, OSDs. When one of the OSDs in a placement group fails, a *replacement OSD* is assigned to it. The primary OSD is responsible for reading the required data from the surviving OSDs, reconstructing the missing block, and sending it to the replacement OSD for permanent storage.

Ceph's design imposes certain limitations on our evaluation. When the failed OSD and the primary OSD belong to different *locality groups*, the repair data must be transferred across groups. In complex network topologies, this might incur cross-rack or cross-zone traffic that LRCs were designed to avoid. In addition, degraded reads are currently implemented by reconstructing the entire object at the primary OSD. This means that all k data blocks are read, even if only r blocks are required to repair the missing block. As a result, for degraded reads, there is no observable difference between MDS codes and LRCs.

We chose to use Ceph despite these limitations. As far as we know, it is the only open-source distributed storage system that implements LRCs as part of its main distribution. Furthermore, at the time we began this research, it was the only system to support online erasure coding, without requiring that objects are first replicated and then erasure-coded in the background.

LRC plugin. In Ceph, erasure codes are implemented as plugins. We used the Jerasure Erasure Code plugin [4], which contains an implementation of Reed-Solomon based on the Jerasure [21] and GF-Complete [20] libraries. We used the Locally Repairable Erasure Code plugin (LRC plugin) [5] to implement Azure-LRC and Azure-LRC+1[1]. This plugin first attempts to reconstruct the missing blocks from the surviving blocks in its locality group. If the block does not belong to any group, or if other blocks in the group are unavailable, it will be reconstructed from the global parities.

We made two adjustments in the LRC plugin. First,

[1]Ceph's LRC plugin actually implements Pyramid codes, and not Azure-LRC. However, the data read by these two codes in all single-node failure scenarios is identical. The precise parity calculations and fault tolerance of Azure-LRC are outside the scope of this study.

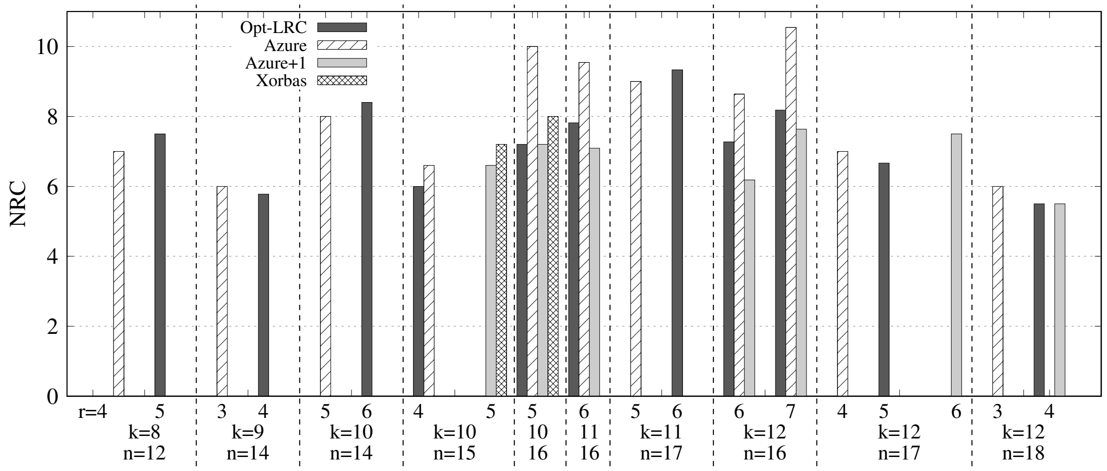

Figure 12: NRC of codes with $d \geq 4$. Azure-LRC and Optimal-LRC are the most flexible codes, defined for all (k,n) combinations.

we prevented CRUSH from rebalancing the placement groups after a node failure. This prevented the primary OSD from reading data that was not strictly required for repair. Second, we adjusted the LRC plugin to read the minimum amount of required blocks for recovery with global parities. The original implementation greedily reads *all* the remaining blocks in the stripe, which artificially increased the repair cost.

Optimal-LRC implementation. We implemented the encoding in Optimal-LRC as a multiplication by a $k \times n$ generator matrix created from the polynomial described in [13]. When creating the matrix, we transformed the k columns corresponding to the data blocks to ensure *systematic encoding* in which the data blocks are not encoded and are stored on the storage nodes in their original form. We further optimized the generator matrix to ensure that the decoding process of local recovery would consist only of XOR operations, avoiding finite field operations. We used Matlab to construct the generator matrix for each (n,k,r) Optimal-LRC in our evaluation.[2]. Beyond these initial calculations, the encoding and decoding processes of Optimal-LRC are equivalent to those of the original Ceph LRC implementation. The differences in encoding and decoding complexities are negligible compared to the I/O and network times of a large-scale storage system [6, 11, 12, 15, 19]. Similarly, there is no significant difference in the overhead of their implementation and metadata storage and maintenance.

Amazon EC2 deployment. We deployed our Ceph cluster on 20 instances in the Amazon Elastic Compute Cloud (EC2). We used `t2.medium` instances, each equipped with two Intel Xeon processors and 4GiB RAM [2]. We allocated two storage volumes to each instance, and used them to initialize two OSDs, resulting in a cluster of 40 OSDs. An additional `t2.2xlarge` instance hosted the monitor, metadata server, and client.

EC2 data centers belong to different *regions*, which correspond to distinct geographical locations. Each region contains several *availability zones*, which are connected by low latency links and guarantee failure tolerance within the region [3]. We deployed our cluster in a single availability zone in the Frankfurt region in all experiments except the multi-zone one. Unless stated otherwise, we use General Purpose SSD as storage devices [1].

In our basic "node repair" experiment, we populated the cluster with 200GB of data, written as 64MB objects. These objects are distributed across 512 placement groups. Thus, each OSD stored, on average, 5GB of data, and additional parity blocks according to the evaluated code. We killed one OSD daemon on one instance and removed this OSD from the cluster. This initialized the repair process, which was performed by the primary OSD in each affected placement group. We recorded the amount of data read from each device and the CPU utilization of each instance, until the full recovery of the cluster. We describe variations of this experiment with foreground workload and with slower storage below.

6 Results

Amount of data read and transferred. Figure 13 shows the number of blocks read by each code during repair, normalized to the number of data blocks on the failed OSD. We also present the ARC and NRC of each code, for comparison. We use an (n,k) Reed-Solomon in each configuration as our baseline. The results show the considerable reduction in repair cost achieved by LRCs, and that full-LRCs achieve a larger reduction, as shown in our theoretical evaluation.

For a given (n,k,r) combination, both ARC and NRC can predict which code will incur the the highest and lowest repair costs. At the same time, they are both inaccurate in their prediction of the actual repair cost. The reason for this inaccuracy is different for each metric. ARC in-

[2]Our implementation of Optimal-LRC in Ceph and its construction in Matlab will be made available as open-source projects.

Code	ARC	Adjusted ARC	NRC	Adjusted NRC	Data read	Time (s)
(13,10) RS	10	10	13	13	11.71	89
(14,10,5) Azure	5.71 (0.57)	5.5 (0.55)	8 (0.62)	7.7 (0.59)	6.74 (0.58)	59 (0.66)
(14,10,6) Azure	5.85 (0.58)	5.6 (0.56)	8.2 (0.63)	7.84 (0.6)	6.86 (0.59)	57 (0.64)
(15,10,5) Azure+1	4.4 (0.44)	4.5 (0.45)	6.6 (0.51)	6.75 (0.52)	5.96 (0.51)	57 (0.64)
(15,10,6) Azure+1	4.53 (0.53)	4.59 (0.46)	6.8 (0.52)	6.88 (0.53)	6.08 (0.52)	57 (0.64)

Table 1: Adjusted ARC and NRC according to block distribution on the failed OSD. The adjusted NRC corresponds to the actual amount of data read for recovery. The values in parenthesis show the costs normalized to Reed-Solomon with the same k and d.

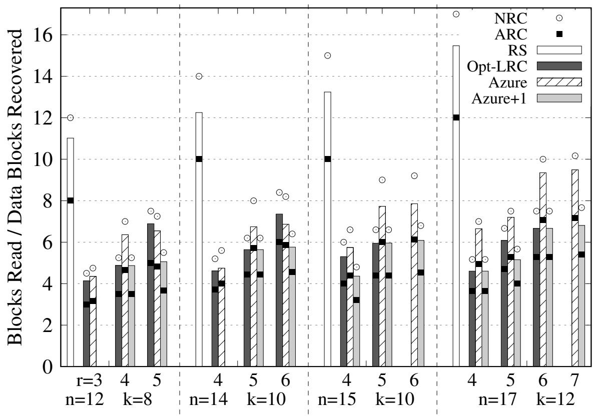

Figure 13: The number of average read blocks per data block repaired, compared to expected ARC and NRC.

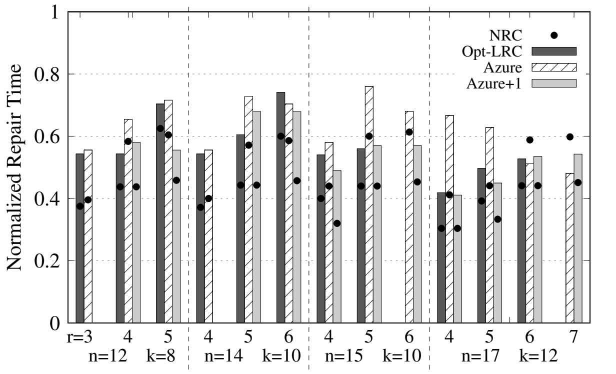

Figure 14: Recovery time of LRCs normalized to Reed-Solomon with the same k and n.

herently underestimates the absolute cost, because it does not take into account the code's overhead. As a result, it is not useful for comparing codes with different storage overheads. For example, the ARC of (14,10) Reed-Solomon and (15,10) Reed-Solomon is 10 for both, but they read 12.53 and 13.24 blocks per data block recovered, respectively.

The inaccuracy of NRC is the result of our limited evaluation setup. Although CRUSH attempts to uniformly distribute data and parity blocks on all OSDs, its mapping is deterministic, and the actual distribution with 40 OSDs is not perfectly uniform. As a result, some OSDs store more blocks than others, and the percentage of data and parity blocks on each OSD is different. We verified that this is the cause of the inaccuracy - by distinguishing between the blocks on the failed OSD according to the number of blocks required for their repair and observing that their percentage is different than expected. We adjusted the NRC and ARC in several setups according to the observed distribution, although we still assumed 5GB of data on each OSD[3].

Table 1 shows the detailed metric and results for Reed-Solomon, Azure-LRC, and Azure-LRC+1, when $k = 10$, and $d = 4$. The required storage overhead is different for each code, which makes it difficult to directly compare their repair costs. The mapping of OSDs to placement groups is also different for each n. The table shows the calculated and the adjusted ARC and NRC of all codes, with the repair cost of each LRC compared to that of Reed-Solomon in parenthesis. The adjusted NRC provides a fairly accurate prediction of the amount of data read for recovery (we discuss the recovery time below). This confirms that in a large-scale storage system with uniform block distribution, the NRC can accurately predict the average repair cost of an entire storage node.

The amount of data transferred between nodes was almost identical to the amount of data read. We verified that the differences were caused by the role of the primary OSD in the reconstruction process: when the primary stored one of the blocks required for reconstruction, it did not have to transfer this block to another OSD. On the other hand, the primary always had to transfer the reconstructed block to the replacement OSD. In light of this simple correlation, we omit the amount of data transferred from the rest of our discussion.

Repair time. Figure 14 shows the recovery time of LRCs normalized to Reed-Solomon with the same k and n (normalizing to Reed-Solomon with the same d yields equivalent results). Our results show that the reduction in the amount of data read for repair does not directly translate to a reduction in repair time. This is the result of additional bottlenecks in the system, such as queuing and batching delays. We verified that the CPU utilization is the same for all codes, ruling out encoding costs as a bottleneck. However, the I/O bandwidth utilized by the codes was slightly different. Reed-Solomon typically achieved a higher throughput than the LRCs—it reads considerably more data than the other codes, which allows it to saturate the storage devices. Thus, the reduction in repair time achieved by the LRCs was smaller than that predicted by NRC. Overall, the full-LRCs achieved the greatest reduc-

[3]Ceph does not report the number of data blocks stored on each OSD, and we could not distinguish between data blocks and local parity blocks because they require the same number of blocks for repair.

Code	NRC	SSD	Opt HDD	Cold HDD
Reed-Solomon	15	100	115	303
Azure-LRC	6.6 (0.44)	58 (0.58)	65 (0.56)	134 (0.44)
Azure-LRC+1	4.8 (0.32)	49 (0.49)	53 (0.46)	134 (0.44)
Optimal-LRC	6 (0.4)	54 (0.54)	57 (0.49)	143 (0.47)

Table 2: NRC of all codes and their recovery time in seconds. $n = 15$ and $k = 10$ for all codes and $r = 4$ for the LRCs, with the repair time normalized to Reed-Solomon in parentheses.

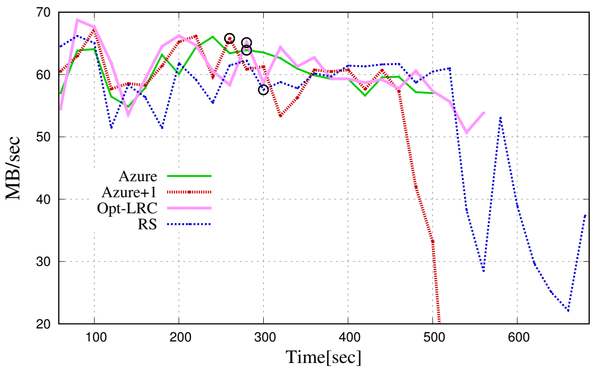

Figure 15: Throughput of RADOS benchmark during repair with LRC in (15,10,4) and RS(15,10).

tion in repair time.

Different storage types. LRCs reduce the amount of data read during recovery, and thus their benefit is expected to increase with the cost of storage I/O. We repeated our repair experiment for one configuration, replacing the SSD storage volumes with two types of hard drives, Optimized HDD and Cold HDD, with a maximum throughput of 500 and 250 IOPS, respectively [1]. The amount of data read from all storage types was the same. Table 2 shows the repair time, in seconds, for all the codes and storage types. As we expected, in the setups where the repair time of Reed-Solomon was longer, the reduction in repair time achieved by all LRCs was higher and closer to the reduction predicted by NRC.

Foreground workloads. Local repair is also designed to minimize the interference with application workloads running in the system at the time of failure. To evaluate this interference, we repeated the repair experiment with the (15,10,4) configuration, in which each LRC has a different NRC. We ran a Ceph benchmark called RADOS Bench [32], which writes objects for a given amount of time (220 seconds in our experiment), reads all the objects, and terminates. For this experiment, we increased the number of outstanding recovery requests allowed per OSD from 15 to 150. We killed one OSD 100 seconds after the benchmark started to read. The repair process took place while the benchmark was still reading the data, but the system recovered before the benchmark terminated.

Figure 15 shows the throughput of the benchmark's I/O requests during its read phase. The black circles mark the time at which recovery was fully completed, and the measurements continue until the benchmark terminates. The differences between the codes were smaller than we expected. This is the result of Ceph's restrictions on repair throughput, and of the high I/O parallelism of SSDs. Nevertheless, the results show that the different codes completed their repair in the order of their NRC: Azure-LRC+1 was the fastest and Reed-Solomon the slowest. The throughput reduction experienced by the benchmark was greatest with Reed-Solomon and smallest with the full-LRCs—Azure-LRC+1 and Optimal-LRC.

Multiple zones. The first LRCs were motivated by the goal of restricting the repair cost to the locality of the failed node. In production systems, this means that blocks in the same group are assigned to a group of nodes on the same rack or in the same zone of the datacenter [11]. To evaluate the different LRCs in a similar environment, we repeated the repair experiment when our instances were deployed on three availability zones in the same EC2 region (N. Virginia). In this experiment, we deployed six instances in each zone, with a total of 18 instances running 36 OSDs. To make up for the reduced I/O bandwidth within in each zone, we replaced the General Purpose SSDs with Provisioned IOPS SSDs, which increased the maximum IOPS per volume from 150 to 2500.

We edited the CRUSH map, instructing CRUSH to assign placement groups to OSDs such that groups are allocated in the same zone [35]. We also ensured that the primary OSD resides in the same zone as the failed OSD. We ran this experiment twice. In the first setup, both the primary OSD and the failed OSD belonged to a data group. In the second setup, both the primary OSD and the failed OSD belonged to a global-parity group.

For full-LRCs, both setups are equivalent: all blocks are reconstructed from blocks in the same group. For Reed-Solomon, all recovery scenarios follow the second setup: recovery requires blocks from different groups. For data-LRCs, data and local parity blocks are recovered according to the first setup, and global parities are recovered according to the second setup. We calculated the weighted average of these two setups to obtain the expected recovery time for each code.

We used (15,8,4) as our configuration, having excluded it from our previous analysis due to its high overhead. Nevertheless, it has the desirable property that all the groups in all codes have the same size (5). This ensures that all placement groups include the same number of OSDs in each zone. In this configuration, the full LRCs are equivalent in their distribution of data and parity blocks to groups. We use Azure-LRC+1 as our full-LRC in this experiment.

For comparison, we repeated this experiment with all OSDs in a single zone, but with the same restriction on the allocation of OSDs to groups. This setup eliminates the cross-zone network bottleneck, with I/O parallelism limited as in the three-zone experiment. Our baseline is

Code	NRC	Reads	1 zone		3 zones
			Baseline	3 groups	
Reed-Solomon	15	14.42	121	179	190
Azure-LRC	10	9.73	88 (0.73)	158 (0.88)	162 (0.85)
Azure-LRC+1	7.5	7.21	80 (0.66)	148 (0.82)	148 (0.78)

Table 3: Number of blocks read per lost data block and repair time when running on one zone and on three zones, with the repair time normalized to Reed-Solomon in parentheses.

the unrestricted setup we used in the rest of this section.

Table 3 shows the amount of data read and the weighted average of the repair time in this experiment. It shows that restricting the number of nodes that participate in the repair process significantly reduces its throughput. When all the OSDs are deployed in the same zone, this restriction increases the repair time by 48% to 85%. The increase is lower for Reed-Solomon because it can still utilize twice as many OSDs than the LRCs. The addition of the cross-zone network bottleneck further reduces the repair time of Reed-Solomon (by 6%) and of Azure-LRC (by 2.5%), but does not affect Azure-LRC+1 which does not incur any cross-zone transfers for repair.

These results demonstrate the well-known tradeoff between I/O parallelism and locality. They confirm that data-LRCs and full-LRCs are expected to achieve the highest benefit in large-scale deployments, where sufficient I/O parallelism can be achieved within a single zone.

7 Related Work

Efforts to reduce the repair cost can be classified into two main lines of research: LRCs, which attempt to reduce the repair cost by reducing the number of nodes participating in the repair process [8, 10, 11, 17, 28], and Regenerating Codes, which strive to attain the same goal by reducing the network bandwidth utilized during repair [6, 24, 25].

The benefits of codes with locality were first realized in [10] before the actual notion was isolated into a standalone concept in the information-theory community [8]. The basic code construction of Pyramid codes [10] assumes that a global parity relation of an MDS code is subdivided into two (or more) local parity check equations which can be used for local repair. Importantly, this work indicated possible savings in repair cost, thereby propelling further research on LRC codes.

In particular, [11] developed LRC codes and observed substantial savings in the repair cost of Microsoft Azure storage attained by using them, and [8] developed the coding-theoretic side of the notion of LRCs. Another relevant work [37] builds on Azure-LRC to dynamically adjust the system's overall storage overhead and average recovery speed by migrating hot and cold data to arrays with more or fewer parity nodes, respectively. Finally, a family of non-MDS codes called Sector Disk codes [15, 19] addresses the recovery of a failed block within an otherwise healthy node. The codes constructed in these works add parity blocks that allow efficient recovery of bad hard-disk sectors or SSD blocks. The above codes were implemented and evaluated independently of one another. Our study is the first to present a comparative framework for codes designed with different properties and overheads, and for different sets of parameters.

Minimum storage regenerating (MSR) codes [6, 25] are a class of MDS codes designed to optimize recovery network bandwidth rather than the number of accessed storage devices. MSR codes and related families, such as RotatedRS [12], Hitchhiker-XOR [23], Butterfly [7] and Zigzag [31] codes, divide each data and parity block into smaller chunks, such that only a subset of each block's chunks are required for the repair of a failed node. Paper [22] constructs an MSR code that reduces the amount of data read from some of the surviving nodes but is applicable only for clusters with $n = 2 \times k$. These codes reduce the *rebuilding ratio*—the portion of the surviving nodes' data that must be read during recovery. All MDS codes with the same d have the same overhead, and can be directly compared by their rebuilding ratio. However, this metric is also limited in its ability to predict recovery costs in a real system: these costs depend on the granularity of the non-sequential I/O accesses incurred when reading arbitrary chunks from each block [18].

Alternative approaches reduce recovery costs of existing codes. An approach introduced in [33] and developed in [9] considers linear repair schemes of Reed-Solomon codes for reducing their network repair bandwidth. *Repair pipelining* improves the utilization of the network bandwidth of all nodes participating in the recovery [16]. *Lazy repair* delays node recovery to amortize its costs over more than one failure [29]. This reduces the fault tolerance of the system, which is equivalent to reducing d. LSTOR relies on attached non-volatile memory for caching additional parity blocks [27], effectively increasing the storage overhead of the system. Our comparative framework, using NRC, can also be extended to evaluate the above approaches.

8 Conclusions

In this study, we performed the first systematic comparison of full-LRCs and data-LRCs. To that end, we implemented a new full-LRC, Azure-LRC+1, and extended and implemented Optimal-LRC for a comparison covering a wide range of system parameters. We demonstrated the limitations of existing metrics and introduced NRC—a new metric that successfully models full-node repair cost. Our theoretical analysis demonstrated the non-trivial correlation between NRC and the cost of degraded reads, and the tradeoff between them and the code's fault tolerance. Our evaluation in a Ceph cluster on Amazon EC2 further showed how this benefit of full-LRCs and data-LRCs depends on the underlying storage devices, network topology, and foreground application load.

References

[1] Amazon EBS volumes. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumes.html, 2017. [Accessed: 2017-09-24].

[2] Amazon EC2 instance types. https://aws.amazon.com/ec2/instance-types, 2017. [Accessed: 2017-09-22].

[3] Amazon EC2 regions and availability zones. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html, 2017. [Accessed: 2017-09-22].

[4] Jerasure erasure code plugin. http://docs.ceph.com/docs/hammer/rados/operations/erasure-code-jerasure/, 2017. [Accessed: 2017-09-24].

[5] Locally repairable erasure code plugin. http://docs.ceph.com/docs/hammer/rados/operations/erasure-code-lrc/, 2017. [Accessed: 2017-09-24].

[6] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, Sept 2010.

[7] E. En-Gad, R. Mateescu, F. Blagojevic, C. Guyot, and Z. Bandic. Repair-optimal MDS array codes over $GF(2)$. In *IEEE International Symposium on Information Theory (ISIT)*, 2013.

[8] P. Gopalan, C. Huang, H. Simitci, and S. Yekhanin. On the locality of codeword symbols. *IEEE Transactions on Information Theory*, 58(11):6925–6934, November 2012.

[9] V. Guruswami and M. Wootters. Repairing reed-solomon codes. In *48th Annual ACM SIGACT Symposium on Theory of Computing (STOC)*, 2016.

[10] C. Huang, M. Chen, and J. Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *Trans. Storage*, 9(1):3:1–3:28, Mar. 2013.

[11] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In *USENIX Annual Technical Conference (ATC)*, 2012.

[12] O. Khan, R. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *10th Usenix Conference on File and Storage Technologies (FAST)*, 2012.

[13] O. Kolosov, A. Barg, I. Tamo, and G. Yadgar. Optimal LRC codes for all lenghts n $\leq$ q. *ArXiv e-prints*, abs/1802.00157, Feb. 2018.

[14] J. Li and X. Tang. Optimal exact repair strategy for the parity nodes of the $(k+2,k)$ zigzag code. *IEEE Transactions on Information Theory*, 62(9):4848–4856, Sept 2016.

[15] M. Li and P. P. C. Lee. STAIR codes: A general family of erasure codes for tolerating device and sector failures. *Trans. Storage*, 10(4):14:1–14:30, Oct. 2014.

[16] R. Li, X. Li, P. P. C. Lee, and Q. Huang. Repair pipelining for erasure-coded storage. In *USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[17] F. Oggier and A. Datta. Self-repairing homomorphic codes for distributed storage systems. In *Proc. 2011 IEEE INFOCOM*, pages 1215–1223, 2011.

[18] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Guyot, E. En-Gad, and Z. Bandic. Opening the chrysalis: On the real repair performance of MSR codes. In *14th Usenix Conference on File and Storage Technologies (FAST)*, 2016.

[19] J. S. Plank and M. Blaum. Sector-disk (SD) erasure codes for mixed failure modes in RAID systems. *Trans. Storage*, 10(1):4:1–4:17, Jan. 2014.

[20] J. S. Plank, K. M. Greenan, and E. L. Miller. Screaming fast Galois field arithmetic using Intel SIMD instructions. In *11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[21] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *7th Usenix Conference on File and Storage Technologies (FAST)*, 2009.

[22] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

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

[24] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to

the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[25] K. V. Rashmi, N. B. Shah, and P. V. Kumar. Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction. *IEEE Transactions on Information Theory*, 57(8):5227–5239, Aug 2011.

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

[27] E. Rosenfeld, N. Amit, and D. Tsafrir. Using disk add-ons to withstand simultaneous disk failures with fewer replicas. *7th Annual Workshop on the Interaction amongst Virtualization, Operating Systems and Computer Architecture (WIVOSCA)*, 2013.

[28] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: novel erasure codes for big data. In *39th International Conference on Very Large Data Bases (VLDB)*, 2013.

[29] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *International Conference on Systems and Storage (SYSTOR)*, 2014.

[30] I. Tamo and A. Barg. A family of optimal locally recoverable codes. *IEEE Transactions on Information Theory*, 60(8):4661–4676, Aug 2014.

[31] I. Tamo, Z. Wang, and J. Bruck. Zigzag codes: MDS array codes with optimal rebuilding. *IEEE Transactions on Information Theory*, 59(3):1597–1616, March 2013.

[32] F. Wang, M. Nelson, S. Oral, S. Atchley, S. Weil, B. W. Settlemyer, B. Caldwell, and J. Hill. Performance and scalability evaluation of the ceph parallel file system. In *Proceedings of the 8th Parallel Data Storage Workshop. ACM*, 2013.

[33] Z. Wang, A. Dimakis, , and J. Bruck. Rebuilding for array codes in distributed storage systems. In *GLOBECOM Workshops (GC Wkshps)*, pages 1905–1909. IEEE, 2010.

[34] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[35] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *ACM/IEEE Conference on Supercomputing (SC)*, 2006.

[36] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. RADOS: A scalable, reliable storage service for petabyte-scale storage clusters. In *2nd International Workshop on Petascale Data Storage (PDSW): Held in Conjunction with Supercomputing*, 2007.

[37] M. Xia, M. Saxena, M. Blaum, and D. A. Pease. A tale of two erasure codes in HDFS. In *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[38] M. Ye and A. Barg. Explicit constructions of MDS array codes and RS codes with optimal repair bandwidth. In *2016 IEEE International Symposium on Information Theory (ISIT)*, 2016.