Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage

Yuchong Hu, Liangfeng Cheng, and Qiaori Yao, *Huazhong University of Science & Technology;* Patrick P. C. Lee, *The Chinese University of Hong Kong;* Weichun Wang and Wei Chen, *HIKVISION*

https://www.usenix.org/conference/fast21/presentation/hu

This paper is included in the Proceedings of the 19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

Open access to the Proceedings of the 19th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage

Yuchong Hu[†], Liangfeng Cheng[†], Qiaori Yao[†], Patrick P. C. Lee[‡], Weichun Wang[*], Wei Chen[*]
[†]*Huazhong University of Science & Technology* [‡]*The Chinese University of Hong Kong* [*]*HIKVISION*

Abstract

Erasure coding is a low-cost redundancy mechanism for distributed storage systems by storing stripes of data and parity chunks. *Wide stripes* are recently proposed to suppress the fraction of parity chunks in a stripe to achieve extreme storage savings. However, wide stripes aggravate the repair penalty, while existing repair-efficient approaches for erasure coding cannot effectively address wide stripes. In this paper, we propose *combined locality*, the first mechanism that systematically addresses the wide-stripe repair problem via the combination of both parity locality and topology locality. We further augment combined locality with efficient encoding and update schemes. Experiments on Amazon EC2 show that combined locality reduces the single-chunk repair time by up to 90.5% compared to locality-based state-of-the-arts, with only a redundancy of as low as 1.063×.

1 Introduction

Erasure coding is an established low-cost redundancy mechanism for protecting data storage against failures in modern distributed storage systems [25, 34, 47]; in particular, Reed-Solomon (RS) codes [61] are widely adopted in today's erasure coding deployment [26, 45, 47, 59, 72]. At a high level, for some configurable parameters n and k (where $k < n$), RS codes compose multiple *stripes* of n chunks, including k original uncoded data chunks and $n-k$ coded parity chunks, such that any k out of n chunks of the same stripe suffice to reconstruct the original k data chunks (see §2.1 for details). Each stripe of n chunks is distributed across n nodes to tolerate any $n-k$ node failures. RS codes incur a *minimum* redundancy of $\frac{n}{k}\times$ (i.e., no other erasure codes can have a lower redundancy than RS codes while tolerating any $n-k$ node failures). In contrast, traditional replication incurs a redundancy of $(n-k+1)\times$ to tolerate the same number of any $n-k$ node failures. For example, Facebook f4 [47] uses $(14,10)$ RS codes to tolerate any four node failures with a redundancy of 1.4×, while replication needs a redundancy of 5× for the same four-node fault tolerance. With proper parameterization of (n,k), erasure coding can limit the redundancy to at most 1.5× (see Table 1).

Conventional wisdom suggests that erasure coding parameters should be configured in a *medium* range [53]. Table 1 lists the parameters (n,k) used by state-of-the-art production systems. We see that the number of tolerable failures $n-k$ is typically three or four, while the stripe size n is no more than 20. One major reason of choosing a moderate stripe size is to limit the *repair penalty* of erasure coding, in which repairing any single lost chunk needs to retrieve multiple available chunks of the same stripe for decoding the lost chunk (e.g., k chunks are retrieved in (n,k) RS codes). A larger stripe size n, and hence a larger k for tolerating the same $n-k$ node failures, implies more severe bandwidth and I/O amplifications in repair and hence compromises storage reliability.

Storage systems	(n,k)	**Redundancy**
Google Colossus [25]	(9,6)	1.50
Quantcast File System [49]	(9,6)	1.50
Hadoop Distributed File System [3]	(9,6)	1.50
Baidu Atlas [36]	(12,8)	1.50
Facebook f4 [47]	(14,10)	1.40
Yahoo Cloud Object Store [48]	(11,8)	1.38
Windows Azure Storage [34]	(16,12)	1.33
Tencent Ultra-Cold Storage [8]	(12,10)	1.20
Pelican [12]	(18,15)	1.20
Backblaze Vaults [13]	(20,17)	1.18

Table 1: Common parameters of (n,k) in state-of-the-art erasure coding deployment. Note that a similar table is also presented in [22], while we add Azure and Pelican here.

While erasure coding effectively mitigates storage redundancy, we explore further redundancy reduction under erasure coding to achieve *extreme* storage savings; for example, a redundancy reduction of 14% (from 1.5× to 1.33×) can translate to millions of dollar savings in production [52]. This motivates us to explore *wide stripes*, in which n and k are very large, while the number of tolerable failures $n-k$ remains three to four as in state-of-the-art production systems. Wide stripes are studied in storage industry (e.g., VAST [9]), and provide an opportunity to achieve *near-optimal redundancy* (i.e., $\frac{n}{k}$ approaches one) with the maximum possible storage savings. For example, VAST [9] considers a setting of $(n,k) = (154, 150)$, thereby incurring only a redundancy of 1.027× . We argue that the significant storage efficiency of wide stripes is attractive for both *cold* and *hot* distributed storage systems. Erasure coding is traditionally used by cold storage systems (e.g., backup and archival applications), in which data needs to be persistently stored but is rarely accessed [2, 10, 12]. Wide stripes allow cold storage systems to achieve long-term data durability at extremely low cost. Erasure coding is also adopted by hot storage systems (e.g.,

in-memory key-value stores) to provide data availability for key-value objects that are frequently accessed in the face of failures and stragglers [18, 57, 73, 74]. Wide stripes allow hot storage systems to significantly reduce expensive hardware footprints (e.g., DRAM for in-memory key-value stores).

While wide stripes achieve extreme storage savings, they further aggravate the repair penalty, as the repair bandwidth (i.e., the amount of data transfers during repair) increases with k. Many existing repair-efficient approaches for erasure-coded storage leverage *locality* to reduce the repair bandwidth. There are two types of locality: (i) *parity locality*, which introduces extra local parity chunks to reduce the number of available chunks to retrieve for repairing a lost chunk [14, 27, 34, 39, 51, 63]; and (ii) *topology locality*, which takes into account the hierarchical nature of the system topology and performs local repair operations to mitigate the cross-rack (or cross-cluster) repair bandwidth [31, 32, 56, 65, 66, 68].

However, existing locality-based repair approaches still mainly focus on stripes with a small k (e.g., $k = 12$ [34] and $k = 6$ [32]). They inevitably increase the redundancy or degrade the repair performance for wide stripes as k increases (§2.3). The reason is that the near-optimal redundancy of wide stripes reduces the benefits brought by either parity locality or topology locality (§3.5).

In this paper, we present *combined locality*, a new repair mechanism that systematically combines both parity locality and topology locality to address the repair problem in wide-stripe erasure coding. Combined locality associates local parity chunks with a small subset of data chunks (i.e., parity locality) and localizes a repair operation in a limited number of racks (i.e., topology locality), so as to provide better trade-offs between redundancy and repair performance than existing locality-based state-of-the-arts. In addition, we revisit the classical encoding and update problems for wide-stripe erasure coding under combined locality and design the corresponding efficient schemes. Our contributions include:

- We are the *first* to systematically address the wide-stripe repair problem. We propose combined locality, which mitigates the cross-rack repair bandwidth under ultra-low storage redundancy. We examine the trade-off between redundancy and cross-rack repair bandwidth for different locality-based schemes (§3).
- We design ECWide, which realizes combined locality to address two types of repair: single-chunk repair and full-node repair. We also design (i) an efficient encoding scheme that allows the parity chunks of a wide stripe to be encoded across multiple nodes in parallel, and (ii) an inner-rack parity update scheme that allows parity chunks to be locally updated within racks to reduce cross-rack transfers (§4).
- We implement two ECWide prototypes, namely ECWide-C and ECWide-H, to realize combined locality. The former is designed for cold storage, while the latter builds on a Memcached-based [5, 6] in-memory key-value store for hot storage (§5). The source code of our prototypes is now available at **https://github.com/yuchonghu/ecwide**.
- We compare via Amazon EC2 experiments ECWide-C and ECWide-H with two existing locality-based schemes: (i) Azure's Local Reconstruction Codes (Azure-LRC) [34] adopted in production, and (ii) the recently proposed topology-locality-based repair approach [32, 65] that minimizes the cross-rack repair bandwidth for fast repair. We show that combined locality significantly reduces the single-chunk repair time by up to 87.9% and 90.5% of the above two schemes, respectively, while incurring a redundancy of as low as $1.063\times$ only. We also validate the efficiency of our encoding and update schemes (§6).

2 Background and Motivation

We provide the background details of erasure coding for distributed storage (§2.1), and state the challenges of deploying wide-stripe erasure coding (§2.2). We describe how existing studies exploit locality to address the repair problem (§2.3), and motivate the idea of our combined locality design (§2.4).

2.1 Erasure Coding for Distributed Storage

Consider a distributed storage system that organizes data in fixed-size *chunks* spanning across a number of storage nodes, such that erasure coding operates in units of chunks. Depending on the types of storage workloads, the chunk size used for erasure coding can vary significantly, ranging from as small as 4 KiB in in-memory key-value storage (i.e., hot storage) [18, 73, 74], to as large as 256 MiB [59] in persistent file storage (i.e., cold storage) for small I/O seek costs. Erasure coding can be constructed in different forms, among which RS codes [61] are the most popular erasure codes and widely deployed (§1).

To deploy RS codes in distributed storage, we configure two integer parameters n and k (where $k < n$). An (n,k) RS code works by encoding k fixed-size (uncoded) *data chunks* into $n-k$ (coded) *parity chunks* of the same size. RS codes are *storage-optimal* (a.k.a. *maximum distance separable (MDS)* in coding theory terms), meaning that any k out of the n chunks suffice to reconstruct all k data chunks (i.e., any $n-k$ lost chunks can be tolerated for data availability), while the redundancy (i.e., $\frac{n}{k}$ times the original data size) is the minimum among all possible erasure code constructions. We call each set of n chunks a *stripe*. A distributed storage system contains multiple stripes that are independently encoded, and the n chunks of each stripe are stored in n different nodes to provide fault tolerance against any $n-k$ node failures.

Mathematically, each parity chunk in an (n,k) RS code is formed by a linear combination of the k data chunks of the same stripe based on the arithmetic of the Galois Field $GF(2^w)$ in w-bit words [53] (where $n \leq 2^w$). Specifically, let $D_1, D_2, \cdots, D_k$ be the k data chunks of a stripe, and $P_1, P_2, \cdots, P_{n-k}$ be the $n-k$ parity chunks of the same stripe. Each parity

chunk P_i ($1 \leq i \leq n-k$) can be expressed as $P_i = \sum_{j=1}^{k} \alpha_{i,j} D_j$, where $\alpha_{i,j}$ denotes some encoding coefficient. In this work, we focus on *Cauchy RS codes* [15, 55], where the encoding coefficients are defined based on the Cauchy matrix, so that we can construct *systematic* RS codes (i.e., the k data chunks are included in a stripe for direct access).

2.2 Challenges of Wide-Stripe Erasure Coding

We explore wide-stripe erasure coding with both large n and k, so as to achieve an ultra-low redundancy $\frac{n}{k}$ (i.e., approaching one). However, it poses three performance challenges.

Expensive repair. Erasure coding is known to incur the repair penalty, and it is even more severe for wide stripes. For an (n,k) RS code, the conventional approach for repairing a single lost chunk is to retrieve k available chunks from other non-failed nodes, implying that the bandwidth and I/O costs are amplified k times. Even though new erasure code constructions can mitigate the repair bandwidth and I/O costs (e.g., regenerating codes [23] or locally repairable codes [27, 33, 34, 51, 63]), the repair bandwidth and I/O amplifications still exist and become more prominent as k increases, as proven by theoretical analysis [23].

The high repair penalty of wide stripes manifests differently in cold and hot storage workloads. For cold storage workloads with large chunk sizes, the repair bandwidth is much more significant for large k. For example, if we configure a wide stripe with $k = 128$ and the chunk size is 256 MiB [59], the single-chunk repair bandwidth becomes 32 GiB. We may interpolate that the daily repair bandwidth of 180 TiB for the $(14, 10)$ RS code [59] will increase to 2.25 PiB for $k = 128$. For hot storage workloads with small chunk sizes, although its single-chunk repair bandwidth is much less than in cold storage, a large k incurs a significant tail latency under frequent accesses, as the repair is now more likely bottlenecked by any straggler node out of the k non-failed nodes.

Expensive encoding. The (per-stripe) encoding overhead of erasure coding becomes more prominent as k increases (the same arguments hold for decoding). In an (n,k) RS code, each parity chunk is a linear combination of k data chunks (§2.1), so the computational overhead increases linearly with k. Most importantly, as k increases, it becomes more difficult for the encoding process to fit the input data of a wide stripe into CPU cache, leading to significant encoding performance degradations. Figure 1 shows the encoding throughput on three Intel CPU families versus k, using the Intel ISA-L encoding APIs [4]. Here, we fix a chunk size of 64 MiB and $n-k = 4$. We see that the encoding throughput remains high from $k = 4$ to $k = 16$, but drops dramatically as k further increases from $k = 32$ onwards; for example, the throughput drops by 43-70% from $k = 4$ to $k = 128$.

Expensive updates. The (per-stripe) update overhead of erasure coding is significant: if any data chunk of the same stripe has been updated, all $n-k$ parity chunks need to be updated.

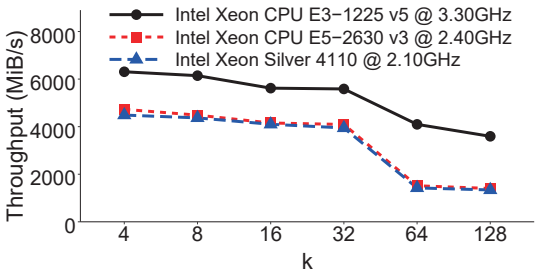


Figure 1: Encoding throughput on different Intel CPU families versus k for a chunk size of 64 MiB and $n-k = 4$.

Wide stripes suffer the same expensive update issue as in traditional stripes of moderate sizes.

2.3 Locality in Erasure-coded Repair

The main challenge of wide-stripe erasure coding is the repair problem. Existing studies on the erasure-coded repair problem have led to a rich body of literature, and many of them focus on using *locality* to reduce the repair bandwidth, including *parity locality* and *topology locality*.

Parity locality. Recall that an (n,k) RS code needs to retrieve k chunks for repairing a lost chunk. Parity locality adds *local parity chunks* to reduce the number of surviving chunks (and hence the repair bandwidth and I/O) for repairing a lost chunk. Its representative erasure code construction is the *locally repairable codes (LRCs)* [27, 33, 34, 51, 63]. Take Azure's Local Reconstruction Codes (Azure-LRC) [34] as an example. Given three configurable parameters n, k, and r (where $r < k < n$), an (n,k,r) Azure-LRC encodes each local group of r data chunks (except the last group, which may have fewer than r data chunks) into a local parity chunk, so that the repair of a lost chunk now only accesses r surviving chunks ($r < k$). It also contains $n-k-\lceil \frac{k}{r} \rceil$ *global parity chunks* encoded from all data chunks. Azure-LRC satisfies the *Maximally Recoverable* property [34] and can tolerate any $n-k-\lceil \frac{k}{r} \rceil + 1$ node failures.

Figure 2(a) shows the $(32, 20, 2)$ Azure-LRC [34]. It has 20 data chunks (denoted by $D_1, D_2, \ldots, D_{20}$). It has 10 local parity chunks, in which the ℓ-th local parity chunk $P_\ell[i\text{-}j]$ (where $1 \leq \ell \leq 10$) is a linear combination of data chunks $D_i, D_{i+1}, \ldots, D_j$. It also has two global parity chunks $Q_1[1\text{-}20]$ and $Q_2[1\text{-}20]$, each of which is a linear combination of all 20 data chunks. All the above 32 chunks are placed in 32 nodes to tolerate any three node failures. Thus, the $(32, 20, 2)$ Azure-LRC has a single-chunk repair bandwidth of two chunks (e.g., repairing D_1 needs to access D_2 and $P_1[1\text{-}2]$), while incurring a redundancy of $1.6\times$. In contrast, the $(23, 20)$ RS code also has 20 data chunks and is tolerable against any three node failures. Its single-chunk repair bandwidth is 20 chunks, yet its redundancy is only $1.15\times$. In short, parity locality *reduces the repair bandwidth but incurs high redundancy*.

Topology locality. Existing erasure-coded storage systems [34, 47, 58, 60, 63] (including Azure-LRC) place each chunk of a stripe in a distinct node residing in a distinct rack. This

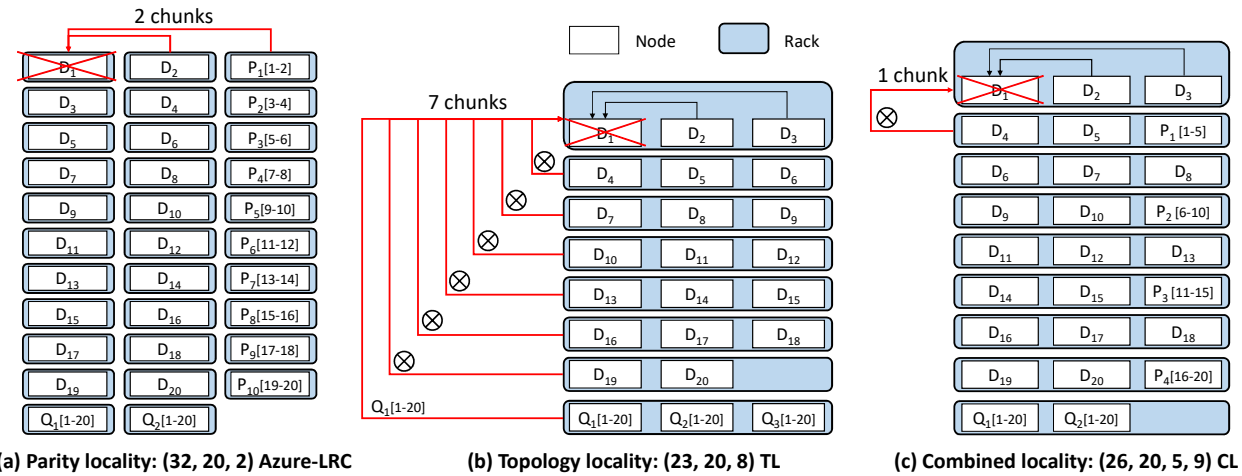


Figure 2: Examples of three locality-based schemes, each of which stores 20 data chunks and can tolerate any three node failures.

provides tolerance against the same numbers of node failures and rack failures, but the repair incurs substantial cross-rack bandwidth, which is often much more constrained than inner-rack bandwidth [20].

Recent studies [31, 32, 65] exploit *topology locality* to reduce the cross-rack repair bandwidth by localizing the repair operations within racks, at the expense of reduced rack-level fault tolerance. They store the chunks of a stripe in multiple nodes within a rack, and split a repair operation into inner-rack and cross-rack repair sub-operations. The cross-rack repair bandwidth is *provably* minimized, subject to the minimum redundancy [31, 32, 65]. Some similar studies focus on minimizing the cross-cluster repair bandwidth via inner-cluster repair sub-operations [56, 66, 68]. We define a topology locality scheme as (n,k,z) TL, in which (n,k) RS-coded chunks are placed in z racks (or clusters).

Figure 2(b) shows the $(23,20,8)$ TL that places 20 data chunks and three RS-coded parity chunks in 23 nodes that reside in eight racks, so as to tolerate any three node failures and one rack failure. The $(23,20,8)$ TL has the minimum redundancy of $1.15\times$, but transfers seven cross-rack chunks to repair a lost chunk. For example, repairing D_1 needs to retrieve $Q_1[1\text{-}20]$ and six chunks that are linear parts of $Q_1[1\text{-}20]$ from other racks, so that D_1 can be solved from $Q_1[1\text{-}20]$ by canceling out the linear parts, D_2, and D_3. The single-chunk repair bandwidth is higher than that of the $(32,20,2)$ Azure-LRC (i.e., two chunks). In short, topology locality *achieves the minimum redundancy, but incurs high cross-rack repair bandwidth*.

2.4 Motivating Example

For wide stripes with a large k, neither parity locality (high redundancy) nor topology locality (high repair penalty) can effectively balance the trade-off between redundancy and repair penalty. This motivates us to combine both types of locality to obtain a better trade-off and hence make wide stripes practically applicable.

Figure 2(c) shows the idea. We encode 20 data chunks into 26 chunks via the $(26,20,5)$ Azure-LRC. We place the chunks across nine racks, and denote the scheme by the $(26,20,5,9)$ CL (see §3.1 for definition). In this case, repairing the lost chunk D_1 can be solved by canceling out D_2, D_3, D_4, and D_5 from $P_1[1\text{-}5]$. The single-chunk repair bandwidth is only one cross-rack chunk, less than both the $(32,20,2)$ Azure-LRC (two chunks) and the $(23,20,8)$ TL (seven chunks). Meanwhile, the redundancy is $1.3\times$, much closer to the minimum redundancy than the $(32,20,2)$ Azure-LRC ($1.6\times$).

3 Combined Locality

In this section, we present *combined locality*, which exploits the combination of parity locality and topology locality to reduce the cross-rack repair bandwidth subject to limited redundancy for wide-stripe erasure coding. We provide definitions and state our design objective (§3.1), and show our design idea of combined locality (§3.2). We analyze and select the suitable LRC construction for combined locality (§3.3). We present the details of the combined locality mechanism (§3.4), and analyze its trade-off between redundancy and cross-rack repair bandwidth (§3.5). Finally, we present reliability analysis on combined locality (§3.6). Table 2 summarizes the notation.

3.1 Design Objective

We define the combined locality mechanism as (n,k,r,z) CL, which combines (n,k,r) Azure-LRC and (n,k,z) TL across z racks (note that we justify our choice of Azure-LRC in §3.3). Our primary objective of the combined locality mechanism is to determine the parameters (n,k,r,z) that minimize the cross-rack repair bandwidth, subject to: (i) the number of tolerable node failures (denoted by f) and (ii) the maximum allowed redundancy (denoted by γ). For wide-stripe erasure coding, we consider a large k (e.g., $k = 128$) for a typical fault tolerance level shown in Table 1 (e.g., $f = 4$).

Notation	Description
n	total number of chunks of a stripe
k	number of data chunks of a stripe
r	number of retrieved chunks to repair a lost chunk
z	number of racks to store a stripe
c	number of chunks of a stripe in a rack
f	number of tolerable node failures of a stripe
γ	maximum allowed redundancy

Table 2: Notation for combined locality.

Here, we ensure that the maximum number of chunks of a stripe residing in each rack (denoted by c) cannot be larger than the number of tolerable node failures f of a stripe; otherwise, a rack failure can lead to data loss. Thus, we require:

$$c \leq f. \tag{1}$$

Each of the first $z-1$ racks stores c chunks of a stripe and the last rack stores the $n-c(z-1)$ ($\leq c$) remaining chunks.

We focus on optimizing two types of repair operations: single-chunk repair and full-node repair (§4.1). Both repair operations assume that each failed stripe has exactly one failed chunk as in most prior studies (§7), including those on parity locality [27, 33, 34, 51, 63] and topology locality [31, 32, 65]. For the failed stripes with multiple failed chunks, we resort to the conventional repair that retrieves k available chunks for reconstructing all failed chunks as in RS codes.

3.2 Design Idea

To achieve the objective of combined locality, we observe from Figure 2 that combined locality repairs a data chunk by downloading $r-1$ data chunks plus one local parity chunk (i.e., the repair bandwidth is r chunks). Since combined locality places some of the r chunks in identical racks, it can apply a local repair to the chunks in each rack, so as to reduce the cross-rack repair bandwidth. Intuitively, if c increases (i.e., more chunks of a stripe can reside in one rack), a local repair can include more chunks, thereby further reducing the cross-rack repair bandwidth. Thus, we aim to find the largest possible c. Recall that $c \leq f$ (Equation (1)). If $c = f$, then the cross-rack repair bandwidth can be minimized.

Thus, the construction of (n,k,r,z) CL is to ensure $c = f$. However, there are different constructions of (n,k,r) LRCs that provide different levels of fault tolerance f [35]. Thus, our idea is to select the appropriate LRC construction that has the highest fault tolerance (§3.3).

3.3 LRC Selection

We consider four representative LRCs discussed in [35].

- **Azure-LRC [34]:** It computes a local parity chunk as a linear combination of r data chunks of each local group, and computes the global parity chunks via RS codes. Note that repairing a global parity chunk needs to retrieve k chunks.

	(n,k,r)	$(16,10,5)$
Azure-LRC [34]	$f = n-k-\lceil k/r \rceil + 1$	$f = 5$
Xorbas [63]	$f \leq n-k-\lceil k/r \rceil + 1$	$f = 4$
Optimal-LRC [69]	$f \leq n-k-\lceil k/r \rceil + 1$	$f = 4$
Azure-LRC+1 [35]	$f = n-k-\lceil k/r \rceil$	$f = 4$

Table 3: Number of tolerable node failures f for different LRCs for $(n,k,r) = (16,10,5)$ [35].

- **Xorbas [63]:** It differs from Azure-LRC in that it allows each global parity chunk to be repairable by at most r chunks, which may include the other global parity chunks and the local parity chunks.
- **Optimal-LRC [69]:** It divides all data chunks and global parity chunks into local groups of size r, and adds a local parity to each local group to allow the repair of any lost chunk by at most r chunks.
- **Azure-LRC+1 [35]:** It builds on Azure-LRC by adding a new local parity chunk for all global parity chunks, allowing the local repair of any lost global parity chunk.

Table 3 shows the number of tolerable node failures f for a practical setting $(n,k,r) = (16,10,5)$ [35]. Note that Xorbas and Optimal-LRC give their upper bounds of f, but in fact the bounds are not attainable for some parameters, including $(n,k,r) = (16,10,5)$ [35]. Table 3 shows that Azure-LRC has the largest f under the same (n,k,r), so it can be the appropriate selection of LRC for combined locality.

The reason why Azure-LRC achieves the highest fault tolerance f is that it neither introduces extra local parity chunks that are linearly dependent on the global parity chunks (e.g., Optimal-LRC and Azure-LRC+1), nor makes the global parity chunks linearly dependent on the local parity chunks (e.g., Xorbas). In fact, for a given level of redundancy, adding linear dependency does not improve fault tolerance.

Note that Azure-LRC needs to download k chunks to repair a global parity chunk, which may be inefficient in repairing a failed node that stores multiple global parity chunks. Nevertheless, we argue that the number of global parity chunk accounts for a small fraction for wide stripes with a large k. For example, in the $(128,120,24)$ Azure-LRC, which contains three global parity chunks, only 3/128 = 2.34% of the chunks stored in each node are global parity chunks. Also, the cross-rack repair bandwidth of a single global parity chunk can be significantly reduced via topology locality. In our following discussion, unless otherwise specified, we focus on a single-chunk repair for a data chunk or a local parity chunk.

3.4 Construction of (n,k,r,z) CL

We provide the construction of (n,k,r,z) CL as follows. Here, we focus on one stripe that has k data chunks with a fixed number of tolerable node failures f subject to the maximum allowed redundancy γ (i.e., $\frac{n}{k} \leq \gamma$). The construction comprises two steps: (i) finding the parameters for (n,k,r) Azure-LRC, and (ii) placing all n chunks across z racks for local repair operations.

Step 1. Given (n,k,r) Azure-LRC, Table 3 states that:

$$n = k + \lceil k/r \rceil + f - 1. \quad (2)$$

Due to $\frac{n}{k} \leq \gamma$, we have:

$$\lceil k/r \rceil \leq k(\gamma - 1) - f + 1. \quad (3)$$

We can obtain the minimum value of r that satisfies Equation (3), denoted by r_{min}. Since r represents the single-chunk repair bandwidth (§3.2), r_{min} refers to the minimum single-chunk repair bandwidth.

Step 2. Based on r_{min}, we proceed to minimize the cross-rack repair bandwidth. First, we can obtain the value of n from Equation (2) and r_{min}. Next, we place these n chunks across n nodes that reside in z racks as follows. For each local group, we put $r+1$ chunks (note that $r = r_{min}$ here), including r data chunks and the corresponding local parity chunk, into $(r+1)/c$ different racks (for the simplicity of discussion, we assume that $r+1$ is divisible by c to have a symmetric distribution of chunks across racks). Thus, for any lost chunk in a rack, we can perform a local repair over the $(r+1)/c$ racks, such that the cross-rack repair bandwidth is $(r+1)/c-1$ chunks collected from the other $(r+1)/c-1$ racks. By setting $c = f$ to minimize the cross-rack repair bandwidth (§3.2), the minimum cross-rack repair bandwidth is $(r+1)/f-1$ chunks.

Figure 2(c) illustrates the $(26,20,5,9)$ CL with $k = 20$ and $f = 3$. Each local group of $r+1 = 6$ chunks (where $r = r_{min} = 5$) is stored in $(r+1)/f = 2$ racks. The cross-rack repair bandwidth is only one chunk (i.e., $(r+1)/f-1 = 1$).

3.5 Trade-off Analysis

Each set of the parameters (n,k,r,z) in combined locality yields the corresponding set of values of redundancy and cross-rack repair bandwidth based on the results in §3.4. We can also derive the values for Azure-LRC and topology locality in terms of k and f. For Azure-LRC, we obtain its redundancy via Equation (2) and cross-rack repair bandwidth as r chunks (assuming each chunk is stored in a distinct rack). For topology locality, we obtain its redundancy subject to $f = n-k$ and cross-rack repair bandwidth as the number of racks minus one (in chunks) (i.e., $\lceil n/f \rceil - 1$) (Figure 2(b)). Table 4 lists the redundancy and cross-rack repair bandwidth for Azure-LRC, topology locality, and combined locality, represented as (n,k,r) Azure-LRC, (n,k,z) TL, and (n,k,r,z) CL, respectively.

Figure 3 plots the results of Table 4 for $k = 128$ and $f = 2,3,4$ subject to the maximum allowed redundancy $\gamma = 1.1$. We set k as a sufficiently large value for wide stripes, and set f as in state-of-the-arts (Table 1). We set γ to close to one to achieve extreme storage savings with wide stripes.

Each point in Figure 3 represents a trade-off between redundancy and cross-rack repair bandwidth for a specific set of parameters. Note that topology locality has three points for the three respective values of f; in contrast, Azure-LRC and combined locality have three curves for the three respective values of f, since they have an additional parameter r that leads to different points along each curve for different values of r. We only plot the points that satisfy $r = r_{min}$ for the minimum cross-rack repair bandwidth.

	Redundancy	**Cross-rack repair bandwidth**
(n,k,r) Azure-LRC	$\frac{k+\lceil k/r \rceil+f-1}{k}$	r
(n,k,z) TL	$\frac{k+f}{k}$	$\lceil (k+f)/f \rceil - 1$
(n,k,r,z) CL	$\frac{k+\lceil k/r \rceil+f-1}{k}$	$(r+1)/f-1$

Table 4: Redundancy and cross-rack repair bandwidth (in chunks) given k and f for (n,k,r) Azure-LRC, (n,k,z) TL, and (n,k,r,z) CL.

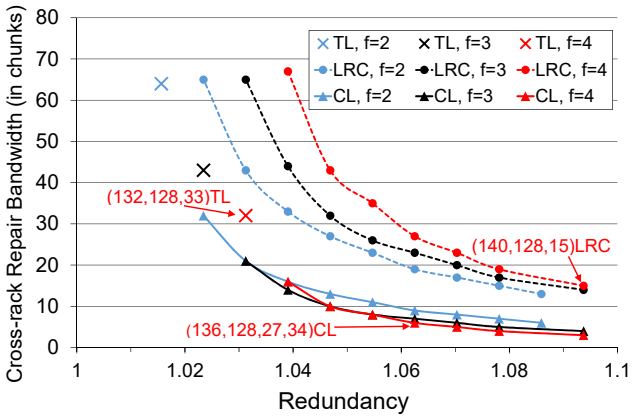


Figure 3: Trade-off between redundancy and cross-rack repair bandwidth for Azure-LRC (LRC), topology locality (TL), and combined locality (CL) for $k = 128$ and $f = 2,3,4$.

Combined locality outperforms both Azure-LRC and topology locality in terms of the trade-off between redundancy and cross-rack repair bandwidth via the combination of both parity locality and topology locality. Take $f = 4$ as an example. For topology locality, the $(132,128,33)$ TL has the minimum redundancy $1.031\times$, yet its cross-rack repair bandwidth reaches 32 chunks, even though many racks perform local repair operations. The $(140,128,15)$ Azure-LRC largely reduces the cross-rack repair bandwidth to $r = 15$ chunks via parity locality, yet its redundancy $(1.094\times)$ is not close to the minimum one. The reason is that Azure-LRC's redundancy is $\propto (1/r)$, while its cross-rack repair bandwidth is $\propto r$ (Table 4), so r should be small for small cross-rack repair bandwidth, at the expense of incurring higher redundancy. In contrast, for combined locality, the $(136,128,27,34)$ CL not only has closer redundancy (i.e., $1.063\times$) to the minimum one, but also further significantly reduces the cross-rack repair bandwidth to at most $(r+1)/f-1 = 6$ chunks (we show a more precise calculation in §3.6), a reduction of 60% compared to Azure-LRC. The reason is that the cross-rack repair bandwidth of combined locality is $\propto (r/f)$ (Table 4), so it has lower cross-rack repair bandwidth under limited redundancy.

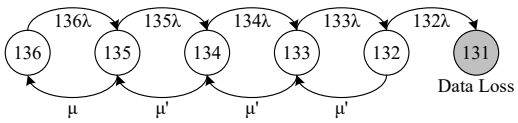


Figure 4: Markov model for (136,128,27,34) CL.

3.6 Reliability Analysis

We analyze the mean-time-to-data-loss (MTTDL) metric via Markov modeling as in prior studies [19,26,32,34,63,67]. We compare six different codes with $f = 4$: (i) $(16,12)$ RS, (ii) $(16,12,6)$ Azure-LRC, (iii) $(132,128)$ RS, (iv) $(132,128,33)$ TL, (v) $(140,128,15)$ Azure-LRC, and (vi) $(136,128,27,34)$ CL. The former two codes are moderate-stripe codes, while the latter four codes are wide-stripe codes.

Figure 4 shows the Markov model for $(136,128,27,34)$ CL; other codes are modeled similarly. Each state represents the number of available nodes of a stripe. For example, State 136 means that all nodes are healthy, while State 131 means data loss. We make two assumptions to simplify our analysis. First, we assume that data loss always occurs whenever there exist five failed nodes, yet in reality some combinations of five failed nodes remain repairable [34] (e.g., the loss of five local parity chunks). Thus, the reliability of $(136,128,27,34)$ CL is an underestimate; we make similar treatments when we model Azure-LRC. Second, we only focus on independent node failures, but do not consider rack failures with multiple nodes failing simultaneously (e.g., a power outage [19]). Our justification is that node failures are much more common than rack failures [47]. We plan to relax the assumptions in our future work.

Our reliability modeling follows the prior work [34]. Let λ be the failure rate of each node. Thus, the state transition rate from State i to State $i-1$ (where $132 \le i \le 136$) is $i\lambda$, since any one of the i nodes in State i fails independently. To model repair, let μ be the repair rate of a failed node from State 135 to State 136, and μ' be the repair rate for each node from State i to State $i+1$ (where $132 \le i \le 134$). We assume that the repair time of a single-node failure is proportional to the amount of repair traffic. Specifically, let N be the total number of nodes in a storage system, S be the capacity of each node, B be the network bandwidth of each node, and ε be the fraction of available network bandwidth of each node for repair due to rate throttling. If a single node fails, the repair load is evenly distributed over the remaining $N-1$ nodes, and the total available network bandwidth for repair is $\varepsilon(N-1)B$. Thus, we have $\mu = \varepsilon(N-1)B/(CS)$, where C is the single-node repair cost (which is derived below). If multiple nodes fail, we set $\mu' = 1/T$, where T denotes the time of detecting multiple node failures and triggering a multi-node repair, based on the assumption that the multi-node repair is prioritized over the single-node repair [34].

We compute C as the average cross-rack repair bandwidth. Take $(136,128,27,34)$ CL as an example. There exist $\lceil \frac{k}{r} \rceil = 5$ local groups, in which the first four local groups (each with $r+1 = 28$ chunks) span seven racks (i.e., the cross-rack repair bandwidth is six chunks), while the last local group (with 21 chunks) spans six racks (i.e., the cross-rack repair bandwidth is five chunks). For the remaining $n-k-\lceil \frac{k}{r} \rceil = 3$ global parity chunks (which reside in one rack), we repair each of them by accessing the other $z-1 = 33$ racks, each of which sends one cross-rack chunk computed from an inner-rack repair sub-operation as in topology locality. Thus, we have $C = (6\times 112 + 5\times 21 + 33\times 3)/136 = 6.44$ chunks.

$1/\lambda$ (years)	2	4	10
(16,12) RS	2.47e+11	7.87e+12	7.66e+14
(16,12,6) Azure-LRC	4.38e+11	1.40e+13	1.36e+15
(132,128) RS	6.33e+05	1.53e+07	1.20e+09
(132,128,33) TL	1.61e+06	4.64e+07	4.24e+09
(140,128,15) Azure-LRC	2.06e+06	6.20e+07	5.82e+09
(136,128,27,34) CL	5.82e+06	1.82e+08	1.75e+10

Table 5: MTTDLs of codes (in years) for varying $1/\lambda$ (years) and $B = 1$ Gb/s.

B (Gb/s)	0.5	1	10
(16,12) RS	3.96e+12	7.87e+12	7.83e+13
(16,12,6) Azure-LRC	7.00e+12	1.40e+13	1.39e+14
(132,128) RS	1.01e+07	1.53e+07	1.09e+08
(132,128,33) TL	2.57e+07	4.64e+07	4.20e+08
(140,128,15) Azure-LRC	3.29e+07	6.20e+07	5.85e+08
(136,128,27,34) CL	9.30e+07	1.82e+08	1.78e+09

Table 6: MTTDLs of codes (in years) for varying B (Gb/s) and $1/\lambda = 4$ years.

We configure the default parameters as follows. We set $N = 400$, $S = 16$ TB, $\varepsilon = 0.1$, and $T = 30$ minutes [34]. We also set the mean-time-to-failure $1/\lambda = 4$ years and $B = 1$ Gb/s [63]. We show the MTTDL results for varying λ (Table 5) and varying B (Table 6).

We see that $(136,128,27,34)$ CL has a lower MTTDL than $(16,12)$ RS and $(16,12,6)$ Azure-LRC with moderate stripes, but achieves a significantly higher MTTDL than other locality-based schemes for wide stripes by minimizing the cross-rack repair bandwidth for a single-node repair. For example, when $B = 1$ Gb/s and $1/\lambda = 4$ years, the MTTDL gain of $(136,128,27,34)$ CL is $10.90\times$ of $(132,128)$ RS, $2.92\times$ of $(132,128,33)$ TL, and $1.94\times$ of $(140,128,15)$ Azure-LRC.

In general, combined locality achieves a higher MTTDL gain when $1/\lambda$ increases or B decreases. The former implies that multiple node failures are less probable, while the latter implies that the cross-rack bandwidth is more constrained. In either case, minimizing the cross-rack repair bandwidth for a single-node repair is critical for a high MTTDL gain.

4 Design

We design ECWide, a wide-stripe erasure-coded storage system that realizes combined locality. ECWide addresses the challenges of achieving efficient repair, encoding, and updates in wide-stripe erasure coding (§2.2), with the following goals:

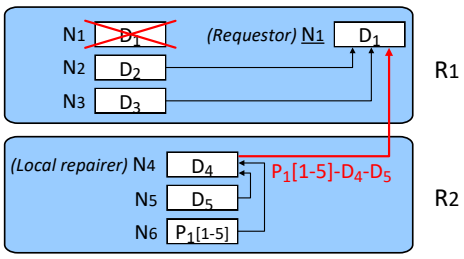


Figure 5: Repair in ECWide.

- **Minimum cross-rack repair bandwidth:** ECWide minimizes the cross-rack repair bandwidth via combined locality (§4.1).
- **Efficient encoding:** ECWide applies *multi-node encoding* that supports efficient encoding for wide stripes (§4.2).
- **Efficient parity updates:** ECWide applies *inner-rack parity updates* that allow both global and local parity chunks to be updated mostly within local racks (§4.3).

4.1 Repair

ECWide realizes combined locality for two types of repair operations: single-chunk repair and full-node repair.

Single-chunk repair. ECWide realizes two steps of combined locality in repair (§3.4). Consider a storage system that organizes data in fixed-size chunks given k, f, and γ. In Step 1, ECWide determines the parameters n and r via Equations (2) and (3). It then encodes k data chunks into $n-k$ local/global parity chunks. In Step 2, ECWide selects $(r+1)/f$ racks for each local group, and places all $r+1$ chunks of each local group into $r+1$ different nodes evenly across these racks (i.e., f chunks per rack). Since the above two steps ensure that the cross-rack repair bandwidth for a single-chunk repair is minimized as $(r+1)/f-1$ chunks (§3.4), ECWide only needs to provide the following details for the repair operation.

Figure 5 describes the repair of a lost chunk D_1 in rack R_1. Specifically, ECWide selects one node $\underline{N_1}$ (called the *requestor*) in R_1 to be responsible for reconstructing the lost chunk. It also selects one node N_4 (called the *local repairer*) in rack R_2 to perform local repair. N_4 then collects all chunks D_5 and $P_1[1\text{-}5]$ within R_2, computes an encoded chunk $P_1[1\text{-}5]-D_4-D_5$ (assuming that $P_1[1\text{-}5]$ is the XOR-sum of $D_1, D_2, \ldots, D_5$ for simplicity), and sends the encoded chunk to the requestor $\underline{N_1}$. Finally, $\underline{N_1}$ collects data chunks D_2 and D_3 within R_1, and solves for D_1 by cancelling out D_2 and D_3 from the received encoded chunk $P_1[1\text{-}5]-D_4-D_5$.

Full-node repair. A full-node repair can be viewed as multiple single-chunk repairs for multiple stripes (i.e., one lost chunk per stripe), which can be parallelized. However, each single-chunk repair involves one requestor and multiple local repairers, so multiple single-chunk repairs may choose identical nodes as requestors or local repairers, thereby overloading the chosen nodes and degrading the overall full-node repair performance. Thus, our goal is to choose as many different nodes to be requestors and local repairers as possible for effective parallelization of multiple single-chunk repairs.

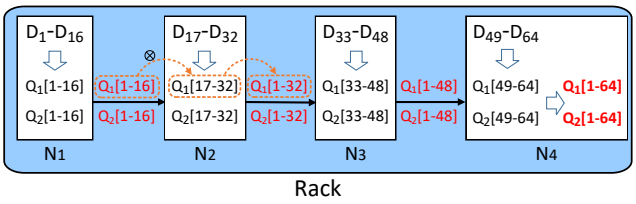


Figure 6: Multi-node encoding in ECWide.

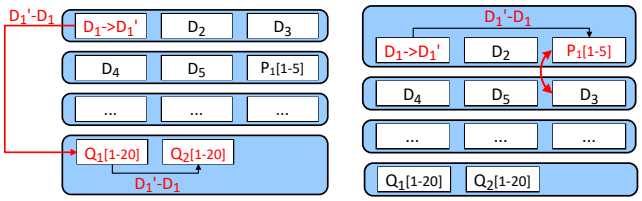
To this end, ECWide designs a least-recently-selected method to select nodes as requestors or local repairers, and implements it via a doubly-linked list and a hashmap. The doubly-linked list holds all node IDs to track which node has been recently selected or otherwise, and the hashmap holds the node ID and the node address of the list. We can then obtain the least-recently-selected node as the requestor or local repairer by simply selecting the bottom one of the list and updating the list via hashmap in O(1) time.

4.2 Encoding

Recall from §2.2 that single-node encoding for wide stripes leads to significant performance degradation for a large k. We observe that the current encoding implementation (e.g., Intel ISA-L [4] and QFS [49]) often splits data chunks of large size (e.g., 64 MiB) into smaller-size data slices and performs slice-based encoding with hardware acceleration (e.g., Intel ISA-L) or parallelism (e.g., QFS). To encode a set of k data slices that are parts of k data chunks, the CPU cache of the encoding node prefetches successive slices from each of the k data chunks. If k is large, the CPU cache may not be able to hold all prefetched slices, thereby degrading the encoding performance of the successive slices.

To overcome the limitation of single-node encoding, we consider a *multi-node encoding* scheme that aims to achieve high encoding throughput for wide-stripes. Its idea is to divide a single-node encoding operation with a large k into multiple encoding sub-operations for a small k across different nodes. It is driven by three observations: (i) the encoding performance of stripes with a small k (e.g., $k=16$) is fast (Figure 1 in §2.2); (ii) the parity chunks are linear combinations of data chunks (§2.1), so a parity chunk can be combined from multiple *partially* encoded chunks of different subsets of k data chunks; and (iii) the bandwidth among the nodes within the same rack is often abundant.

Figure 6 depicts the multi-node encoding scheme with $k=64$, assuming that two global parity chunks $Q_1[1\text{-}64]$ and $Q_2[1\text{-}64]$ are to be generated. ECWide first evenly distributes all 64 data chunks across four nodes N_1, N_2, N_3, and N_4 in the same rack. It lets each node (e.g., N_1) encode its 16 local data chunks (e.g., $D_1, D_2, \ldots, D_{16}$) into two partially encoded chunks (e.g., $Q_1[1\text{-}16]$ and $Q_2[1\text{-}16]$). The first node N_1 sends $Q_1[1\text{-}16]$ and $Q_2[1\text{-}16]$ to its next node N_2. N_2 combines the



(a) Global parity updates (b) Local parity updates

Figure 7: Inner-rack parity updates in ECWide.

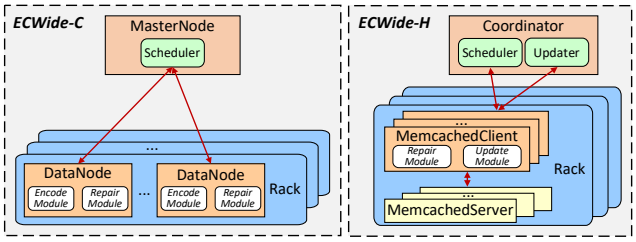


Figure 8: Architecture of ECWide.

two received partially encoded chunks with its local partially encoded chunks $Q_1[17\text{-}32]$ and $Q_2[17\text{-}32]$ to form two new partially encoded chunks $Q_1[1\text{-}32]$ and $Q_2[1\text{-}32]$, which are sent to the next node N_3. Similar operations are performed in N_3 and N_4. Finally, N_4 generates the final global parity chunks $Q_1[1\text{-}64]$ and $Q_2[1\text{-}64]$. Note that the partially encoded chunks are encoded in parallel and forwarded from N_1 to N_4 via fast inner-rack links, so as to efficiently calculating the global parity chunks of wide stripes.

ECWide needs to generate local parity chunks under combined locality, yet the local parity chunks can be more efficiently encoded from r data chunks of each local group in a single node, as r is typically much smaller than k. In addition, ECWide needs to distribute all data chunks, local parity chunks, and global parity chunks to different racks. Such a distribution incurs cross-rack data transfers; minimizing the cross-rack data transfers for the encoding of wide stripes is our future work.

4.3 Updates

To alleviate the expensive parity update overhead in wide stripes (§2.2), we present an *inner-rack parity update* scheme for wide stripes. Its idea is to limit both global and local parity updates within the same rack as much as possible, so as to mitigate cross-rack data transfers.

Figure 7(a) depicts how to perform inner-rack parity updates for two global parity chunks $Q_1[1\text{-}20]$ and $Q_2[1\text{-}20]$ (also shown in Figure 2(c)). ECWide places all the global parity chunks $Q_1[1\text{-}20]$ and $Q_2[1\text{-}20]$ in the same rack, which is always feasible without violating rack-level tolerance given that $c = f$ and the number of global parity chunks is often no more than f. In this case, when a data chunk D_1 is updated to D_1', ECWide first transfers a *delta chunk* $D_1' - D_1$ across racks for the global chunk $Q_1[1\text{-}20]$ (§2.1). It updates $Q_1[1\text{-}20]$ by adding $\alpha(D_1' - D_1)$, where α is the encoding coefficient of D_1 in $Q_1[1\text{-}20]$. ECWide updates the other global parity chunk $Q_2[1\text{-}20]$ by transferring the delta chunk only via inner-rack data transfers. Note that ECWide only incurs one cross-rack transferred chunk for updating all global parity chunks.

Figure 7(b) depicts how to perform inner-rack parity updates for the local parity chunk $P_1[1\text{-}5]$. For each stripe, ECWide first records the update frequency of data chunks of each rack and finds the most update-intensive rack for each local group. If $P_1[1\text{-}5]$ does not reside in the most update-intensive rack of its group, then ECWide swaps $P_1[1\text{-}5]$ for a random data chunk (say D_3) in the most update-intensive rack. In this way, $P_1[1\text{-}5]$ is moved to the rack that is the most update-intensive, so that the local parity updates can mostly be performed within the rack without incurring cross-rack data transfers.

If a data chunk is updated, it is important to ensure that all global and local parity chunks of the same stripe are consistently updated. ECWide may handle consistent parity updates in a state-of-the-art manner, for example, by leveraging a piggybacking method to improve the classical two-phase commits as one-phase commits [74].

5 Implementation

We implement ECWide (§4) in three major modules: a *repair* module that performs the repair operations based on combined locality, an *encode* module that performs multi-node encoding, and an *update* module that performs inner-rack parity updates. We implement two prototypes of ECWide, namely ECWide-C and ECWide-H, for cold and hot storage systems, respectively, as shown in Figure 8.

ECWide-C. ECWide-C is mainly implemented in Java with about 1,500 SLoC, while the encode module is implemented in C++ with about 300 SLoC based on Intel ISA-L [4]. It has a `MasterNode` that stores metadata and organizes the repair and encoding operations with a `Scheduler` daemon, as well as multiple `DataNodes` that store data and perform the repair and multi-node encoding operations. Note that ECWide-C does not consider the update module, assuming that updates are rare in cold storage.

For the repair operation, the `Scheduler` triggers the repair module of each `DataNode` that serves as a local repairer. Such `DataNodes` (which serve as local repairers) send the partially repaired results to the `DataNode` that serves as the requestor, which finally reconstructs the lost chunk. For the encoding operation, the `Scheduler` selects a rack and triggers the encode module of each involved `DataNode` in the rack. Those involved `DataNodes` perform multi-node encoding, and the `DataNode` that serves as the destination node generates all global parity chunks.

ECWide-H. ECWide-H builds on the Memcached in-memory key-value store (v1.4) [6] and libMemcached (v1.0.18) [5] for hot storage. It is implemented in C with about 3,000 SLoC. It follows a client-server architecture. It contains

MemcachedServers that store key-value items, as well as MemcachedClients that perform the repair and parity update operations. It also includes the Coordinator for managing metadata. The Coordinator includes a Scheduler daemon that coordinates the repair and parity update operations and an Updater daemon that analyzes the update frequency status. Note that ECWide-H does not include the encode module as in ECWide-C, since the chunk size in erasure-coded in-memory key-value stores is often small (e.g., 4 KiB [18, 73, 74]) and a single-node CPU cache is large enough to prefetch all data chunks of a wide stripe for high encoding performance (§4.2).

For the repair operation, ECWide-H performs the same way as ECWide-C, except that it uses MemcachedClients as local repairers. For the updates of global parity chunks, the Scheduler locates the rack where the global parity chunks reside, and triggers the update modules of MemcachedClients to perform the inner-rack parity updates. For the updates of local parity chunks, the Updater first triggers the swapping, in which the two involved MemcachedClients exchange the corresponding chunks. The inner-rack parity updates for the local parity chunks can be later performed. Note that some existing in-memory systems (e.g., Cocytus [74]) also deploy multiple Memcached instances in a single physical node and have a form of hierarchical topology that is suitable for topology locality.

6 Evaluation

We conduct our experiments on Amazon EC2 [1] with a number of m5.xlarge instances connected by a 10 Gb/s network. One instance represents a MasterNode for ECWide-C or a Coordinator for ECWide-H (§5), while the other instances represent the DataNodes for ECWide-C or the MemcachedClients/MemcachedServers for ECWide-H. To simulate the heterogeneous bandwidth within a rack and across racks, we partition nodes into logical racks and assign one dedicated instance as a *gateway* in each rack. The instances within the same logical rack can communicate directly via the 10 Gb/s network, while the instances in different racks communicate via the gateways. We use the Linux traffic control command tc [7] to limit the outgoing bandwidth of each gateway to make cross-rack bandwidth constrained. In our experiments, we vary the gateway bandwidth from 500 Mb/s up to 10 Gb/s.

We set the chunk size as 64 MiB for ECWide-C and 4 KiB for ECWide-H (§2.2). We plot the average results of each experiment over ten runs. We also plot the error bars for the minimum and maximum results over the ten runs. Note that the error bars may be invisible in some plots due to the small variance.

We present the experimental results of ECWide-C and ECWide-H for combined locality (CL), compared with Azure-LRC (LRC) and topology locality (TL) that represent state-of-the-art locality-based schemes. We show that CL outperforms LRC and TL for both single-chunk repair and full-node repair. We also show the efficiency of our multi-node encoding and inner-rack parity update schemes.

6.1 ECWide-C Performance

Experiment A.1 (Repair). We evaluate the repair performance of LRC, TL, and CL using ECWide-C. Here, we let $32 \leq k \leq 64$ and $2 \leq f \leq 4$, and configure different gateway bandwidth settings. For (n,k,r,z) CL, we deploy $n+1$ instances, including n instances as DataNodes and one instance as MasterNode. We select two types of LRC and two types of CL for each set of f and k with different r. We also compute the corresponding redundancy of each scheme based on Table 4. Given k, f, and r, we can compute $n = k + \lceil \frac{k}{r} \rceil + f - 1$ and $z = \lceil \frac{n}{f} \rceil$. Thus, in the following discussion, we only show the values of k, f, and r.

Figures 9(a)-9(e) show the average single-chunk repair times of LRC, TL, and CL for different values of k and f, under the gateway bandwidth of 1 Gb/s and 500 Mb/s. CL always outperforms LRC and TL under the same k, f, and the gateway bandwidth, while TL with the minimum redundancy often performs the worst. For example, in Figure 9(c), when the gateway bandwidth is 1 Gb/s, the single-chunk repair time of CL with $r = 7$ is 0.8 s, while those of LRC with $r = 7$ and TL are 3.9 s and 9.0 s, respectively; equivalently, CL reduces the single-chunk repair times of LRC and TL by 79.5% and 91.1%, respectively.

CL shows a higher gain compared to LRC under smaller gateway bandwidth. For example, in Figure 9(c), when the gateway bandwidth is 500 Mb/s, the gain of CL over LRC is 82.1%, which is higher than 79.5% when the gateway bandwidth is 1 Gb/s. The reason is that CL minimizes the cross-rack repair bandwidth, so its performance gain is more obvious when the gateway bandwidth is more constrained.

Also, the single-chunk repair time of CL increases when only r increases (see $r = 7$ and $r = 11$ in Figure 9(c)), and keeps stable when only k changes (see Figure 9(a)-9(c)). The empirical results are consistent with the theoretical results in Table 4, as the single-chunk cross-rack repair bandwidth is equal to $(r+1)/f - 1$.

Figure 9(f) shows the average full-node repair rates of LRC, TL, and CL for different values of f; we also compare CL with and without the least-recently-selected (LRS) method (§4.1). We fix $k = 64$, $r = 11$, and the gateway bandwidth as 1 Gb/s. To mimic a single node failure, we erase 64 chunks from 64 stripes (i.e., one chunk per stripe) in one node. We then repair all the erased chunks simultaneously. Note that practical storage systems often store many more chunks per node, yet each chunk of the failed node is independently associated with one stripe. Thus, we expect that using 64 chunks sufficiently provides stable performance. From the figure, we see that CL shows a higher full-node repair rate than TL and LRC. Its full-node repair rate increases with f, as the single-chunk cross-rack repair bandwidth is equal to $(r+1)/f - 1$. Also, CL

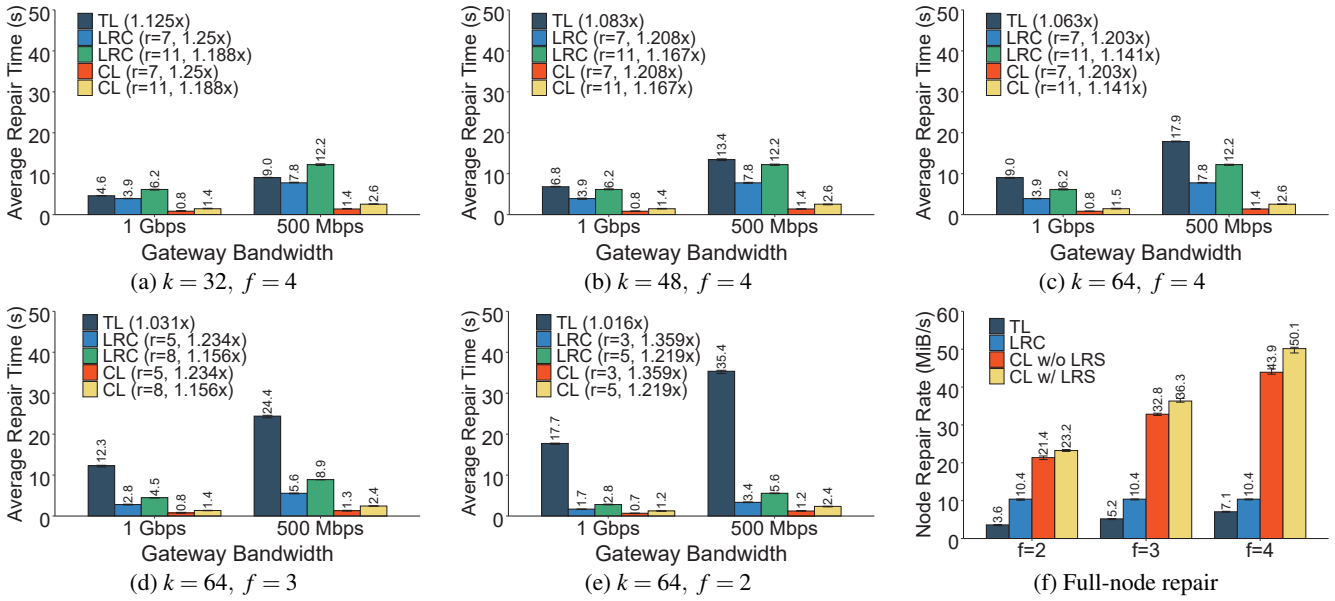


Figure 9: Experiment A.1: Average single-chunk repair time (in seconds) for different k and f under the gateway bandwidth of 1 Gb/s and 500 Mb/s (figures (a)-(e)), and average full-node repair rate for different f (figure (f)).

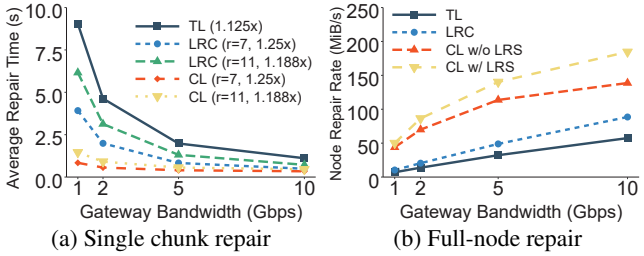


Figure 10: Experiment A.1: Single-chunk repair time and full-node repair rate for different gateway bandwidth.

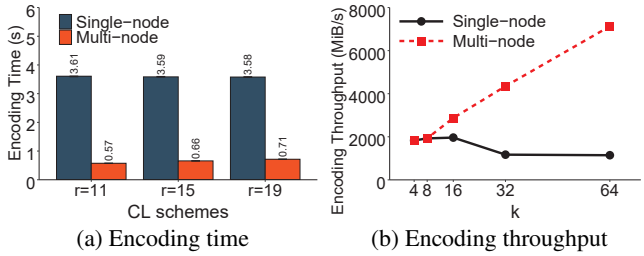


Figure 11: Experiment A.2: Encoding time and encoding throughput for single-node encoding and multi-node encoding.

with LRS increases the full-node repair rate by 14.1% when $f = 4$ compared to CL without LRS, thereby demonstrating the efficiency of the LRS method.

Finally, Figure 10 shows how the average single-chunk repair time and the average full-node repair rate vary with the gateway bandwidth, ranging from 1 Gb/s to 10 Gb/s. Here, we fix $k = 64$ and $f = 4$. From Figure 10(a), CL still outperforms LRC and TL in single-chunk repair under all gateway bandwidth settings, although the difference becomes smaller as the gateway bandwidth increases. For example, when the gateway bandwidth is 10 Gb/s, the single-chunk repair time of CL with $r = 7$ (0.34 s) reduces those of LRC with $r = 7$ (0.49s) and TL (1.11s) by 30.6% and 69.4%, respectively. Also, from Figure 10(b), CL maintains its performance gain in full-node repair over LRC and TL, and LRS brings further improvements.

One limitation of our current implementation is that the full-node repair performance is not fully optimized. We can further improve the throughput by state-of-the-art repair parallelization techniques, such as parity declustering [30], PPR [46], and repair pipelining [41]. Nevertheless, all coding schemes are fairly evaluated under the same implementation setting. If we use parallelization techniques for all coding schemes, we expect that CL should maintain its performance gain by reducing the cross-rack repair bandwidth and I/O. We pose this issue as future work.

Experiment A.2 (Encoding). We measure the average encoding time of CL per stripe. Here, we fix $k = 64$ and $f = 4$, and let $11 \le r \le 19$. Figure 11(a) shows the results of single-node encoding and multi-node encoding. We see that multi-node encoding shows significantly lower encoding time than single-node encoding. For example, when $r = 11$, multi-node encoding reduces 84% of the encoding time compared to single-node encoding.

We further measure the average encoding throughput. Here, we fix $4 \le k \le 64$ and $f = 4$. Figure 11(b) shows the results of single-node encoding and multi-node encoding. Multi-node encoding achieves significantly high encoding throughput when k is large, since many nodes in the same rack can share their computational resources to accelerate the encoding operation. On the other hand, single-node encoding has low throughput when k is large, consistent with our findings in

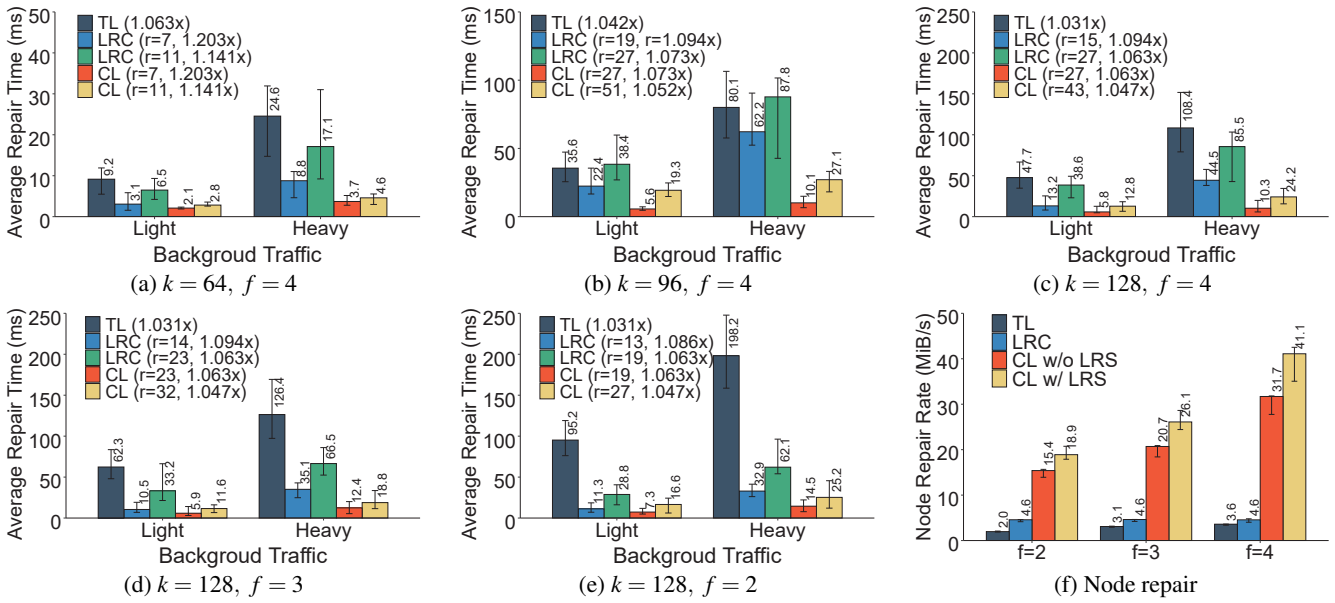


Figure 12: Experiment B.1: Average single-chunk repair time (in milliseconds) for different k and f under different types of background traffic (figures (a)-(e)), and average full-node repair rate for different f.

Figure 1. Note that Figure 1 shows higher throughput than Figure 11(b), since the former only considers the computation part of the encoding operations, while the latter also includes disk I/O for reading chunks for encoding in addition to encoding computation.

6.2 ECWide-H Performance

Experiment B.1 (Repair). We evaluate the repair performance of LRC, TL, and CL using ECWide-H. For (n,k,z) TL and (n,k,r,z) CL, we deploy $n+8z+1$ instances, including n instances as `MemcachedServers`, $8z$ instances as `MemcachedClients` (with eight instances in each of the z racks), and one instance as `Coordinator`.

We consider two deployment scenarios of ECWide-H with different loads of background traffic. To mimic background traffic, in addition to the existing `MemcachedClients` in ECWide-H, we add extra Memcached clients in the background (called *background clients*) that continuously issue read requests to `MemcachedServers`. Specifically, we consider a case of *light* background traffic where each `MemcachedServer` serves 20 background clients, and a case of *heavy* background traffic where each `MemcachedServer` serves 80 background clients.

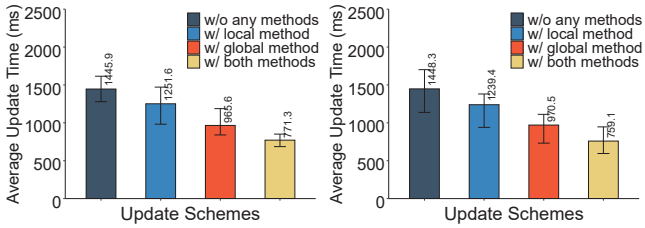
Figures 12(a)-12(e) show the average single-chunk repair times of LRC, TL, and CL for different k and f under the light and heavy background traffic loads. Here, we let $64 \le k \le 128$ and $2 \le f \le 4$. As in Experiment A.1 (§6.1), we select two types of LRC and two types of CL for each set of k and f with different r. We also compute the corresponding redundancy of each scheme based on Table 4. Similar to Experiment A.1, we see that CL still performs the best in hot storage workloads. For example, in Figure 12(c) under heavy background traffic, the single-chunk repair time of CL with $r = 27$ is 10.3 ms, while those of LRC with $r = 27$ and TL are 85.5 ms and 108.4 ms, respectively; equivalently, CL reduces the single-chunk repair times of LRC and TL by 87.9% and 90.5%, respectively. Also, CL with $r = 27$ only incurs a redundancy of $1.063\times$, close to the minimum redundancy of TL ($1.031\times$).

In addition, CL under heavy background traffic shows a higher performance gain compared to the light one, similar to Experiment A.1 that compares the gateway bandwidth of 500 Mb/s to that of 1 Gb/s. The reason is that the single-chunk repair performance is more likely bottlenecked by the limited available bandwidth under heavy background traffic, in which the performance gain of CL is more prominent.

Figure 12(f) shows the average full-node repair rate for different values of f. Here, we fix $k = 64$ and $r = 7$, and focus on light background traffic. From the figure, CL shows a higher full-node repair rate than LRC and TL, and CL with LRS increases the full-node repair rate by 29.7% when $f = 4$ compared to CL without LRS. Also, the full-node repair rate of CL increases with f, consistent with the results in Figure 9(f) (see Experiment A.1 in §6.1) for the same reason.

Experiment B.2 (Updates). We evaluate the update time of a chunk with and without the inner-rack parity updates for global and local parity chunks using (136,128,27,34) CL (§4.3); without the inner-rack parity updates, we assume that each parity chunk is updated directly by the corresponding delta chunk. We use workloads generated by Yahoo! Cloud Serving Benchmark (YCSB) [21] with two read-to-update ratios, namely read-mostly (95%:5%) and update-intensive (50%:50%).

Figure 13 shows the average update times of different update schemes. Compared to without inner-rack parity updates,



(a) read:update = 95%:5% (b) read:update = 50%:50%

Figure 13: Experiment B.2: Average update time (in milliseconds) with inner-rack parity update methods for local and global parity chunks under different read/update ratios, using (136,128,27,34) CL.

the inner-rack parity updates for global parity chunks reduce the update time by up to 33.2%, the inner-rack parity updates for local parity chunks reduce the update time by up to 14.4%, and the inner-rack parity updates for both local and global parity chunks reduce the update time by up to 47.6%.

7 Related Work

Wide-stripe erasure coding. Wide stripes have been commercially adopted (e.g., VAST [9]), but little is known about the design details, and there is no rigorous analysis on the fundamental properties of wide-stripe erasure coding in real deployment. Some studies in the literature address the wide-stripe problem from different perspectives. Li et al. [42] address the read-retry problem in hard disks using local erasure coding, where $n = 1024$ and $n - k \leq 20$. Haddock et al. [28] use general-purpose GPUs to improve encoding/decoding efficiency, where $n = 24$ and $k = 20$. Some studies consider large stripes, where both k and $n - k$ are large, for distributed storage using low-density parity-check (LDPC) codes [54] or rateless codes [44], but the minimum redundancy that they consider (e.g., 1.167× [44] and 1.5× [54]) remains higher than that in our wide-stripe problem. Our work is the first to systematically study the performance issues in wide-stripe erasure coding, including repair, encoding, and updates.

Erasure coding in distributed storage. Erasure coding has been widely studied in distributed storage (see surveys [11, 52]). As erasure coding has higher performance overhead than replication, it is often used in cold storage that treats data persistence as a first-class citizen as opposed to access performance [10, 12]. To ensure data reliability, fast repair is critical for erasure coding in cold storage. Most studies address the repair issue via either proposing new erasure codes that minimize the repair bandwidth [23, 34, 50, 58, 60, 63, 71], or designing repair-efficient techniques that mitigate the repair time [41, 46]. Erasure coding is also considered in hot storage that requires high data access performance. One notable example is erasure coding in in-memory key-value storage, in which existing studies mainly address caching [57], data management [43, 73, 74], and consistent hashing [18, 70]. Some studies focus on updates in erasure-coded storage, and mainly address performance [16, 37] and consistency [17]. Existing studies on erasure coding mainly focus on small k and m. In contrast, we focus on the application of wide stripes in both cold and hot storage.

Locality in erasure coding. Many studies exploit either parity locality or topology locality to improve the performance of erasure coding. In terms of parity locality, *locally repairable codes* [27, 33, 34, 51, 63] reduce the repair bandwidth and I/O costs by associating local parity chunks with different groups of fewer than k data chunks. *Product codes* [24, 29, 40] associate local parities with both horizontal and vertical groups of data chunks for high fault tolerance. Several studies exploit *hierarchical parity locality* to associate local parity chunks with different levels of groups of data chunks to handle multiple failures [38, 62]. In terms of topology locality, existing studies exploit rack-level locality to reduce cross-rack data transfers in repair or update operations. Some studies propose repair-optimal erasure code constructions [31,32,56] that minimize the cross-rack repair bandwidth, while the others design new techniques for efficient repair [65, 66] or updates [64]. Our work combines both parity locality and topology locality to solve the wide-stripe problem, especially on reducing the repair bandwidth for wide stripes.

8 Conclusions

Wide stripes are a new notion for erasure-coded distributed storage to achieve extreme storage savings. We propose combined locality, a novel repair mechanism that combines parity locality and topology locality to address the repair problem effectively for wide stripes. We design ECWide, a prototype system that realizes combined locality. We further design multi-node encoding and inner-rack parity updates to improve the encoding and update performance, respectively. We implement ECWide for both cold and hot storage systems, and our Amazon EC2 experiments demonstrate the efficiency of ECWide in repair, encoding, and updates.

Acknowledgement

We thank our shepherd, Cheng Huang, and the anonymous reviewers for their comments. This work was supported by National Natural Science Foundation of China (61872414), Key Laboratory of Information Storage System Ministry of Education of China, and the Research Grants Council of Hong Kong (AoE/P-404/18). The corresponding author is Yuchong Hu.

References

[1] Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`, Retrieved in Jan 2021.

[2] Amazon S3 Glacier & S3 Glacier Deep Archive. `https://aws.amazon.com/glacier/`, Retrieved in Jan 2021.

[3] HDFS erasure coding. `https://hadoop.apache.org/docs/current/hadoop-project-`

dist/hadoop-hdfs/HDFSErasureCoding.html, Retrieved in Jan 2021.

[4] Intel ISA-L. https://github.com/intel/isa-l, Retrieved in Jan 2021.

[5] libMemcached. https://libmemcached.org/libMemcached.html, Retrieved in Jan 2021.

[6] Memcached. https://memcached.org, Retrieved in Jan 2021.

[7] tc. https://linux.die.net/man/8/tc, Retrieved in Jan 2021.

[8] Tencent ultra-cold storage system optimization with Intel ISA-L - a case study. https://software.intel.com/content/www/us/en/develop/articles/tencent-ultra-cold-storage-system-optimization-with-intel-isa-l-a-case-study.html, Retrieved in Jan 2021.

[9] VastData. https://vastdata.com/providing-resilience-efficiently-part-ii/, Retrieved in Jan 2021.

[10] F. André, A.-M. Kermarrec, E. Le Merrer, N. Le Scouarnec, G. Straub, and A. Van Kempen. Archiving cold data in warehouses with clustered network coding. In *Proc. of ACM Eurosys*, page 21, 2014.

[11] S. B. Balaji, M. N. Krishnan, M. Vajha, V. Ramkumar, B. Sasidharan, and P. V. Kumar. Erasure coding for distributed storage: An overview. *CoRR*, abs/1806.04437, 2018. http://arxiv.org/abs/1806.04437.

[12] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, and A. Rowstron. Pelican: A building block for exascale cold data storage. In *Proc. of USENIX OSDI*, 2014.

[13] B. Beach. Backblaze Vaults: Zettabyte-Scale Cloud Storage Architecture. https://www.backblaze.com/blog/vault-cloud-storage-architecture/, 2017.

[14] M. Blaum, J. L. Hafner, and S. Hetzler. Partial-MDS codes and their application to RAID type of architectures. *IEEE Trans. on Information Theory*, 59(7):4510–4519, 2013.

[15] J. Blömer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, UC Berkeley, Aug 1995.

[16] J. C. Chan, Q. Ding, P. P. C. Lee, and H. H. Chan. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. In *Proc. of USENIX FAST*, pages 163–176, 2014.

[17] Y. L. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure coding objects across global data centers. In *Proc. of USENIX ATC*, pages 539–551, 2017.

[18] L. Cheng, Y. Hu, and P. P. C. Lee. Coupling decentralized key-value stores with erasure coding. In *Proc. of ACM SoCC*, pages 377–389, 2019.

[19] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. G. Sirer. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *Proc. of USENIX ATC*, pages 31–43, 2015.

[20] Cisco Systems. Oversubscription and density best practices. https://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/storage-networking-solution/net_implementation_white_paper0900aecd800f592f.html, 2015.

[21] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of ACM SoCC*, pages 143–154, 2010.

[22] H. Dau, I. M. Duursma, H. M. Kiah, and O. Milenkovic. Repairing Reed-Solomon codes with multiple erasures. *IEEE Trans. on Information Theory*, 64(10):6567–6582, 2018.

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

[24] K. S. Esmaili, L. Pamies-Juarez, and A. Datta. CORE: Cross-object redundancy for efficient data repair in storage systems. In *Proc. of IEEE BigData*, 2013.

[25] A. Fikes. Storage architecture and challenges. http://cloud.google.com/files/storage_architecture_and_challenges.pdf, 2010.

[26] D. Ford, F. Labelle, F. I. Popovici, M. Stokel, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, Oct 2010.

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

[28] W. Haddock, P. V. Bangalore, M. L. Curry, and A. Skjellum. High performance erasure coding for very large stripe sizes. In *Proc. of IEEE SpringSim*, pages 1–12, 2019.

[29] J. L. Hafner. HoVer erasure codes for disk arrays. In *Proc. of IEEE/IFIP DSN*, 2006.

[30] M. Holland, G. A. Gibson, and D. P. Siewiorek. Architectures and algorithms for on-line failure recovery in

redundant disk arrays. *Distributed Parallel Databases*, 2(3):295–335, 1994.

[31] H. Hou, P. P. C. Lee, K. W. Shum, and Y. Hu. Rack-aware regenerating codes for data centers. *IEEE Trans. on Information Theory*, 65(8):4730–4745, 2019.

[32] Y. Hu, X. Li, M. Zhang, P. P. C. Lee, X. Zhang, P. Zhou, and D. Feng. Optimal repair layering for erasure-coded data centers: From theory to practice. *ACM Trans. on Storage*, 13(4):33, 2017.

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

[34] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In *Proc. of USENIX ATC*, Jun 2012.

[35] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg. On fault tolerance, locality, and optimality in locally repairable codes. In *Proc. of USENIX ATC*, pages 865–877, 2018.

[36] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. Atlas: Baidu's key-value storage system for cloud data. In *Proc. of IEEE MSST*, pages 1–14, 2015.

[37] H. Li, Y. Zhang, Z. Zhang, S. Liu, D. Li, X. Liu, and Y. Peng. PARIX: Speculative partial writes in erasure-coded systems. In *Proc. of USENIX ATC*, 2017.

[38] J. Li and B. Li. Beehive: Erasure codes for fixing multiple failures in distributed storage systems. *IEEE Trans. on Parallel and Distributed Systems*, 28(5):1257–1270, 2016.

[39] M. Li, R. Li, and P. P. C. Lee. Relieving both storage and recovery burdens in big data clusters with R-STAIR codes. *IEEE Internet Computing*, 22(4):15–26, 2018.

[40] M. Li, J. Shu, and W. Zheng. GRID codes: Strip-based erasure codes with high fault tolerance for storage systems. *ACM Trans. on Storage*, 4(4):1–22, 2009.

[41] R. Li, X. Li, P. P. C. Lee, and Q. Huang. Repair pipelining for erasure-coded storage. In *Proc. of USENIX ATC*, pages 567–579, 2017.

[42] Y. Li, H. Wang, X. Zhang, N. Zheng, S. Dahandeh, and T. Zhang. Facilitating magnetic recording technology scaling for data center hard disk drives through filesystem-level transparent local erasure coding. In *Proc. of USENIX FAST*, 2017.

[43] W. Litwin, R. Moussa, and T. Schwarz. LH*RS: A highly-available scalable distributed data structure. *ACM Trans. on Database Systems*, 30(3):769–811, 2005.

[44] M. Luby, R. Padovani, T. J. Richardson, L. Minder, and P. Aggarwal. Liquid cloud storage. *ACM Trans. on Storage*, 15(1):2, 2019.

[45] P. Luse and K. Greenan. Swift object storage: Adding erasure code. *SNIA Education September*, 2014.

[46] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi. Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage. In *Proc. of ACM Eurosys*, page 30. ACM, 2016.

[47] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's Warm BLOB Storage System. In *Proc. of USENIX OSDI*, pages 383–398, 2014.

[48] P. Narayanan, S. Samal, and S. Nanniyur. Yahoo cloud object store-object storage at exabyte scale. `https://yahooeng.tumblr.com/post/116391291701/yahoo-cloud-object-store-object-storage-at`, 2017.

[49] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. In *Proc. of ACM VLDB*, 2013.

[50] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Guyot, E. E. Gad, and Z. Bandic. Opening the chrysalis: On the real repair performance of MSR codes. In *Proc. of USENIX FAST*, 2016.

[51] D. S. Papailiopoulos and A. G. Dimakis. Locally repairable codes. *IEEE Trans. on Information Theory*, 60(10):5843–5855, Oct 2014.

[52] J. S. Plank and C. Huang. Tutorial: Erasure coding for storage applications. Slides presented at USENIX FAST 2013, Feb 2013.

[53] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. OHearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proc. of USENIX FAST*, 2009.

[54] J. S. Plank and M. G. Thomason. A practical analysis of low-density parity-check erasure codes for wide-area storage applications. In *Proc. of IEEE/IFIP DSN*, 2004.

[55] J. S. Plank and L. Xu. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *Proc. of IEEE NCA*, pages 173–180, 2006.

[56] N. Prakash, V. Abdrashitov, and M. Médard. The storage versus repair-bandwidth trade-off for clustered storage systems. *IEEE Trans. on Information Theory*, 64(8):5783–5805, 2018.

[57] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proc. of USENIX OSDI*, pages 401–417, 2016.

[58] 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 *Proc. of USENIX FAST*, 2015.

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

[60] 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 *Proc. of ACM SIGCOMM*, 2014.

[61] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[62] B. Sasidharan, G. K. Agarwal, and P. V. Kumar. Codes with hierarchical locality. In *Proc. of IEEE ISIT*, pages 1257–1261, 2015.

[63] 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 *Proc. of ACM VLDB Endowment*, pages 325–336, 2013.

[64] Z. Shen and P. P. C. Lee. Cross-rack-aware updates in erasure-coded data centers: Design and evaluation. *IEEE Trans. on Parallel and Distributed Systems*, 31(10):2315–2328, Oct 2020.

[65] Z. Shen, P. P. C. Lee, J. Shu, and W. Guo. Cross-rack-aware single failure recovery for clustered file systems. *IEEE Trans. on Dependable and Secure Computing*, 17(2):248–261, Mar/Apr 2020.

[66] Z. Shen, J. Shu, Z. Huang, and Y. Fu. ClusterSR: Cluster-aware scattered repair in erasure-coded storage. In *Proc. of IEEE IPDPS*, pages 42–51. IEEE, 2020.

[67] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *Proc. of ACM SYSTOR*, pages 1–7, 2014.

[68] J.-y. Sohn, B. Choi, S. W. Yoon, and J. Moon. Capacity of clustered distributed storage. *IEEE Trans. Information Theory*, 65(1):81–107, 2018.

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

[70] K. Taranov, G. Alonso, and T. Hoefler. Fast and strongly-consistent per-item resilience in key-value stores. In *Proc. of ACM EuroSys*, page 39, 2018.

[71] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy, S. Hussain, and S. Nandi. Clay codes: moulding MDS codes to yield an MSR code. In *Proc. of USENIX FAST*, page 139, 2018.

[72] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. of USENIX OSDI*, pages 307–320, 2006.

[73] M. M. Yiu, H. H. Chan, and P. P. C. Lee. Erasure coding for small objects in in-memory KV storage. In *Proc. of ACM SYSTOR*, page 14, 2017.

[74] H. Zhang, M. Dong, and H. Chen. Efficient and available in-memory KV-store with hybrid erasure coding and replication. In *Proc. of USENIX FAST*, pages 167–180, 2016.