Morph: Efficient File-Lifetime Redundancy Management for Cluster File Systems

Timothy Kim*, Sanjith Athlur*, Saurabh Kadekodi§, Francisco Maturana*
Dax Delvira*, Arif Merchant§, Gregory R. Ganger*, K. V. Rashmi*
*Carnegie Mellon University §Google

Abstract

Many data services tune and change redundancy configurations of files over their lifetimes to address changes in data temperature and latency requirements. Unfortunately, changing redundancy configs (*transcode*) is IO-intensive. The Morph cluster file system introduces new transcode-efficient redundancy schemes to minimize overheads as files progress through lifetime phases. For newly ingested data, commonly stored via 3-way replication, Morph introduces a *hybrid redundancy* scheme that combines a replica with an erasure-coded (EC) stripe, reducing both ingest IO and capacity overheads while enabling free transcode to EC by deleting replicas. For subsequent transcodes to wider, more space-efficient EC configs, Morph exploits *Convertible Codes*, which minimize data read for EC transcode, and introduces new block placement policies to maximize their effectiveness.

Analysis of data ingest and transcode activity in Google storage clusters shows the current massive IO load and the potential savings from Morph's approach—transcode IO can be reduced by over 95%, and total ingest+transcode IO can be reduced by 50–60% while also reducing capacity overheads for newly ingested data by 20%. Experiments evaluating a Morph implementation in HDFS show that these benefits can be realized in a real system without hidden increases in complexity, tail latency, or degraded-mode latency.

1 Introduction

Distributed file systems (DFSs) spread PBs-to-EBs of data over 10Ks-to-100Ks disks and use redundancy for fault tolerance [8, 17, 49]. When ingested, most data is stored via 3-way replication to minimize throughput and tail latency concerns. After a few hours or days, most data is erasure coded (EC) for capacity-efficient fault tolerance.

Generally speaking, k chunks of data protected with $n - k$ chunks of parities (redundancy) make up a EC(k,n) stripe.

SOSP '24, November 4–6, 2024, Austin, TX, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695981

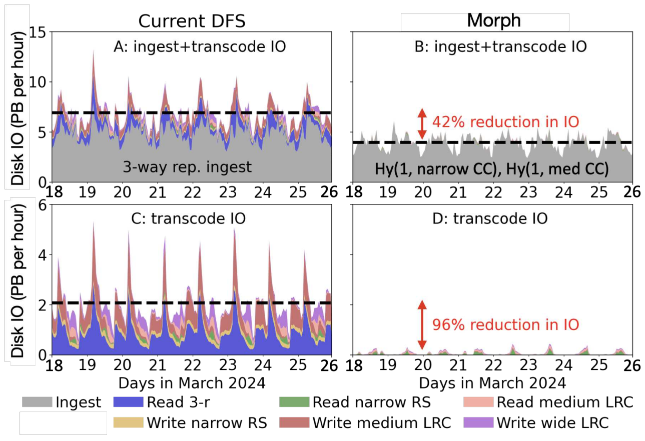


Figure 1. Morph reduces ingest+transcode IO on real-world cluster data. The graphs on the left show one week of measured hourly ingest+transcode IO (top=A) and transcode-only IO (bottom=C) for one of the largest Google data services. The graphs on the right show the IO to provide the same data protections and lifetime transitions using Morph. The colors categorize IO for the different lifetime transitions: ingest, transcode from 3-way replication to a narrow EC (e.g., RS(8,12)) after a day or so, transcode from narrow EC to medium LRC (e.g., LRC(32,38)) after a month or so, and transcode from medium LRC to wide LRC (e.g.,LRC(64,74)) after several months.

The values of k and n dictate data durability, availability and storage cost and can also affect read and write performance. For instance, a EC(6,9) stripe of the commonly-used Reed-Solomon code can tolerate 3 simultaneous chunk failures at $\frac{3}{6}$=50% storage overhead. As such, different values of k and n are often selected for different files' contents.

Indeed, the applications (data management services) that create most files in large clusters now change (transcode) the redundancy schemes of those files one or more times during their lifetime to reduce space overhead as data cools. Since different services have different reliability and performance needs, the services choose the redundancy scheme and not the DFS. In most file systems [14, 25, 43], transcode is not directly supported as a native DFS operation. Instead, an application transcodes a file by reading the file's data, writing it into a new file with the new EC, and deleting the original.

In large-scale cluster file systems, huge amounts of IO are used for establishing and changing redundancy. As a concrete example, Fig. 1A shows measured ingest+transcode IO for one of the largest data services in a Google storage cluster going through the lifetime phases presented in Fig. 2. Fig. 1C zooms in on the transcode portion, which is 20–33% of the 5–13PB/hour total.

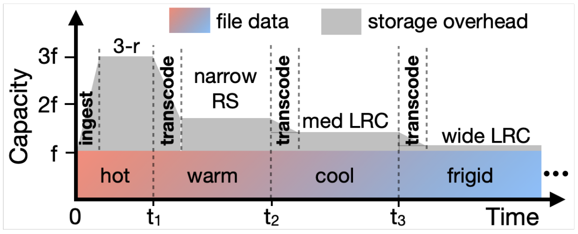


Figure 2. A file of f bytes is ingested in 3-r. Then, at t_1 the file is transcoded to a narrow RS scheme, at t_2 is transcoded to a medium-width LRC, and at t_3 is transcoded to wide LRC with the least storage overhead. Each transcode requires $2 \times f$ IO as the file is read, re-encoded, and written to the new redundancy scheme. Transcode happens in reaction to the file's cooling data temperature over time.

We present Morph, a cluster file system that reduces IO requirements for redundancy throughout file lifetimes. Morph introduces new redundancy schemes that make both ingest and transcode more efficient and makes transcode a native DFS operation. For ingest, it introduces a new *hybrid redundancy* scheme that combines one (or two) replicas with an EC stripe. This reduces ingest IO and eliminates transcode IO for the file's first lifetime transition, since the replica can simply be deleted to convert to EC. To make other transitions efficient, Morph adopts and specializes for a recently proposed theoretical code construction framework, called *Convertible Codes* (CC) [37, 40, 41], that minimizes data read when transcoding from one EC to another, often avoiding reading any data chunks (just the parities).

The result is large reduction in ingest+transcode IO. Comparing Fig. 1B to Fig. 1A, we see that Morph would reduce the IO required by ≈40%. About half of the reduction comes from a 20% reduction in ingest IO, because the hybrid scheme of one replica plus an EC stripe is a 2.4× data expansion rather than 3× for 3-way replication. The capacity consumed is also reduced accordingly. The other half comes from a >95% reduction in transcode IO (compare Fig. 1D to Fig. 1C) required for file lifetime transitions.

Realizing Morph's promise has required overcoming a number of correctness and efficiency challenges. First, ingest uses 3-way replication to achieve high-write throughput and low write and read tail latencies (since the newest data is warmest), and to avoid in-line EC which comes with incremental parity update complexity from read-modify-write. A straightforward realization of the hybrid redundancy scheme would jeopardize all of these requirements. Morph introduces a write protocol that replicates (3-way) across servers, but rapidly computes the EC parities in the background and usually deletes the then-unneeded extra replicas before they are written to disk. Second, while recent work in the field of coding theory has provided the theoretical foundations for Convertible Codes, incorporating this new class of codes requires system design changes to realize the promised benefits in practice. For example, EC-to-EC file transcode requires processing multiple EC stripes at once, such as when combining each sequence of five EC(6,9) stripes in the file into a valid EC(30,33) stripe (with all n=33 on different servers), which can require moving some data chunks if not placed properly. Morph introduces data placement policies to avoid such data movement and employs parity placement policies to maximize server-local computation.

We implement Morph in HDFS [15], enabling head-to-head comparison to a popular open-source DFS. Experiments on a small academic cluster show that Morph achieves a 58% reduction in disk IO and 55% reduction in network IO, compared to HDFS, for a mix of data ingest and file lifetime transitions. Our analytical evaluations of Morph applied to month-long traces of large-scale production data service applications at Google similarly show 40-50% reductions in IO for observed ingest and transcode activity. In addition, we confirm that reads and writes with a hybrid redundancy scheme perform almost identically to replication across workloads and cluster conditions. We also show that a practical implementation of Convertible Codes achieves significant IO and computational savings, directly reflecting the theoretical savings while not compromising encode or decode times for data writes and reconstruction events.

Contributions. This paper makes five primary contributions: (1) It introduces a new *hybrid redundancy* scheme that combines replication with erasure coding to provide replication-like performance, lower space consumption, and near-zero transcode-to-EC overhead. (2) It describes data and parity placement policy augmentations that minimize both disk IO and network IO for transcode operations with minimal impact on other placement considerations. (3) It describes how Morph makes transcode a native DFS operation and combines Convertible Codes with (1) and (2) in a real system. (4) It shows that Morph can be implemented efficiently, outperforms HDFS for both ingest and each transcode step, and offers large reductions in the overall ingest+transcode IO based on production traces from Google. (5) It establishes that Convertible Codes, which were so far studied only theoretically, can indeed be incorporated into real-world DFSs and that they provide significant IO and computational savings for transcode operations in production traces.

2 Background and Motivation

Exascale storage clusters contain 100Ks of disks deployed on 1000s of servers managed by a distributed file system (DFS). Each file's contents are stored in a set of fixed-size chunks using the redundancy scheme its creator specifies.

Data redundancy schemes. Replication and erasure coding are the main redundancy mechanisms used to ensure data availability and durability. Generally, replication has lower average and tail write latency (see Fig. 3) at the expense of higher storage overhead, with 3-way replication ("3-r") being the most common replication configuration in practice[15, 17, 43]. Since replicas are exact copies, any replica can be read to service a read request. In fact, some

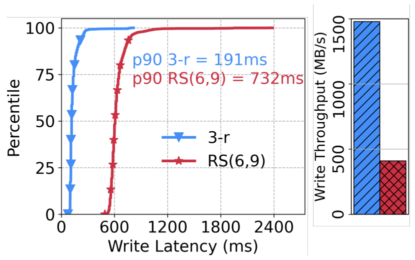


Figure 3. 3-r outperforms RS(6,9). Write latency and throughput are shown for creating 8MB files. Both median and tail (p90) write latency for RS(6,9) is almost 4× higher than 3-r. Throughput is 68% lower, and read performance suffers, especially for degraded-mode reads.

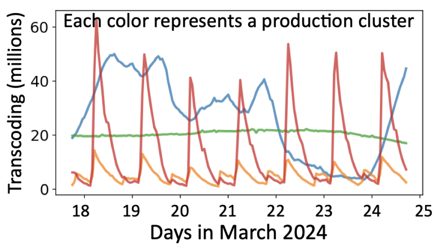


Figure 4. File transitions per hour in four Google storage clusters. Each line shows the millions of file transitions pending+performed per hour (including early life to mid-life and mid-life to late-life) in one storage cluster, each transition consuming large amounts of cluster resources.

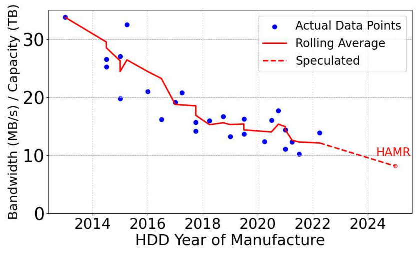


Figure 5. HDD sustained bandwidth per capacity trend in the last decade [4]. The red empty circle denotes the speculated bandwidth per capacity cost with the introduction of disk technologies like HAMR (heat assisted magnetic recording) [1–3].

DFS clients will initiate reads of more than one replica in parallel and use the first response in order to decrease tail latency [9].

Erasure coding (EC) entails associating each k equal-sized chunks with r parities to make up an n-chunk *EC stripe*. Compared to replication, EC has lower storage overhead ($\frac{r}{k}$) at the cost of generally higher latencies and lower IO efficiency. Also, when a data chunk is unavailable in an EC stripe, reconstructing it requires reading multiple other data chunks in the stripe plus a parity chunk and mathematically decoding the missing chunk. A client read that decodes a missing chunk is called a degraded-mode read.

Multiple classes of erasure codes exist [10, 16, 21, 26, 34, 45, 46, 54] but the two most common constructions in large-scale storage clusters are Reed-Solomon (RS) [54] and Locally Recoverable Codes (LRC) [21, 27]. An RS(k, n) stripe (also written as k-of-n) can tolerate any arbitrary combination of $n - k$ chunk failures. However, RS stripes incur the cost of high reconstruction overheads. A recovery in EC(k,n) stripe implies DFS has to read k *of* n blocks from the stripe. This overhead is exacerbated with wider stripes as k increases. LRCs are popular due to their low reconstruction overheads. The k data chunks are organized into l local groups, each group protected with its own parity, in addition to r *global* parities. We denote this LRC with a (k, l, r) scheme [27]. A single failure in a local group can be recovered using k/l local chunks. There are numerous LRC constructions [7, 20, 23, 31, 33, 50–52]; discussing their differences is beyond the scope of this paper. [1]

Data age, temperature and redundancy. Fig. 2 illustrates the lifetime of a file, such as for the data service shown in Fig. 1, in terms of the redundancy schemes used. Recently created data is often popular in its *early life*. Of course, data reuse is absorbed by large caches, but early life data often requires higher availability and IO performance, and therefore is often stored using replication because directly storing in EC has performance implications as shown in Fig. 3. It is common for a large storage cluster to ingest 100s of PB daily, mostly stored in 3-r (200% storage overhead).

As data ages, it tends to cool, such that IO performance requirements are less stringent. Data in *mid-life* ranges from warm to cool in terms of access frequency, and is often stored using EC to reduce storage overhead. Narrow RS codes or narrow LRCs, where the EC stripe width (k) is typically below 20 chunks, are popular choices for mid-life data. The narrow width of the EC stripe keeps tail latencies from going too high, especially during client degraded mode reads. Narrow RS or LRCs typically have 25–50% storage overhead.

Late-life data is cold or frigid. Online archives, backups, and years-old videos are examples of data in late-life. Client accesses are rare but can occur, and thus, maximizing storage efficiency is the main focus of redundancy scheme choice. Wide LRCs are typical for late-life data, with k as large as 80 chunks [31], or even 150 chunks per EC stripe [11], with 10–20% storage overheads.

Transcodes are frequent and IO-intensive. Modern DFSs allow files to appropriately change redundancy schemes based on their stage of life. These transcodings have become a common operation performed by data services today. Fig. 4 shows the number of file transcodes per hour for four Google exascale storage clusters. The millions of transcodes each hour include early to mid, mid to late, and sometimes multiple configurations in between. In most DFSs, however, transcode is not a native DFS operation—instead, an application does it by reading file data and re-writing it as a new file with the target scheme. This *read–re-encode–write (RRW)* operation requires significant disk IO and network bandwidth to read data, write parities, *and re-write data*. Several systems such as HDFS, Ceph and most popular public cloud infrastructures we know of use this approach [14]. As illustrated in Fig. 1, huge amounts of IO (2PB/hour in this example) are used for transcoding files via RRW.

Increasing IO-per-byte-stored bottleneck. Per-HDD capacity increases over the years at a faster rate than per-HDD bandwidth: ≈11.8%/year vs. ≈5.1%/year. Consequently, the available bandwidth-per-TB has been decreasing at an

[1]Readers can refer to the tutorial [12] on erasure-coding for more details.

alarming ≈8.5%/year rate. Further, the onset of new disk technologies like HAMR is expected to exacerbate the problem, as illustrated in Fig. 5. Thus, it is critical to design cluster file systems to minimize IO demand.

3 Morph Overview

Morph is a cluster file system that reduces cluster IO requirements by addressing the large amount of IO used to establish redundancy (during ingest) and change it over file lifetimes (via transcode). It does so while continuing to satisfy application constraints related to data durability, storage efficiency, and access latency by introducing new redundancy schemes that match those properties for existing alternatives (3-r and RS) but reduce IO for ingest and transcode. Thus, applications can continue to optimize their redundancy choices as before while using less IO.

Morph's design views the lifetime of a file in two parts: (1) Early life and transition to mid-life, when data generally requires lower average and tail latency, and (2) transitions from mid-life onwards, when data generally becomes progressively colder and performance increasingly shrinks in importance relative to storage overhead.

For early-life, Morph introduces a new *hybrid redundancy* scheme that simultaneously employs both replication and EC, along with specifically designed placement and read/write protocols. We show that applications can use this hybrid redundancy instead of 3-way replication to achieve early life goals (performance and durability like 3-r) but with lower ingest (file creation) IO and near-zero-IO transcode to EC for mid-life, via a new explicit transcode() interface.

For mid-life, Morph leverages recent advances in coding theory and employs a new class of codes called *Convertible Codes* (CC) [40, 41] in place of RS codes. CC provide the same reliability properties and configuration flexibility, but they enable much more IO-efficient transcode from one EC scheme to another. Whereas applications need to explicitly opt into choosing hybrid redundancy during file creation (ingest), CC are a drop-in replacement for existing EC mechanisms. Applications can specify EC parameters as before and expect the same performance and reliability guarantees. If, however, the target EC configurations are well-chosen (as explained below), Convertible Codes allow Morph to significantly reduce IO overheads, especially in a transcode-native DFS designed specifically to exploit the new codes.

The next two sections describe the design and challenges overcome for each phase.

4 Early to Mid Life – Hybrid Redundancy

Data in early life is often "hot" and thus demands high performance requirements such as fast write throughput and low read latencies. Erasure coding cannot meet these high performance demands (see Fig. 3), leaving replication as the defacto option for new files. However, replication bears high storage

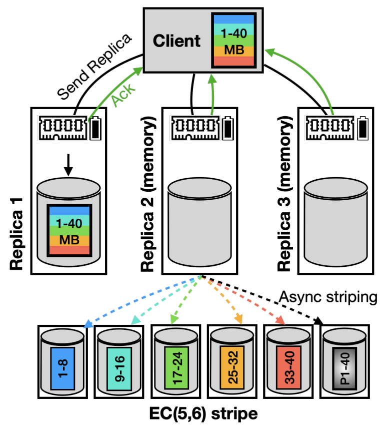


Figure 6. Client issuing a 40 MB spanning write in $\mathrm{Hy}(1, \mathrm{EC}(5,6))$. The replica is sent to 3 nodes and is acknowledged from each after reaching memory-backed RAM. The first node persists the replica to disk while the second node asynchronously sends data chunks with their parities to other nodes. Both the second and third node delete their temporary replicas once parities are persisted. In its resting state, $\mathrm{Hy}(1, \mathrm{EC}(5,6))$ has 1 replica, 5 data chunks, and 1 parity chunk on disk.

overheads after ingest and poses challenges for transitioning into more space-efficient storage options later on. Therefore, we propose an alternative *hybrid redundancy* storage scheme. Morph's hybrid redundancy offers better capacity-efficiency than replication, fulfills the necessary properties for hot data, and enables efficient transitions as data cools.

4.1 Definition

Hybrid redundancy stores data in both replicated (replica blocks) and erasure-coded form (EC stripe) simultaneously. Fig. 6 shows an example of a hybrid scheme with 1 replica block and an EC stripe with 5 data chunks and one parity chunk. Note that the data chunks of the EC stripe contain the same data as the replicas and act as another copy despite being in a different format. The hybrid scheme is generally applicable to any combination of replicated and EC formats as it is simply a combination of the two.

A $\mathrm{Hy}(c, \mathrm{EC}(k, n))$ scheme represents c copies for each replicated block and an EC stripe consisting of k data chunks protected by $n-k$ parity chunks. Morph presents the $\mathrm{Hy}(1, \mathrm{EC}(k, n))$ scheme, that provides sufficient durability (one extra replica over an already durable EC stripe) and lower space overhead than 3-r ($\frac{n}{k}\times$ versus 2×), while meeting read and write performance requirements. In addition, Morph also supports $\mathrm{Hy}(2, \mathrm{EC}(k, n))$ scheme with two replicas instead of one.

While the idea of hybrid redundancy is simple, it encounters several challenges. We describe these challenges and Morph's solutions below.

4.2 Hybrid Writes

Hybrid redundancy writes data in both replicated and EC form during ingest. Though beneficial for future phase transitions, writing the EC stripe when the file is first written poses challenges with write performance and reliability.

In many DFSs, writes are often deemed durable when the data reaches the *in-memory* buffer cache (battery-backed RAM) on 3 nodes, say when writing a 3-r file. Consequently, the ingest latency is bottlenecked by the slowest of three nodes when receiving data. To maintain these durability and write latency expectations, Morph handles small writes and spanning writes (large writes spanning multiple EC chunks) differently.

For small writes (writes spanning only one EC chunk), Morph always sends data to 2 replicas *in-memory* and to the corresponding EC chunk in the stripe. The write is acknowledged after all three nodes receive the data, thereby achieving durability and slowest-of-3 write latency requirements. The parity computation is offloaded from the client, and delegated to one of the servers hosting a replica. c of the 2 replicas and the EC chunk get persisted to disk immediately after receipt of data. The remaining $2-c$ replicas are evicted from memory only after the parities are computed and persisted to disks to maintain durability. Thus, while $\text{Hy}(2, \text{EC}(k, n))$ persists both the replicas to disk, $\text{Hy}(1, \text{EC}(k, n))$ only persists one of the two replicas and discards the other. In the event that a replica needs to be evicted from memory or when a file is closed before parities get persisted, both replicas are persisted even in the case of $\text{Hy}(1, \text{EC}(k, n))$.

For large writes spanning multiple EC chunks, the aforementioned approach can lead to higher tail latencies. For writes spanning all k chunks, the client waits for the slowest-of-k nodes holding the chunks of the EC stripe in addition to the two replica nodes. To overcome this challenge, Morph writes all data to 3 replicas (instead of 2) *in-memory*. The client waits for acknowledgement from these three replicas (thereby maintaining the durability and latency requirements) as shown in Fig. 6. One of the nodes storing a temporary replica assumes the role of a *striper*, and writes data to the n data nodes in the striped form in the background (*async striping*). Similar to the small writes case, while c of the 3 replicas are persisted to disk immediately, others are discarded after persistence of parities. We meet the slowest-of-3 latency requirements at the cost of server-to-server network IO and server memory resources.

Appendability guarantees. A popular attribute of replicated files unavailable to EC files is the ability to append without modifying parities. Morph provides this ability by computing parities only when the EC stripe is complete. As hybrid writes are made durable through the replicas and not parities, parity computation can be arbitrarily delayed until all the data chunks are present.

4.3 Hybrid Reads

Replication offers the intrinsic advantage of having r copies of data, enabling read latency that reflects the *best of* r. Reads to an EC stripe (striped reads), on the other hand, can span up to k blocks, in which case the read samples the *worst of* k. However, striped reads can potentially utilize up to k disks in parallel to improve read throughput compared to a single disk in a replica read. Hybrid redundancy offers the read latency of replication and the read throughput of striped reads by dynamically adapting its read strategy to client access patterns.

Latency-sensitive workloads. Reading data from a hybrid file can be performed in one of two ways (1) read from a replica or (2) read from the EC stripe. We find at Google that most client reads are small (i.e., $\leq$1 MB) latency-sensitive reads. For this type of read, a hybrid scheme with $c = 2$ performs exactly the same as 3-r at both the median and tail. As the read does not span across multiple EC chunks (i.e., read from a single chunk), option (2) becomes equivalent to option (1). Even in the case when $c = 1$, two copies of the data can be requested before resorting to a degraded mode read. We find analytically that probability of a degraded mode read from the striped portion of a hybrid scheme is very low (tail-of-the-tail). See Appendix B for an example.

Throughput-heavy workloads. In fact, there are client workloads that *benefit* from hybrid redundancy due to the availability of a striped read. A striped read can achieve greater read throughput than a sequential replicated read by concurrently reading from multiple disks. For large file scans, Morph automatically reads in parallel from the stripe chunks rather than the replicas. As large reads become bottlenecked by bandwidth rather than IOPS, striped reads *outperform* replica reads.

Degraded reads. When some data chunk is unavailable, fetching several more data chunks and decoding on the critical path induces latency on the critical path for erasure-coded files. Morph circumvents this issue by instead reading from a replica when a data chunk is unavailable.

4.4 Fault Tolerance and Data Recovery

In general, a $\text{Hy}(c, \text{EC}(k, n))$ scheme is capable of handling any arbitrary $c + (n-k)$ failures. For any range of data, there exist c replicas, 1 set of data chunks, and $n-k$ parity chunks to recover from. Hybrid redundancy provides simple and efficient reconstruction protocols. In the case of a replica chunk failure, the replica can be recovered by either (1) reading another replica with the same contents if $c > 1$ or (2) reading from the EC stripe if (1) is not an option. In the case of an EC data chunk failure, the data can be recovered by (1) reading the relevant portion of a replica or (2) reading the other data chunks and a parity chunk. The same options are available during an EC parity chunk failure, but the entire EC data stripe must be read either from the replica or the data chunks to re-compute the parity.

4.5 Transitions from Hybrid to EC

Designing hybrid redundancy to be an exact combination of the replicated and EC components is a conscious choice meant to facilitate the future transition to EC. A transition from replication to EC forces the file to be rewritten, which has significant overheads (i.e., reading file, rewriting file, writing new parities) as the data needs to be eventually written out *in a different format*. Clients often know apriori which EC scheme the data will transition to and can build that information into data ingest with hybrid redundancy. Thus, a transition from hybrid to erasure-coded is simply a localized metadata change and the deletion of the replicas.

5 Mid to Late Life – Convertible Codes

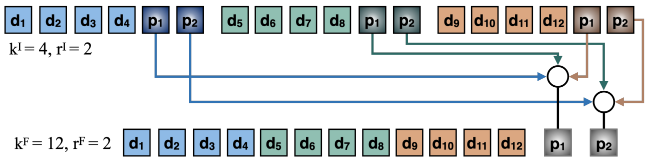
In mid to late life (post-replication period), a file sees multiple transitions between EC schemes, each scheme wider and more capacity efficient than the previous. Morph leverages recent advances in coding theory to minimize transcode IO in this phase of a file's life. Specifically, Morph uses a class of codes called Convertible Codes (CC) [40, 41] instead of RS codes and uses a class of codes called Locally Recoverable Convertible Codes (LRCC) [42] instead of LRCs. While the theory behind CC and LRCC promises reduction in transcode IO, several challenges need to be overcome to realize the benefits in practice. We discuss the codes, challenges and system design decisions and optimization done in Morph to leverage these codes below.

5.1 Codes used by Morph

Convertible Codes (CC) [40, 41] are a new class of erasure codes that are designed to significantly reduce transcode IO overheads. The key idea is to design the initial parities to enable maximum information extraction from existing parities during transcode, so that the new parities can be generated without having to read *all* the data chunks. These codes are an alternative to traditional codes (e.g., Reed-Solomon codes). Specifically, CC have the same parameters and same fault tolerance properties as traditional codes, and only the parity coefficients are different. The carefully designed parity coefficients help in reducing transcode IO. CC enable Morph to support transcoding from any-to-any EC schemes.

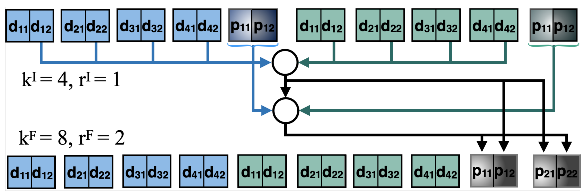
The efficacy of transcoding under CC depends on the initial and final EC parameters. In the best case, when integral number of stripes are *merged* to create a stripe and the number of parities does not increase, new parities can be calculated solely by reading the existing parities and none of the data blocks. An example of this is shown in Fig. 7. If integral number of stripes are merged but number of parities per stripe increase, in addition to parities, reading *some* parts of the data blocks will be necessary. There is still a reduction in transcode IO, albeit smaller. An example is shown in Fig. 8. We provide more details, including a discussion on other transcoding scenarios, in Appendix A.

Morph builds on the ideas behind CC and presents efficient transcode with LRCs as well. A possible warm-to-cool transcode is from CC to LRCC (Fig. 2). In the best case, each local group in LRCC is formed using an integral number of initial stripes of the CC, and the number of final global



d_1 d_2 d_3 d_4 p_1 p_2 d_5 d_6 d_7 d_8 p_1 p_2 d_9 d_{10} d_{11} d_{12} p_1 p_2
$k^I = 4$, $r^I = 2$
$k^F = 12$, $r^F = 2$
d_1 d_2 d_3 d_4 d_5 d_6 d_7 d_8 d_9 d_{10} d_{11} d_{12} p_1 p_2

Figure 7. Transcode using CC to merge three stripes of CC(4,6) into one stripe of CC(12,14). Instead of 12 data blocks, only the 6 parity blocks are read for transcode, reducing IO by 50%.



$d_{11}d_{12}$ $d_{21}d_{22}$ $d_{31}d_{32}$ $d_{41}d_{42}$ $p_{11}p_{12}$ $d_{11}d_{12}$ $d_{21}d_{22}$ $d_{31}d_{32}$ $d_{41}d_{42}$ $p_{11}p_{12}$
$k^I = 4$, $r^I = 1$
$k^F = 8$, $r^F = 2$
$d_{11}d_{12}$ $d_{21}d_{22}$ $d_{31}d_{32}$ $d_{41}d_{42}$ $d_{11}d_{12}$ $d_{21}d_{22}$ $d_{31}d_{32}$ $d_{41}d_{42}$ $p_{11}p_{12}$ $p_{21}p_{22}$

Figure 8. Transcode using CC where number of parities increase. Two stripes of CC(4,5) are transcoded to one stripe of CC(8,10), by reading all the parity blocks and only half of each of the data blocks, resulting in 25% reduction in the amount of data read for trancode.

parities is at most one less than the initial number of parities per stripe. Here, the local parity of each local group is computed by only reading the *first* parity of each of the constituent initial stripes. The global parities are computed by only reading the remaining parities. Consider the scenario of CC(6,9) to LRCC(24,4,2) (24 data blocks, 4 local groups (with one local parity), and 2 global parities), where four initial stripes are merged to form a single LRCC stripe. The first parity of each initial stripe remains unchanged and is used as the corresponding local parity in the LRCC stripe. The other two parities of each initial stripe are then merged to form the two global parities of the LRCC stripe.

The cool to frigid transcode uses similar mechanics (i.e., from LRCC(k^I,l^I,r^I) to LRCC(k^F,l^F,r^F)) by independently leveraging the properties of CC for computing both the local and global parities. See Appendix A for more details.

Next, we discuss how Morph takes transcode overheads for each of these scenarios into consideration when suggesting the best parameters to use.

5.2 Choosing CC-friendly EC scheme parameters

The efficacy of transcode depends on the initial and final EC parameters since the amount of reduction in IO achieved by CC and LRCC depends on the code parameters. Morph strategically suggests EC scheme parameters such that they are "CC-friendly" in order to reap the maximum IO benefits, without sacrificing any other aspects such as durability or space overhead. For instance, consider an application that transcodes its files from EC(6,9) to EC(27,30). Using a final scheme of EC(24,27) instead, improves transcode IO overhead by about 40%, with better durability and a trivial decline in space efficiency. Morph provides the strategic parameters as suggestions to applications, who make the final decision on the choice of the parameters.

Morph uses following heuristics to identify EC parameters that minimize transcode IO. When transcoding from EC(k^I,n^I) with r^I parities, to EC(k^F,n^F) with r^F parities: first, choose the final scheme such that the number of data chunks in the final stripe is an integral multiple of the initial number of data chunks. Second, keep the number of parities constant. However, when transitioning to wide schemes, it becomes imperative to add extra parities for maintaining minimum reliability. In such cases, minimize $k^F/k^I*(r^I+k^I*\frac{r^F-r^I}{r^F})$[41].

While Morph uses CC to reduce the transcode IO and suggests optimal parameters to use, the choice of when to transcode and parameters to trancode into remain with the application. To use CC across all scenarios, the application needs to pre-determine the EC schemes to transcode into at the time of file writing. This is indeed realistically feasible, as most transcoding events today are driven by a programmed schedule. For instance, over 75% of the transcode events shown in Fig. 4 are commonly occurring, pre-determined operations known by the service in advance.

5.3 Convertible Codes in Systems

Convertible Codes can be applied in DFSs to significantly reduce transcoding overheads. However, maximizing the benefits of Convertible Codes requires being aware of potential future transitions and purposefully placing data and parity chunks onto the appropriate disks and racks. This section describes system decisions and block placement heuristics that facilitate transcoding with Convertible Codes.

Transcoding in DFS. Transcoding occurs at the EC file granularity, each of which consists of a logical sequence of data chunks from EC stripes. Morph's design creates new EC stripes around *sequential data chunks*. For example, transcoding to a 6-of-9 scheme creates a set of 3 parities for data chunks 1-6, 7-12, etc... in the file. This strategy offers several benefits: 1) no additional metadata is required as stripe membership can be inferred based on data block ordering and EC scheme, 2) data placement decisions are aware of which blocks may end up merged later, 3) determining new stripes is computationally cheap.

Data separation. EC guarantees adequate data reliability by ensuring that no two chunks of the same stripe are placed onto the same disk or server. Chunk placement policies enforce this property when the data is first written. However, when transcoding to a target scheme, the new EC stripe potentially consists of chunks that were originally stored on the same server. To provide sufficient reliability, overlapping chunks must be moved to an unoccupied server, incurring disk and network IO. Morph eliminates this overhead by planning for future stripe widths when the file is originally written. As new stripes are formed from sequential data blocks, Morph separates sequential sets of chunks in the initial placement decision. Specifically, we determine the LCM of all potential future stripe widths or k^*. Chunks within each set of k^* consecutive chunks are placed on different servers without overlap.

Parity placement strategy. We observe when merging with Convertible Codes and $r^I = r^F$, each new parity chunk can be computed solely using the parities they replace. Therefore, we choose to place parity chunks *from different stripes* that may be merged together on the same node. This decision enables *local parity merges* that do not incur network overheads. When merging multiple stripes, the corresponding parity of each stripe is read from and combined into a new parity chunk on the same node. We argue that such placement of parities between stripes does not compromise the reliability of any given stripe. Any given stripe is capable of tolerating the same number of failures, independent of failures in the other stripes it shares parity servers with. Furthermore, the total amount of reconstruction work for a failed server does not change as each server already contains chunks across many different stripes, a very small fraction of which are the parities of a single file.

6 Implementation

We implement Morph as an extension to the Hadoop Distributed File System (HDFS). This section describes its (1) hybrid redundancy implementation and (2) robust transcoding module that supports general transcoding operations.

6.1 Hybrid Redundancy in a DFS

HDFS supports replicated and EC storage natively. Morph offers hybrid as a third class of storage with support for client read/write and background detection and recovery of block failures.

System block metadata. Clients can specify parameters for a hybrid file based on the client's replication factor (c) and the directory's erasure-coding policy (k, n). A hybrid file consists of a list of *hybrid blocks*. A hybrid block is internally an EC stripe joined to a list of replica blocks. An EC stripe consists of metadata for data and parity chunks while a replica block consists of metadata for replicated chunks. Maintaining a hybrid block as a single entity that has an internal nested structure serves to simplify metadata retrieval operations for recovery/reads and facilitate the metadata transition to future states.

Client hybrid write pipeline. HDFS writes replicas in a pipeline where data is sent from client to Datanode and then mirrored from Datanode to Datanode until the end of the chain. The client considers the write as durable when the acknowledgment propagates back up through the pipeline to the client. Morph's hybrid write pipeline similarly mirrors data through replica chunks, but additionally stripes to the EC data chunks at the end of the chain. When the pipeline is initialized, the striper instantiates and maintains a connection in parallel to all k Datanodes in the stripe. Fig. 9 shows a hybrid pipeline for a write to a Hy$(2, \text{CC}(4, n))$ block and

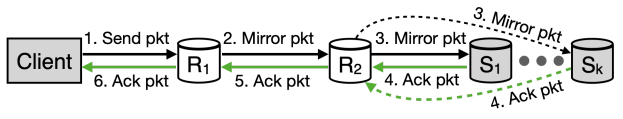


Figure 9. Hybrid pipelined write in HDFS. Writes propagate down a chain and are striped by the last node. Note that in step 3, any given packet is mirrored to only the relevant node that stores the data chunk.

the striper forwarding the packet to the first data chunk in the stripe, S1. After S1 is full, the striper will forward to S2.

In a hybrid pipeline, replicas and data chunks are not necessarily finished at the same time. Therefore, we add a new flag to packets that specifies to finalize data chunks in addition to the replica chunks. This mechanism allows keeping the data chunks open for appends from multiple stripers on their local filesystem until explicitly notified, significantly improving write performance.

Parity computation options. The client is provided three options for EC parity computation: 1) client computes parities synchronously, 2) Datanode computes parities asynchronously, 3) parities are disabled.

Synchronous encoding: The client maintains a stripe buffer of size $k *$ block size MB. When the stripe buffer is full, the client encodes the stripe's parities and sends them to the Datanodes to be appended to the appropriate parity chunk. The client observes the computational latency of encoding and sending parities for faster additional durability.

Asynchronous encoding: One Datanode is designated to be the striper for all of the hybrid block's write pipelines, ensuring that all data in the stripe passes through this node. As the striper receives data, it additionally writes the data to an elastic buffer pool. A background thread on the Datanode polls the buffer to encode and write the parities when the stripe is full. To guarantee the best client performance, writes to the buffer do not block and clients do not wait for parities to be received by the Datanode. By default Morph encodes parities asynchronously for high throughput and fast durability.

No encoding: Parities are not persisted at rest, and thus, data is kept durable solely by extra copies. This option provides the greatest throughput benefits as the system does exactly the same amount of work as for $c + 1$ replication but compromises on extra durability.

Hybrid block placement policies. Morph keeps block placement modifications to the minimum and simply excludes or includes Datanodes from the policy decisions. For hybrid writes, Morph initially allocates EC stripe chunk locations based on the existing EC block placement policy. Then, all locations in the EC stripe are excluded from any of the corresponding replica block's placement decisions. This guarantees no overlap between replica and EC chunks.

Client read optimizations. Morph implements a modified version of a *hedged read* for hybrid files. A hedged read initially requests the data from one replica chunk; if the initial read passes a certain time threshold, the client requests from another replica chunk in parallel. If there are no more replica chunks available, the client performs a striped read. These steps are repeated until all copies are exhausted or the first result is received, at which point the outstanding requests are canceled.

Morph optimizes for large, throughput-bound read requests separately. When the client requests a read that spans the contents of an entire EC stripe, Morph automatically performs a striped read. If a chunk in the stripe is slow or unresponsive, the client will either request a replica chunk if all the missing data is present on that single chunk or, if not, a parity chunk to perform a local degraded mode read.

Failure detection and recovery. Block failure is detected in the Namenode by observing Datanode heartbeats. When a chunk is detected as corrupt or missing, the Namenode identifies the block (hybrid, replicated, or EC stripe) that the chunk is a part of based on a mask of its block ID. This mechanism already exists in HDFS and is similarly leveraged for hybrid blocks. When hybrid block failure is detected, the relevant metadata (replica, data, parity chunk locations, EC scheme) is bundled and sent to a Datanode for reconstruction. The Datanode follows the steps outlined in Sec. 4 to reconstruct the chunk.

6.2 Transcoding as a first-class operation

Fig. 10 shows the various steps and data structures a transcoding operation goes through until completion.

Transcoding interface. Morph exposes a new interface at the Namenode defined as transcode(filename, policy). This new service allows clients to set a file's EC scheme and the DFS to enact the change asynchronously.

Transcoding flow. Morph implements the transcoding module within the Namenode, which exposes a simple interface to the client library in which clients specify a target EC policy for a file or directory. The Namenode forms new stripes across sequential data chunks and stores the new pending stripes in the awaiting transcoding queue (ATQ). Upon each Datanode heartbeat to the Namenode, the Namenode polls the ATQ until the maximum amount of work is allotted or the queue is empty. The event is allocated into the work queue of the least busy target parity Datanode and simultaneously tracked in the undergoing transcoding map (UTM). The UTM is a concurrent map of a file ID and its pending stripes, and another map of a stripe to its pending parity chunks. Morph uses a bitmap to track completion status to minimize the memory footprint of the UTM. The transcode completion is signaled when all parity chunks of all stripes of the file are complete.

Crash consistency during transcoding. The transcode completion signal serves several purposes. First, it allows the deletion of old parities to be delayed until all new parities are transcoded. This decision guarantees that throughout the transcoding process, normal reads, degraded reads, and reconstruction of the stripe remain available to clients and

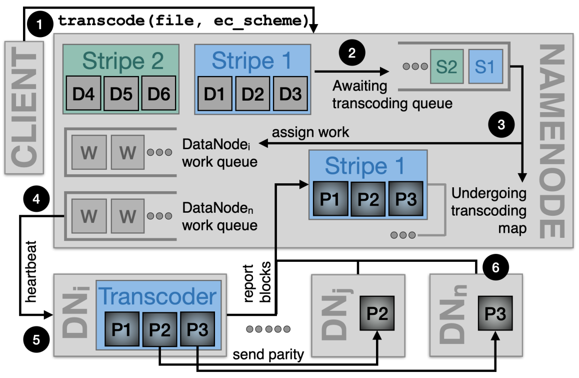


Figure 10. Morph transcode architecture in the context of HDFS components. In this diagram, a file is transcoded into two new stripes, each with 3 data chunks and 3 new parity chunks. Note that the Namenode only handles the assigning of transcode work to Datanodes and does not actually read or write the file data.

system pipelines. Second, it triggers the atomic switch in metadata to the new EC scheme and block locations, which keeps file state consistent with parity data while the file is undergoing transcoding. Third, it is used as a reference point for the filesystem state in the event of a Namenode failure, avoiding the need to persist any of the metadata state.

Transcoding-aware block placement policy. Morph preconfigures a default set of EC policies available to files and determines the LCM of the width, or k^*, of each of those policies. As the client writes to a file, a custom block placement policy excludes the locations of the other chunks in the set of k^* chunks from the data chunk placement decision. Additionally, the $n-k$ parity chunk locations originally designated for that file are excluded from the data chunk placements and included for the parity chunk placements. This strategy guarantees 1) every set of k^* data chunks are uniquely placed, 2) data chunks and parity chunks do not overlap, and 3) parity chunks across stripes that may be merged are co-located.

Convertible Codes implementation. We implement both CC [37, 39] and LRCC [42]. When the number of parities change, *vector* codes are employed which enable efficiently computing new parities by reading only the parity chunks and a *fraction* of each data chunk. This is possible as a portion of each new parity chunks is pre-computed when the file is initially written. The specific part of the new parity chunk to pre-compute is strategically chosen such that the portion of data blocks to read during transcode is physically contiguous on disk, significantly improving IO efficiency. This approach is based on the "hop-and-couple" optimization for reconstruction-efficient vector codes introduced in [45]. For example, when transcoding from CC(6,7) to CC(12,14), our design ensures that disks retrieve a single contiguous chunk of 4 MB, as opposed to retrieving 8 disjoint chunks of a half MB from each of the 12 data blocks. More details about CC vector codes are provided in Appendix A.

7 Evaluation

Morph significantly reduces disk+network IO and storage capacity for ingest+transcode. This section shows that (1) Morph reduces storage overhead and IO requirements for file creation (ingest) and lifetime transitions, (2) Morph's hybrid redundancy does not sacrifice on client read/write performance and latency compared to 3-r, (3) applying Convertible Codes achieves faster, more IO-efficient transcode.

Experimental setup. Our evaluation includes both measurements on a small academic cluster and benefits extrapolated from production cluster traces at Google. The experimental evaluations were run on a 29-node cluster of one Namenode, 23 Datanodes, and five client nodes. For the *baseline* DFS, we use unchanged HDFS [49] v3.3.1, whereas Morph uses the same HDFS version enhanced as described in Sections 3–6. Each machine has a Quad-Core AMD Opteron Processor and 128 GB RAM. Each Datanode has a 1 TB Ext4 file system atop a 7200 RPM Hitachi Ultrastar HDD. Each client node spawns up to 8 worker threads to generate read/write traffic. The nodes in the cluster are connected via a 40 GbE network. We use DFS-perf [22] to introduce artificial client load into the cluster and measure client performance, blktrace [6] and seekwatcher [35] to measure block device IO and cluster throughput, and Ganglia [36] to aggregate per node memory and compute usage.

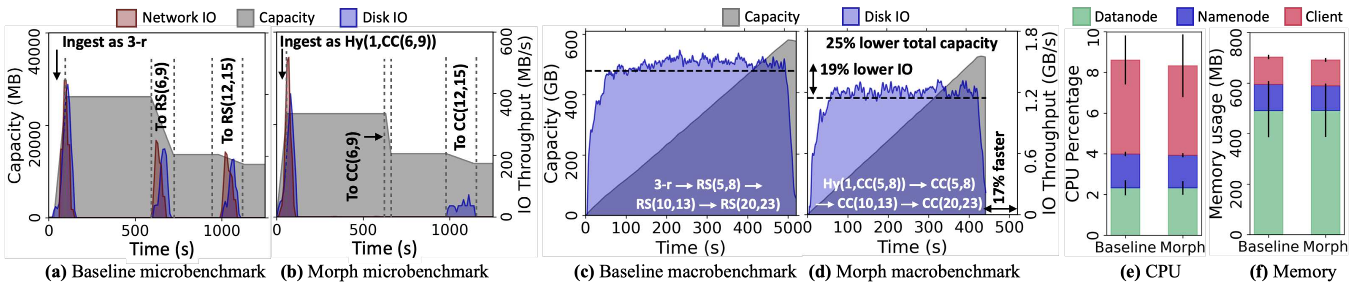
7.1 Morph makes lifetime transitions seamless

This section quantifies the end-to-end savings in IO and storage overhead. We show both microbenchmarks and macrobenchmarks evaluated in our cluster followed by an analysis of potential savings for workload traces captured in production at Google.

Morph lifetime savings. First, we use a microbenchmark to quantify the capacity and disk/network IO overheads of ingest+transcode for a single 8 GB file following a similar transition path as the data illustrated in Fig. 2.

Fig. 11a shows the capacity (left Y-axis), disk IO (blue) and network IO (red) overheads in baseline HDFS where data is ingested as 3-r, resulting in 24 GB of disk usage. Ten minutes after ingest, the data is transcoded from 3-r to RS(6,9) by reading and then writing the data in EC form with its new parities. The data is subsequently transcoded 15 minutes later to RS(12,15) in the same RRW process. Across the entire lifetime of the file (ingest+transcode), we measure the total network+disk IO to be 124 GB, an IO amplification of 15.5×.

Fig. 11b shows the same lifetime transitions but in Morph where the data is instead ingested in Hy(1, CC(6, 9)). During ingest, Morph incurs a storage overhead of 150% compared to baseline's 200%—a 25% reduction. The first transcode from Hy(1, CC(6, 9)) to CC(6,9) is free as it only consists of deleting the replica, thus resulting in zero disk or network IO. In the subsequent transcode to CC(12,15), Morph uses Convertible Codes to merge parities of stripes locally on Datanodes



(a) Baseline microbenchmark (b) Morph microbenchmark (c) Baseline macrobenchmark (d) Morph macrobenchmark (e) CPU (f) Memory

Figure 11. Micro and macrobenchmarks evaluated in a cluster with 29 nodes comparing capacity, IO throughput, and compute/memory in baseline HDFS (3-r ingest + RRW transcoding) against Morph (hybrid ingest + CC transcodings). Figures (a) and (b) are microbenchmarks showing 8 GB ingest transcoded to EC(6,9) and then EC(12,15). Morph with Hy(1, CC(6, 9)) ingest (b) is best with 25% storage overhead reduction and 55% network and 58% disk IO reduction. Figures (c) and (d) are macrobenchmarks showing continuous ingest+transcode of files going through 3 transitions in their lifetime. For the same amount of work, Morph finishes 17% faster, requires 19% lower disk IO throughput, and uses 25% lower capacity compared to baseline. Figures (e) and (f) compare macrobenchmark compute and memory usage respectively. Morph reduces resource usage in both dimensions via more efficient transitions.

without incurring network IO, as described in Sec. 5.3. Across the entire file lifetime, we measure the total network+disk IO to be 54 GB, an IO amplification of 6.75×. In comparison to baseline, Morph achieves a 58% and 55% reduction in disk and network IO bandwidths respectively.

Savings in a steady-state ingest+transcode workload. We evaluate Morph's benefits and performance by simulating continuous ingest+transcode traffic in an academic cluster. After ingestion, each file is transcoded first to EC(5,8), then to EC(10,13), and finally to EC(20,23). For the baseline, we configure the cluster to ingest at a rate of ∼1100 MB/s (∼45% cluster load) and transcode at a rate of ∼300 MB/s (∼12% cluster load). Recent studies indicate that ∼56% cluster utilization is generally representative of average datacenter load [44], and we choose the same relative transcode to ingest rate based on Google's production cluster as shown in Fig. 1. In Morph, we ingest and transcode the same number of files as the baseline to compare the two systems executing the exact same work.

Fig. 11c and Fig. 11d show disk IO and capacity for baseline and Morph respectively. Since no files are deleted, we observe capacity rising for the duration of the experiment in both systems. However, Morph achieves lower capacity overheads due to ingesting data as Hy(1, CC(5, 8)) as opposed to 3-r, resulting in a total storage overhead reduction of 25%. Furthermore, Morph achieves better write and transcode latencies (plots omitted due to space) than the baseline. The p50 and p90 transcode latency is 7× higher in the baseline, and Morph demonstrates a (∼15–20%) reduction in write latency. We attribute this improvement to the overall decrease in background activity from transcoding traffic. In particular, transcoding from Hy(1, CC(5, 8)) to CC(5,8) is *free*, CC(5,8) to CC(10,13) requires 50% less IO, and CC(10,13) to CC(20,23) requires 72% less IO than baseline. Moreover, the average disk IO bandwidth needed for Morph for the duration of the workload is 19% lower. Finally, without any compromise in reliability or performance, Morph executes the macrobenchmark 17% faster than the baseline.

Fig. 11e and Fig. 11f show average CPU and memory usage broken down across Namenode, Datanode, and client node for the steady-state ingest+transcode workload. First, we observe the client averages less total compute resources in Morph as the client handles transcode via RRW in baseline, while Morph uses the explicit transcode interface exposed by the Namenode. Despite the system handling transcode in Morph, we still observe a slight decrease in compute in both the Namenode and Datanodes attesting to the efficiency and simplicity of the transition process. In our experiment, the Datanode processes for both baseline and Morph were allocated 512 MB of battery-backed buffer cache to enable fast acknowledgements back to the client. Despite Morph using the buffer cache to achieve fast hybrid writes as described in Sec. 4.2, we find that Morph achieves all its benefits without allocating additional memory.

Savings in production workloads. We analytically calculate the benefits of Morph using traces from large-scale production storage clusters at Google. Fig. 12 shows a month-long trace at hour-granularity of ingest+transcode IO traffic from two services, each with PB-scale IO footprints. Note that Service A is the same application as shown in Fig. 1. Data in both services is ingested as 3-r, as it is hot and latency-sensitive to both reads and writes (subplot A of Services A, B). As data cools, Service A transcodes either into a narrow RS code (approximately 15-wide) or a medium-width LRC (approximately 40-wide), depending on the type of file and its desired performance / storage-overhead tradeoffs[2]. Once the data written in narrow RS cools down further, it is transcoded into medium LRC. Finally, data that is frigid is transcoded from medium LRC to wide LRC (approximately 60-wide). Just the transcoding operations zoomed in are shown in subplots C of Services A, B. Each transition is performed via RRW. Service B only transcodes once after ingest, as it goes directly from 3-r to a very wide LRC (approximately 80-wide), shown in the A and C subplots of Service B.

[2]Note that which files are to be transcoded into narrow RS versus medium LRC is chosen by the application and not in the purview of Morph.

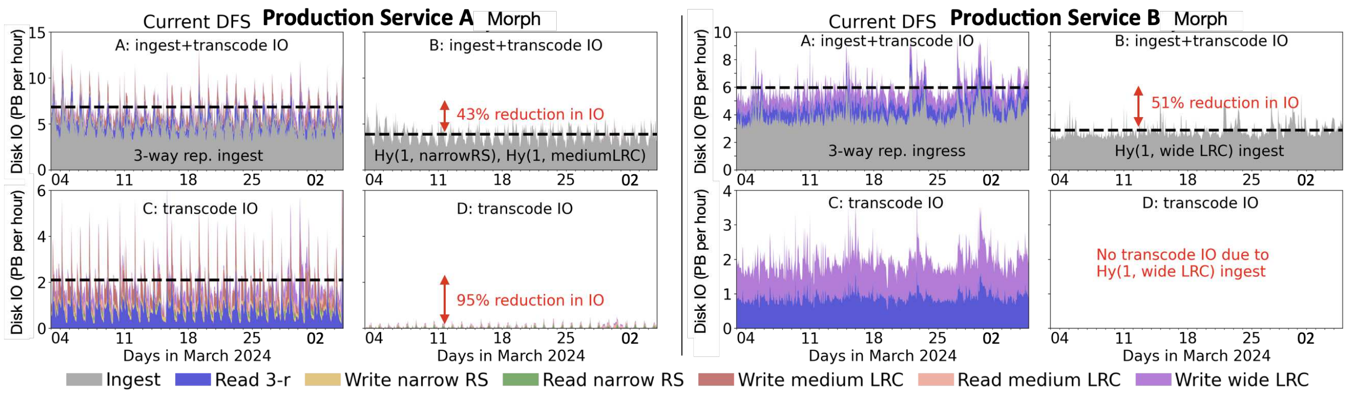


Figure 12. Lifetime transitions for a month-long trace from each of two large services in production exascale storage clusters at Google.

Subplots B and D of Fig. 12 (Services A, B) show the ingest+transcode IO that would be incurred in Morph using aggregate calculations across stripes. During ingest, depending on whether the first transcoding step is to the narrow RS or medium LRC, Morph accordingly can write data in Hy(1,narrow CC) or Hy(1,medium CC) in the case of Service A, and Hy(1,wide LRCC) in the case of service B. As shown above, Morph would incur a lower ingest storage overhead, which in this case would be 20% for Service A and 28% for Service B. Since the first transcoding step incurs no IO in case of Morph, transcoding to narrow, or medium CC for Service A, and wide LRCC for Service B would be free. As there are no more transitions for Service B, the transcoding IO incurred by Morph would be zero (shown in subplot D of Service B). CC aids transitions from narrow-to-medium and medium-to-wide CC in Service A would incur an IO cost, but as shown in subplot D of Service A, the disk IO throughput is 95% lower compared to today's DFS. Overall, Service A would reduce its disk IO throughput for all transitions for a month by 43% on average, whereas Service B would reduce disk IO throughput by 51%, implying huge efficiency gains for petascale and exascale storage clusters.

7.2 Hybrid redundancy matches or outperforms 3-r

We now compare client write and read performance across various hybrid schemes to 3-r and RS in an experimental setup. We vary the number of client threads, *t*, across our experiments to determine performance at low (t=12), medium (t=25), and high load (t=40). We find that hybrid redundancy achieves effectively the same (or at times better) performance, attesting to its ability to replace 3-r for early-life hot data.

Measuring hybrid write performance. Early-life data is often appended and requires low write latency, as described in Sec. 4.2. Fig. 13a shows write latency across four ingest options for 8 MB files. Notably, both types of hybrid ingest perform almost identically to 3-r with less than a 2% slowdown at the median, whereas direct writes to RS(6,9) are significantly slower (6× slowdown). For a small write (≤8 MB) workload, the client must wait for RS(6,9) to compute and write out three additional parities on its critical

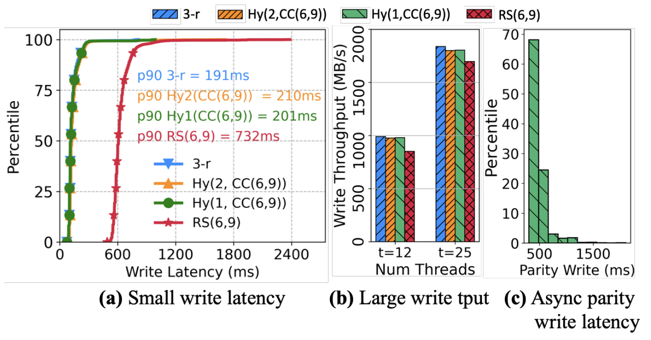


Figure 13. Write latency (a) for small writes (8 MB), throughput (b) for large streaming writes (120 MB) across various ingest methods, and time to persist parities (c) asynchronously in hybrid writes. In all cases, both forms of hybrid achieve the same ingest performance as 3-r while RS sees a significant decline in performance.

path. Meanwhile hybrid writes, similar to 3-r, can quickly return back to the client after the three copies are propagated and handle any EC asynchronously.

We also compare write throughput across the ingest options for large file (>100 MB) writes in Fig. 13b. We observe both hybrid writes (1-copy and 2-copy) perform as well as 3-r with less than a 2% decrease in write throughput at medium load (25 threads) and a 1% decrease at low load (12 threads). Both hybrid schemes also see a 6% increase in throughput compared to RS(6,9). We attribute this moderate increase to the durability guarantees provided by HDFS EC. EC writes puts parities on the critical path but does not compute them until the end of the stripe, thereby leaving data in an incomplete stripe unprotected. Hybrid redundancy does not face any durability issues with incomplete stripes as it leverages redundant replicas.

Parity computation and persistence can play a crucial role in affecting hybrid write latency. Specifically, parities must be persisted quickly enough to delete the temporary replica before it gets flushed to disk. Fig. 13c shows the distribution of time taken to persist parities in a write-only load. We measure 95% of parities are written within 500ms of the first write to the stripe, confirming that the additional replicas are not necessarily kept in buffer-cache for a long time.

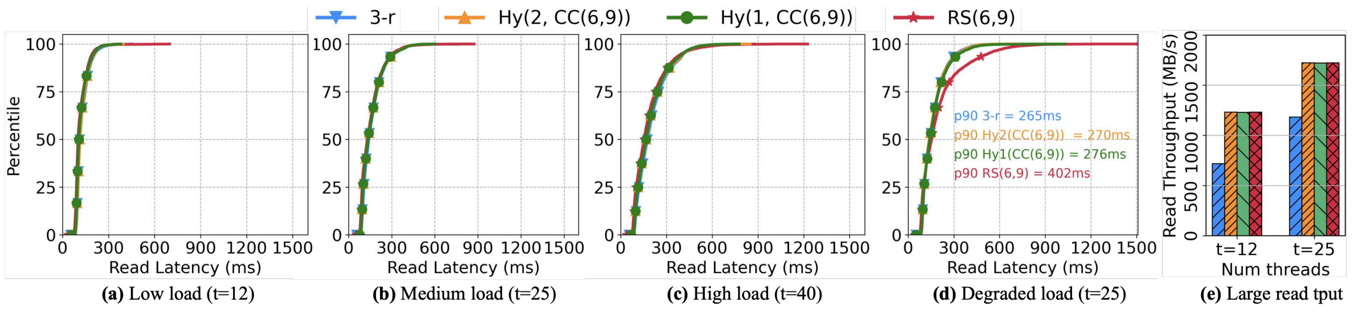


Figure 14. Read latency for small reads (8 MB) across varying loads (a-c) and with nodes in the cluster down (d). And (e) read throughput for large stripe-spanning reads (48 MB). Hybrid and 3-r read latencies are similar in all conditions while RS performs worse in degraded mode. Read throughput for large reads improves as a result of better stripe parallelism.

Measuring hybrid read performance. We now compare the latency of client reads between 3-r, Hy(1, CC(6, 9)), Hy(2, CC(6, 9)), and RS(6,9). Reads are "hedged" such that a second parallel request to a replica is made at p95 latency, which is a standard read optimization tactic. In the case of Hy(1, CC(6, 9)), the read is directed to the CC stripe if the replica does not respond in time. Fig. 14a-d compares 8 MB read latency for different redundancy options in varying scenarios. Specifically, Fig. 14a, Fig. 14b, Fig. 14c show performance at 30%, 55%, and 80% cluster load respectively. We observe no significant difference in performance across all benchmarks between any of the redundancy options. We note, however, that the tail of RS extends much further as a result of only having a single chunk to read from. The other schemes are capable of hedge-reading from an alternate copy while RS is forced to either wait or do a degraded mode read.

For Fig. 14d, we measure read latency when 10% of the nodes in the cluster are down to observe performance in an extreme scenario with several degraded mode reads. Both Hy(1, CC(6, 9)) and Hy(2, CC(6, 9)) perform the same as 3-r at the median and only see a 2% and 4% increase in p90 latency respectively, despite Hy(1, CC(6, 9)) performing a striped read 10% of the time and a degraded mode read for 5% of the striped reads. RS(6,9), however, observes a palpable decrease in performance towards the tail (52% increase in latency) as a result of read-amplification and EC compute during degraded mode reads.

For throughput, hybrid reads can perform parallel reads from the stripe, rather than the replica(s), especially in the case of streaming large reads as described in Sec. 4.3 and Sec. 6. In Fig. 14e, we show the throughput benefits of Morph automatically prioritizing striped reads instead of replica reads for large stripe spanning reads (48 MB). With 12 threads, the read throughput increases by 71% compared to 3-r, and with 25 threads the increase is 46%, as the increase is bottlenecked by disk bandwidth.

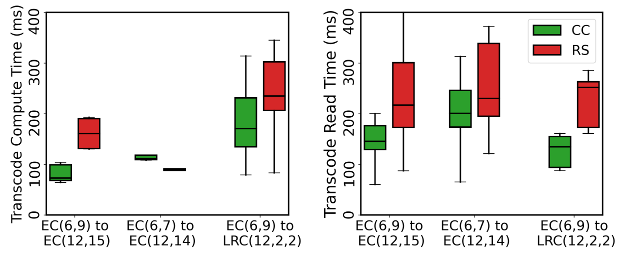
7.3 Convertible Codes minimize transcode IO

This section evaluates a practical implementation of Convertible Codes [37–39] and the latency and bandwidth benefits from applying these codes to mid-to-late life transitions. We draw several comparisons between CC and RS codes; recall that the latter must read all data chunks to compute new parities.

Fig. 15 compares RS and CC across two key metrics: latency to compute new parities (Fig. 15a) and latency to read the corresponding data necessary for transcoding (Fig. 15b). We choose three different scenarios: (1) when the number of parities stay constant (e.g., EC(6,9) to EC(12,15)), (2) when the number of parities increase (e.g., EC(6,7) to EC(12,14)) and (3) for a non-LRC to LRC transcoding (e.g., EC(6,9) to LRC(12,2,2)). We measure transcode latencies when twenty 96 MB files are transcoded in parallel.

First, when transcoding from EC(6,9) to EC(12,15), the median computational latency reduces by 50% when compared to RS, which can be attributed to halving the computation matrix (12-wide in RS compared to 6-wide in CC). We also observe a 40% reduction in read latency. Transcoding with CC requires much fewer concurrent disk reads and network transfers (6 parities in CC as opposed to 12 data chunks in RS). Second, when transcoding from EC(6,7) to EC(12,14), CC waits for data from 14 disks and network transfers (as opposed to 12 in RS), but reads 33% less data overall. This is reflected in a 17% drop in read latency, albeit at the expense of longer compute latency. The compute latency increases because the process involves separating out the pre-computed fraction of parities from existing parities. This is an acceptable trade-off since disk bandwidth is increasingly becoming a more contested resource, as shown in Fig. 5.

Third, we show transcoding from EC(6,9) to an LRC EC(12,16) with 2 local groups. Two initial stripes are combined to form a final stripe. The first parity of the two stripes become the local parities after transcoding, while the remaining two parities get converted to global parities. Similar to the first case, CC achieves 30% improvement in read latency, and about 50% improvement in compute latency. Further, we confirm that encode(write)/decode(reconstruction) *compute* latencies stay the same as RS when parities per stripe don't change. When parities change in the second case, CC encoding is 80% slower and decode is 150% slower, since it involves pre-computing a fraction of parity block during encode and separating it out during decode. However, note that compute latencies are



(a) Compute latency **(b)** Read latency

Figure 15. Comparison of compute and read latencies for transcode using CC and RS for three cases: when (A) parities stay same, (B) parity increases by 1, (C) EC to LRC. In A and C, we observe 30–40% reduction in read latency and compute latency cut by $\approx$ 50%. In B, separating the pre-computed fraction of parities delays computation of parities, but reduces read latency by $\approx$ 17% and read bandwidth by $\approx$ 25%.

negligible, when compared to read/write operations to/from Datanodes, and have minimal effect on overall performance.

The above examples show the common case of transcoding where multiple stripes *merge* to form a single stripe. However, note that CC offers significant advantages when transcoding to any general scheme. For instance, we evaluate transcoding EC(6,9) stripes to k-wide stripes for all cases where $6 < k <= 30$. CC achieves 45% IO reduction on average (33% in the worst case) when parities per stripe stay the same, and 20% IO savings on average (12.5% in the worst case) when we add an extra parity. A full exploration of these benefits is provided in Appendix A.

8 Related Work

Large-scale cluster file systems have long relied on redundancy for fault tolerance, starting with replication being the primary means of redundancy [8, 18]. Subsequently, academic research [5, 19, 24] and industry [13, 25, 27, 48, 53, 57] cluster file systems have integrated erasure codes into the storage system to achieve space-efficient redundancy. These are the common options in modern cluster storage.

A hybrid storage scheme using a combination of erasure-coding and replication has been proposed for a distributed database system called ER-Store [32], as one of three storage options used for tablets depending on their data temperature. Morph and its hybrid redundancy scheme differs from ER-store in several important ways: (1) ER-Store's hybrid scheme is used for "warm" storage, with 3-way replication used for "hot" data, so ER-Store does not provide a protocol for fully durable, low-latency ingest directly into it; ingesting into hybrid redundancy requires a protocol like Morph's and is the source of significant benefit in DFSs; (2) ER-Store transitions between storage options by re-writing tablets, like is done in traditional DFSs, as opposed to the IO-efficient approaches enabled and used in Morph; (3) ER-Store provides no support for changing EC scheme parameters, which is common throughout a file's lifetime.

The idea of transitioning between codes in a cluster has been considered in previous works. The paper [55] considers transitioning between two specific codes, and thus not general (any-to-any). A recent line of work has studied how disk failure rates vary over time and proposed changing EC parameters accordingly [30]. In this context, multiple systems have been proposed to reduce the IO spikes of such transitions [28, 29]. However, the total IO consumed in those systems is still high due to the need to read all the data chunks. The DFS-native transcode service in Morph uses CC and LRCC to avoid having to read all the data chunks and help in reducing the total IO load in these systems.

StripeMerge [56] supports merging two narrow stripes of a carefully designed k-of-n code to generate one wider stripe of a $2k$-of-n' code, while keeping the number of parities fixed. Morph differs from StripeMerge in three important ways: (1) Morph is a DFS that supports transcoding of individual files while maintaining the usual practice of restricting EC stripes to data within file, for security and complexity reasons. StripeMerge assumes that any two stripes in the system could be merged, and so searches throughout the cluster for ones with data on separate disks, whereas Morph explicitly places file data and parity to achieve this within files. (2) StripeMerge supports only the one merge scenario (described above), whereas Morph supports efficient general transcoding (any to any). (3) Morph is implemented in and compared with HDFS, handling all the metadata, sequencing, and crash consistency issues.

9 Conclusion

Morph introduces new redundancy and placement schemes to reduce file ingest and transcode overheads in DFSs. Morph's new hybrid redundancy reduces ingest and early-life transcode IO, and Convertible Codes reduce subsequent EC-to-EC transcode IO. For large Google data services, Morph's techniques would reduce transcode IO by over 95% and total ingest+transcode IO by 40–50%, while also reducing capacity required for newly ingested data by about 20%—without compromising on performance or data reliability.

10 Acknowledgements

We thank our shepherd Kang Chen and the anonymous reviewers for their invaluable feedback. We extend special thanks to Larry Greenfield, Mustafa Uysal, and numerous other researchers and engineers at Google. This research is generously supported in part by NSF grants CNS1956271, CNS1901410, and CAREER 1943409, by a Sloan Foundation Fellowship, and by a VMware Systems Research Award. We also thank the members and companies of the PDL consortium (Amazon, Google, Hitachi, Honda, HPE, IBM, Intel, Jane Street, Meta, Microsoft, NetApp, Oracle, Pure Storage, Salesforce, Samsung, Seagate, Two Sigma, Western Digital) and VMware for their interests, insights, feedback, and support.

References

[1] How Lasers Could Unlock Hard Drives With 10 Times More Data Storage. https://www.popularmechanics.com/technology/a20078/heating-magnets-lasers-could-be-the-key-magnetic-recording/.

[2] Seagate: HAMR is nailing it – no looming 20TB to 30TB capacity problem. https://blocksandfiles.com/2021/09/24/seagate-hamr-on-course-no-looming-20-to-30tb-capacity-problem/.

[3] Seagate Reveals HAMR HDD Roadmap: 32TB First, 40TB Follows. https://www.tomshardware.com/news/seagate-reveals-hamr-roadmap-32-tb-comes-first.

[4] HDD User Benchmarks. http://hdd.userbenchmark.com/, (accessed July 5, 2023).

[5] Thomas E Anderson, Michael D Dahlin, Jeanna M Neefe, David A Patterson, Drew S Roselli, and Randolph Y Wang. Serverless network file systems. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 109–126, 1995.

[6] Alan D Brunelle. Block I/O layer tracing: blktrace. *HP, Gelato-Cupertino, CA, USA*, 57, 2006.

[7] Han Cai, Ying Miao, Moshe Schwartz, and Xiaohu Tang. A construction of maximally recoverable codes with order-optimal field size. *IEEE Transactions on Information Theory*, 68(1):204–212, 2021.

[8] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows Azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

[9] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

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

[11] Facebook. VAST Erasure Coding. https://vastdata.com/blog/introducing-rack-scale-resilience, 2020.

[12] FAST. Ec tutorial. https://web.eecs.utk.edu/~jplank/plank/papers/2013-02-11-FAST-Tutorial.pdf, 2013.

[13] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[14] Apache Software Foundation. Hdfs erasure coding. *URL https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html*, 2023.

[15] Apache Software Foundation. Hdfs architecture. *https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html*, 2024.

[16] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[17] S. Ghemawat, H. Gobioff, and S.T. Leung. The Google File System. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[19] Garth A Gibson, David F Nagle, Khalil Amiri, Jeff Butler, Fay W Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *ACM SIGOPS operating systems review*, 32(5):92–103, 1998.

[20] Parikshit Gopalan, Cheng Huang, Bob Jenkins, and Sergey Yekhanin. Explicit maximally recoverable codes with locality. *IEEE Transactions on Information Theory*, 60(9):5245–5256, 2014.

[21] Parikshit Gopalan, Cheng Huang, Huseyin Simitci, and Sergey Yekhanin. On the locality of codeword symbols. *IEEE Transactions on Information theory*, 58(11):6925–6934, 2012.

[22] Rong Gu, Qianhao Dong, Haoyuan Li, Joseph Gonzalez, Zhao Zhang, Shuai Wang, Yihua Huang, Scott Shenker, Ion Stoica, and Patrick PC Lee. DFS-PERF: A scalable and unified benchmarking framework for distributed file systems. *EECS Dept., Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016-133*, 2016.

[23] Venkatesan Guruswami, Lingfei Jin, and Chaoping Xing. Constructions of maximally recoverable local reconstruction codes via function fields. *IEEE Transactions on Information Theory*, 66(10):6133–6143, 2020.

[24] John H Hartman and John K Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems (TOCS)*, 13(3):274–310, 1995.

[25] Dean Hildebrand and Denis Serenyi. Colossus under the hood: a peek into Google's scalable storage system. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system, 2021.

[26] Cheng Huang, Minghua Chen, and Jin Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *ACM Transactions on Storage (TOS)*, 2013.

[27] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. Erasure Coding in Windows Azure Storage. In *USENIX Annual Technical Conference (ATC)*, 2012.

[28] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, KV Rashmi, and Gregory R Ganger. Tiger: Disk-Adaptive redundancy without placement restrictions. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 413–429, 2022.

[29] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, KV Rashmi, and Gregory Ganger. Pacemaker: Avoiding heart attacks in storage clusters with disk-adaptive redundancy. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, 2020.

[30] Saurabh Kadekodi, K V Rashmi, and Gregory R Ganger. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity. In *USENIX File and Storage Technologies (FAST)*, 2019.

[31] Saurabh Kadekodi, Shashwat Silas, David Clausen, and Arif Merchant. Practical Design Considerations for Wide Locally Recoverable Codes (LRCs). In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 1–16, 2023.

[32] Zijian Li and Chuqiao Xiao. Er-store: A hybrid storage mechanism with erasure coding and replication in distributed database systems. *Scientific Programming*, 2021:1–13, 09 2021.

[33] Gaojun Luo and Xiwang Cao. Constructions of optimal binary locally recoverable codes via a general construction of linear codes. *IEEE Transactions on Communications*, 2021.

[34] David JC MacKay. Fountain codes. *IEE Proceedings-Communications*, 152(6):1062–1068, 2005.

[35] Chris Mason. Seekwatcher. *URL http://oss. oracle. com/˜ mason/seekwatcher*, 2008.

[36] Matt Massie, Bernard Li, Brad Nicholes, Vladimir Vuksan, Robert Alexander, Jeff Buchbinder, Frederiko Costa, Alex Dean, Dave Josephsen, Peter Phaal, et al. *Monitoring with Ganglia: tracking dynamic host and application metrics at scale.* " O'Reilly Media, Inc.", 2012.

[37] Francisco Maturana, VS Chaitanya Mukka, and KV Rashmi. Access-optimal linear mds convertible codes for all parameters. In *2020 IEEE International Symposium on Information Theory (ISIT)*, pages 577–582. IEEE, 2020.

[38] Francisco Maturana and KV Rashmi. Convertible codes: new class of codes for efficient conversion of coded data in distributed storage. In *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[39] Francisco Maturana and KV Rashmi. Bandwidth cost of code conversions in the split regime. In *2022 IEEE International Symposium on Information Theory (ISIT)*, pages 3262–3267. IEEE, 2022.

[40] Francisco Maturana and KV Rashmi. Convertible codes: Enabling efficient conversion of coded data in distributed storage. *IEEE Transactions on Information Theory*, 68(7):4392–4407, 2022.

[41] Francisco Maturana and KV Rashmi. Bandwidth cost of code conversions in distributed storage: Fundamental limits and optimal constructions. *IEEE Transactions on Information Theory*, 2023.

[42] Francisco Maturana and KV Rashmi. Locally repairable convertible codes: Erasure codes for efficient repair and conversion. In *2023 IEEE International Symposium on Information Theory (ISIT)*, pages 2033–2038. IEEE, 2023.

[43] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231, 2021.

[44] Phitchaya Mangpo Phothilimthana, Saurabh Kadekodi, Soroush Ghodrati, Selene Moon, and Martin Maas. Thesios: Synthesizing accurate counterfactual i/o traces from i/o samples. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 1016–1032, New York, NY, USA, 2024. Association for Computing Machinery.

[45] K V Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. *ACM Special Interest Group on Data Communication (SIGCOMM)*, 2014.

[46] Korlakai Vinayak Rashmi, Nihar B Shah, and P Vijay 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, 2011.

[47] KV Rashmi, Nihar B Shah, and Kannan Ramchandran. A piggybacking design framework for read-and download-efficient distributed storage codes. *IEEE Transactions on Information Theory*, 63(9):5802–5820, 2017.

[48] Philip Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[49] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. The Hadoop distributed file system. In *IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2010.

[50] Natalia Silberstein, Ankit Singh Rawat, O Ozan Koyluoglu, and Sriram Vishwanath. Optimal locally repairable codes via rank-metric codes. In *2013 IEEE International Symposium on Information Theory*, pages 1819–1823. IEEE, 2013.

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

[52] Itzhak Tamo, Dimitris S Papailiopoulos, and Alexandros G Dimakis. Optimal locally repairable codes and connections to matroid theory. *IEEE Transactions on Information Theory*, 62(12):6661–6671, 2016.

[53] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the Panasas parallel file system. In *FAST*, volume 8, pages 1–17, 2008.

[54] Stephen B Wicker and Vijay K Bhargava. *Reed-Solomon codes and their applications.* John Wiley & Sons, 1999.

[55] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. A tale of two erasure codes in HDFS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 213–226, Santa Clara, CA, February 2015. USENIX Association.

[56] Qiaori Yao, Yuchong Hu, Liangfeng Cheng, Patrick PC Lee, Dan Feng, Weichun Wang, and Wei Chen. Stripemerge: Efficient wide-stripe generation for large-scale erasure-coded storage. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 483–493. IEEE, 2021.

[57] Zhe Zhang, Amey Deshpande, Xiaosong Ma, Eno Thereska, and Dushyanth Narayanan. Does erasure coding have a role to play in my data center. *Microsoft research MSR-TR-2010*, 52, 2010.

Supplemental material for Morph

The appendix is not peer-reviewed.

A Convertible Codes

The theoretical framework of Convertible Codes (CC) [37, 40, 41] provides broad constructs for transcoding regimes. (1) **Merge**: where multiple EC stripes are combined to create a single EC stripe of a target scheme, i.e, from k-of-n^I to a λk-of-n^F scheme. This is the most optimal regime, where none of the the data blocks need to be read in the best case. This is particularly effective since most transcoding events are to wider schemes in cluster storage systems to increase space efficiency. Fig. 7 shows an example of this case of transcoding. (2) **Split**: where a single EC stripe is divided into multiple EC stripes of a target scheme, i.e., from a λk-of-n^I scheme to a k-of-n^F scheme. This regime offers less impressive transcoding IO savings when compared to merge regime, as most of the data chunks still need to be read from. However, we encounter this regime less often in practice. (3) **General**: where any general set of parameters is supported by using merge and split regimes as building blocks.

Convertible codes can be categorized into two sub-classes based on the number of parities in a stripe in the initial (r^I) and final (r^F) EC scheme: (1) when they stay the same ie. $r^I = r^F$, where optimal transcode overheads can be achieved with scalar codes, and (2) when $r^I \neq r^F$, which require the use of vector codes. While scalar codes are easier to implement and have lower computational complexity, construction of vector codes is more involved, and come with higher computational complexity. We provide background on both sub-classes below.

Case 1 - $r^I = r^F$: When number of parities per stripe remain same, we employ Access-Optimal Convertible Codes[37]. The most dramatic savings are observed in the *merge regime* where data is transcoded from a k-of-n^I scheme to a λk-of-n^F scheme. We portrayed this in the example in Fig. 7. No data blocks have to be read during transcoding.

In the *split regime*, where data is transcoded from a λk-of-n^I scheme to a k-of-n^F scheme, CC is able to transcode by reading the r^I parity blocks and only the first $(\lambda - 1)k$ blocks of the initial λk data blocks, as shown in Fig. 16. On transcoding from EC(12,14) to EC(4,6), each of the four data blocks that would make up the first two final stripes are read to compute the corresponding parities, similar to transcoding with RS-encoding. For computing the parities for the third stripe, instead of reading all of the remaining four data blocks, we read the two initial parities, and the eight data blocks that were already read. As a result, we read a total of 10 blocks instead of 12.

A combination of the above two regimes is used to transcode stripes in the *general regime.* For example, to transcode from EC(6,9) to EC(15,18), 5 EC(6,9) initial stripes are combined to form 2 EC(15,18) final stripes. While only parities are read for

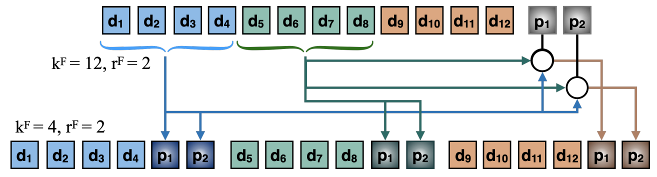


Figure 16. Access-optimal split: Splitting one stripe of EC(12,14) into three stripes of EC(4,6)

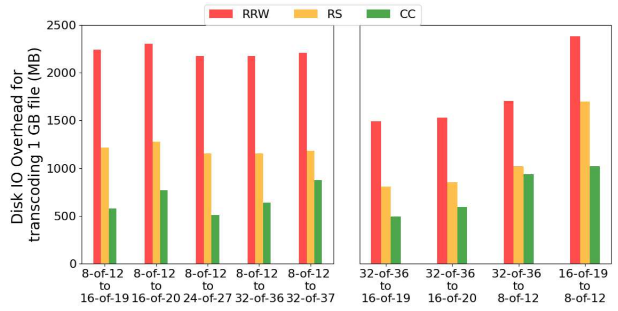


Figure 17. Disk IO/bandwidth overhead for transcoding 1 GB files for different initial and target EC schemes across merge(left) and split(right) regimes and the two sub-classes $r^I = r^F$ and $r^I \neq r^F$. The best gains come in the merge regime when number of parities stay constant.

four of the initial stripes, all the data blocks of the third stripe need to be read. As a consequence, CC achieves transcoding with 40% less IO overhead.

Case 2a - $r^I < r^F$: When the number of parities per stripe increases during transcoding, the inherent advantages of the Convertible Codes' construction discussed above vanish. This is because, the information for $(r^F - r^I)$ new parities do not exist in any of the initial parities. In these scenarios, we employ another class of Convertible Codes, Bandwidth-Optimal Convertible Codes (BWO-CC) [41], which extend the capabilities of Convertible Codes described in case 1, to minimize the total volume of data read.

An example is illustrated in Fig. 8. Although a higher number of blocks are read from (higher "access cost"), the total data downloaded is minimized (lower "bandwidth cost"). Contrary to scalar codes such as Reed-Solomon, BWO-CC operate as a vector code. Here, each EC symbol is represented as a vector or an array of data chunks. This design allows embedding additional information within segments of the parity vectors - called *piggybacking* [47], which can be later decoded for efficient transcoding.

Each data block is (logically) divided into r^F chunks resulting in r^F substripes. During the encoding process, despite requiring only r^I parities initially, all r^F parities are computed for the first r^I substripes. The parities are derived using the Access-Optimal CC construction described in case 1. These are then added to each of the r^I parities of the remaining $r^F - r^I$ substripes. Upon eventual transcoding of the file, only the data from the last $r^F - r^I$ substripes is read along with the parities. The efficiency arises because the new parity data for the initial r^I sub-stripes can be decoded using

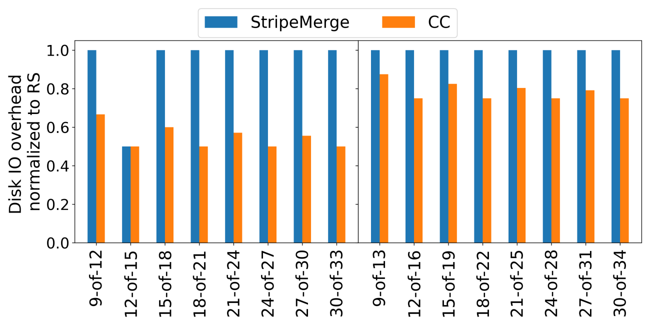


Figure 18. Disk IO overheads normalized to the baseline RS, when transcoding from 6-of-9 to schemes on k-of-n schemes on X-axis. Parities stay constant in the left plot, while they increase by 1 on the right. The transcode IO savings with CC vary, but significant when compared to RS and Stripe Merge across any general target scheme. In contrast, StripeMerge only helps in one scenario 6-of-9 to 12-of-15.

the rest of the retrieved data. This process is shown in the example in Fig. 8 (in the main paper).

Case 2b - $r^I > r^F$: When the number of parities per stripe decreases during transcoding, Access-Optimal CC is sufficient to provide optimal transcode IO reductions for the merge regime. However, for the *split* regime[39], i.e. when the initial stripe is broken into integral number of smaller stripes, use of vector codes as discussed in case 2a is needed to reduce the IO overheads. Since the split regime construction serves as a building block for the general regime, in this case, the general regime also benefits from Piggybacking.

Analysis and Comparison. Fig. 17 shows the disk and network bandwidth used for transcoding in the merge regime (left) and split regime (right). In the merge regime, we calculate the read bandwidth to transcode from 8-of-12 ($r^I = 4$) to various schemes with $16 \geq k^F \geq 32$ and $3 \geq r^F \geq 5$.

The best gains are observed in the merge regime when $r^I = r^F$, with $> 50\%$ IO reduction when compared to RS in the examples provided. These benefits scale with the initial stripe width. For instance, merging two stripes of EC(17,20) to form a stripe of EC(34,37) saves $>$ 80% of bandwidth overhead when compared to RS. The gains are lower for the cases when $r^I < r^F$. Transcoding from 16-of-19 to 8-of-12, Convertible Codes utilizes 40% lower bandwidth than RS. A similar pattern emerges with a 26% bandwidth reduction when transcoding from 8-of-12 to 32-of-37.

Fig. 18 illustrates how CC fares against RS and StripeMerge in the general regime, while transcoding from EC(6,9) to EC with k-wide stripes, where $6 < k <= 30$. CC achieves 45% IO reduction on average (33% in the worst case) when parities per stripe stay the same (left), and 20% IO savings on average (12.5% in the worst case) when we add an extra parity in the final stripe (right). In contrast, StripeMerge saves IO in only one instance, on transcoding to EC(12,15). CC therefore opens up a rich trade-off space for application developers to carefully balance transcode IO, durability and space efficiency.

Transcoding in late life: LRCC Here, the properties of CC are leveraged independently for computing both the local as well as the global parities. The best case occurs when multiple stripes of LRCCs are merged to create a single stripe. Here, each post-transcoding local group is created using an integral number of initial local groups using the efficient transcode operation of CC. The global parities of the stripe are computed using the global parities of the constituent initial stripes.

B Analytical estimation of degraded stripe read probability from a Hy$(1, \mathrm{CC}(k, n))$

We can analytically estimate the likelihood of a degraded mode read from a Hy$(1, \mathrm{CC}(k, n))$ scheme. Let us assume that the probability of a chunk missing is f. So, when $c = 1$, i.e., one replica and a CC(k,n) stripe, the probability of degraded mode read is $f \times \sum_{i=1}^{n-k} f^i \times (1-f)^{n-i}$, where the first term corresponds to the replicated chunk being unavailable and the second term corresponds to the data chunk from CC(k,n) being unavailable. To estimate an upper bound, suppose 1% of all chunks are simultaneously unavailable; $f = 0.01$,then say for a Hy$(1, \mathrm{CC}(6, 9))$ scheme, probability of degraded mode reads is 0.00009, a very rare scenario (tail-of-the-tail).