A Tale of Two Erasure Codes in HDFS

Mingyuan Xia, *McGill University;* Mohit Saxena, Mario Blaum, and David A. Pease, *IBM Research Almaden*

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

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

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

ISBN 978-1-931971-201

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

A Tale of Two Erasure Codes in HDFS

Mingyuan Xia*, Mohit Saxena†, Mario Blaum† and David A. Pease†
*McGill University, †IBM Research Almaden
mingyuan.xia@mail.mcgill.ca,{msaxena,mmblaum,pease}@us.ibm.com

Abstract

Distributed storage systems are increasingly transitioning to the use of erasure codes since they offer higher reliability at significantly lower storage costs than data replication. However, these codes tradeoff recovery performance as they require multiple disk reads and network transfers for reconstructing an unavailable data block. As a result, most existing systems use an erasure code either optimized for storage overhead or recovery performance.

In this paper, we present HACFS, a new erasure-coded storage system that instead uses *two different erasure codes* and dynamically adapts to workload changes. It uses a fast code to optimize for recovery performance and a compact code to reduce the storage overhead. A novel conversion mechanism is used to efficiently upcode and downcode data blocks between fast and compact codes. We show that HACFS design techniques are generic and successfully apply it to two different code families: Product and LRC codes.

We have implemented HACFS as an extension to the Hadoop Distributed File System (HDFS) and experimentally evaluate it with five different workloads from production clusters. The HACFS system always maintains a low storage overhead and significantly improves the recovery performance as compared to three popular single-code storage systems. It reduces the degraded read latency by up to 46%, and the reconstruction time and disk/network traffic by up to 45%.

1 Introduction

Distributed storage systems storing multiple petabytes of data are becoming common today [4, 25, 2, 15]. These systems have to tolerate different failures arising from unreliable components, software glitches, machine reboots, and maintenance operations. To guarantee high reliablity and availablity despite these failures, data is replicated across multiple machines and racks. For example, the Google File System [11] and the Hadoop Distributed File System [4] maintain three copies of each data block. Although disk storage seems inexpensive today, replication of the entire data footprint is simply infeasible at massive scales of operation. As a result, most large-scale distributed storage systems are transitioning to the use of erasure codes [3, 2, 15, 20], which provide higher reliability at significantly lower storage costs.

The trade-off for using erasure codes instead of replicating data is performance. If a data block is three-way replicated, it can be reconstructed by copying it from one of its available replicas. However, for an erasure-coded system, reconstructing an unavailable block requires fetching multiple data and parity blocks within the code stripe, which results in significant increase in disk and network traffic. Recent measurements on a Facebook's data warehouse cluster [19, 20] storing multiple petabytes of erasure-coded data, required a median of more than 180 Terabytes of data transferred to recover from 50 machine-unavailability events per day.

This increase in the amount of data to be read and transferred during recovery for an erasure-coded system results in two major problems: *high degraded read latency* and *longer reconstruction time*. First, a read to an unavailable block requires multiple disk reads, network transfers and compute cycles to decode the block. The application accessing the block waits for the entire duration of this recovery process, which results in higher latencies and degraded read performance. Second, a failed or decommissioned machine, or a failed disk results in significantly longer reconstruction time than in a replicated system. Although, the recovery of data lost from a failed disk or machine can be performed in the background, it severely impacts the total throughput of the system as well as the latency of degraded reads during the reconstruction phase.

As a result, the problem of reducing the overhead of recovery in erasure-coded systems has received signifi-

*Work done as an intern at IBM Research Almaden

cant attention in the recent past both in theory and practice [19, 2, 20, 15, 24, 3, 14, 26]. Most of the solutions tradeoff between two dimensions: storage overhead and recovery cost. Storage overhead accounts for the additional parity blocks for a coding scheme. Recovery cost is the total number of blocks required to reconstruct a data block after failure.

In general, most production systems use a *single erasure code*, which either optimizes for recovery cost or storage overhead. For example, Reed-Solomon [21] is a popular family of codes used in Google's ColossusFS [2], Facebook's HDFS [3], and several other storage systems [20, 25, 16]. The Reed-Solomon code used in ColossusFS has a storage overhead of 1.5x, while it requires six disk reads and network transfers to recover a lost data block. In contrast, the Reed-Solomon code used in HDFS reduces the storage overhead to 1.4x, but has a recovery cost of ten blocks. The other popular code family is the Local Reconstruction Codes (LRC) [15, 24, 26], and has similar tradeoffs.

In this paper, we present Hadoop Adaptively-Coded Distributed File System (*HACFS*), a new erasure-coded storage system, which instead uses *two different erasure codes* from the same code family. It uses a *fast code* with low recovery cost and a *compact code* with low storage overhead. It exploits the data access skew observed in Hadoop workloads [9, 7, 22, 5] to decide the initial encoding of data blocks. The HACFS system uses the fast code to encode a small fraction of the frequently accessed data and provide overall low recovery cost for the system. It uses the compact code to encode the majority of less frequently accessed data blocks and maintain a low and bounded storage overhead.

After initial encoding, the HACFS system dynamically adapts to workload changes by using two novel operations to convert data blocks between the fast and compact codes. *Upcoding* blocks initally encoded with fast code into compact code enables the HACFS system to reduce the storage overhead. Similarly, *downcoding* data blocks from compact code to fast code representation lowers the overall recovery cost of the HACFS system. The upcode and downcode operations are very efficient and only update the associated parity blocks while converting blocks between the two codes.

We have designed and implemented HACFS as an extension to the Hadoop Distributed File System [3]. We find that adaptive coding techniques in HACFS are generic and can be applied to different code families. We successfully implement adaptive coding in HACFS with upcode and downcode operations designed for two different code families: Product codes [23] and LRC codes [15, 24, 26]. In both cases, HACFS with adaptive coding using two codes outperforms HDFS with a single Reed-Solomon [2, 3] or LRC code [15]. We evaluate our design on an HDFS cluster with workload distributions obtained from production environments at Facebook and four different Cloudera customers [9].

The main contributions of this paper are as follows:

- We design HACFS, a new erasure-coded storage system that adapts to workload changes by using two different erasure codes - a fast code to optimize recovery cost of degraded reads and reconstruction of failed disks/nodes, and a compact code to provide low and bounded storage overhead.
- We design a novel conversion mechanism in HACFS to efficiently up/down-code data blocks between the two codes. The conversion mechanism is generic and we implement it for two code families – Product and LRC codes – popularly used in distributed storage systems.
- We implement HACFS as an extension to HDFS and demonstrate its efficacy using two case studies with Product and LRC family of codes. We evaluate HACFS by deploying it on a cluster with real-world workloads and compare it against three popular single code systems used in production. The HACFS system always maintains a low storage overhead, while improving the degraded read latency by 25-46%, reconstruction time by 14-44%, and network and disk traffic by 19-45% during reconstruction.

The remainder of the paper is structured as follows. Section 2 motivates HACFS by describing the different tradeoffs for erasure-coded storage systems and HDFS workloads. Section 3 and 4 present the detailed description of HACFS design and implementation. Finally, we evaluate HACFS design techniques in Section 5, and finish with related work and conclusions.

2 Motivation

In this section, we describe the different failure modes and recovery methods in erasure-coded HDFS [3]. We discuss how the use of erasure codes within HDFS reduces storage overhead, however it increases the recovery cost. This motivates the need to design HACFS, which exploits the data access characteristics of Hadoop workloads to achieve better recovery cost and storage efficiency than the existing HDFS architecture.

Failure Modes and Recovery in HDFS. HDFS has different failure modes, for example, block failure, disk failure, and a decommisioned or failed node. The causes of these failures may be diverse such as hardware failures, software glitches, maintenance operations, rolling upgrades that take certain percentage of nodes offline, and hot-spot effects that overload particular disks. Most of these failures typically result in the unavailability of a single block within an erasure code stripe. An erasure

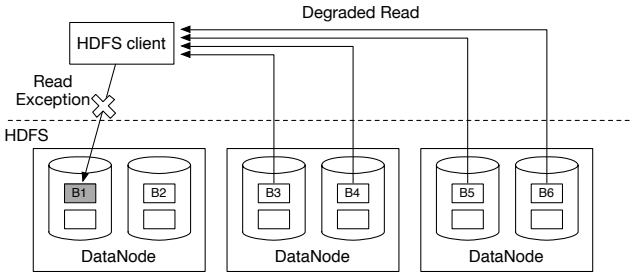


Figure 1: **Degraded reads for HDFS block failure:** *The figure shows a degraded read for an HDFS client reading an unavailable block B_1. The HDFS client retrieves available data and parity blocks and decodes the block B_1.*

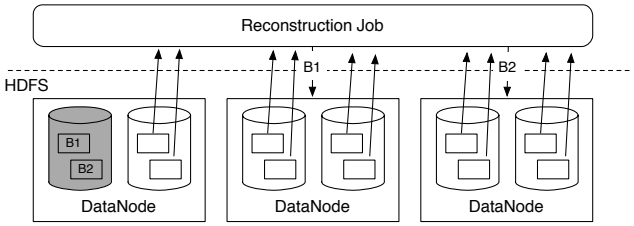


Figure 2: **Reconstruction for HDFS node/disk failure or decomissioned nodes:** *The figure shows a reconstruction MapReduce job launched to recover from a disk failure on an HDFS DataNode. The reconstruction job executes parallel recovery of the lost blocks B_1 and B_2 on other live DataNodes by retrieving available data and parity blocks. An HDFS client accessing a lost block encounters degraded reads during the reconstruction phase.*

code stripe is composed of multiple data blocks striped across different disks or nodes in an HDFS cluster. Over 98% of all failure modes in Facebook's data-warehouse and other production HDFS clusters require recovery of a single block failure [20, 19]. Another 1.87% have two blocks missing, and just less than 0.05% are three or more block failures. As a result, most recent research on erasure-coded storage systems has focused on reducing the recovery cost of single block failures [24, 20, 15, 16].

The performance of an HDFS client or a MapReduce task can be affected by HDFS failures in two ways: degraded reads and reconstruction of an unavailable disk or node. Figure 1 shows a degraded read for an HDFS client reading an unavailable data block B_1, which returns an exception. The HDFS client recovers this block by first retrieving the available data and parity blocks within the erasure-code stripe from other DataNodes. Next, the HDFS client decodes the block B_1 from the available blocks. Overall, the read to a single block B_1 is delayed or degraded by the time it takes to perform several disk reads and network transfers for available blocks, and the time for decoding. Reed-Solomon codes used in two production filesystems - Facebook's HDFS [3] and Google ColossusFS [2] - require between 6-10 network transfers and several seconds for completing one degraded read (see Section 5.3).

A failed disk/node or a decomissioned node typically requires recovery of several lost data blocks. When a DataNode or disk failure is detected by HDFS, several MapReduce jobs are launched to execute parallel recovery of lost data blocks on other live DataNodes. HDFS places data blocks in an erasure-code stripe on different disks and nodes. As a result, the reconstruction of most disk and node failures effectively requires recovery of several single block failures similar to degraded reads.

Figure 2 shows the network transfers required by the reconstruction job running on live DataNodes. An HDFS client trying to access lost blocks B_1 and B_2 during the reconstruction phase encounters degraded reads. Overall, the reconstruction of lost data from a failed disk or node results in several disk reads and network transfers, and can take from tens of minutes to hours for complete recovery (see Section 5.4).

Erasure Coding Tradeoffs. Figure 3 show the recovery cost and storage overhead for Reed-Solomon family of codes widely used in production systems [2, 3, 25]. In addition, it also shows the recovery cost and storage overhead of three popular erasure-coded storage systems: Google ColossusFS [2], Facebook HDFS [3], and Microsoft Azure Storage [15].

Google ColossusFS and Facebook HDFS use two different Reed-Solomon codes - $RS(6,3)$ and $RS(10,4)$- that encode six and ten data blocks within an erasure-code stripe with three and four parity blocks respectively. As a result, they have a recovery cost of six and ten blocks, and storage overheads of 1.5x and 1.4x respectively. Microsoft Azure Storage uses an LRC code - LRC_{comp}, which reduces the storage overhead to 1.33x and has a similar recovery cost of six blocks as Google ColossusFS. It encodes twelve data blocks with two global and two local parity blocks (see Section 3.3 for more detailed description on LRC codes). In contrast, three-way data replication provides recovery cost of one block, but a higher storage overhead of 3x. In general, most erasure-codes including Reed-Solomon and LRC codes trade-off between recovery cost and storage overhead, as shown in Figure 3.

In this work, we focus on the blue region in Figure 3 to achieve recovery cost for HACFS less than that of both Reed-Solomon and LRC codes used in ColossusFS and Azure. We further exploit the data access skew in Hadoop workloads to maintain a low storage overhead for HACFS and keep it bounded between the storage overheads of these two systems.

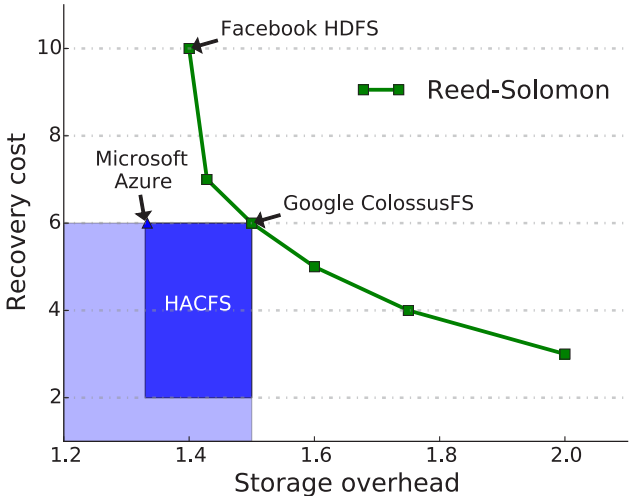


Figure 3: **Recovery Cost vs. Storage Overhead:** *The figure shows the tradeoff between recovery cost and storage overhead for the popular Reed-Solomon family of codes. It also shows three production storage systems each using single erasure code.*

Data Access Skew. Data access skew is a common characteristic of Hadoop storage workloads [9, 7, 22, 5]. Figure 4 shows the frequency of data accessed in production Hadoop clusters at Facebook and four different Cloudera customers [9]. All workloads show skew in data access frequencies. The majority of the *data volume* is cold and accessed only a few times. Similarly, the majority of the *data accesses* go to a small fraction of data, which is hot. In addition, HDFS does not allow in-place block updates or overwrites. As a result, the read accesses primarily characterize this data access skew.

Why HACFS? The HACFS design aims to achieve the following goals:

- *Fast degraded reads* to reduce the latency of reads when accessing lost or unavailable blocks.
- *Low reconstruction time* to reduce the time for recovering from failed disks/nodes or decommissioned nodes, and the associated disk and network traffic.
- *Low storage overhead* that is bounded under practical system constraints and adjusted based on workload requirements.

As shown in Figure 3, the use of a single erasure code tradesoff recovery cost for storage overhead. To achieve the above design goals, the HACFS system uses a combination of two erasure codes and exploits the data access skew within the workload. We next describe HACFS design in more detail.

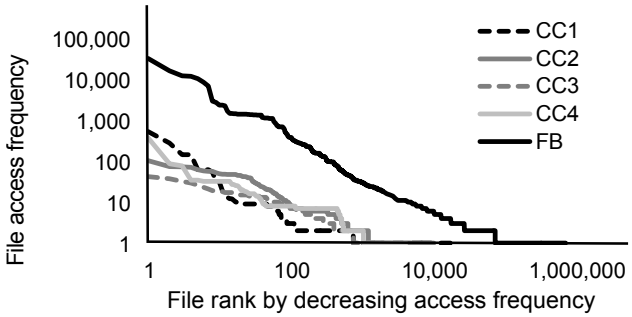


Figure 4: **Data Access Skew in Hadoop Workloads:** *The figure shows the data access distributions of Hadoop workloads collected from production clusters at Facebook (FB) and four different Cloudera customers (CC-1,2,3,4). Both axes are on log scale.*

3 System Design

In this section, we first describe how the HACFS system adapts between the two different codes based on workload characteristics to reduce recovery cost and storage overhead. We next discuss the application of HACFS's adaptive coding to two different code families: Product codes with low recovery cost and LRC codes with low storage overhead.

3.1 Adaptive Coding in HACFS

The HACFS system is implemented as an extension to the HDFS-RAID module [3] within HDFS. We show our extensions to HDFS-RAID as three shaded components in Figure 5. The adaptive coding module maintains the system states of erasure-coded data and manages state transitions for ingested and stored data. It also interfaces with the erasure coding module, which implements the different coding schemes.

System States. The adaptive coding module of HACFS manages the system state. The system state tracks the following *file state* associated with each erasure-coded data file: file size, last modification time, read count and coding state. The file size and last modification time are attributes maintained by HDFS, and used by HACFS to compute the total data storage and write age of the file. The adaptive coding module also tracks the read count of a file, which is the total number of read accesses to the file by HDFS clients. The coding state of a file represents if it is three-way replicated or the erasure coding scheme used for it. The file state can be updated on a create, read or write operation issued to the file from an HDFS client.

The adaptive coding module also maintains a *global state*, which is the total storage used for data and parity. Every block in a replicated data file is replicated at three different nodes in the HDFS cluster and the two replicas account for the parity storage. In contrast, every

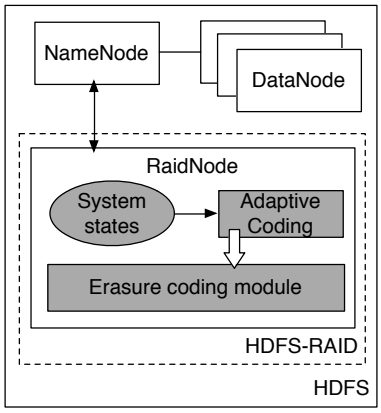


Figure 5: **HACFS Architecture:** *The figure shows the different components of the HACFS architecture implemented as extensions to HDFS in the shaded modules.*

block in an erasure-coded data file has exactly one copy. Each erasure-coded file is split into different erasure code stripes, with blocks in each stripe distributed across different nodes in the HDFS cluster. Each erasure-coded data file also has an associated parity file whose size is determined by the coding scheme. The global state of the system is updated periodically when the adaptive coding module initiates state transitions for erasure-coded files. A state transition corresponds to a change in the coding state of a file and is invoked by using the following interfaces to the erasure-coding module.

Coding Interfaces. As shown in Table 1, the *erasure coding module* in HACFS system exports four major interfaces for coding data: encode, decode, upcode and downcode. The *encode* operation requires a data file and coding scheme as input, and generates a parity file for all blocks in the data file. The *decode* operation is invoked on a degraded read for a block failure or as part of the reconstruction job for a disk or node failure. It also requires the index of the missing or corrupted block in a file, and reconstructs the lost block from the remaining data and parity blocks in the stripe using the input coding scheme.

The adaptive coding module invokes upcode and downcode operations to adapt with workload changes and convert a data file representation between the two coding schemes. As we show later in Section 3.2 and 3.3, both of these conversion operations only update the associated parity file when changing the coding scheme of a data file. The *upcode* operation transforms data from a fast code to a compact code representation, thus reducing the size of the parity file to achieve lower storage overhead. It does not require reading the data file and is a parity-only transformation. The *downcode* operation transforms from a compact code to a fast code representation, thus reducing the recovery cost. It requires reading both data and parity files, but only changes the parity file. We next explain how HACFS uses these interfaces for transitioning files between different coding schemes based on the file state and global system state.

Function	Input	Output
encode	data file, codec	parity file
decode	data file, parity file, codec, lost block index	recovered block
upcode	parity file, original fast codec, new compact codec	parity file encoded with compact codec
downcode	data file, parity file, original compact codec, new fast codec	parity file encoded with fast codec

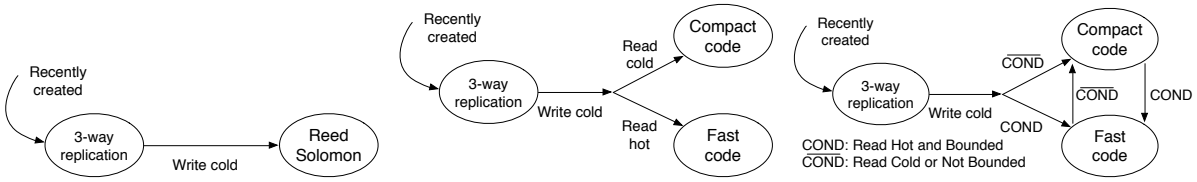
Table 1: **The HACFS Erasure Coding Interfaces**

State Transitions. The HACFS system extends HDFS-RAID to use different erasure coding schemes for files with different read frequency, thus achieving the low recovery cost of a fast code and the low storage overhead of a compact code. We first describe the basic state machine used in HDFS-RAID and then elaborate on the HACFS extensions.

As shown in Figure 6(a), a recently created file in HDFS-RAID is classified as write hot based on its last modified time and therefore three-way replicated. The HDFS-RAID process (shown as RaidNode in Figure 5) scans the file system periodically to select write cold files for erasure coding. It then schedules several MapReduce jobs to encode all such candidate files with a Reed-Solomon code [3]. After encoding, the replication level of these files is reduced to one and the coding state changes to Reed-Solomon. As HDFS only supports appends to files, a block is never overwritten and these files are only read after being erasure-coded.

Figure 6(b) shows the first extension of the HACFS system. It replicates write hot files similar to HDFS-RAID. In addition, HACFS also accounts for the read accesses to data blocks in a file. All write cold files are further classified based on their read counts and encoded with either of the *two different erasure codes*. Read hot files with a high read count are encoded with a *fast code*, which has a low recovery cost. Read cold files with a low read count are encoded with a *compact code*, which has a low storage overhead.

However, a read cold file can later get accessed and turn into a read hot file, thereby requiring low recovery cost. Similarly, encoding all files with the fast code may result in a higher total storage overhead for the system. As a result, the HACFS system needs to adapt to the workload by converting files between fast and compact codes (as shown in Figure 6(c)). The conversion for a file is guided by its own file state (read count) as well as the global system state (total storage). When the total storage consumed by data and parity blocks exceeds a configured system *storage bound*, the HACFS system se-



(a) Single Code in HDFS (b) Two Codes in HACFS (c) Adaptive Coding in HACFS

Figure 6: **Execution States:** *The figure shows the functional state machines for HDFS and two extensions for HACFS. Transitions between different states are triggered by the adaptive coding module, which invokes the coding interface exported by the erasure-coding module in HACFS.*

	PC_{fast}	PC_{comp}	LRC_{fast}	LRC_{comp}
DRC	2	5	2	6
RC	2	5	3.25	6.75
SO	1.8x	1.4x	1.66x	1.33x
MTTF	1.4×10^{12}	2.1×10^{11}	6.1×10^{11}	8.9×10^{10}

Table 2: **Fast and Compact Codes**: This table shows the two codes in the Product and LRC code families used for adaptive coding in HACFS (*DRC*: Degraded Read Cost, *RC*: Reconstruction Cost, *SO*: Storage Overhead, *MTTF*: Mean-Time-To-Failure in years).

lects some files encoded with fast code and upcodes them to the compact code. Similarly, it selects some replicated files and encodes them directly into the compact code. The HACFS system selects these files by first sorting them based on their read counts and then upcodes/encodes the files with lowest read counts into compact code to make the total storage overhead bounded again.

The downcode operation transitions a file from compact to fast code. As a result, it reduces the recovery cost of a future degraded read to a file, which was earlier compact-coded but has been recently accessed. As shown in Figure 4, the data access skew for Hadoop workloads result in a small fraction of read hot files and large fraction of read cold files. This skew allows HACFS to reduce the recovery cost by encoding/downcoding the read hot files with a fast code and reduces the storage overhead by encoding/upcoding a large fraction of read cold files with a compact code.

Fast and Compact Codes. The adaptive coding techniques in HACFS are generic and can be applied to different code families. We demonstrate its application to two code families: Product codes [23] with low recovery cost and LRC codes [15, 24] with low storage overhead. Table 2 shows the three major characteristics useful for selecting fast and compact codes from a code family. The fast code must have a low recovery cost for degraded reads and reconstruction. The compact code must have a low storage overhead. Finally, the reliability of both codes measured in terms of mean-time-to-failure for data loss must be greater than that for three-way replication (3.5×10^9 years) [13]. In addition, the HACFS system requires a storage bound, which can be set from the practical requirements of the system or can be optimally tuned close to the storage overhead of the compact code. We use a storage bound of 1.5x with Product codes and 1.4x with LRC codes in the two case studies of the HACFS system.

We next describe the design of the *erasure coding module* in HACFS for Product and LRC codes in Section 3.2 and 3.3 respectively.

3.2 Product Codes

We now describe the construction and coding interfaces of Product codes used in the HACFS system.

Encoding and Decoding. Figure 7 shows the construction of a Product code, PC_{fast} or $PC(2 \times 5)$, which has a stripe with two rows and five columns of data blocks. The encode operation for PC_{fast} retrieves the ten data blocks from different locations in the HDFS cluster and generates two horizontal, five vertical and one global parity. The horizontal parities are generated by transferring the five data blocks in each row and performing an XOR operation on them. A vertical parity only requires two data block transfers in a column. The global parity can be constructed as an XOR of the two horizontal parities. The decode operation for a Product code is invoked on a block failure. A single failure in any data or parity block of the PC_{fast} code requires only two block transfers from the same column to reconstruct it.

As a result, the PC_{fast} code can achieve a very low recovery cost of two block transfers at the cost of a high storage overhead of eight parity blocks for ten data blocks (1.8x). We choose the PC_{comp} or $PC(6 \times 5)$ as the compact code (see Figure 7), which provides a lower storage overhead of 1.4x and higher recovery cost of five block transfers (see Table 2). In addition, both fast and compact Product codes have reliability better than three-way replication. We select a storage bound of 1.5x for the HACFS system with these Product codes since it is close to the storage overhead of the PC_{comp} code. This bound also matches the practical limits prescribed by the Google ColossusFS [2], which uses the Reed-Solomon $RS(6, 3)$ code similarly optimized for recovery cost.

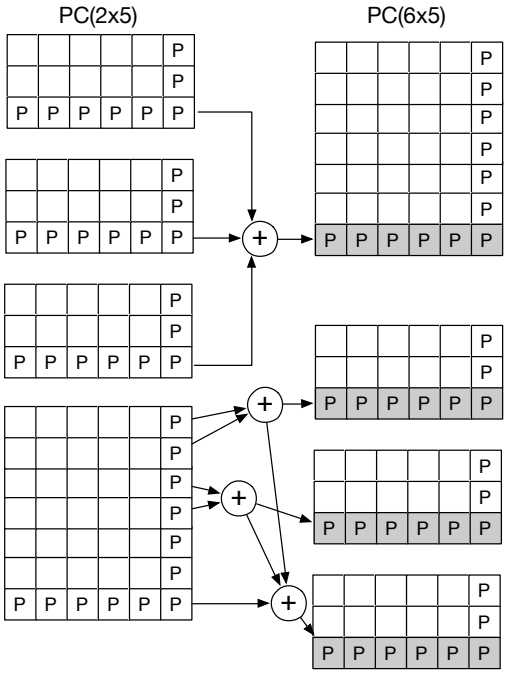


Figure 7: **Product Code - Upcode and Downcode Operations:** *The figure shows the upcode and downcode operations on the data and parity blocks for Product codes. The shaded horizontal parity blocks in the output code are only computed, and the remaining blocks remain unchanged from the input code.*

Upcoding and Downcoding. Figure 7 shows upcoding from PC_{fast} to PC_{comp} and downcoding from PC_{comp} to PC_{fast} codes. Upcode is a very efficient *parity-only conversion* operation for Product codes. All data and vertical parity blocks remain unchanged in upcoding from the PC_{fast} to PC_{comp} code. Further, it only performs XOR over the old horizontal and global parity blocks of the three PC_{fast} codes to compute the new horizontal and global parity blocks of the PC_{comp} code. As a result, the upcode operation does not require any network transfers of the data blocks from the three PC_{fast} codes to compute the new parities in the PC_{comp} code.

The downcode operation converts a PC_{comp} code into three PC_{fast} codes. Only the horizontal and global parities change between the PC_{comp} code and the three PC_{fast} codes. However, computing the horizontal and global parities in the first two PC_{fast} codes requires network transfers and XOR operations over the data blocks in the two horizontal rows of the PC_{comp} code. The horizontal and global parities in the third PC_{fast} code is computed from the those of the old PC_{comp} code and those newly computed ones of the first two PC_{fast} codes. This optimization saves on the network transfers of two horizontal rows of data blocks. Similar to the upcode operation, data and vertical parity blocks in the resulting three PC_{fast} codes remain unchanged from the PC_{comp} code and do not require any network transfers.

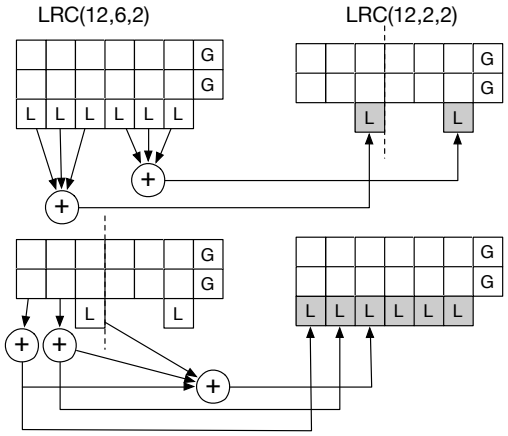


Figure 8: **LRC Code - Upcode and Downcode Operations:** *The figure shows the upcode and downcode operations with data and parity blocks for LRC codes. The shaded blocks are only computed during these operations, and remaining blocks remain unchanged.*

3.3 LRC Codes

We now describe the construction and coding interfaces of the erasure coding module using LRC codes in HACFS.

Encoding and Decoding. Figure 8 shows the construction of the LRC_{fast} or $LRC(12, 6, 2)$, with twelve data blocks, six local parities, and two global parities. The encode operation for an LRC code computes the local parities by performing an XOR over a *group* of data blocks. Two data blocks in each column form a different group in LRC_{fast}. The two global parities are computed by performing a Reed-Solomon encoding over all of the twelve data blocks [24]. The Reed-Solomon encoding of the global parities has properties similar to the LRC code construct used in Microsoft Azure Storage [15] for the most prominent single block failure scenarios. The decode operation for LRC_{fast} code is similar to Product Codes for data and local parity blocks. Any single failure in data or local parity blocks for LRC_{fast} requires two block transfers from the same column to reconstruct it. However, a failure in a global parity block requires all twelve data blocks to reconstruct it using the Reed-Solomon decoding.

Degraded reads from an HDFS client only occur on data blocks, while reconstructing a failed disk or node can also require recovering lost global parity blocks. As a result, the degraded read cost for the *fast code* - LRC_{fast} or $LRC(12, 6, 2)$ - is very low at two blocks (see Table 2). Unlike Product codes, the average reconstruction cost for the LRC_{fast} code is *asymmetri-*

cal to its degraded read cost since reconstruction requires twelve block transfers for global parity failures: $\frac{(12+6)*2+2*12}{12+6+2}$ or 3.25 blocks. However, the storage overhead for an LRC_{fast} code is 1.66x corresponding to eight parity blocks required for twelve data blocks.

We use the LRC_{comp} or $LRC(12, 2, 2)$ code used in Azure [15] as the *compact code* for adaptive coding in HACFS. The LRC_{comp} code has a lower storage overhead of 1.33x due to fewer local parities. However, each of its two local parities is associated with a group of six data blocks. Thus, recovering a lost data block or local parity requires six block transfers from its group in the LRC_{comp} code. The global parities require twelve data block transfers for recovery. As a result, the LRC_{comp} code also has a lower recovery cost for degraded reads than its reconstruction cost similar to the LRC_{fast} code. Both LRC codes are more reliable than three-way replication. We select a storage bound of 1.4x for the HACFS system with LRC codes since it is close to the storage overhead of LRC_{comp} code and lower than HACFS with Product codes.

Upcoding and Downcoding. Upcode and downcode operations for Product codes require merging three PC_{fast} codes into a PC_{comp} code and splitting a PC_{comp} code into three PC_{fast} codes respectively. The LRC codes can be upcoded and downcoded in a similar manner. However, such upcoding and downcoding with LRC codes requires several data block transfers to compute the new local and global parities. As a result, we use a more efficient code collapsing technique for the LRC upcode and downcode operations. This does not require computing the global parities again because collapsing converts exactly one LRC_{fast} code to one LRC_{comp} code (and reverse for downcoding).

Figure 8 shows the LRC upcode operation by computing the new local parities in LRC_{comp} code and preserving the global parities from the LRC_{fast} code. The two local parities in the LRC_{comp} code are computed as an XOR over three local parities in the LRC_{fast} code. As a result, the HACFS system requires only six network transfers to compute the two new local parities of the LRC_{comp} code in an upcode operation. The downcode operation computes two of the three new local parities in LRC_{fast} code from the data blocks in the individual columns of the LRC_{comp} code. The third local parity is computed by performing an XOR over the two new local parities and the old local parity in the LRC_{comp} code. Overall, the downcode operation requires ten block transfers for computing the new local parities. The global parities remain unchanged and do not require any network transfers in the downcode operation as well.

4 Implementation

We have implemented HACFS as an extension to the HDFS-RAID [3] module in the Hadoop Distributed File System (HDFS). The HDFS-RAID module is implemented by Facebook to support a single erasure code for distributed storage in an HDFS cluster. Our implementation of HACFS spans nearly 2 K lines of code, contained within the HDFS-RAID module, and requires no modification to other HDFS components such as the NameNode or DataNode.

Erasure Coding in HDFS. The HDFS-RAID module overlays erasure coding on top of HDFS and runs as a RaidNode process. The RaidNode process periodically queries the NameNode for new data files that need to be encoded and for corrupted files that need to be recovered. The RaidNode launches a MapReduce job to compute the parity files associated with data files on different DataNodes for the encode operation. The decode operation is invoked as part of a degraded read or the reconstruction phase.

A read from an HDFS client requests the block contents from a DataNode. A degraded read can occur due to failures on DataNodes such as a CRC check error. In those cases, the HDFS client queries the RaidNode for locations of the available blocks in the erasure-code stripe required for recovery. The client then retrieves these blocks and performs decoding itself to recover the failed block. The recovered block is used to serve the read request, but it is not written back to HDFS since most degraded reads are caused by transient failures that do not necessarily indicate data loss [20, 24].

When a disk or node failure is detected, the NameNode updates the list of corrupted blocks and lost files. The RaidNode then launches two MapReduce reconstruction jobs, one to recover lost data blocks and the other for lost parity blocks. The reconstruction job retrieves the available blocks for decoding, recovers the lost blocks using the decode operation, writes the recovered blocks to HDFS, and informs the NameNode of successful recovery. If there is a file which has many errors and can not be recovered, then it is marked as permanently lost.

HACFS and Two Erasure Codes. Figure 5 shows the three major components of HACFS implementation: erasure coding module, system states and the adaptive coding module. In addition, we also implement a fault injector to trigger degraded reads and data reconstruction.

The HDFS-RAID only supports a single Reed-Solomon erasure code for encoding data. We implement two new code families as part of the HACFS erasure coding module: Product and LRC codes. The erasure coding module in HACFS exports the same encode/decode interfaces as HDFS-RAID. In addition, the erasure coding

module also provides two new upcode/downcode interfaces to the extended state machine implemented in the adaptive coding module of HACFS. The upcode operation either merges three fast codes for Product codes or collapses one fast code for LRC codes into a new compact code of smaller size. Downcoding performs the reverse sequence of steps. Both operations change the coding state of the data file and reduce its replication level to one.

The adaptive coding module tracks the system states and invokes the different coding interfaces. As desribed earlier, the HDFS-RAID module selects the three-way replicated files which are write cold based on their last modification time for encoding. The extended state machine implemented as part of the adaptive coding module in HACFS further examines these candidate files based on their read counts. It retrieves the coding state of all files classified as read cold and launches MapReduce jobs to upcode them into the compact code. Similarly, if the global system storage exceeds the bound, it upcodes the files with the lowest read counts into the compact code. If the global system storage is lower than the bound or a cold file has been accessed, the adaptive coding module downcodes the file into the fast code and also updates its coding state.

On a disk or node failure, the RaidNode in HACFS launches MapReduce jobs to recover lost data and parity blocks similar to HDFS-RAID. It prioritizes the jobs for reconstructing data over parities to quickly restore data availability for HDFS clients. There are four different types of reconstruction jobs in HACFS, which recover data and parity files encoded with fast and compact codes. Data files encoded with a fast code have a lower recovery cost, but they are also fewer in number than compact-coded data files. As a result, the reconstruction of data files encoded with a fast code is prioritized first among all reconstruction jobs. This prioritization also helps to reduce the total number of degraded reads during the reconstruction phase since fast-coded files get accessed more frequently.

We also implement a fault injector outside HDFS to simulate different modes of block, disk and node failures on DataNodes. The fault injector deletes a block from the local file system on a DataNode, which is detected by the HDFS DataNode as a missing block, and triggers a degraded read when an HDFS client tries to access it. The fault injector simulates a disk failure by deleting all data on a given disk of the target DataNode and then restarting the corresponding DataNode process. A node failure is injected by killing the target DataNode process itself. In both disk and node failure, the NameNode updates the list of lost blocks, and then the RaidNode launches the MapReduce jobs for reconstruction.

5 Evaluation

We evaluate HACFS's design techniques along three different axes: degraded read latency, reconstruction time and storage overhead.

5.1 Methods

Experiments were performed on a cluster of eleven different nodes, each of which is equipped with 24 Intel Xeon E5645 CPU cores running at 2.4 GHz, six 7.2 K RPM disks each of 2 TB capacity, 96 GB of memory, and 1 Gbps network link. The systems run Red Hat Enterprise Linux 6.5 and HDFS-RAID [3]. We use the default HDFS filesystem block size of 64 MB.

The HACFS system uses adaptive coding with fast and compact codes from Product and LRC code families. We refer these two different systems as: HACFS-PC using PC_{fast} and PC_{comp} codes, and HACFS-LRC using LRC_{fast} and LRC_{comp} codes. We compare these two HACFS systems against three HDFS-RAID systems using exactly one of these codes for erasure coding: Reed-Solomon $RS(6,3)$ code, Reed-Solomon $RS(10,4)$ code, and $LRC(12,2,2)$ or LRC_{comp} code. These three codes are used in production storage systems: $RS(6,3)$ used in Google Colossus FS [2], $RS(10,4)$ used in Facebook HDFS-RAID [3], and LRC_{comp} used in Microsoft Azure Storage [15]. We configure the storage overhead bound for HACFS-PC and HACFS-LRC systems as 1.5x (similar to Colossus FS) and 1.4x (similar to Facebook's HDFS-RAID) respectively.

We use the default HDFS-RAID block placement scheme to evenly distribute data across the cluster ensuring that no two blocks within an erasure code stripe reside on the same disk. We measure the degraded read latency by injecting single block failures (as described in Section 4) for a MapReduce grep job that is both network and I/O intensive. We measure the reconstruction time by deleting all blocks on a disk. The block placement scheme ensures that the lost disk does not have two blocks from the same stripe. As a result, the NameNode starts the reconstruction jobs in parallel using the remaining available disks. We report the completion time and network bytes transferred for reconstruction jobs averaged over five different executions.

We use five different workloads collected from production Hadoop clusters in Facebook and four Cloudera customers [9]. Table 3 shows the distribution of each workload: number of files accessed, percentage of files accounting for 90% of the total accesses, percentage of data volume corresponding to such files, and percentage of reads in all accesses to such files.

5.2 System Comparison

HACFS improves degraded read latency, reconstruction

	HACFS-PC			HACFS-LRC		
	Colossus FS	HDFS-RAID	Azure	Colossus FS	HDFS-RAID	Azure
Degraded Read Latency	25.2%	46.1%	25.4%	21.5%	43.3%	21.2%
Reconstruction Time	14.3%	43.7%	21.4%	-3.1%	32.2%	5.6%
Storage Overhead	2.3%	-4.7%	-10.2%	7.7%	1.1%	-4.2%

Table 4: **System Comparison**. The table shows the percentage improvement of two HACFS systems using Product and LRC codes for recovery performance and storage overhead over three single code systems: Google ColossusFS using $RS(6,3)$, Facebook HDFS-RAID using $RS(10,4)$, and Microsoft Azure Storage using LRC_{comp} codes.

Workload	Files	Hot Files	% Hot Data	% Hot Reads
CC1	20.1K	1.2K	5.9	86.1
CC2	10.2K	1.6K	15.7	75.9
CC3	2.1K	1.1K	52.4	75.5
CC4	5.2K	1.4K	26.9	85.2
FB	802K	103K	12.8	90.2

Table 3: **Workload Characteristics**: The table shows the distributions of five different workloads from Hadoop clusters deployed at four different Cloudera customers (CC1/2/3/4) and Facebook (FB).

time, and provides low and bounded storage overhead. We begin with a high-level comparison of HACFS using adaptive coding on Product and LRC codes with three single code systems: ColossusFS, HDFS-RAID and Azure.

Degraded Read Latency. Table 4 shows the percentage improvement of adaptive coding in HACFS with Product and LRC codes averaged over five different workloads. HACFS reduces the degraded latency by 25-46% for Product codes and 21-43% for LRC codes compared to three single-coded systems. This improvement in HACFS primarily comes from the use of fast codes (PC_{fast} and LRC_{fast}) for hot data, which is primarily dominated by read accesses (see Table 3). As a result, the degraded read latency of HACFS is lower than all of the three other production systems relying on $RS(6,3)$, $RS(10,4)$ and LRC_{comp} codes. We describe these results in more detail for each of the five different workloads in Section 5.3.

Reconstruction Time. HACFS improves the reconstruction time to recover from a disk or node failure by 14-43% for Product codes and up to 32% for LRC codes. The reconstruction time is dominated by the volume of data and parity blocks lost in a disk or node failure. The fast and compact Product codes used in HACFS have a lower reconstruction cost than the two LRC codes. As described in Section 3.3, this is because LRC codes have a higher recovery cost for failures in local and global parity blocks than data blocks. As a result, the HACFS system with LRC codes takes slightly longer to reconstruct a lost disk than ColossusFS, which uses the $RS(6,3)$ code with a symmetric cost to recover from data and parity failures. We discuss these results in more detail in Section 5.4.

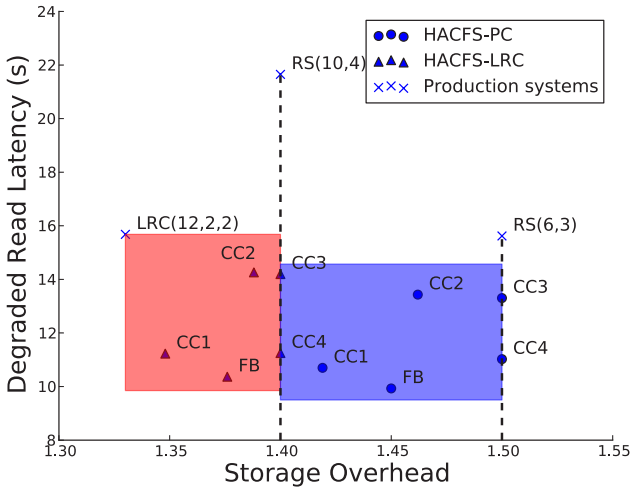


Figure 9: **Degraded Read Latency:** The figure shows the degraded read latency and storage overhead for two HACFS systems and three single code systems.

Storage Overhead. HACFS is designed to provide low and bounded storage overheads. The Azure system using the LRC_{comp} code has the lowest storage overhead (see Table 2), and is up to 4-10% better than the two HACFS systems. The HDFS-RAID system using $RS(10,4)$ has about 5% lower storage overhead than HACFS optimized for recovery with Product codes. However, the HACFS system with LRC codes has storage overheads lower or comparable to the three single-coded production systems [2, 15, 24]. This is primarily because adaptive coding in HACFS bounds the storage overhead by 1.5x for Product codes and by 1.4x for LRC codes. We discuss the storage overheads of each system across different workloads in Section 5.3.

5.3 Degraded Read Latency

HACFS uses a combination of recovery-efficient fast codes (PC_{fast} and LRC_{fast}) and storage-efficient compact codes (PC_{comp} and LRC_{comp}). Figure 9 shows the degraded read latency on y-axis and storage overhead on x-axis for the five different workloads. A normal read from an HDFS client to an available data block can take up to 1.2 seconds since it requires one local

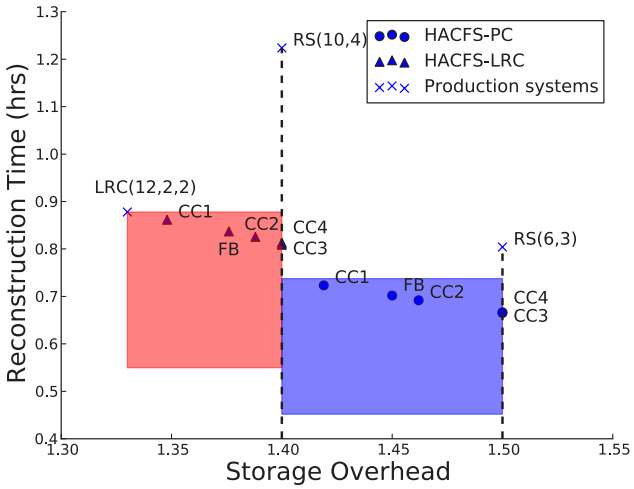


Figure 10: **Reconstruction Time:** The figure shows the reconstruction time to recover from data loss with two HACFS systems and three single code systems.

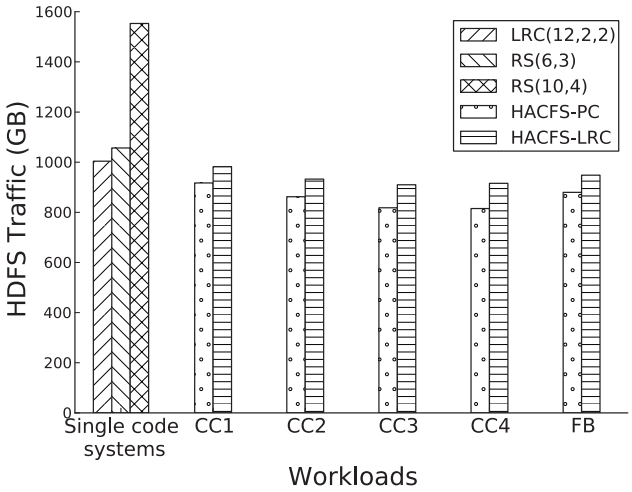


Figure 11: **Reconstruction Traffic:** The figure shows the reconstruction traffic for recovering lost data and parity blocks with two HACFS systems and three single code systems.

disk read and one network transfer if the block is remote. In contrast, a degraded read can require multiple network transfers, and takes between 16-21 seconds for the three single coded systems. These systems do not adapt with the workload and only use a single code. As a result, their degraded read latency and storage overhead is the same across all five workloads. Adaptive coding in HACFS reduces the degraded read latency by 5-10 seconds for three workloads (CC1, CC4 and FB), which have a higher percentage of reads to hot data encoded with the fast code (85-90%, see Table 3). The two shaded boxes in Figure 9 demonstrate that HACFS adapts to the characteristics of the different workloads. However, HACFS always outperforms the three single coded systems since all of them require more blocks to be read and transferred over the network to decode a missing block.

Both HACFS systems have a lower storage overhead for workloads (CC1, CC2 and FB) with a higher percentage of cold files (85-95%) encoded with the compact codes. The lowest possible storage overhead for HACFS is shown by the left boundary of the two shaded regions marked with 1.33x and 1.4x for the compact codes (LRC_{comp} and PC_{comp} codes respectively). In addition, HACFS also bounds the storage overhead by 1.5x for Product codes and 1.4x for LRC codes. As a result, workloads (CC3 and CC4) with fewer cold files still never exceed these storage overheads marked by the right edges in the two shaded regions. If we do not enforce any storage overhead bounds, these two workloads benefit even further by a reduction of 6-20% in their degraded read latencies.

5.4 Reconstruction Time

Figure 10 shows the reconstruction time of the three single code systems and HACFS system with adaptive coding when a disk with 100 GB of data fails. The reconstruction job launches map tasks on different DataNodes to recreate the data and parity blocks from the lost disk. The time to reconstruct a cold file encoded with a compact code is longer than that for a fast code. The HACFS system with Product codes outperforms the three single code systems for all five workloads. It takes about 10-35 minutes less reconstruction time than the three single code systems. This is because both fast and compact Product codes reconstruct faster than the two Reed-Solomon codes and the LRC_{comp} code.

The HACFS system with the use of faster LRC_{fast} code for reconstruction outperforms the LRC_{comp} code with the lowest storage overhead. However, the HACFS system with LRC codes is generally worse for all workloads than the $RS(6,3)$ code used in the ColossusFS. This is because both LRC codes used in HACFS have a recovery cost for global parities higher than the $RS(6,3)$ code (see Table 2).

Figure 11 shows the reconstruction traffic measured as HDFS read and writes incurred by the reconstruction job to recover 100 GB of data and additional parity blocks lost from the failed disk. The reconstruction job reads all blocks in the code stripe for recovering the lost blocks, and then writes the recovered data and parity blocks back to HDFS. The HDFS-RAID system using the $RS(10,4)$ code results in the highest traffic: 1550 GB of reconstruction traffic for 100 GB of lost data. This is close to the theoretical reconstruction traffic of nearly fifteen blocks per lost data block, including ten block reads for

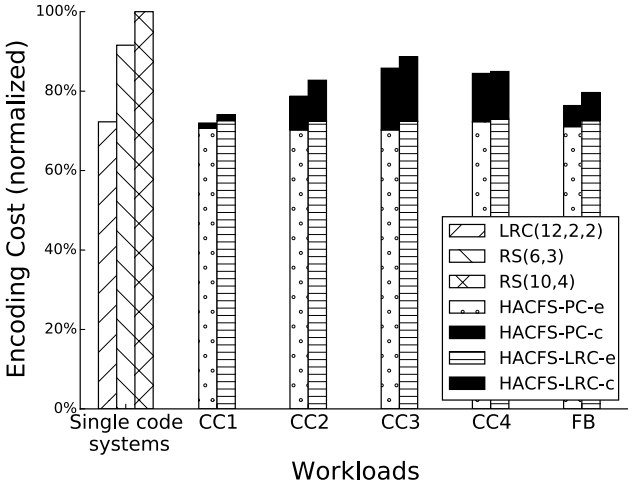


Figure 12: **Encoding Cost:** The figure shows the encoding time for three single code systems and two HACFS systems normalized over the HDFS-RAID ($RS(10,4)$). The black bars show the *c*-onversion time component of the total *e*-ncoding time for the two HACFS systems.

data recovery, four block reads for parity recovery, and then writes of the recovered data block and parity blocks ($RS(10,4)$ uniformly stores 0.4 parity blocks with each data block on a disk). Similarly, LRC_{comp} in Azure and $RS(6,3)$ in Colossus, require nearly ten HDFS block read/writes for recovering a block from the lost disk.

The HACFS system with Product codes always requires fewer blocks for reconstruction than the three single code systems: between eight blocks for CC3 and CC4 workloads and nine blocks for CC1, CC2 and FB workloads (with more than 85% cold files) based on the data skew distributions. The HACFS system with LRC codes requires more blocks for global parity recovery than Product codes. As a result, its reconstruction traffic is close to $RS(6,3)$ and LRC_{comp} codes at nearly ten blocks per lost data block.

5.5 Encoding and Conversion Time

Figure 12 shows the encoding cost for initial encoding of three-way replicated data in three single code systems, and the encoding cost for initial encoding and later conversion between the fast and compact codes in the two HACFS systems. We normalize the encoding cost per block to eliminate the differences in dataset sizes across the five workloads. All compared systems are based on HDFS-RAID implementation, which schedules the encoding and conversion operations as MapReduce jobs in background to minimize their impacts on user jobs. As a result, we show the impact of encoding cost for all systems relative to $RS(10,4)$ used in HDFS-RAID in Figure 12. Encoding cost is a function of the coding scheme used for data blocks and does not change with workload for the three single code systems. Reed-Solomon codes used in HDFS-RAID and ColossusFS have the highest encoding cost because of complex Galois Field operations required to compute parity blocks [18]. LRC code in Azure uses such operations only to compute global parities and uses cheaper XOR operations for all local parities. Similarly, HACFS with Product codes only uses XOR operations for encoding. As a result, the encoding time component of the two HACFS systems is similar to the LRC codes in Azure for all workloads.

The HACFS system also converts (upcodes/downcodes) data between fast and compact codes. The conversion cost is only high when the HACFS system aggressively converts blocks to limit the storage overhead by upcoding hot files into compact code. As a result, the three workloads (CC2, CC3 and CC4) with a higher percentage of hot data spend up to 18% of total encoding time for conversion operations. For these workloads, the total encoding and conversion cost of the HACFS systems is up to 16% higher than the Azure system using a single LRC code. In general, the encoding cost of the two HACFS systems is about 3-28% lower than the single-code ColossusFS and HDFS-RAID systems using Reed-Solomon codes for all workloads.

6 Related Work

Our work builds on past work on distributed storage systems, faster recovery techniques, and tiered storage systems.

Distributed Storage Systems. Petabytes of storage is becoming common with the fast growing data requirements of modern systems today. Erasure codes offer an attractive alternative to provide lower storage overhead than data replication. As a result, many distributed filesystems such as Google ColossusFS [2], Facebook HDFS [3], and IBM General Parallel File System [6] are moving to the use of erasure codes. Many popular object stores used for cloud storage, for example, OpenStack Swift [17], Microsoft Azure Storage [15] and Cleversafe [1] are also adopting erasure codes for low storage overhead. However, most of these systems use a single erasure code and address the recovery cost by trading for storage overhead. In contrast, HACFS is the first system that uses a combination of two codes to dynamically adapt with workload changes and provide both low recovery cost and storage overhead.

Faster Recovery for Erasure-Codes. Recently, there has been a growing focus on improving recovery performance for erasure-coded storage systems. Reed-Solomon codes [21] offer optimal storage overhead but generally have high recovery cost. Rotated Reed-Solomon codes have been proposed as an alternative con-

struction, which requires fewer data reads for faster degraded read recovery [16]. HitchHiker proposes a new encoding technique by dividing a single Reed-Solomon code stripe into two correlated substripes and improves recovery performance [20]. However, both of them trade extra encoding time for faster recovery. In contrast, adaptive coding techniques in HACFS provide lower recovery cost without increasing encoding time. In general, adaptive coding can be applied to most code families, which tradeoff between storage overhead and recovery cost. We have found efficient up/downcode operations for applying adaptive coding to different constructs of Reed-Solomon code and other modern storage codes such as PMDS [8] and HoVer [12]. For example, we devised up/downcode operations for converting m (n,r) Reed-Solomon codes into a (mn, r) Reed-Solomon code using a parity-only conversion scheme.

Tiered Storage Systems. Adaptive coding in HACFS is inspired by tiering in RAID architectures [27, 3, 10]. AutoRAID [27] provides a two-level storage hierarchy within the storage controller. It automatically migrates data between different RAID levels to provide high I/O performance for active data and low storage overhead for inactive data. Similarly, HACFS migrates data between fast and compact erasure codes, however with the objective to reduce extra network transfers for recovery in distributed storage. Facebook's HDFS-RAID [3] and DiskReduce [10] propose tiered storage by asynchronously migrating data between replicated and erasure-coded storage tiers. HACFS extends this further by splitting the erasure-coded storage tier into two parts to optimize for both storage overhead and recovery performance.

7 Conclusions

Distributed storage systems extensively deploy erasure-coding today for lower storage overhead than data replication. However, most of these systems trade storage overhead for recovery performance. We present a novel erasure-coded storage system, which uses two different erasure codes and dynamically adapts by converting between them based on workload characteristics. It uses a fast code for fast recovery performance and a compact code for low storage overhead. In the future, as we move to cloud storage, it will be important to revisit similar erasure-coding tradeoffs, and extend adaptive coding techniques to large-scale object stores in the cloud.

Acknowledgements

We thank Jim Hafner for his valuable feedback at different stages of this work. We also thank our shepherd Cheng Huang and the anonymous FAST reviewers for their helpful comments and useful feedback.

References

[1] Cleversafe object storage. `http://www.cleversafe.com`.

[2] Colossus, successor to Google File System. `http://static.googleusercontent.com/media/research.google.com/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf`.

[3] Facebook's erasure coded hadoop distributed file system (HDFS-RAID). `https://github.com/facebook/hadoop-20`.

[4] The Hadoop Distributed File System. `http://wiki.apache.org/hadoop/HDFS`, 2007.

[5] Cloudera: What do real-life apache hadoop workloads look like? `http://blog.cloudera.com/blog/2012/09/what-do-real-life-hadoop-workloads-look-like/`, September 2012.

[6] An introduction to GPFS version 3.5. `http://www-03.ibm.com/systems/resources/introduction-to-gpfs-3-5.pdf`, August 2012.

[7] ABAD, C. L., ROBERTS, N., LU, Y., AND CAMPBELL, R. H. A storage-centric analysis of mapreduce workloads: File popularity, temporal locality and arrival patterns. In *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC)* (Washington, DC, USA, 2012), IISWC '12, IEEE Computer Society, pp. 100–109.

[8] BLAUM, M., HAFNER, J. L., AND HETZLER, S. Partial-MDS codes and their application to RAID type of architectures. *IEEE Transactions on Information Theory 59*, 7 (2013), 4510–4519.

[9] CHEN, Y., ALSPAUGH, S., AND KATZ, R. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow. 5*, 12 (Aug. 2012), 1802–1813.

[10] FAN, B., TANTISIRIROJ, W., XIAO, L., AND GIBSON, G. DiskReduce: RAID for data-intensive scalable computing. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage* (New York, NY, USA, 2009), PDSW '09, ACM, pp. 6–10.

[11] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[12] HAFNER, J. L. HoVer erasure codes for disk arrays. In *Proceedings of the International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2006), DSN '06, IEEE Computer Society, pp. 217–226.

[13] HAFNER, J. L., AND RAO, K. Notes on reliability models for non-MDS erasure codes. IBM Tech Report RJ10391 (A0610-035), 2006.

[14] HUANG, C., CHEN, M., AND LI, J. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. *Trans. Storage 9*, 1 (Mar. 2013), 3:1–3:28.

[15] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (2012), USENIX ATC'12, USENIX Association, pp. 2–2.

[16] KHAN, O., BURNS, R., PLANK, J., PIERCE, W., AND HUANG, C. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST'12, USENIX Association, pp. 20–20.

[17] LUSE, P., AND GREENAN, K. Swift object storage: Adding erasure codes. `http://www.snia.org/sites/default/files/Luse_Kevin_SNIATutorialSwift_Object_Storage2014_final.pdf`, 2014.

[18] PLANK, J. S., GREENAN, K. M., AND MILLER, E. L. Screaming fast Galois Field arithmetic using Intel SIMD instructions. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies* (2013), FAST'13, USENIX Association, pp. 299–306.

[19] RASHMI, K. V., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems* (2013), HotStorage'13, USENIX Association, pp. 8–8.

[20] RASHMI, K. V., SHAH, N. B., GU, D., KUANG, H., BORTHAKUR, D., AND RAMCHANDRAN, K. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of ACM SIGCOMM* (2014), SIGCOMM'14.

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

[22] REN, K., KWON, Y., BALAZINSKA, M., AND HOWE, B. Hadoop's adolescence: An analysis of Hadoop usage in scientific workloads. *Proc. VLDB Endow. 6*, 10 (Aug. 2013), 853–864.

[23] ROTH, R. *Introduction to Coding Theory*. Cambridge University Press, New York, NY, USA, 2006.

[24] SATHIAMOORTHY, M., ASTERIS, M., PAPAILIOPOULOS, D., DIMAKIS, A. G., VADALI, R., CHEN, S., AND BORTHAKUR, D. XORing elephants: novel erasure codes for big data. In *Proceedings of the 39th international conference on Very Large Data Bases* (2013), PVLDB'13, VLDB Endowment, pp. 325–336.

[25] SCHMUCK, F., AND HASKIN, R. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST '02, USENIX Association.

[26] TAMO, I., AND BARG, A. A family of optimal locally recoverable codes. *IEEE Transactions on Information Theory 60*, 8 (2014), 4661–4676.

[27] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst. 14*, 1 (Feb. 1996), 108–136.