Zebra: Demand-aware Erasure Coding for Distributed Storage Systems

Jun Li, Baochun Li
Department of Electrical and Computer Engineering, University of Toronto, Canada
{*junli, bli*}*@ece.toronto.edu*

***Abstract*—Erasure coding has been increasingly replacing replication in distributed storage systems, thanks to its lower storage overhead with the same level of failure tolerance. However, with lower storage overhead, the reconstruction overhead of erasure codes can increase significantly as well. Under the ever-changing workload, in which the data access can be highly skewed, it is difficult to achieve a well trade-off between the storage overhead and the reconstruction overhead.**

In this paper, we propose *Zebra*, a framework that encodes data into multiple tiers by their demand. Given the overall storage overhead and the number of failures to tolerate, Zebra determines the parameters of erasure coding in each tier by solving a geometric programming problem. Based on the demand of data, Zebra can dynamically assign data into the corresponding tiers to minimize the overall reconstruction overhead, and achieve a flexible tradeoff between the storage overhead and the reconstruction overhead in multiple tiers, such that hot data can enjoy less overhead of reconstruction and cold data can be stored with lower storage overhead. When demand changes, Zebra can adjust itself accordingly with a marginal amount of network transfer.

I. Introduction

Distributed storage systems [1], [2], [3] store a massive amount of data over a large number of commodity servers. Due to the nature of the commodity hardware, as well as other reasons like software glitches, power failures, and upgrade and maintenance operations, servers in distributed storage systems are subject to frequent failures on a daily basis. For example, in a Facebook cluster with 3000 servers, 50 failures that lead to data unavailability can be expected every day [4]. Therefore, in distributed storage systems, redundant data must be stored such that a specific number of failures can be tolerated.

The naive way to store redundant data in a distributed storage system is replication. However, replication is very expensive in terms of storage overhead, especially for data at a petabyte scale. For example, with 3-way replication, to store 10 PB of data, we need to spend additional 20 PB to store the other two copies. Hence, distributed storage systems are migrating from replication to erasure coding [5], [6], [7], [8], [9], as erasure coding can tolerate more failures with much less storage overhead [10].

With Reed-Solomon (RS) codes [11], the most common choice of erasure codes in distributed storage systems, we can encode r parity blocks from k data blocks of the same size, such that any k among these $k + r$ blocks can recover the original data. In other words, at most r failures can be tolerated. When $k = 10$ and $r = 4$, we can tolerate at most 4 failures with only $1.4x$ storage overhead. With the increasing of k, we can achieve even lower storage overhead while tolerating the same number of failures.

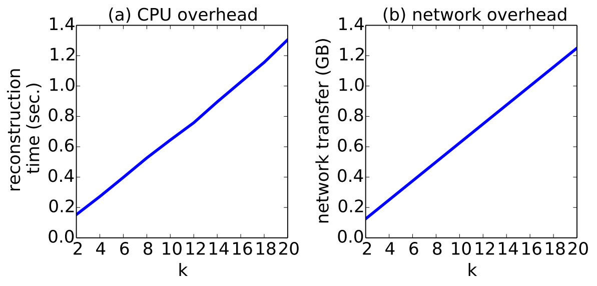


Fig. 1. CPU and network overhead to reconstruct one block with RS code, where each block contains 64 MB.

However, RS codes can incur a significantly higher overhead when we need to reconstruct an unavailable block. With the $(k = 10, r = 4)$ RS code, for example, if one block is not available, we need to obtain 10 blocks to reconstruct it. The reconstruction in a distributed storage system is a universal operation, which can also be triggered when the distributed storage system is performing a degraded read of a block, *i.e.,* reading data in a temporarily unavailable block. For example, when a server is temporarily unavailable, all accesses to the data blocks it stores will have to be performed by degraded reads. In a degraded read, the overhead of reconstruction can lead to higher access latency. Under typical values of k, the network transfer incurred by reconstructions can be huge: in a Facebook's cluster the daily median of top-of-rack network transfer incurred by reconstruction can be as much as 180 TB [12]. Fig. 1 illustrates the overhead of time and network transfer to reconstruct

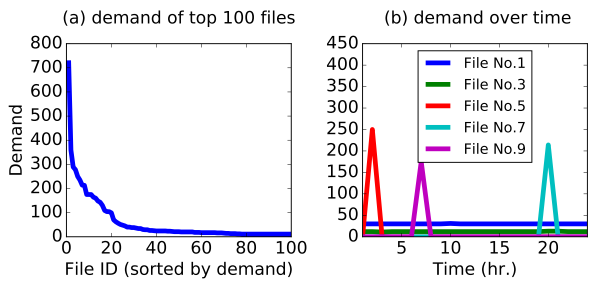


Fig. 2. The demand skewness of files in a Facebook workload trace.

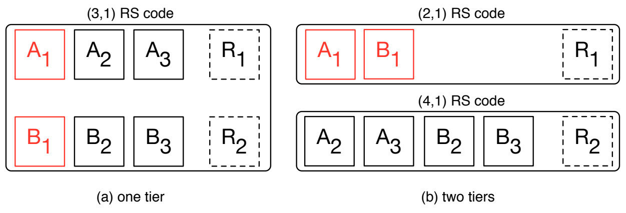


Fig. 3. Comparison of data encoded in one and two tiers of RS codes. A_i and B_i are data blocks, $i = 1, 2, 3$. R_1 and R_2 are parity blocks encoded by corresponding RS codes. Colored red, A_1 and B_1 are hot blocks with 100 visits per unit of time, while other blocks are cold with only 10 visits.

one block of 64 MB with RS codes[1]. We can see that both the time and network transfer incurred by the reconstruction increases linearly with k. In other words, a smaller value of k means less reconstruction overhead, leading to less network transfer for data reconstruction as well as lower latency for degraded reads.

Therefore, in a distributed storage system, while we desire for a small value of k to achieve low reconstruction overhead, it takes a large k to save storage overhead. Currently, most distributed storage systems deploy only one erasure code to encode data, optimized either for storage overhead or reconstruction overhead. However, in practical distributed storage systems, the demand of data can be highly skewed. In Fig. 2a, we show the demand of data in a workload trace from Facebook [14]. This workload contains more than 10^4 files while we only show the 100 most demanded files in the figure, with all the rest having no more than 11 visits. We can see that a very small portion of data are highly demanded while the rest are barely touched. With only one erasure code we can not accommodate cold data with low storage overhead while achieving low reconstruction overhead for hot data. Moreover, the demand for the data can change over time dynamically. For example, in Fig. 2b, while file No. 1 and 3 have consistent demand over time, the other three files only have transient high demand at some time and have no visit any other time. Therefore, it is also hard to find an erasure code that can work well adaptively with the ever-changing demand.

In this paper, we propose *Zebra*, a novel framework for distributed storage systems deploying RS codes. According to the demand of data, Zebra can split data into multiple tiers such that data in different tiers are encoded with RS codes with different values of parameters. By solving geometric programming problems, Zebra determines the parameter values of RS codes in each tier, such that hot data can be reconstructed with low overhead and cold data can enjoy low storage overhead at the same time. When demand changes, Zebra can migrate data accordingly into different tiers or change the parameter values of the tier, while carefully controlling overhead in the migration.

We run simulations under various workload traces to evaluate the performance of the Zebra framework. The evaluation results show that Zebra can reduce overall reconstruction overhead, especially by 54.9% for the hot data. The cold data, on the other hand, will incur less storage overhead to maintain their tolerance against failures. With the ever-changing workload, we demonstrate that the network transfer of migration can be well controlled, such that it occupies no more than 2.4% of the network transfer of demand.

II. Motivation and Examples

In this section, we present the general idea that motivates the design of the Zebra framework. We assume that in a distributed storage system, data are stored in blocks of 64 MB, where we compute 1 parity block from every 3 data blocks. In other words, a $(3, 1)$ RS code is deployed in this distributed storage system.

As a toy example in Fig. 3a, assume that we have 6 data blocks in total, stored with two additional parity blocks. In this way, we can tolerate a failure of any single block, with 1.33x storage overhead. However, the demand of the 6 data blocks is not equal to each other, where A_1 and B_1 are highly demanded with 100 visits per unit of time, and all other four blocks are visited by merely 10 times per unit of time. Suppose that each data block has equal chance to be unavailable. When one of them becomes unavailable, we need to obtain 3 other blocks to reconstruct it until it has been repaired on a replacement server. In a unit of time, this means that $\frac{1}{6}(2 \times 100 + 4 \times 10) \times 3 \times 64$ MB $= 7.5$ GB on average will be read from disks and transferred through network per unit of time.

Under the Zebra framework, on the other hand, we can encode these 6 data blocks into two tiers, with two RS codes. We encode the two hot blocks with a $(2, 1)$ RS code and the other four blocks with a $(4, 1)$ RS code. In each tier, we still have 1 parity blocks in this way, maintaining the same storage overhead. Meanwhile, we

[1]The time of the reconstruction is measured using the zfec library [13] running on an Intel Core i7 processor.

can still tolerate the failure of any single block. However, this time we can significantly reduce the average number of blocks to visit per unit of time. Though the cold blocks need to obtain 4 blocks to reconstruct, the hot blocks have dominant demand with much less blocks to obtain. On average per unit of time, we need to have $\frac{1}{6}(2 \times 2 \times 100 + 4 \times 4 \times 10) \times 64$ MB $= 5.83$ GB to read, saving corresponding disk I/O and network transfer by 22.23%. For the two hot blocks in particular, their reconstruction overhead can simply be reduced from 3 to 2 blocks, *i.e.*, 33.3% reduction in time, disk I/O, and network transfer.

In this example, we deploy two tiers of RS codes. However, Zebra is not limited to only two tiers. In fact, we can flexibly deploy any number of tiers inside the Zebra framework with different parameter values of RS codes, where the number of tiers and the parameter values can be efficiently calculated according to the demand of data, as well as the requirements of storage overhead and failure tolerance.

III. Related Work

At the scale of petabyte storage, erasure coding has become more and more attractive to distributed storage systems because of its low storage overhead and high failure tolerance. Hence, many distributed storage systems, such as HDFS [9], [15], Openstack Swift [5], Google file system [6], and Windows Azure storage [8], are moving towards or have deployed erasure coding as an alternative to replications, where in most cases RS codes are chosen by these distributed storage systems. However, they all choose only one kind of erasure code with fixed parameters. In other words, it is hard to trade well between storage overhead and reconstruction overhead of erasure codes, under the dynamical workload with highly skewed data demand [16].

Traditional RS codes can incur a high overhead of reconstruction when some of the data are not available due to failures inside distributed storage systems [12], [17]. There has been a growing attention of improving the overhead of reconstruction of erasure codes. For example, locally repairable codes [18] can achieve low reconstruction overhead by allowing unavailable data to be reconstructed from a small number of other servers. Similar ideas have been applied in the design of other erasure codes [8], [17], [19], [20], [21]. On the other hand, another family of erasure codes, called regenerating codes, are designed to achieve the optimal network transfer in the reconstruction [22]. All these erasure codes, however, are optimized for their own objectives over all encoded data, unaware of that the demand of data can be high skewed. As data with different demand may have different performance objectives, applying one single erasure code over all data may not achieve all their objectives. Different from these erasure codes, in Zebra we propose to dynamically assign data into multiple tiers by their demand so as to achieve a flexible tradeoff between storage and reconstruction overhead.

Some distributed storage systems, such as HDFS [23], allow data to be stored under a tiered architecture, where data in different tiers are stored by different erasure codes or replication with preconfigured parameters and can be automatically migrated between different tiers [15], [24]. However, all these systems require users to configure the erasure code used in each tier as well as their parameters statically. Hence, they can not well adapt themselves to the ever-changing workload. Besides, as in different tiers the storage overhead incurred by the corresponding erasure codes will also be different, it is hard to control the overall storage overhead. The Zebra framework, on the other hand, does not need to specify parameters of each tier or even the number of tiers. According to the demand of data, Zebra can configure itself flexibly, where only the overall storage overhead and failure tolerance need to be manually specified.

IV. System Model

In this paper, we assume that in a distributed storage system, data are stored in *blocks* with the same size. This is a common practice in distributed storage systems (*e.g.*, HDFS [3]).

Assume that we have N blocks in total, and each block B_i is associated with demand of d_i visits per unit of time, $i = 1, \ldots, N$. We also assume that any r block failures should be tolerated without data loss. Hence, in the Zebra framework each block will be encoded with a (k_i, r) RS code, where we call k_i as the rank of block B_i. For convenience, we let $D = (d_1, \ldots, d_N)$, and $K = (k_1, \ldots, k_N)$. We also want to control the overall storage overhead such that the overall storage space consumed is no more than C times of the original data.

In this model, we assume that the RS codes deployed in the distributed storage system should be systematic. In other words, a (k, r) systematic RS code will compute $k + r$ blocks that directly contains k original data blocks. The other r blocks are known as parity blocks. In this way, we can always directly obtain any data block without decoding as long as it is available. Hence, we can assume that all demand will go directly to the corresponding data blocks rather than parity blocks, unless the demanded data blocks are unavailable.

We now use this model to represent the way to encode data in one or multiple tiers with RS codes. Fig. 3 (without loss of generality, we can rewrite A_i as B_{i+3}, $i = 1, 2, 3$) illustrates two examples of this model where $N = 6$, $C = \frac{4}{3}$, and $D = \{100, 10, 10, 100, 10, 10\}$. In Fig. 3a, $k_i = 3$ for all i, *i.e.,* all blocks are encoded in one tier with a $(3, 1)$ RS code. On the other hand, in

Fig. 3b, we have $K = \{2, 4, 4, 2, 4, 4\}$ such that the six blocks are encoded into two tiers with a $(2, 1)$ and a $(4, 1)$ RS code.

In this way, we can see that the number of tiers in the model do not need to be explicitly defined as blocks with the same rank can naturally be categorized into the same tier. Once the rank of each block is equal to each other, for example, it becomes the conventional case of only one tier. Hence, we are not limited by a given number of tiers, and we can easily change the number of tiers when demand changes.

In this paper, our objective is to minimize the overall overhead of reconstruction with respect to the constraint of the overall storage overhead C. As shown in Fig. 1, the reconstruction overhead of a block increases linearly with its rank. We assume that each block has the same chance to be unavailable. Thus, the chance of reconstructing a block should also increase linearly with its demand. Combining all blocks together, we can define the overall reconstruction overhead as $\sum_{i=1}^{N} d_i k_i = D \cdot K$.

Besides the overall reconstruction overhead, we need to control the overall storage overhead, which can be computed with D and K in the model. Since each block has the same size, we assume that the size of each block is 1 for convenience. Thus, the storage space consumed to store block B_i and its parity is $1 + \frac{r}{k_i}$. The sum of storage space of all blocks is $N + \sum_{i=1}^{N} \frac{r}{k_i}$. Since the total storage space we can use under the constraint of the overall storage overhead C is CN, we can write this constraint as $\sum_{i=1}^{N} \frac{1}{k_i} \leq \frac{(C-1)N}{r}$.

Therefore, we can solve K to minimize the overall reconstruction overhead with respect to the storage overhead by the following integer geometric programming problem.

$$\min \quad D \cdot K \tag{1}$$

$$\text{s.t.} \quad \sum_{i=1}^{N} \frac{1}{k_i} \leq \frac{(C-1)N}{r}, \tag{2}$$

$$k_i \in \mathbb{Z}^+. \tag{3}$$

Typically, a geometric programming problem can be easily converted to a convex optimization problem and solved efficiently [25]. However, there are some other issues that makes it challenging to achieve a practical solution by directly solving this problem.

First, in a distributed storage system there can be an extremely large number of data blocks. For example, in HDFS the default block size is 64 MB. If there are 1 PB of data stored in HDFS, there will be over 10^7 blocks in total. No solver of convex optimization problems can solve our model in a reasonable amount of time. Besides, the geometric programming problem in (1)-(3) is an integer programming problem. This also significantly increases the complexity to solve it.

Second, the solution solved from (1)-(3) is an offline solution. In other words, we need to know the demand in advance before we can get the optimal solution, which makes it impractical. We need to find an online algorithm that can solve K in advance of the demand.

Third, given a solution of this problem, we can't even guarantee that it is feasible. For example, if the solution is $K = \{8, 8, 8, 8, 8, 8\}$, we will need to encode 6 blocks with an $(8, r)$ RS code. This is impossible without rearranging the data blocks into a smaller size. Variable-size blocks, however, will incur significantly more complexity to manage data inside the distributed storage system. In this paper, we retain the assumption of fixed-size blocks and manage to encode data with such solutions without incurring much additional overhead.

In the rest of the paper, we introduce the Zebra framework that solves these practical issues efficiently and then show the encoding scheme under the Zebra framework.

V. Zebra Framework

A. Solving ranks efficiently

In the Zebra framework, we propose a few heuristics to compute ranks of blocks efficiently. We start from temporarily removing the integer constraint (3) and resolving the complexity issues of the non-integer geometric programming problem in (1)-(2) by studying its property.

Without loss of generality, we assume that in D, $d_i \geq d_j$ if $i > j$. In other words, we sort the element in D in a non-ascending order. In this way, an optimal solution of K in (1)-(2) should also be in non-descending order.

We prove this property by contradiction. Assume that there exist i and j in an optimal solution of (1)-(2) such that $k_i > k_j$ where $i < j$, and then the reconstruction overhead of block i and j is $d_i k_i + d_j k_j$. If $d_i > d_j$, it is straightforward that $d_i k_i + d_j k_j > d_i k_j + d_j k_i$. Therefore, we can get a solution with even lower overall reconstruction overhead by exchange the rank of block B_i and B_j, which is contradictory to the assumption that the original solution is optimal. On the other hand, if $d_i = d_j$, we can assign a new rank, $\frac{2k_i k_j}{k_i + k_j}$, to both block B_i and B_j to get a lower overall reconstruction overhead, while still satisfying the condition in (2). This is also contradictory to the assumption that the original solution is optimal.

From this property, we can directly get a corollary that $k_i = k_j$ if $d_i = d_j$. In other words, if two block have the same demand, they will also have the same rank in the optimal solution. Therefore, inspired by this property, we can refine the original model to significantly decrease the complexity to solve it, by reducing the number of variables to solve.

First, we assume that all blocks of the same file should have the same or similar demand in a distributed storage system. In practice, if the demand of blocks in a single file is not the same, we can use the demand of the hottest block of the file as the demand of the file. The intuition of this assumption is that typically a distributed storage system that stores large-size files will have distributed data processing system running upon it, such as Hadoop and Spark, which will visit each block of the file distributively. Therefore, once a file is visited, all of its blocks will probably be visited with the same demand. Notice that if all blocks of the same file have the same demand, this step will not hurt the optimality of the solution.

Second, we extend this property by assuming that blocks with similar demand will also have similar ranks. Thus, we can classify the demand of all blocks into discrete categories. The simplest way is to set a parameter t where any demand that falls into the interval $(tx-t, tx]$ will be approximated as $tx, \forall x \in \mathbb{Z}^*$. Hence, files with similar demand will be merged into the same one, and we can further reduce the complexity to solve the model. For the cold data, this is especially useful, as cold data typically occupy a very large portion of all the data, yet with similar demand of very little values. Thus, we can quickly categorize cold data into few intervals, and then thousands of files can be grouped into few ones.

After these two steps, we can refine the model such that there are n files, where each file F_i is associated with size w_i and demand d_i, $i = 1, \ldots, n$. The sum of of the size of each file must be N, *i.e.*, $\sum_{i=1}^{n} w_i = N$. Each file will be encoded with a (k_i, r) RS code, and the overall storage overhead should be no more than C. Hence, the problem to solve the optimal K can be redefined as

$$\min \quad \sum_{i=1}^{n} w_i d_i k_i \tag{4}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} \frac{w_i}{k_i} \leq \frac{(C-1)N}{r} \tag{5}$$

$$k_i > 0, \forall i. \tag{6}$$

This is still a geometric programming problem. Notice that here we remove the constraint (3) that each k_i must be an integer. In the Zebra framework, we will solve this problem first, and then round k_i to an integer. For now we always round k_i to $\lceil k \rceil$ in the approximated solution. In this way, we won't break the requirement of storage overhead in (4), while the worst case of additional reconstruction overhead is $\sum_{i=1}^{n} w_i d_i$. Thus, the approximation ratio of this solution is $1 + \frac{1}{\min_i k_i}$.

In practice, the approximation ratio of a real workload can be close to 1. To demonstrate this gap, we run the simulation on the workload from Facebook that we show in Fig. 2. In this workload, there are 15565 files in total. If we store data into blocks of 64 MB, there will be 1.8×10^7 blocks. The result in Fig. 4a shows that we can reduce the number of files into 55 even when $t = 1$. Notice that when $t = 1$, we actually don't merge files unless their demand is exactly the same. When we increase the value of t, we can even further reduce the complexity of solving k. In fact, we can always solve K within 0.1 seconds for this workload, with any value of t.

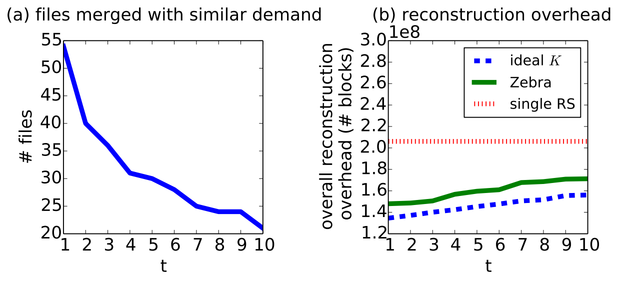


Fig. 4. Complexity and approximation ratio of the refined model.

We calculate the overall reconstruction overhead in Fig. 4b. The overall reconstruction overhead is calculated by (4), which represents the reconstruction overhead of all block visits under the given demand. We assume that any two block failures should be tolerated, *i.e.*, $r = 2$, and the overall storage overhead should be no more than 1.25. We can see that the performance of the approximated solution of K (approximated K) is close to the non-integer solution of K (ideal K) in (4)-(6), with 11.4% more reconstruction overhead in the worst case. Compared to the single erasure code, which we use $(8, 2)$ RS code in this case to meet the requirement of storage overhead, the approximated solution of K can save up to 28.2% reconstruction overhead.

B. *Encoding data in multiple tiers*

From the ranks solved above, we can now encode data into multiple tiers. The number of tiers is determined by merging files with the same rank into one tier. We actually encode blocks in each tier with a (k_i, r) RS code, where k_i is the rank of the file and r is the number of failures to tolerate. In particular, for the blocks of rank 1, *i.e.*, $k_i = 1$, they will be replicated with $1 + r$ copies. Since data in each tier are migrated from multiple files, the chance of having a tier with no more than k_i blocks is negligible. Inside each tier, we encode every k_i blocks into r parity blocks with the (k_i, r) RS code where we term such a group of $k_i + r$ blocks as a *stripe*. All blocks in each stripe will be stored into different servers. If we have remaining blocks or the total number of blocks is less than k_i, we will temporarily encode them with an (l, r) RS code if the number of remaining blocks is l. Since $l < k_i$, this will incur additional storage overhead. However, given the large number of blocks stored in a distributed storage system, this additional storage overhead is marginal.

VI. Deploying Zebra with Online Demand

A. Online demand

Before we deploy the Zebra framework in any practical scenarios, we should be aware that ranks inside the Zebra framework can only be constructed with the given demand, however, they must work well with the demand in the future.

To meet this requirement with online demand, the demand D we use in the model will not be purely determined by the most recent demand, but a linear combination of demand over a longer period of time. Specifically, we split the time into intervals. For example, if the interval is one hour, we measure the demand of each file every hour, and the demand measured in this hour is the most recent demand in the next interval.

If we use the most recent demand to solve K, we may imprudently increase k_i of some block if this block gets transiently high demand in the latest interval and in the next interval there will be no such high demand. Therefore, we also need to consider the consistency of the demand besides the latest demand. In this paper, we use α to achieve a flexible tradeoff between the consistency and the transiency of the demand. Assume that D_0 is the most recent demand and D is the demand we use to calculate K in the last interval. After this interval, we are going to update the demand D as

$$(1 - \alpha)D + \alpha D_0, \alpha \in [0, 1]. \tag{7}$$

It is straightforward that when $\alpha = 1$, we will always use the latest demand to calculate K used in the next interval. On the other hand, as α goes to 0, the latest demand will be less and less taken into account. When $\alpha = 0$, D won't be updated at any time. In this way, the transient demand will get smoother over time. Therefore, we won't be easily tricked by the transient demand. In other words, a smaller α can help to make it more consistent in the updated demand D over time.

Once again, we run the simulation with the hourly updated demand on the workload from Facebook, with $r = 2$, $C = 1.25$, and $t = 1$. Fig. 5 illustrates the results. Compared to the single RS code, Zebra can work well with online demand, and we can on average save 60.8% of reconstruction overhead in general. We can observe that in this workload, the transiency is quite significant (from the overhead of the single RS code), and thus with a larger α we can slightly better adapt to the general workload change. In fact, the best choice of α depends on the characteristics of the workload, and we will show the results with more workload in Sec. VII.

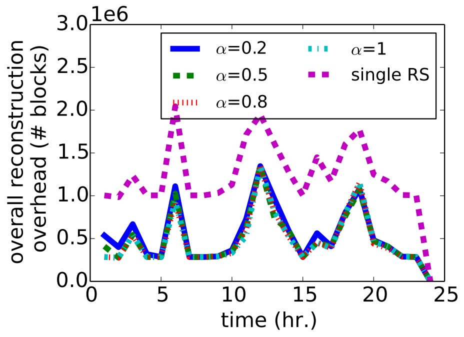


Fig. 5. Overall reconstruction overhead with demand updated online.

B. Data migration

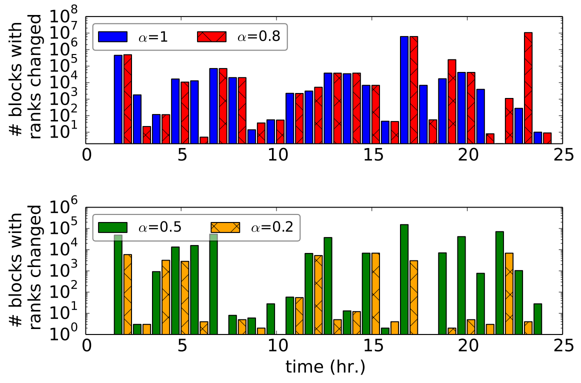
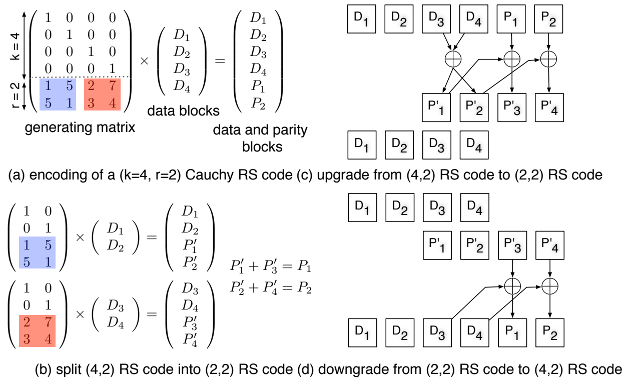


Fig. 6. The number of blocks to migrate into different ranks after each hour.

Once the demand changes after each time interval, we will need to update the files if their ranks are changed. This may involve quite a lot of migration overhead, especially the network transfer. An naive way to update files with new ranks is to compute new parity blocks with the new RS code and then remove the old parity blocks. To encode r parity blocks from k data blocks with a (k, r) RS code, we need to transfer at least $k + r - 1$ blocks, by computing all parity blocks on one server and sending $r - 1$ ones to other servers. In other words, if the rank of a file is updated, the traffic to generate new parity blocks will be even more than the amount of this file. Fig. 6 shows the number of blocks that have different ranks after each hour. We can see that the migration overhead varies with different values of α, and it is hard to predict (as we have more blocks to migrate when $\alpha = 0.8$ than $\alpha = 1$. In fact, there can be 650.5 TB of data to migrate to new RS codes at some hour when $\alpha = 0.8$. In other words, we need to generate new parity blocks for almost $\frac{2}{3}$ of all data. This will not only hinder the migration process from finishing quickly, but hurt the performance of the data access as well. Hence, though migration is unavoidable to meet the ever-changing demand, our objective is to reduce the network transfer for migration to a marginal level, for any values of α.

In this paper, we propose a different way to encode data such that we can control the migration overhead without increasing the overall reconstruction overhead significantly. We use Cauchy RS codes [26], [27] in Zebra, which contains a Cauchy matrix in its generating matrix. With a (k, r) RS code, we have a $(k + r) \times k$ matrix G as its *generating matrix*, such that the encoding operation can be formalized as the multiplication of the generating matrix and the k data blocks, as illustrated in Fig. 7a. In particular, if the first k rows of G are an identity matrix, the corresponding RS code will be systematic. In this way, we can write G as $G = \begin{bmatrix} I \\ \hat{G} \end{bmatrix}$. In a Cauchy RS code, the matrix $\hat{G}$ is a Cauchy matrix. An advantage of Cauchy RS code is that all encoding operations can be converted into XOR operations. More importantly, Cauchy matrix makes it easy to migrate from one RS code into another RS code with significantly less overhead, since any submatrix of a Cauchy matrix is still a Cauchy matrix (Fig. 7b).

Because of this property, we can easily downgrade or upgrade data between an (mk, r) and a (k, r) RS code, $m \in \mathbb{Z}^+$. We show in Fig. 7c and Fig. 7d how we can migrate between these two RS codes. To downgrade from a (k, r) RS code to an (mk, r) RS code, by applying the property of the Cauchy matrix, we only need to XOR the r parity blocks in the m stripes together. To upgrade from an (mk, r) RS code to a (k, r) RS code, we need to generate the parity blocks in the $m - 1$ stripes under the (k, r) RS code and XOR the new parity blocks and the existing parity blocks into the parity blocks of the last stripe. We show in Table I the network transfer of both upgrade and downgrade, as well as the network transfer of the naive migration scheme. We can see that the Cauchy matrix can help to save network transfer in both cases, when



k = 4
r = 2
generating matrix
data blocks
data and parity blocks
(a) encoding of a (k=4, r=2) Cauchy RS code
(c) upgrade from (4,2) RS code to (2,2) RS code
$P'_1 + P'_3 = P_1$
$P'_2 + P'_4 = P_2$
(b) split (4,2) RS code into (2,2) RS code
(d) downgrade from (2,2) RS code to (4,2) RS code

Fig. 7. Construction of the Cauchy RS code and its migration.

TABLE I
NETWORK TRANSFER (NO. OF BLOCKS) BETWEEN (mk, r) AND (k, r) RS CODES.

	without Cauchy matrix	with Cauchy matrix
downgrade	$mk + r - 1$	$(m - 1)r$
upgrade	$m(k + r - 1)$	$(m - 1)(k + 2r - 1)$

$m < 2 + \frac{k-1}{r}$. Apparently when $m = 2$ this condition can always be satisfied, and we will use this property to save the migration overhead.

However, we can rely on the Cauchy matrix only when the rank of the new code is multiple times or can be divided into the rank of the old code. To maximize this effect, we can also change the way to solve ranks. We set $k_{\max}$, an upper bound of ranks of all blocks, such that all k_i should be no more than $k_{\max}$. Moreover, the rank of all blocks should be a divisor of $k_{\max}$. Given k_i solved from (4)-(6) with an additional constraint $k_i \leq k_{\max}, \forall k_i$, we encode the corresponding block with a (k, r) RS code where k is the minimum divisor of $k_{\max}$ that is no less than k_i.

When we migrate m stripes of blocks encoded with a (k, r) RS code into one stripe with an (mk, r) RS code, we may also need to move data blocks because two blocks in different stripes may be stored in the same server. In Zebra, if $k_{\max}$ is set, we store every $k_{\max}$ data blocks into different servers, and compute parity blocks with the corresponding (k, r) RS code which will be stored into other servers. When we need to migrate into an (mk, r) RS code, we will not need to move any data blocks, but only migrate parity blocks as described above. This method can also be applied to the upgrade case as well. Apparently, we can maximize the effect of the Cauchy matrix by wisely selecting the value of $k_{\max}$. For example, when $k_{\max} = 16$, we have 5 available ranks $(1, 2, 4, 8, 16)$. Thus, when downgrading or upgrading to any neighbor ranks we can always exploit the Cauchy matrix to save network transfer (with $m = 2$). For some other values of $k_{\max}$, some rank may not be integer multiples of its neighboring rank (*e.g.*, 4 and 3 when $k_{\max} = 12$), we will have to remove all existing parity blocks and generate new parity blocks with the new ranks.

In Fig. 8, we compare the performance with different values of $k_{\max}$. We run the Facebook workload by updating ranks of data every hour and calculate all the network transfer incurred by the migration, with $C = 1.25$, $r = 2$, $t = 1$, and $\alpha = 0.8$. We can see that all values of $k_{\max}$ in Fig. 8 can significantly reduce the network transfer such that only 2.2% of the original network transfer will be incurred ($k_{\max} = 16$). The overall reconstruction overhead, on the other hand, will increase by between 8.7% ($k_{\max} = 24$) and 29.8% ($k_{\max} = 16$).

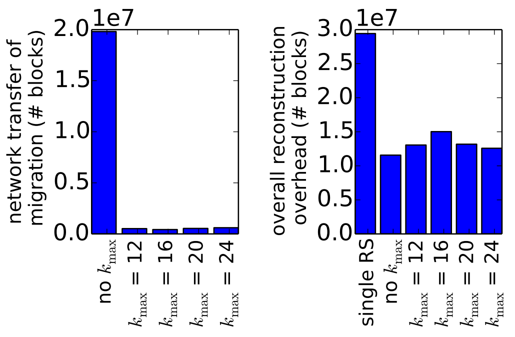


Fig. 8. Network transfer incurred by the migration and the overall reconstruction overhead with different values of $k_{\max}$, when $\alpha = 0.8$.

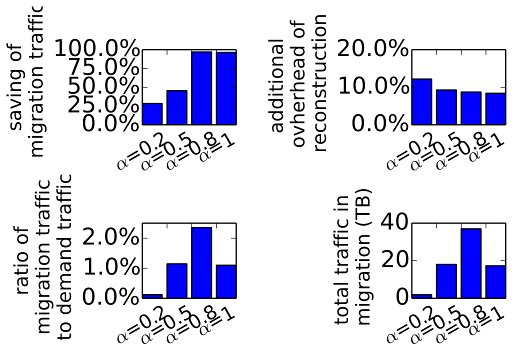


Fig. 9. Comparison of network transfer incurred by migration with various values of α, when $k_{\max} = 24$.

We show the results of other values of α in Fig. 9. We can see that with any values of α, the total network transfer for migration will be reduced to no more than 37 TB, which occupies less than 2.4% of all traffic incurred by demand. Meanwhile, at most 12.1% more reconstruction overhead will be incurred. Hence, we reduce the network transfer for migration to a marginal level without incurring much additional reconstruction overhead.

VII. Simulation

A. Methodology

We evaluate the performance of the Zebra framework with more workloads in this section. The two workloads [14] used in this paper are obtained from a 600-machine cluster at Facebook running MapReduce jobs, each of which spans 24 hours. In the workload, the demand of HDFS files is recorded. We show characteristics of the two workloads in Table II, and we have been using the FB2 workload to evaluate the design of Zebra throughout this paper so far. From Table II we can see that both workloads have a significant skewness of data demand, where more than half of data have demand less than the 1.6 and only a small portion has very high demand.

TABLE II
Characteristics of workloads used in the simulation.

workload	size	# files	max demand	mean demand
FB1	0.96 PB	15223	688	1.6
FB2	1.07 PB	16256	721	1.5

Like previous evaluations, we divide the time in each workload into intervals, where each time interval spans the same amount of time. In our simulation, we set each interval as one hour. In each interval, we calculate the rank of files with the demand updated by (7). The new file that appears for the first time in an interval will be stored with with a default rank. The default rank is calculated as the minimum divisor of $k_{\max}$ that is no less than $\frac{r}{C-1}$, and can hence achieve the required storage overhead C with r failures to tolerate. In the simulation, the size of each block is 64 MB. After each interval, the ranks of all existing files will be updated. Given the rank of each file, we can further calculate the average overhead of reconstruction per visit, the storage overhead of blocks, and the network transfer of the migration.

In the simulation, besides Zebra and one RS code, we add another scheme that encodes data into two tiers of RS codes. This scheme is similar to the method proposed in [24] which implements two tiers with two other preconfigured erasure codes. In our simulation, the two tiers both deploy RS codes for the purpose of fair comparison. For convenience, we name the two tiers as hot tier and cold tier, as the hot tier will store hot data with low reconstruction overhead while the cold tier can provide low storage overhead for the cold data. In this scheme, under the constraint of the overall storage overhead, we try to assign as much hot data as possible to the hot tier and store the rest in the cold tier. In the simulation, we deploy a $(4, 2)$ RS code in the hot tier and a $(12, 2)$ RS code in the cold tier.

B. Results

We first evaluate the reconstruction overhead. At each time interval, we calculate the average of the reconstruction overhead per visit, where we define the reconstruction overhead as the number of blocks to read in the reconstruction. In other words, if the rank of a block is k, its reconstruction overhead at that time is k as well. This average reconstruction overhead per visit can be considered as the expected reconstruction overhead of visiting an unavailable block once we have a failure in the distributed storage system. In all the simulations below, we set $C = 1.4$, $r = 2$, $t = 1$, $\alpha = 0.5$, and $k_{\max} = 24$, unless mentioned otherwise.

We show the average reconstruction overhead in Fig. 10, where we sort files by their demand, and then

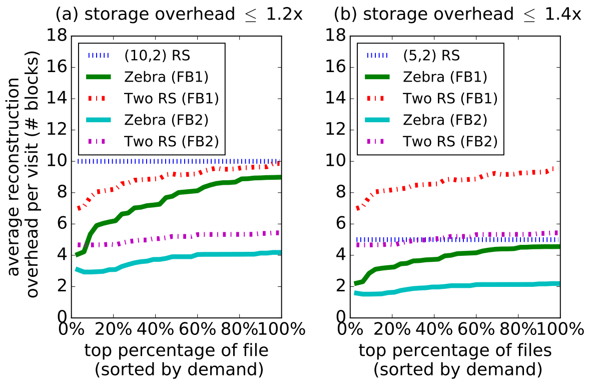


Fig. 10. Average reconstruction overhead per visit with storage overhead 1.2 and 1.4.

calculate the average reconstruction overhead per visit of files in a given percentage of top demanded files with $C = 1.2$ and 1.4, respectively. From Fig. 10a and Fig. 10b, we can see that all data can expect lower reconstruction overhead per visit than the single RS code with the same storage overhead. The reason is that in the Zebra framework, even though cold data may have higher reconstruction overhead than the single RS code, their demand can actually occupy a very small portion of all demand. Hence, on average, the reconstruction overhead per visit of all data can be lower than the single RS code, especially for the hotter data. The two RS codes can also achieve lower average reconstruction overhead than the single RS codes. However, Zebra can achieve better results as it tries to minimize the overall reconstruction overhead. Similar results can also be obtained when we require for a higher value of storage overhead in Fig. 10b, where with two RS codes we can not even compete with single RS codes due to the static configuration of RS codes. On average, we can save the average reconstruction overhead by 53.7% and 54.9% for the top 15% demanded files in Fig. 10a and Fig. 11b, respectively.

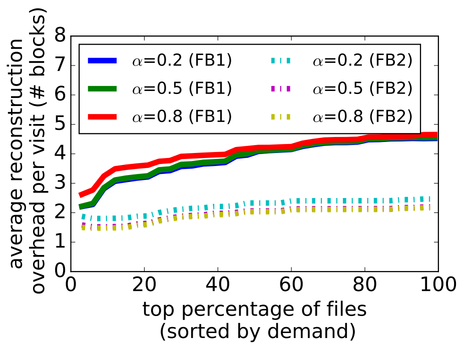


Fig. 11. Average reconstruction overhead with various values of α.

We compare the choices of α in Fig. 11, in which we use the same storage overhead as Fig. 10b. We find that the two workloads demonstrate different reconstruction overhead with different values of α, where FB1 favors lower α for the consistency while FB2 favors higher α for the transiency of the demand. Hence, we believe that the best choice of α depends on the characteristics of the workload.

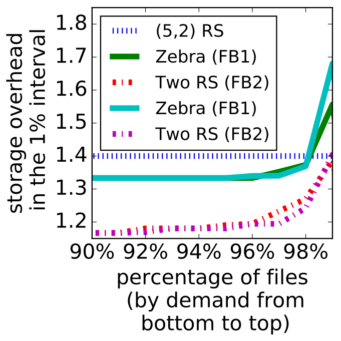


Fig. 12. Storage overhead of data with different demand.

We compare the storage overhead of data with different demand in Fig. 12, where we measure the average storage overhead of files in all time intervals. This time we sort the files by their demand (from bottom to top) and calculate the average storage overhead of data in a 1% interval of files. For example, data points with 99% on the x-axis indicate the average storage overhead of files with demand in the bottom 99%-100% interval, *i.e.,* the top 1% most demanded files. In fact, 90% files have very low demand in both FB1 and FB2, and hence they all have very similar storage overhead. In Fig. 12 we focus on the top 10% files. We can see that at most 2% files have storage overhead higher than the given constraint in Zebra, indicating their extremely high demand. On the other hand, due to the static configuration of the two RS codes, more data with high demand have to be stored in the cold tier with unnecessarily low storage overhead. Hence, with the two RS codes, we can not fully utilize the storage space, and this also explains why two RS codes have higher reconstruction overhead in Fig. 10b.

We illustrate the network transfer incurred by the migration in Fig. 13, by comparing it with the total network transfer to serve the demand. With various constraints of the overall storage overhead, we can see that the total migration traffic never exceeds the 10% of the total demand traffic, thanks to the Cauchy RS codes used in Zebra. Though more migration traffic can be observed with higher storage overhead, we can see that this increased amount of traffic is marginal.

VIII. Conclusions

In this paper, we exploit the skewness of demand in distributed storage systems and propose the Zebra

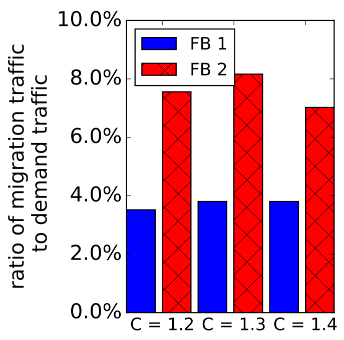


Fig. 13. The ratio of the total network transfer incurred by the migration to the network transfer incurred to serve demand.

framework that can efficiently encode data according to their demand into multiple tiers. By deploying the Zebra framework, we can achieve a much lower reconstruction overhead for the hot data, while spending less storage space to store the cold data. With the ever-changing demand, Zebra can update itself accordingly with a low network transfer in the migration.

Acknowledgment

This research is partially supported by the SAVI NSERC Strategic Networks project, two NSERC Strategic Partnership Projects, and an NSERC Discovery Research Program.

References

[1] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 2011, pp. 143–157.

[2] S. Ghemawat, H. Gobioff, and S. Leung, "The Google File System," in *Proc ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proc. 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010.

[4] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing Elephants: Novel Erasure Codes for Big Data," in *Proc. VLDB Endowment*, 2013.

[5] J. Arnold, "Erasure Codes With OpenStack Swift Digging Deeper," July 2013, https://swiftstack.com/blog/2013/07/17/erasure-codes-with-openstack-swift-digging-deeper/.

[6] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in Globally Distributed File System," in *Proc. USENIX Operating Systems Design and Implementation*, 2010.

[7] "HDFS-RAID," http://wiki.apache.org/hadoop/HDFS-RAID.

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

[9] "[HDFS-7295] Erasure Coding Support inside HDFS," https://issues.apache.org/jira/browse/HDFS-7285.

[10] H. Weatherspoon and J. D. Kubiatowicz, "Erasure Coding vs. Replication: A Quantitative Comparison," in *Proc. International Workshop on Peer-To-Peer Systems (IPTPS)*, 2002.

[11] I. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[12] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster," in *Proc. 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2013.

[13] Z. Wilcox-O'Hearn, "zfec 1.4.24," 2012, http://pypi.python.org/pypi/zfec.

[14] https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository.

[15] W. Wang and H. Kuang, "Saving capacity with HDFS RAID," 2014, https://code.facebook.com/posts/536638663113101/saving-capacity-with-hdfs-raid/.

[16] Y. Chen, S. Alspaugh, and R. Katz, "Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads," *Proc. VLDB Endow.*, vol. 5, no. 12, pp. 1802–1813, August 2012. [Online]. Available: http://dx.doi.org/10.14778/2367502.2367519

[17] D. S. Papailiopoulos, J. Luo, A. G. Dimakis, C. Huang, and J. Li, "Simple Regenerating Codes: Network Coding for Cloud Storage," in *Proc. of IEEE INFOCOM*, 2012.

[18] D. Papailiopoulos and A. Dimakis, "Locally Repairable Codes," in *Proc. IEEE International Symposium on Information Theory Proceedings (ISIT)*, July 2012, pp. 2771–2775.

[19] F. Oggier and A. Datta, "Self-repairing Homomorphic Codes for Distributed Storage Systems," in *Proc. of IEEE INFOCOM*, Apr. 2011.

[20] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A "HitchHiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers," in *Proc ACM SIGCOMM*, 2014.

[21] C. Huang, M. Chen, and J. Li, "Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems," *ACM Trans. Storage*, vol. 9, no. 1, pp. 3:1–3:28, March 2013. [Online]. Available: http://doi.acm.org/10.1145/2435204.2435207

[22] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network Coding for Distributed Storage Systems," *IEEE Trans. Inform. Theory*, vol. 56, no. 9, pp. 4539–4551, 2010.

[23] D. Borthakur, "HDFS Architecture Guide," *Hadoop Apache Project*, http://hadoop.apache.org/common/docs/current/hdfs_design.pdf.

[24] M. Xia, M. Saxena, M. Blaum, and D. A. Pease, "A Tale of Two Erasure Codes in HDFS," in *Proc. USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[25] S. Boyd and L. Vandenberghe, *Convex Optimization*. New York: Cambridge University Press, 2004.

[26] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman, "An XOR-based Erasure-resilient Coding Scheme," International Computer Science Institute, Tech. Rep. Technical Report TR-95-048, August 1995.

[27] J. Plank and L. Xu, "Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications," in *Proc. IEEE International Symposium on Network Computing and Applications*, July 2006, pp. 173–180.