Enabling Efficient and Reliable Transition from Replication to Erasure Coding for Clustered File Systems

Runhui Li, Yuchong Hu, and Patrick P. C. Lee

Abstract—To balance performance and storage efficiency, modern clustered file systems often first store data with replication, followed by encoding the replicated data with erasure coding. We argue that the commonly used random replication does not take into account erasure coding in its design, thereby raising both performance and availability issues in the subsequent encoding operation. We propose *encoding-aware replication*, which carefully places the replicas so as to (i) eliminate cross-rack downloads of data blocks during the encoding operation, (ii) preserve availability without data relocation after the encoding operation, and (iii) maintain load balancing across replicas as in random replication before the encoding operation. We conduct extensive HDFS-based testbed experiments and discrete-event simulations, and demonstrate the performance gains of encoding-aware replication over random replication.

Index Terms—replication, erasure codes, distributed storage systems, experiments and implementation

1 Introduction

Clustered file systems (CFSes) ensure data availability by striping data with redundancy across different nodes in different racks. Two redundancy schemes are commonly used: (i) *replication*, which creates identical replicas for each data block, and (ii) *erasure coding*, which transforms original data blocks into an expanded set of encoded blocks, such that any subset with a sufficient number of encoded blocks can reconstruct the original data blocks. Replication improves read performance as it can load-balance read requests across multiple replicas. On the other hand, erasure coding improves storage efficiency with much less storage redundancy, while it achieves the same or even higher fault tolerance than replication based on reliability analysis [32], [36]. For example, Azure reportedly reduces the storage overhead from 3× incurred by 3-way replication to 1.33× via erasure coding, leading to over 50% of operational cost saving for storage [19].

Recent studies [14], [19], [32] demonstrate the feasibility of adopting erasure coding in production CFSes. To balance the trade-off between performance and storage efficiency, CFSes often perform asynchronous encoding [14]: data blocks are first replicated when being stored, and are later encoded with erasure coding in the background. Asynchronous encoding maintains high read performance for new data via replication, while minimizing storage overhead for old data via erasure coding. It also simplifies deployment and error handling, and hides performance degradation [14].

In this paper, we argue that the *encoding* operation (i.e., transforming replicas to erasure-coded blocks) is subject to both performance and availability challenges. First, it may need to retrieve data blocks stored in different racks to generate encoded blocks. This consumes a substantial amount of bandwidth across racks. Cross-rack bandwidth is considered to be a scarce resource in CFSes [8], [11], and is often oversubscribed by a factor of 5 to 20 [2], [4], [17]. Thus, intensive cross-rack data transfers will degrade the performance of normal foreground operations. Second, relocation of encoded blocks may be needed to satisfy the availability requirement (e.g., rack-level fault tolerance). This not only again incurs additional cross-rack data transfers, but also leaves a vulnerable period before relocation is done.

Our observation is that when data blocks are first stored with replication, replica placement plays a critical role in determining both performance and availability of the subsequent encoding operation. One replica placement policy is *random replication* (RR) [9], whose idea is to store replicas across randomly chosen nodes. RR is simple to realize and has been deployed in HDFS [33], Azure [7], and the RAM-based storage system RAMCloud [25]. However, it does not take into account the relations among the blocks when encoding is performed. As we later show, RR brings both performance and availability issues to the subsequent encoding operation.

To this end, we propose *encoding-aware replication* (EAR), which carefully determines the replica placements of the data blocks that will be encoded with erasure coding. The main idea of EAR is that for each group of data blocks to be encoded together, EAR keeps one replica of each data block in the same rack, while storing the remaining replicas in other racks in such a way that the availability requirement is fulfilled. We determine the replica placement by equivalently solving a maximum matching problem. By doing so, EAR avoids downloading data blocks from other racks during the encoding operation, and avoids relocation of encoded blocks after the encoding operation. Also, EAR attempts to randomly distribute replicas as in RR to

- *R. Li and P. Lee are with the Chinese University of Hong Kong, Shatin, N.T., Hong Kong (*{rhli, pclee}@cse.cuhk.edu.hk*).*
- *Y. Hu is with the Huazhong University of Science and Technology, Wuhan, P. R. China (*yuchonghu@hust.edu.cn*).*
- *Corresponding author: Yuchong Hu.*
- *An earlier conference version of this paper appeared at the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2015 [21]. In this journal version, we extend our design to support Local Reconstruction Codes and include new evaluation results.*

maintain load balancing of replicas before the encoding operation. We summarize our contributions as follows.

- We present EAR, a new replica placement algorithm that addresses both performance and availability issues in encoding. EAR supports the classical Reed-Solomon codes [31] and the more recent Local Reconstruction Codes [19].
- We implement EAR on Facebook's HDFS prototype [13], with only few source code modifications.
- We conduct testbed experiments and compare RR and EAR for different topologies. EAR improves encoding throughput over RR in different settings, and the gain can reach over 100% in some cases. Also, EAR reduces the write response time during encoding, and maintains the performance of synthetic MapReduce workloads before encoding.
- We conduct discrete-event simulations based on CSIM 20 [10], and compare RR and EAR for various parameter choices in a 400-node CFS. We show that EAR can improve the encoding throughput of RR by 70% in many cases.
- We examine the replica distribution of EAR, and show that it maintains load balancing in storage and read requests as in RR.

The rest of the paper proceeds as follows. Section 2 presents the problem setting and issues of RR. Section 3 describes the design of EAR. Section 4 presents the implementation details of EAR on HDFS. Section 5 presents our evaluation results. Section 6 reviews related work, and finally Section 7 concludes the paper.

2 Problem

In this section, we formalize the scope of the encoding problem. We also motivate our work via an example.

2.1 System Model

We consider a clustered file system (CFS) architecture, as shown in Figure 1, that stores files over multiple storage *nodes* (or servers). We group the nodes into *racks* (let R be the number of racks), such that different nodes within the same rack are connected via the same top-of-rack switch, while different racks are connected via a network core that is composed of layers of aggregation switches. Cross-rack bandwidth is a scarce resource [8], [11] and often oversubscribed [2], [4], [17]. Thus, we assume that cross-rack data transfer is the performance bottleneck in a CFS architecture. We also consider a CFS that uses append-only writes and stores files as a collection of fixed-size *blocks*, which form the basic read/write data units. Examples of such a CFS includes GFS [16], HDFS [33], and Azure [7]. In this paper, we motivate the problem and our design based on the open-source HDFS implementation by Facebook [13], which supports erasure-coded storage based on HDFS-RAID [18]. Nevertheless, our discussion can be generalized for other CFS implementations.

Replication: Traditional CFS deployments use replication for fault tolerance, by storing multiple replicas of each block in different nodes, where the number of replicas is often set to be three (e.g., GFS [16], HDFS [33], and Azure [7]). One common replica placement policy is collectively called *random replication* (RR) [9], which is used by HDFS [33], Azure [7], and RAMCloud [25]. While the implementation of RR may slightly vary across different CFSes, the main idea of RR is to place replicas across randomly chosen nodes and racks for load balancing, and meanwhile ensure node-level and rack-level fault tolerance. In this paper, we assume that RR follows the default replica placement policy of HDFS [33]: it uses triple replication, such that the first replica is placed on a node in a randomly chosen rack and the two other replicas are replaced on different randomly chosen nodes in a different rack. This protects against either a two-node failure or a single-rack failure.

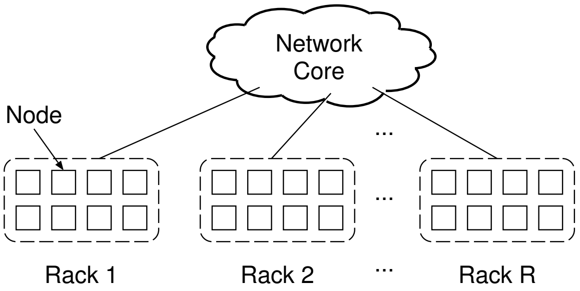


Fig. 1. Example of a CFS architecture.

Erasure coding: Erasure coding is a redundancy alternative that provably incurs less storage overhead than replication under the same fault tolerance [36]. In this paper, we mainly focus on *maximum distance separable (MDS)* codes, which are configured by two parameters n and k (where $k < n$). An (n, k) MDS code transforms k original uncoded blocks (called *data blocks*) to create $n - k$ additional coded blocks (called *parity blocks*), such that any k out of n data and parity blocks can reconstruct all k original data blocks, while the amount of storage redundancy is minimum for the required level of fault tolerance. By distributing the n blocks across n distinct nodes, we can tolerate any $n - k$ node failures. We call the collection of n data and parity blocks to be a *stripe*. Also, we focus on *systematic* codes, i.e., the k data blocks are kept in a stripe. Examples of MDS codes include Reed-Solomon codes [31] and Cauchy Reed-Solomon codes [5]. A CFS stores multiple stripes across all nodes, such that each stripe is encoded independently.

In this paper, we also consider non-MDS codes (e.g., Local Reconstruction Codes [19]), which incur a larger amount of storage redundancy than MDS codes (but still much less than replication) for the same fault tolerance. Unless otherwise specified, we assume that erasure-coded data is constructed by an (n, k) MDS code.

The trade-off of deploying erasure coding is its high performance cost, especially during *recovery* and *updates*. Recovery in erasure coding incurs excessive bandwidth and I/O, since recovering each failed block for an (n, k) code needs to retrieve k blocks for data reconstruction. Extensive studies (e.g., [12], [19], [32]) investigate how to improve recovery performance of erasure coding. Updates in erasure coding also triggers excessive bandwidth and I/O, as updating a data block for an (n, k) code also needs to update $n - k$ parity blocks. Thus, erasure coding is mainly used for

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2678505, IEEE Transactions on Parallel and Distributed Systems

storing warm/cold data that is less frequently accessed but needs long-term persistence [19], [24].

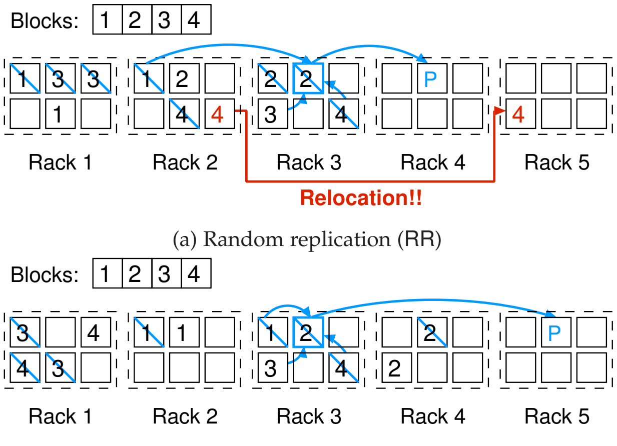
Encoding: Erasure-coded data is usually generated asynchronously in the background (i.e., off the write path) [14], [19], [32], in which all blocks are first replicated when being written to a CFS, and the CFS later transforms the replicas into erasure-coded data. In this paper, we refer to the transformation from replicas to erasure-coded data as the *encoding* operation. The CFS randomly selects a node to perform the encoding operation for a stripe. The encoding operation comprises three steps: (i) the node downloads one replica of each of the k data blocks; (ii) it transforms the downloaded blocks into $n-k$ parity blocks and uploads the parity blocks to other nodes; and (iii) it keeps one replica of each data block and deletes the other replicas. Facebook's HDFS prototype [13] performs asynchronous encoding via a map-only MapReduce job. We elaborate the implementation details in Section 4.

2.2 Issues of Random Replication (RR)

We elaborate how RR potentially harms both performance and availability of the subsequent encoding operation. First, encoding may incur a lot of cross-rack traffic. Facebook's HDFS computes parity blocks for each stripe by downloading and encoding a group of k data blocks from HDFS. However, if the blocks are randomly placed during replication, the encoding operation may have to download data blocks from different racks. Second, encoding may require block relocation to fulfill the fault-tolerance requirement. For example, Facebook's HDFS distributes n blocks of each stripe across n racks to tolerate $n-k$ rack failures [24], [28], [30] It periodically checks for the stripes that violate the rack-level fault tolerance requirement (using the *PlacementMonitor* module), and relocates the blocks if needed (using the *BlockMover* module). We emphasize that block relocation is rare in production CFSes [24], but if it happens, it introduces additional cross-rack traffic. It also leaves a vulnerable period before relocation is completed.

We illustrate the issues of RR via a motivating example. Consider a CFS with 30 nodes evenly grouped into five racks (i.e., six nodes per rack). Suppose that the CFS writes four blocks, denoted by Blocks 1, 2, 3, and 4, with the default 3-way replication. It then encodes the file via a $(5,4)$ erasure code, such that the erasure-coded stripe can tolerate a single-node failure or a single-rack failure. Figure 2(a) shows a possible replica layout of the four data blocks with RR and the subsequent encoding operation. To encode the four data blocks, suppose that a node in Rack 3 is chosen for performing the encoding operation. The chosen node can download Blocks 2, 3, and 4 from other nodes within the same rack, but it needs to download Block 1 from either Rack 1 or Rack 2 to compute parity block P. We call the cross-rack transfer of a data block a *cross-rack download*. We can check that even if we choose a node in another rack, we cannot avoid a cross-rack download.

We further show via simple analysis that when RR is used, it is almost inevitable to have cross-rack downloads in the encoding operation. Suppose that RR uses 3-way replication and places the replicas of each data block in two randomly chosen racks. Thus, the probability that Rack i ($1 \le i \le R$) contains a replica of a particular data block is $\frac{2}{R}$. Given that the replicas of k data blocks to be encoded into a stripe are placed in the same way, the expected number of data blocks stored in Rack i is $\frac{2k}{R}$. If we pick a random node to perform encoding, the expected number of data blocks to be downloaded from different racks is $k - \frac{2k}{R}$, which is almost k if R is large.



(a) Random replication (RR)

(b) Encoding-aware replication (EAR)

Fig. 2. Encoding of four data blocks under RR and EAR.

The same example also shows the availability issue. After we remove the remaining replicas (i.e., the crossed ones in Figure 2(a)), the failure of Rack 2 will result in data loss. Either Block 2 or Block 4 in Rack 2 needs to be relocated to Rack 5 to provide single-rack fault tolerance. Such an availability issue is less likely to occur if R is larger, because k data blocks are more likely to be scattered across k different racks, yet it remains possible.

To summarize, this example shows that RR potentially harms performance (i.e., a data block is downloaded from a different rack) and availability (i.e., blocks need to be relocated). The primary reason is that the replica layout of each data block is independently determined, while the data blocks are actually related when they are encoded together.

This motivates us to explore a different replica placement policy that takes into account the subsequent encoding operation. Figure 2(b) provides insights into the potential gain of the revised replica placement policy, which we call *encoding-aware replication (EAR)*. When the CFS writes the four data blocks with 3-way replication, we always keep one replica in one of the racks (Rack 3 in this case). Thus, if we choose a node in Rack 3 to perform encoding, we avoid any cross-rack download. Also, after encoding, the erasure-coded stripe provides single-rack fault tolerance without the need of relocation.

3 Design

In this section, we elaborate the design of EAR. EAR aims for the following design goals:

- *Eliminating cross-rack downloads:* The node that is selected to perform encoding does not need to download data blocks from other racks during the encoding operation. Note that the node may have to upload parity blocks to other racks in order to achieve rack-level fault tolerance.

- *Preserving availability without block relocation:* Both node-level and rack-level fault tolerance requirements are fulfilled after the encoding operation, without relocating any data or parity block.
- *Maintaining load balancing:* EAR tries to randomly place replicas for simplicity and load balancing as in RR before the encoding operation.

3.1 Eliminating Cross-Rack Downloads

3.1.1 Preliminary Design

The reason why cross-rack downloads occur is that it is unlikely for a rack to contain at least one replica of each of the k data blocks of a stripe. Thus, we present a *preliminary* design of EAR, which jointly determines the replica locations of k data blocks of the same stripe. For each stripe, we ensure that each of the k data blocks of the stripe must have at least one replica placed on one of the nodes within a rack, which we call the *core rack*.

The preliminary EAR works as follows. When a CFS stores each data block with replication, we ensure that the first replica is placed in the core rack, while the remaining replicas are randomly placed in other racks to provide rack-level fault tolerance as in RR. Once the core rack has stored k distinct data blocks, the collection of k data blocks is *sealed* and becomes eligible for later encoding. When the encoding operation starts, we first randomly select a node in the core rack to perform encoding for the stripe. Then the node can download all k data blocks within its rack, and there is no cross-rack download. For example, in Figure 2(b), Rack 3 is the core rack of the stripe, and Blocks 1 to 4 are all within Rack 3 and can be used for encoding.

In practice, the CFS may issue writes from different nodes and racks. We do not need to select a dedicated rack for all stripes as the core rack. Instead, each rack in the CFS can be viewed as a core rack for a stripe. For each data block to be written, the rack that stores the first replica will become the core rack that includes the data block for encoding. When a rack accumulates k data blocks, the k data blocks can be sealed for encoding. Thus, there are multiple core racks at a time, and the encoding of each stripe is handled by one of the core racks. Nevertheless, since stripes are encoded independently, our analysis focuses on a single stripe, and hence a single core rack.

3.1.2 Availability Violation

Our preliminary EAR only ensures that one replica (i.e., the first replica) of each data block resides in the core rack, but does not consider where the remaining blocks are placed after the encoding operation removes the redundant replicas. Thus, block relocation may be needed after the encoding operation so as to maintain rack-level fault tolerance. We elaborate the problem via a simple example. Consider the case where we place three data blocks via 3-way replication and then encode them via a $(4,3)$ erasure code. We also require to tolerate any $n-k=1$ rack failure. Without addressing availability after encoding, the preliminary EAR places the first replica of each data block in the core rack and the remaining two replicas in a randomly chosen rack different from the core rack, as in HDFS [33]. Then the random rack selection for each data block may happen to choose the same rack, meaning that the three replicas of each data block are always placed in the same two racks (i.e., the core rack and the chosen rack). In this case, regardless of how we delete redundant replicas, we must have a rack containing at least two data blocks. If the rack fails, then the data blocks become unavailable, thereby violating the single-rack fault tolerance. In this case, block relocation is necessary after encoding.

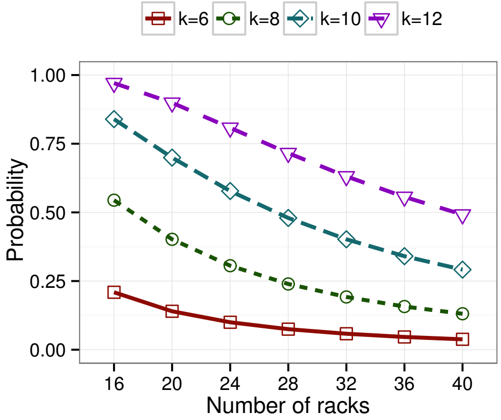


Fig. 3. Probability that a stripe violates rack-level fault tolerance.

As opposed to RR, the preliminary EAR has a smaller degree of freedom in placing replicas across racks. We show via analysis that the preliminary EAR actually has a very high likelihood of violating rack-level fault tolerance and hence triggering block relocation. Suppose that we store data with 3-way replication over a CFS with R racks and later encode the data with an (n,k) erasure code (where $R \geq n$), such that the first replicas of k data blocks are placed in the core rack and the second and third replicas are placed in one of the $R-1$ non-core racks. Thus, there are in total $(R-1)^k$ ways to place the replicas of the k data blocks. We also tolerate $n-k$ rack failures after the encoding operation as in Facebook [24], [28], [30].

Suppose that the second and third replicas of the k data blocks span $k-1$ or k out of $R-1$ non-core racks (the former case implies that the replicas of two of the data blocks reside in the same rack). Then we can ensure that the k data blocks span at least k racks (including the core rack). After we put $n-k$ parity blocks in $n-k$ other racks, we can tolerate $n-k$ rack failures. Otherwise, if the second and third replicas of the k data blocks span fewer than $k-1$ racks, then after encoding, we cannot tolerate $n-k$ rack failures without block relocation. Thus, the probability that a stripe violates rack-level fault tolerance (denoted by f) is:

$$f = 1 - \frac{\binom{R-1}{k} \times k! + \binom{k}{2}\binom{R-1}{k-1} \times (k-1)!}{(R-1)^k}. \qquad (1)$$

Figure 3 plots f for different values of k and R using Equation (1). Note that $k=10$ and $k=12$ are chosen by Facebook [32] and Azure [19], respectively. We see that the preliminary EAR is highly likely to violate rack-level fault tolerance (and hence requires block relocation), especially when the number of racks is small (e.g., 0.97 for $k=12$ and $R=16$).

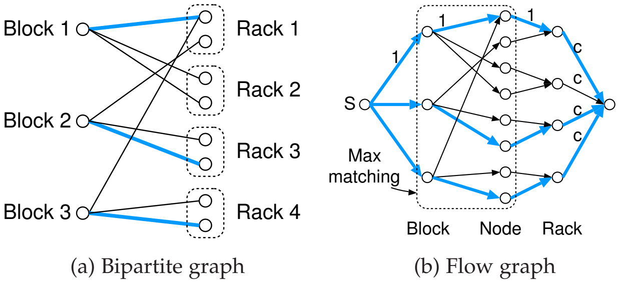


Fig. 4. Examples of a bipartite graph and a flow graph.

3.2 Preserving Availability Without Block Relocation

We now extend the preliminary EAR to preserve both node-level and rack-level fault tolerance after the encoding operation without the need of block relocation. Specifically, we configure an (n, k) erasure code to tolerate $n - k$ node failures by placing n data and parity blocks of a stripe on different nodes. Also, we configure a parameter c for the maximum number of blocks of a stripe located in a rack. Note that this implies that we require $R \geq \frac{n}{c}$, so that a stripe of n blocks can be placed in all R racks. Since a stripe can tolerate a loss of $n-k$ blocks, the CFS can tolerate $\lfloor \frac{n-k}{c} \rfloor$ rack failures. Thus, EAR is designed based on (n, k) and c.

We illustrate the design via an example. We consider a CFS with eight nodes evenly grouped into four racks (i.e., two nodes per rack). We write three data blocks using 3-way replication, and later encode them with a $(4, 3)$ erasure code to tolerate a single node failure. We set $c = 1$, so as to tolerate $\lfloor \frac{4-3}{1} \rfloor = 1$ rack failure.

Given (n, k) and c, our goal is to find a qualified replica placement for k blocks, such that one replica of each of the k blocks is placed in the core rack and other replicas are placed in other racks in such a way that after an encoding operation, both node-level and rack-level fault tolerance can be achieved without block relocation. We determine the replica placement of the k blocks by solving a *maximum matching* problem, whose solution can be found by solving an equivalent maximum flow problem. In the following, we show how we formulate the problem, while we elaborate the exact replica placement algorithm of EAR in Section 3.3.

Specifically, we first map the replica locations of data blocks to a bipartite graph as shown in Figure 4(a), with the vertices on the left and on the right representing blocks and nodes, respectively. We partition node vertices into the racks to which the nodes belong. An edge connecting a block vertex and a node vertex means that the corresponding block has a replica placed on the corresponding node. Since each replica is represented by an edge in the bipartite graph, the replicas of data blocks that are kept (i.e., not deleted) after encoding will form a set of edges. If the set is a maximum matching of the bipartite graph (i.e., every replica is mapped to exactly one node vertex) and no more than c edges from the set is linked to vertices in one rack (i.e., each rack has at most c data blocks), then we fulfill the rack-level fault tolerance requirement. Later, we deliberately place the parity blocks on the nodes that maintain rack-level fault tolerance by assigning parity blocks to the nodes of other racks that have fewer than c blocks of the stripe.

To determine if a qualified maximum matching exists, we now transform the problem to a maximum flow problem. We augment the bipartite graph to a flow graph as shown in Figure 4(b), in which we add source S, sink T, and the vertices representing racks. S connects to every block vertex with an edge of capacity one, meaning that each block keeps one replica after encoding. Edges in the bipartite graph are mapped to the edges with capacity one each. Each node vertex connects to its rack vertex with an edge of capacity one, and each rack vertex connects to T with an edge of capacity c ($c = 1$ in this example), ensuring that each node stores at most one block and each rack has at most c blocks after the encoding operation. If and only if the maximum flow of the flow graph is k, we can find a maximum matching and further determine the replica placement.

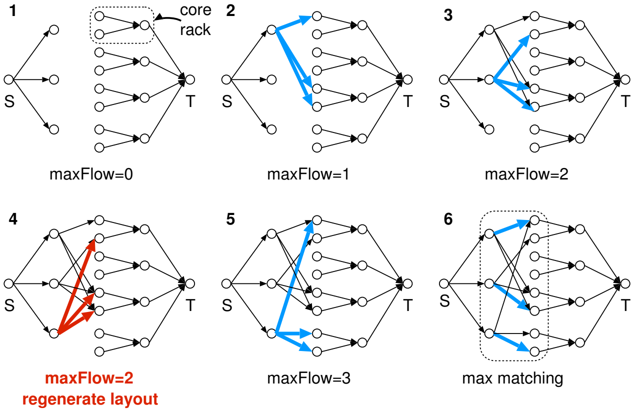


Fig. 5. Illustration of EAR.

3.3 Algorithm

Based on Sections 3.1 and 3.2, we propose an algorithm for EAR to systematically place replicas of data blocks. Its key idea is to randomly place the replicas as in RR, while satisfying the constraints imposed by the flow graph. Specifically, for the i-th data block (where $1 \leq i \leq k$), we place the first replica on one of the nodes in the core rack, and then randomly place the remaining replicas on other nodes based on the specified placement policy. For example, for 3-way replication, we can follow HDFS [33] and place the second and third replicas on two different nodes residing in a randomly chosen rack aside the core rack (which holds the first replica). In addition, we ensure that the maximum flow of the flow graph is i after we place all replicas of the i-th data block.

Figure 5 shows how we place the replicas of each data block for a given stripe. We first construct the flow graph with the core rack (Step 1). We add the first and second data blocks and add the corresponding edges in the flow graph (Steps 2 and 3, respectively). If the maximum flow is smaller than i (Step 4), we repeatedly generate another layout for the replicas of the i-th data block until the maximum flow is i (Step 5). Finally, the maximum matching can be obtained through the maximum flow (Step 6). In our conference version [21], we show that the expected number of iterations for generating a qualified replica layout in Step 5 is generally very small; we omit details here in the interest of space.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2678505, IEEE Transactions on Parallel and Distributed Systems

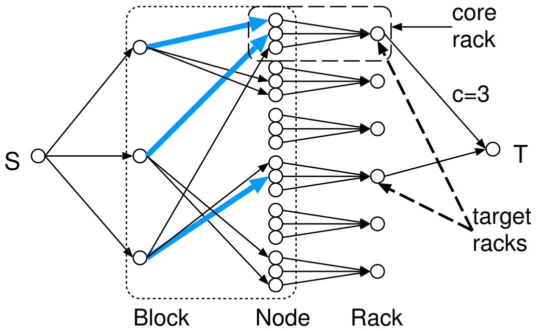


Fig. 6. Revised flow graph with target racks.

3.4 Improving Recovery Performance

To tolerate any $n-k$ rack failures, existing CFS deployments place n data and parity blocks of the same stripe in n different racks (e.g., in Facebook [24], [28], [30], Azure [19], and GFS [15]). One performance issue is that in order to recover a failed block, a node needs to download k blocks. Although one block can be downloaded within the same rack, the other $k-1$ blocks need to be downloaded from other racks. This introduces high cross-rack recovery traffic, which is undesirable [29].

We can reduce the number of racks where a stripe resides in return for lower cross-rack recovery traffic, by setting $c > 1$ to relax the requirement of tolerating $n-k$ rack failures. Specifically, we randomly pick R' out of R racks (where $R' < R$) as *target racks*, such that all data and parity blocks of every stripe must be placed in the target racks. To construct a flow graph for EAR, we keep only the edges from the target racks to the sink, but remove any edges from other non-target racks to the sink. We run the EAR algorithm (see Section 3.3) and find the maximum matching. Note that we require $R' \geq \frac{n}{c}$ (see Section 3.2).

We elaborate it via an example. Suppose that we encode three data blocks with a $(6,3)$ erasure code over a CFS with $R = 6$ racks. If we only require single-rack fault tolerance (i.e., $c = n-k = 3$), we can choose $R' = 2$ target racks and construct the flow graph as in Figure 6. Then we can ensure that after encoding, all data and parity blocks are placed in the target racks.

3.5 Extension for Local Reconstruction Codes

We have thus far focused on MDS codes. We now show how EAR can be extended to support non-MDS codes. We use Azure's Local Reconstruction Codes (LRC) [19] to motivate the extension. Note that our extensions are also applicable to Facebook's Locally Repairable Codes [32].

Overview of LRC: As opposed to (n,k) MDS codes, LRC is configured by three parameters: n, k, and l. An (n,k,l) LRC stripe comprises k data blocks, which are divided into l *local groups* with $\frac{k}{l}$ data blocks each. The data blocks in each local group are encoded to form a *local parity block*. Also, all k data blocks in the stripe are encoded to form $n-k-l$ *global parity blocks*. For example, Figure 7 shows a $(14,10,2)$ LRC stripe, in which there are 10 data blocks (i.e., $D_1, D_2, \cdots, D_{10}$), two local parity blocks (i.e., P_1^L and P_2^L), and two global parity blocks (i.e., P_1^G and P_2^G). P_1^L and P_2^L are encoded from the local groups $\{D_1, D_2, \cdots, D_5\}$ and $\{D_6, D_7, \cdots, D_{10}\}$, respectively, while P_1^G and P_2^G are encoded from all 10 data blocks. One key advantage of LRC is that to reconstruct any single lost data block, we only need to retrieve $\frac{k}{l}$ blocks in the same local group instead of k blocks in the same stripe. For example, if D_1 is lost, it can be reconstructed from D_2, D_3, D_4, D_5, and P_1^L. Since single failure recovery is the most common recovery scenario [15], [19], [29], LRC improves recovery performance in practice [19].

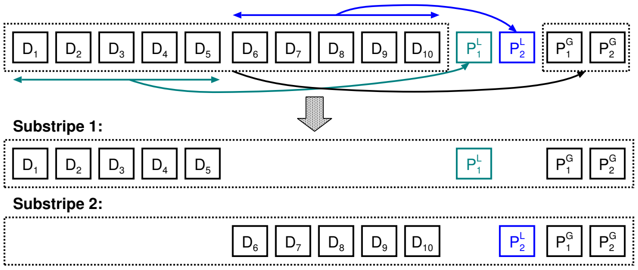


Fig. 7. Example of LRC: a $(14,10,2)$ LRC stripe.

Note that (n,k,l) LRC generally achieves less fault tolerance than (n,k) MDS codes under the same storage redundancy (and hence LRC is non-MDS). For example, while $(14,10)$ MDS codes can tolerate any arbitrary four-block failures, $(14,10,2)$ LRC can only tolerate a subset of four-block failures. To elaborate the fault tolerance property of LRC, we define a *substripe* as the set of blocks that comprise all data blocks in a local group, the corresponding local parity block, and all global parity blocks. For example, Figure 7 shows two substripes, i.e., Substripe 1 and Substripe 2. An LRC stripe can recover a failure if and only if both of the following conditions are satisfied:

1) *Global condition:* The stripe has no more than $n-k$ lost blocks, and
2) *Local condition:* Each substripe has no more than $n-k-l+1$ lost blocks.

Note that MDS codes achieve fault tolerance by satisfying only the global condition, while LRC needs to satisfy both the local and global conditions. The different fault tolerance property of LRC needs a different design of EAR if we encode files with LRC.

EAR for LRC: We now extend EAR for LRC. Our goal is to determine a replica layout that satisfies both global and local conditions of fault tolerance of LRC after encoding, subject to the required level of rack-level fault tolerance. Specifically, we create two flow graphs corresponding to the global and local conditions (call the graphs G_{global} and G_{local}, respectively). Each flow graph is initialized as in the same EAR as in Section 3.3, with one of the racks selected as the core rack. We configure c (i.e., the maximum number of blocks that can be stored in a rack after encoding) for each flow graph, based on the required level of rack-level fault tolerance. We then determine the replica layout of the data blocks for each substripe.

We illustrate the extended algorithm for EAR using $(14,10,2)$ LRC as an example, as shown in Figure 8. Suppose that our goal is to tolerate any single-rack failure after encoding. We first determine the replica layout for

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2678505, IEEE Transactions on Parallel and Distributed Systems

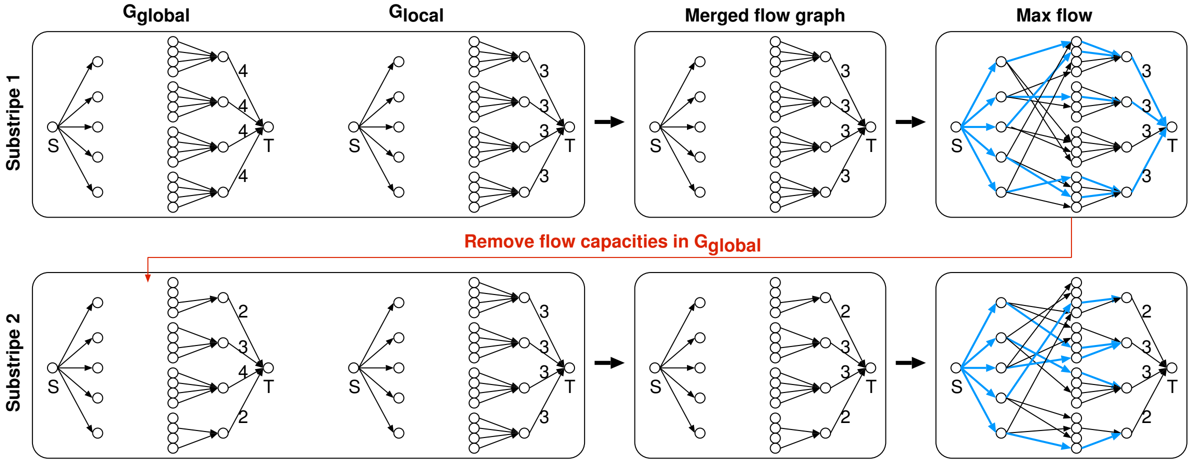


Fig. 8. Illustration of the extended EAR for LRC.

Substripe 1. Initially, we set $c = 4$ and $c = 3$ for G_{global} and G_{local} based on the global and local conditions, respectively. We *merge* both G_{global} and G_{local} into a single flow graph, such that each edge has the minimum capacity of the corresponding edges in G_{global} and G_{local}. This ensures that the replica layout for Substripe 1 satisfies both global and local conditions after encoding. We then generate a valid replica layout using the same EAR algorithm as in Algorithm 3.3, and find the corresponding maximum flow.

We next determine the replica layout for Substripe 2 by initializing G_{global} and G_{local}. However, the global condition also depends on the replica layout of Substripe 1, so we subtract the capacity of each edge in G_{global} by the maximum flow for Substripe 1, as shown in Figure 8. We again merge both G_{global} and G_{local} as above and find a valid replica layout for Substripe 2. We can apply the same idea for general (n, k, l) LRC.

4 Implementation

We implement EAR as an extension to Facebook's HDFS [13]. The source code of our EAR implementation is available at **http://ansrlab.cse.cuhk.edu.hk/software/ear**.

4.1 Overview of Erasure Coding in HDFS

The original HDFS architecture [33] comprises a single *NameNode* and multiple *DataNodes*. The NameNode stores the metadata (e.g., locations) for HDFS blocks, while the DataNodes store HDFS blocks. Facebook's HDFS implements erasure coding in a layer called HDFS-RAID [18], which manages the erasure-coded blocks on HDFS. On top of HDFS, HDFS-RAID introduces a *RaidNode* to coordinate the encoding operation. The RaidNode also periodically checks for any lost or corrupted blocks, and activates recovery for those blocks. Facebook's HDFS supports *inter-file* encoding, such that the data blocks of a stripe may belong to different files.

HDFS-RAID executes encoding through MapReduce [11], which uses a single *JobTracker* to coordinate multiple *TaskTrackers* on MapReduce processing. To perform encoding, the RaidNode first obtains metadata from the NameNode and groups every k data blocks into stripes. It then submits a map-only MapReduce job (without any reduce task) to the JobTracker, which assigns multiple map tasks to run on different TaskTrackers, each of which performs encoding for a subset of stripes. For each stripe, the responsible TaskTracker issues reads to k data blocks in parallel from different DataNodes. Once all k data blocks are received, the TaskTracker computes the parity blocks and writes them to HDFS. Currently, we use the Reed-Solomon codes [31] implemented by HDFS-RAID as our erasure coding scheme. Finally, the replicas of the data blocks are deleted.

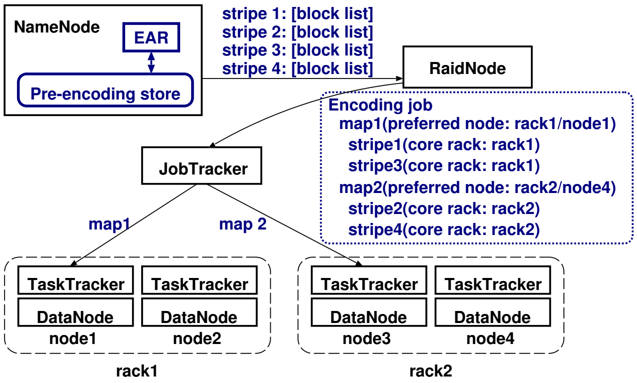


Fig. 9. Integration of EAR into Facebook's HDFS.

4.2 Integration

Figure 9 depicts how we modify HDFS to integrate EAR. Our modifications are minimal, and they can be summarized in three aspects.

First, we add the replica placement algorithm of EAR into the NameNode. EAR returns the following information: (i) which DataNodes the replicas of a data block are to be stored, (ii) which data blocks are encoded into the same stripe in the subsequent encoding operation, and (iii) which replicas of a data block are to be deleted after encoding while ensuring rack-level fault tolerance. We implement a *pre-encoding store* in the NameNode to keep track of each stripe and the list of data block identifiers that belong to the stripe.

We also modify how the RaidNode instructs MapReduce to perform encoding of a stripe in the core rack. To achieve this, we note that the MapReduce framework provides an interface to specify which *preferred node* to run a map task. Specifically, the RaidNode first obtains the list of data block identifiers for each stripe from the pre-encoding store. It then queries the NameNode for the replica locations (in terms of the hostnames of the DataNodes) of each data block. With the returned locations, the RaidNode identifies the core rack of each stripe. When the RaidNode initializes a MapReduce job for encoding, it ensures that a map task encodes multiple stripes that have a common core rack. This is done by attaching the map function with a preferred node, which we choose to be one of the nodes in the common core rack. In this case, the JobTracker tries to schedule this map task to run on the preferred node, or nearby nodes within the core rack, based on locality optimization of MapReduce [11].

The above modifications still cannot ensure that the encoding operation is performed in the core rack, since all nodes in the core rack may not have enough resources to execute a map task for encoding and the JobTracker assigns the map task to a node in another rack. This leads to cross-rack downloads for the encoding operation. Thus, we modify the MapReduce framework to include a Boolean flag in a MapReduce job to differentiate if it is an encoding job. If the flag is true, then the JobTracker only assigns map tasks to the nodes within the core rack. Note that this modification does not affect other non-encoding jobs.

5 Evaluation

We now present evaluation results for EAR through three aspects: (i) testbed experiments, in which we examine the practical deployment of EAR on HDFS; (ii) discrete-event simulations, in which we evaluate EAR in a large-scale setting subject to various parameter choices, and (iii) load balancing analysis, in which we justify EAR maintains load balancing as in RR.

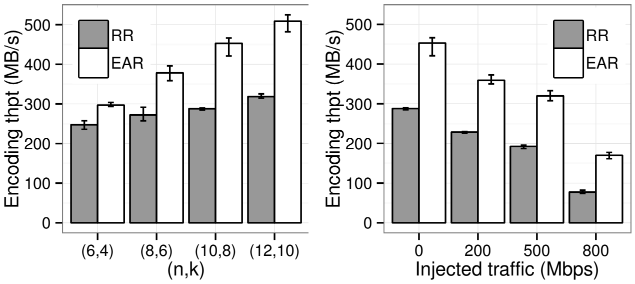
5.1 Testbed Experiments

We consider two CFS topologies, namely the flat topology and the oversubscribed topology. We assume that we encode the stored data with Reed-Solomon codes; we consider LRC codes in simulations (Section 5.2).

5.1.1 Flat Topology

We start with a flat topology, in which we distribute erasure-coded chunks across different nodes, each located in a distinct rack. Specifically, we set up a 13-machine HDFS cluster, in which each machine has an Intel Core i5-3570 3.40GHz quad-core CPU, 8GB RAM, and a Seagate ST1000DM003 7200RPM 1TB SATA disk, and runs Ubuntu 12.04. All machines are interconnected via a 1Gb/s Ethernet switch.

To create a flat topology, we configure each machine to reside in a rack by associating each machine with a unique rack ID, such that one machine (called the *master*) deploys the NameNode, RaidNode, and JobTracker, and each of the remaining 12 machines (called the *slaves*) deploys one DataNode and one TaskTracker. We have validated that the network transfer over the 1Gb/s Ethernet switch is the bottleneck in our testbed.



(a) Different (n, k)

(b) Different UDP sending rates

Fig. 10. Experiment 1: Raw encoding performance of encoding 96 stripes. The endpoints of each error bar show the minimum and maximum values over five runs.

We fix the block size as 64MB, the default of HDFS. Since each rack (machine) has only one DataNode in our testbed, we use 2-way replication and distribute two replicas of each data block in two racks (machines), so as to provide single-rack fault tolerance as in the default HDFS (see Section 2.1). For each encoding job, we configure the RaidNode to launch 12 map tasks. All our results are averaged over five runs.

Experiment 1 (Raw encoding performance): We first study the raw encoding performance without write requests. We consider (n, k) erasure codes with $n = k + 2$, where k ranges from 4 to 10. We write $96 \times k$ data blocks (i.e., 24GB to 60GB of data) to HDFS with either RR or EAR. The RaidNode then submits an encoding job to encode the data blocks, and a total of 96 stripes are created. We evaluate the *encoding throughput*, defined as the total amount of data (in MB) to be encoded divided by the encoding time. Figure 10(a) shows the encoding throughputs of RR and EAR versus (n, k). The encoding throughputs increase with k for both RR and EAR, as we generate proportionally fewer parity blocks. If k increases from 4 to 10, the encoding throughput gain of EAR over RR increases from 19.9% to 59.7%, mainly because more data blocks are downloaded for encoding with a larger k.

We also compare the encoding throughputs of RR and EAR under different network conditions. We group our 12 slave machines into six pairs. For each pair, one node sends UDP packets to another node using the Iperf utility [20]. We consider different UDP sending rates, so that a higher UDP sending rate implies less effective network bandwidth, or vice versa. We fix $(10, 8)$ erasure coding and re-run the above write and encoding operations. Figure 10(b) shows the encoding throughputs of RR and EAR versus the UDP sending rate. The encoding throughput gain of EAR over RR increases with the UDP sending rate (i.e., with more limited network bandwidth). For example, the gain increases from 57.5% to 119.7% when the UDP sending rate increases from 0 to 800Mb/s.

Experiment 2 (Impact of encoding on write performance): We now perform encoding while HDFS is serving write requests. We study the performance impact on both write and encoding operations. Specifically, we fix $(10, 8)$ erasure coding. We first write 768 data blocks, which will later be encoded into 96 stripes. Then we issue a sequence

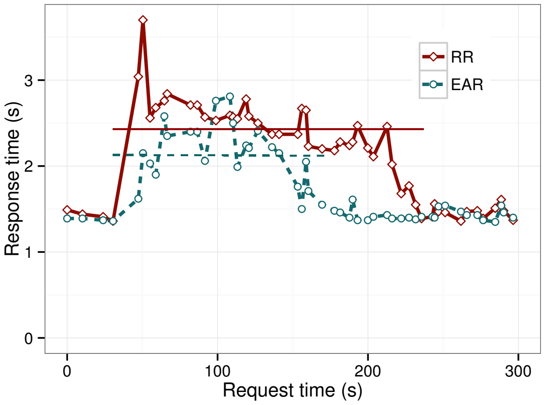


Fig. 11. Experiment 2: Impact of encoding on write performance under RR and EAR. The horizontal lines represent the average write response time during encoding and the duration of the whole encoding operation. For brevity, each data point represents the averaged write response time of three consecutive write requests.

of write requests, each of which writes a single 64MB block to HDFS. The arrivals of write requests follow a Poisson distribution with rate 0.5 requests/s. To repeat our test for five runs, we record the start time of each write request in the first run, and regenerate the write requests at the same start time in the following four runs. After we generate write requests for 30s, we start the encoding operation for the 96 stripes. We measure the response time of each write request and the total encoding time.

Figure 11 plots the response times of the write requests for both RR and EAR. Initially, when no encoding job is running (i.e., before the time 30s), both RR and EAR have similar write response times in around 1.4s. When the encoding operation is running, EAR reduces the write response time by 12.4% and the overall encoding time by 31.6% when compared to RR. This shows that EAR improves both write and encoding performance.

Experiment 3 (Impact of EAR on MapReduce): We study how EAR influences the performance of MapReduce jobs before encoding starts. We use SWIM [35], a MapReduce workload replay tool, to generate synthetic workloads of 50 MapReduce jobs derived from a trace of a 600-node Facebook production CFS in 2009. The generated workloads specify the input, shuffle, and output data sizes of each MapReduce job. Based on the workloads, we first write the input data to HDFS using either RR or EAR. Then we run MapReduce jobs on the input data, and write any output data back to HDFS, again using either RR or EAR. We configure each TaskTracker to run at most four map tasks simultaneously. We measure the runtime of the whole job, which includes reading and processing input data from HDFS, shuffling data, and outputting final results to HDFS.

Figure 12 shows the number of completed jobs versus the elapsed time under either RR or EAR. We observe very similar performance trends between RR and EAR. This shows that EAR preserves MapReduce performance on replicated data as RR.

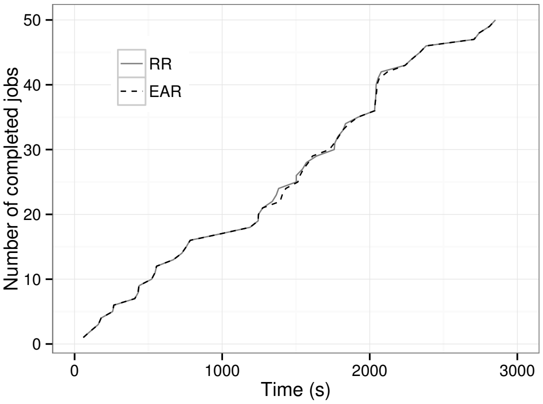


Fig. 12. Experiment 3: Impact of EAR on MapReduce performance. We observe similar performance trends between EAR and RR.

5.1.2 Oversubscribed Topology

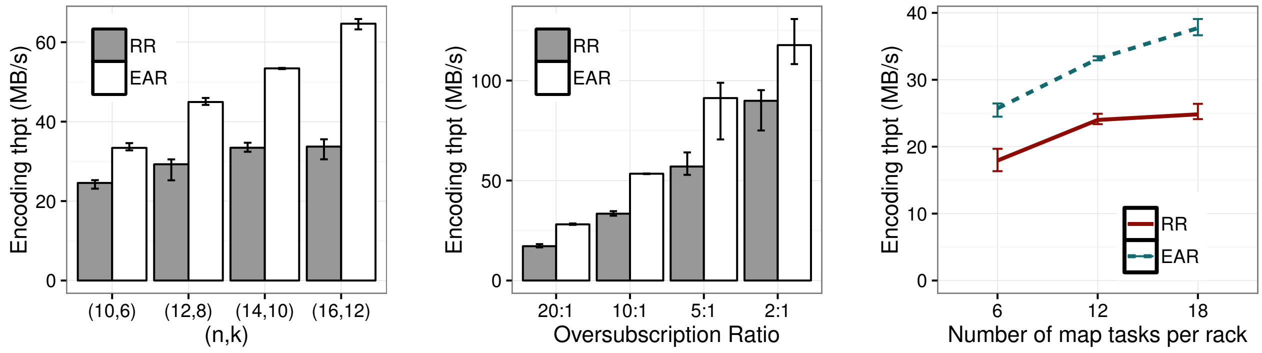
We now consider an oversubscribed topology. We configure a 19-machine Hadoop cluster that comprises a single master and 18 slaves. Each machine is equipped with an Intel Core i5-2400 3.40GHz quad-core CPU, 8GB RAM, and a Seagate ST31000524AS 7200RPM 1TB SATA disk, and runs Ubuntu 12.04. All hosts are connected via a 1Gb/s Ethernet switch. To mimic a network topology in Figure 1, we divide the 18 slaves into nine logical racks with two slaves each. We follow the approach in prior work [1] and use the network utility tool tc to limit the aggregate outgoing bandwidth from one logical rack to other logical racks. For example, if the oversubscription ratio is 2:1, we limit the aggregate outgoing bandwidth to 500Mb/s. We do not limit the bandwidth within the same logical rack.

Experiment 4 (Impact of an oversubscribed topology): We first consider different (n, k) combinations, where $n = k + 4$ and k varies from 6 to 12. We write 72 stripes of data blocks, whose total size ranges from 27GB to 54GB. We consider EAR and RR with three replicas. We fix the oversubscription ratio as 10:1 and configure the RaidNode to launch a 18-task encoding job. Figure 13(a) shows the encoding throughput versus (n, k). The encoding throughput gain of EAR over RR increases from 36.0% to 91.8% as k increases from 6 to 12. The results are consistent with those in a flat topology (see Figure 10(a)).

We also consider different oversubscription ratios. We fix (n, k) to (14,10) and vary the oversubscription ratio from 2:1 to 20:1. Figure 13(b) shows the encoding throughput versus the oversubscription ratio. For both EAR and RR, the encoding throughput decreases as the oversubscription ratio increases, as network resources become more limited. If the oversubscription ratio increases from 2:1 to 20:1, the encoding throughput gain of EAR over RR increases from 31.0% to 63.6%. This implies that EAR is more advantageous in a more resource-constrained environment.

Finally, we consider different numbers of map tasks launched by the encoding job. We fix the oversubscription ratio as 10:1. We re-configure the topology to form a six-rack cluster with three slaves in each rack. We set the number of map tasks to 6, 12 and 18, respectively. Figure 13(c) shows the encoding throughput versus the number of map tasks

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2678505, IEEE Transactions on Parallel and Distributed Systems



(a) Different (n, k)

(b) Different oversubscription ratio

(c) Different number of map tasks per rack

Fig. 13. Experiment 4: Impact of an oversubscribed topology on EAR and RR.

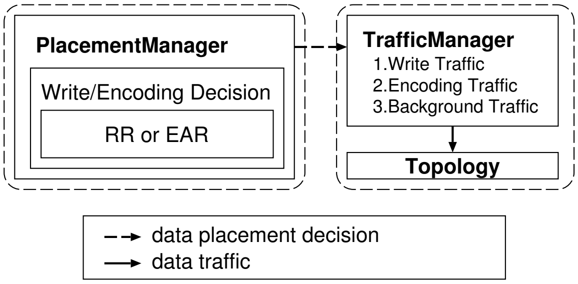


Fig. 14. Simulator overview.

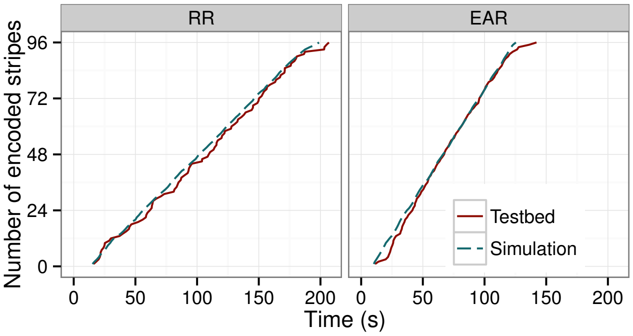


Fig. 15. Experiment 5: Simulator validation.

in the encoding job. Both EAR and RR see higher encoding throughput with more map tasks in the encoding job, since the network resources are better utilized. However, EAR shows more improvement as the number of map tasks increases. For example, when the number of map tasks increases from 12 to 18, the encoding throughput of EAR increases by 14.3%, while that of RR only increases by 3.5%.

5.2 Discrete-Event Simulations

We complement our testbed experiments by comparing RR and EAR via discrete-event simulations in a large-scale CFS architecture. We implement a C++-based discrete-event CFS simulator using CSIM 20 [10]. Figure 14 shows our simulator design. The *PlacementManager* module decides how to distribute replicas across nodes under RR or EAR during replication and how to distribute data and parity blocks in the encoding operation. The *Topology* module simulates the CFS topology and manages both cross-rack and intra-rack link resources. To complete a data transmission request, the *Topology* module holds the corresponding resources for some duration of the request subject to the specified link bandwidth. We assume that a CFS topology only contains nodes that store data (e.g., DataNodes of HDFS) and excludes the node that stores metadata (e.g., the NameNode of HDFS), as the latter only involves a small amount of data exchanges. The *TrafficManager* module generates three traffic streams: write, encoding, and background traffic, and feeds the streams simultaneously to the *Topology* module.

We simulate the write, encoding, and background traffic streams as follows. We first assign a node to perform each of the requests. For each write request, it receives replica placement decisions from the *PlacementManager* module and performs replication based on either RR or EAR. For each encoding request, it first obtains the replica locations of the data blocks for the stripe under RR or EAR from the *PlacementManager* module. It then downloads the data blocks and uploads the parity blocks. For each background request, it transmits a certain amount of data to another node, either in the same rack or a different rack.

Experiment 5 (Simulator validation): We first validate that our simulator can accurately simulate the performance of both RR and EAR. We simulate our testbed with the same topology (12 racks with one DataNode each) and the same link bandwidth (1Gb/s Ethernet). Using the simulator, we replay the write and encode streams using the same setting as in Experiment 2, such that we encode 96 stripes with $(10, 8)$ erasure coding after we issue write requests for 30s. We obtain the averaged simulation results from 30 runs with different random seeds and compare with the results of the testbed experiments. Figure 15 shows the cumulative number of encoded stripes versus the elapsed time from the start of the encoding operations for both testbed experiments and simulations. We observe that the simulator precisely captures the encoding performance under both RR and EAR. Table 1 also shows the averaged write response times in both testbed experiments and discrete-event simulations when the write requests are carried out with and without encoding in the background. We find that in all cases we consider, the response time differences between the testbed experiments and discrete-event simulations are

TABLE 1
Experiment 5: Validation of write response times.

	RR		EAR	
Time (in seconds)	testbed	simulation	testbed	simulation
Without encoding	1.43	1.40	1.42	1.40
With encoding	2.45	2.35	2.13	2.04

less than 4.3%. Thus, our simulator also precisely captures write performance.

Experiment 6 (Impact of parameter choices): We evaluate RR and EAR with our simulator in a large-scale setting. We consider a CFS composed of $R = 20$ racks with 20 nodes each, such that nodes in the same rack are connected via a 1Gb/s top-of-rack switch, and all top-of-rack switches are connected via a 1Gb/s network core. We configure the CFS to store data with 3-way replication, where the replicas are stored in two racks. The CFS then encodes the data with $(14, 10)$ erasure coding that can tolerate 4-node or 4-rack failures, as in Facebook [24]. Although RR may require block relocation after encoding to preserve availability, we do not consider this operation, so the simulated performance of RR is actually over-estimated.

We create 20 encoding processes, each of which encodes 50 stripes. We also issue write and background traffic requests, both of which follow a Poisson distribution with rate 1 request/s. Each write request writes one 64MB block, while each background traffic request generates an exponentially distributed size of data with mean 64MB. We set the ratio of cross-rack to intra-rack background traffic as 1:1.

We consider different parameter configurations. For each configuration, we vary one parameter, and obtain the performance over 30 runs with different random seeds. We normalize the average throughput results of EAR over that of RR for both encoding and write operations, both of which are carried out simultaneously. We present the results in *boxplots* and show the minimum, lower quartile, median, upper quartile, maximum, and any outlier over 30 runs.

Figure 16(a) first shows the results versus k, while we fix $n - k = 4$. A larger k implies less encoding redundancy. It also means that the cross-rack downloads of data blocks for encoding become more dominant in RR, so EAR brings more performance gains. For example, when $k = 12$, the encoding and write throughput gains of EAR over RR are 78.7% and 36.8%, respectively.

Figure 16(b) shows the results versus $n - k$, while we fix $k = 10$. A larger $n - k$ means more data redundancy (i.e., parity blocks). On one hand, since the effective link bandwidth drops, so EAR brings improvements by reducing cross-rack traffic. On the other hand, the gain of EAR over RR is offset since both schemes need to write additional parity blocks. The encoding throughput gain of EAR over RR remains fairly stable at around 70%, yet the write throughput gain of EAR over RR drops from 33.9% to 14.1%.

Figure 16(c) shows the results versus the link bandwidth of all top-of-rack switches and network core. When the link bandwidth is more limited, EAR shows higher performance gains. The encoding throughput gain of EAR reaches 165.2% when the link bandwidth is only 0.2Gb/s. Note that the encoding performance trend versus the link bandwidth is consistent with that of Experiment A.1 obtained from our testbed. The write throughput gain of EAR remains at around 20%.

Figure 16(d) shows results versus the arrival rate of write requests. A larger arrival rate implies less effective link bandwidth. The encoding throughput gain of EAR over RR increases to 89.1% when the write request rate grows to 4 requests/s, while the write throughput gain is between 25% and 28%.

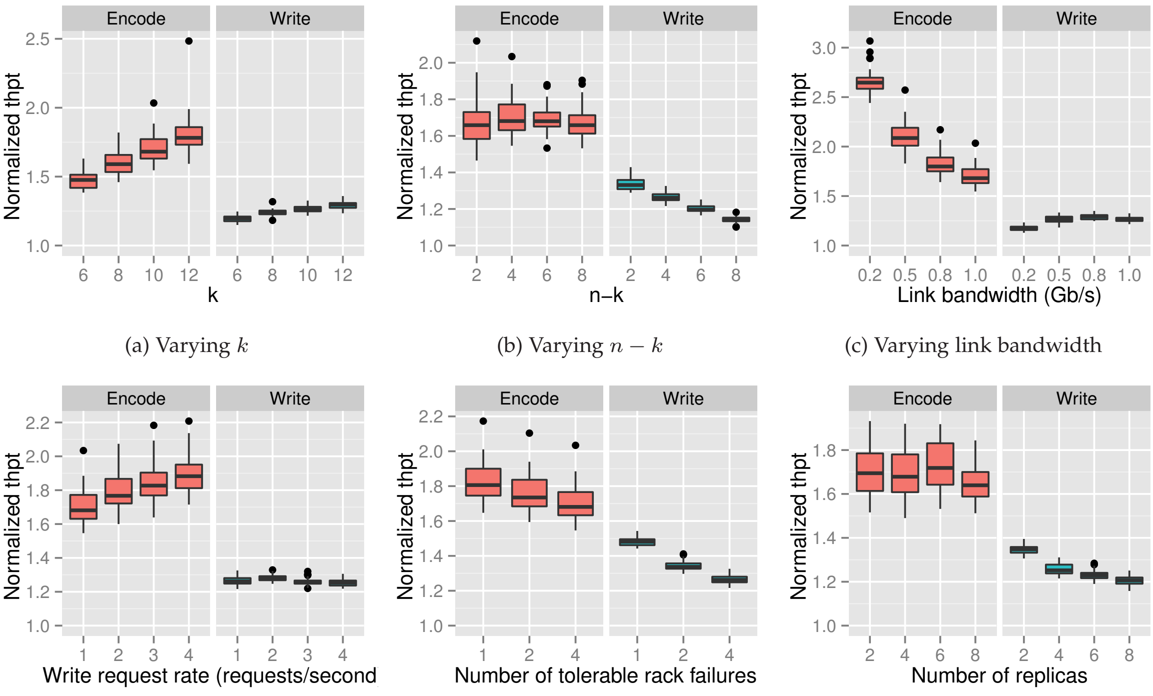
Recall that EAR can vary the rack-level fault tolerance by the parameter c (see Section 3.2). Here, we keep RR to still provide tolerance against $n - k$ rack failures, while we vary the rack-level fault tolerance of EAR. Figure 16(e) shows the throughput results versus the number of rack failures tolerated in EAR. By tolerating fewer rack failures, EAR can keep more data/parity blocks in one rack, so it can further reduce cross-rack traffic. The encoding and write throughput gains of EAR over RR increase from 70.1% to 82.1% and from 26.3% to 48.3%, respectively, when we reduce the number of tolerable rack failures of EAR from four to one.

Finally, Figure 16(f) shows the throughput results versus the number of replicas per data block. Here, we assume that each replica is placed in a different rack, as opposed to the default case where we put three replicas in two different racks. Writing more replicas implies less effective link bandwidth, but the gain of EAR is offset since RR now downloads less data from other racks for encoding. The encoding throughput gain of EAR over RR is around 70%, while the write throughput gain decreases from 34.7% to 20.5% when the number of replicas increases from two to eight.

Experiment 7 (Extension for LRC): Finally, we compare the performance gains when we apply EAR for both MDS codes and LRC. We consider (14,10) MDS codes and (14,10,2)-LRC subject to different rack-level fault tolerance (similar to Figure 16(e)). Figure 17 shows the normalized throughput of EAR over RR. The throughput improvements of LRC is less than that of MDS codes for single-rack and two-rack failure tolerance. The reason is that for LRC, we need to satisfy both the global and local conditions for fault tolerance, while for MDS codes, we only need to address the global condition. Thus, we have more constraints in placing the data and parity blocks in the same rack, and hence trigger more cross-rack traffic in general. Nevertheless, we still observe significant throughput gains of EAR over RR for LRC codes. For example, the encoding and write throughput gains are 78.0% and 35.6%, respectively, for single-rack fault tolerance.

5.3 Load Balancing Analysis

One major advantage of RR is that by distributing data over a uniformly random set of nodes, the CFS achieves both storage and read load balancing [9]. We now show via Monte Carlo simulations that although EAR adds extra restrictions to the random replica placement, it still achieves a very similar degree of load balancing to RR. In particular, we focus on rack-level load balancing, and examine how the replicas are distributed across racks. We consider the replica placement for a number of blocks on a CFS composed of $R = 20$ racks with 20 nodes each. We use 3-way replication,



(a) Varying k

(b) Varying $n - k$

(c) Varying link bandwidth

(d) Varying write request rates

(e) Varying rack-level fault tolerance of EAR

(f) Varying number of replicas

Fig. 16. Experiment 6: Impact of parameter choices on encoding and write performance under EAR and RR. Each plot denotes the normalized throughput of EAR over RR.

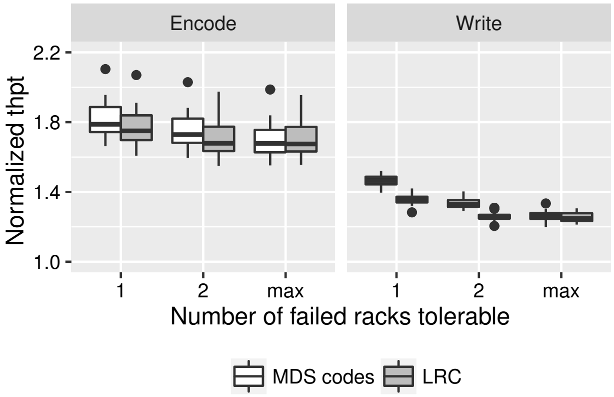


Fig. 17. Experiment 7: Extension for LRC. We compare the performance gains of EAR when we encode data with MDS codes and LRC.

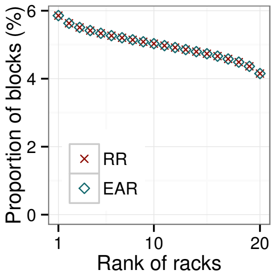


Fig. 18. Experiment 8: Storage load balancing.

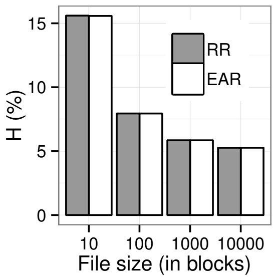


Fig. 19. Experiment 9: Read load balancing.

and the replicas are distributed across two racks as in HDFS [33]. For EAR, we choose $(14, 10)$ erasure coding. We obtain the averaged results over 1,000 runs.

Experiment 8 (Storage load balancing): We first examine the distribution of replicas across racks. We generate the replicas for 1,000 blocks and distribute them under RR or EAR. We count the number of replicas stored in each rack. Figure 18 shows the proportions of replicas of RR and EAR in each rack (sorted in descending order of proportions). Both RR and EAR have very similar distributions, such that the proportions of blocks stored in each rack are 4.1-5.9% for both RR and EAR.

Experiment 9 (Read load balancing): We also examine the distribution of read requests across racks. Suppose that the data blocks in a file are equally likely to be read, and the read requests to a data block are equally likely to be directed to one of the racks that contain a replica of the block. We define a hotness index $H = \max_{1 \le i \le 20}(L(i))$, where $L(i)$ denotes the proportion of read requests to Rack i, where $1 \le i \le 20$. Intuitively, we want H to be small to avoid hot spots. Figure 19 shows H versus the file size, varied from 10 to 10,000 blocks. Both RR and EAR have almost identical H.

6 Related Work

In this section, we review related work on erasure coding in CFSes and replica placement strategies in CFSes.

Erasure coding in CFSes: Researchers have extensively studied the applicability of deploying erasure coding in CFSes. Fan *et al.* [14] augments HDFS with asynchronous encoding to significantly reduce storage overhead. Zhang *et al.* [37] propose to apply erasure coding on the write path of HDFS, and study the performance impact on various MapReduce workloads. Li *et al.* [23] deploy regenerating codes [12] on HDFS to enable multiple-node failure recovery with minimum bandwidth. Silberstein *et al.* [34] propose lazy recovery for erasure-coded storage to reduce bandwidth due to frequent recovery executions. Li *et al.* [22] improve MapReduce performance on erasure-coded storage by scheduling degraded-read map tasks carefully to avoid bandwidth competition. Enterprises have also deployed erasure coding in production CFSes to reduce storage overhead, with reputable examples including Google [15], Azure [19], and Facebook [24], [32].

Some studies propose new erasure code constructions and evaluate their applicability in CFSes. Local repairable codes are a new family of erasure codes that reduce I/O during recovery while limiting the number of surviving nodes to be accessed. Due to the design simplicity, variants of local repairable codes have been proposed and evaluated based on an HDFS simulator [27], Azure [19], and Facebook [32]. RapidRAID codes [26] are new code constructions to address the performance of encoding replicated data. They are constructed such that the encoding operation can be decomposed and distributed across nodes. Our work studies the problem from the perspective of replica placement, and is applicable for general erasure codes. Piggybacked-RS codes [29], [30] embed parity information of one Reed-Solomon-coded stripe into that of the following stripe, and provably reduce recovery bandwidth while maintaining the storage efficiency of Reed-Solomon codes. Note that Piggybacked-RS codes have also been evaluated in Facebook's clusters. Facebook's f4 [24] protects failures at different levels including disks, nodes, and racks, by combining Reed-Solomon-coded stripes to create an additional XOR-coded stripe.

The above studies often assume asynchronous encoding (except the work [37], which performs encoding on the write path). Our work complements these studies by examining the performance and availability of the asynchronous encoding operation itself.

Replica placement in CFSes: Replica placement plays a critical role in both performance and reliability of CFSes. By constraining the placement of block replicas to smaller groups of nodes, the block loss probability can be reduced with multiple node failures [6], [9]. Scarlett [3] alleviates hotspots by carefully storing replicas based on workload patterns. Sinbad [8] identifies the variance of link capacities in a CFS and improves write performance by avoiding storing replicas on nodes with congested links. The above studies mainly focus on replication-based storage, while our work focuses on how replica placement affects the performance and reliability of asynchronous encoding.

7 Conclusions

Given the importance of deploying erasure coding in cluster file systems (CFSes) to reduce storage footprints, this paper studies the problem of encoding replicated data with erasure coding in CFSes. We present encoding-aware replication (EAR), which carefully constructs the replica layout so as to eliminate both cross-rack downloads and block relocation, while attempting to place the replicas as uniformly random as possible. We implement EAR on Facebook's HDFS and show its feasibility in real deployment. We conduct extensive evaluation using testbed experiments, discrete-event simulations, and load balancing analysis, and show that EAR achieves throughput gains of both write and encoding operations, while preserving the even replica distribution, when compared to random replication (RR).

Acknowledgments

This work was supported in part by grants AoE/E-02/08 and CRF-C7036-15 from the University Grants Committee of Hong Kong and the Research Grants Council of Hong Kong, the National Natural Science Foundation of China under Grant No. 61502191 and No. 61502190, and ZTE Corporation under project "data deduplication technology research".

References

[1] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar. ShuffleWatcher: Shuffle-aware Scheduling in Multi-tenant MapReduce Clusters. In *Proc. of USENIX ATC*, 2014.

[2] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proc. of ACM SIGCOMM*, Aug 2008.

[3] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters. In *Proc. of ACM EuroSys*, Apr 2011.

[4] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of ACM IMC*, 2010.

[5] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-Based Erasure-Resilient Coding Scheme. Technical Report TR-95-048, International Computer Science Institute, UC Berkeley, Aug 1995.

[6] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, et al. Apache Hadoop goes realtime at Facebook. In *Proc. of ACM SIGMOD*, Jun 2011.

[7] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of ACM SOSP*, Oct 2011.

[8] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging Endpoint Flexibility in Data-Intensive Clusters. In *Proc. of ACM SIGCOMM*, Aug 2013.

[9] A. Cidon, S. Rumble, R. Stutsman, S. Katti, J. Ousterhout, and M. Rosenblum. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. In *Proc. of USENIX ATC*, 2013.

[10] CSIM. http://www.mesquite.com/products/csim20.htm.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of USENIX OSDI*, Dec 2004.

[12] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Trans. on Info. Theory*, 56(9):4539–4551, Sep 2010.

[13] Facebook's Hadoop. http://goo.gl/fHDloI.

[14] B. Fan, W. Tantisiriroj, and G. Gibson. Diskreduce: Replication as a Prelude to Erasure Coding in Data-Intensive Scalable Computing. Technical Report CMU-PDL-11-112, Carnegie Mellon Univsersity, Parallel Data Laboratory, Oct 2011.

[15] D. Ford, F. Labelle, F. I. Popovici, M. Stokel, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, Oct 2010.

[16] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proc. of ACM SOSP*, Dec 2003.

[17] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. of ACM SIGCOMM*, Aug 2009.

[18] HDFS-RAID. http://wiki.apache.org/hadoop/HDFS-RAID, 2011.

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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2678505, IEEE Transactions on Parallel and Distributed Systems

[20] Iperf. https://iperf.fr/, 2011.

[21] R. Li, Y. Hu, and P. P. C. Lee. Enabling Efficient and Reliable Transition from Replication to Erasure Coding for Clustered File Systems. In *Proc. of IEEE/IFIP DSN*, 2015.

[22] R. Li, P. P. C. Lee, and Y. Hu. Degraded-First Scheduling for MapReduce in Erasure-Coded Storage Clusters. In *Proc. of IEEE/IFIP DSN*, 2014.

[23] R. Li, J. Lin, and P. P. C. Lee. Enabling Concurrent Failure Recovery for Regenerating-Coding-Based Storage Systems: From Theory to Practice. *IEEE Trans.on Computers*, 2015.

[24] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's Warm BLOB Storage System. In *Proc. of USENIX OSDI*, 2014.

[25] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proc. of ACM SOSP*, 2011.

[26] L. Pamies-Juarez, A. Datta, and F. Oggier. RapidRAID: Pipelined Erasure Codes for Fast Data Archival in Distributed Storage Systems. In *Proc. of IEEE INFOCOM*, 2013.

[27] D. Papailiopoulos, J. Luo, A. Dimakis, C. Huang, and J. Li. Simple Regenerating Codes: Network Coding for Cloud Storage. In *Proc. of IEEE INFOCOM*, Mar 2012.

[28] K. V. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth. In *Proc. of USENIX FAST*, 2015.

[29] 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 FacebookWarehouse Cluster. In *Proc. of USENIX HotStorage*, 2013.

[30] 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. of ACM SIGCOMM*, 2014.

[31] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[32] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. In *Proc. of VLDB Endowment*, pages 325–336, 2013.

[33] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of IEEE MSST*, May 2010.

[34] M. Silberstein, L. Ganesh, Y. Wang, L. Alvizi, and M. Dahlin. Lazy Means Smart: Reducing Repair Bandwidth Costs in Erasure-coded Distributed Storage. In *Proc. of ACM SYSTOR*, 2014.

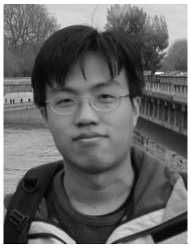
[35] SWIM Project. https://github.com/SWIMProjectUCB/SWIM/wiki.

[36] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, Mar 2002.

[37] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan. Does Erasure Coding Have a Role to Play in my Data Center? Technical Report MSR-TR-2010-52, Microsoft Research, May 2010.

Runhui Li received the B.Eng. degree in Computer Science and Technology from University of Science and Technology of China in 2011 and the Ph.D. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2015. His research interests are in distributed systems and data storage.

Yuchong Hu received the B.Eng. degree in Computer Science and Technology from Special Class for the Gifted Young (SCGY), the University of Science and Technology of China in 2005, and the Ph.D. degree in Computer Software and Theory from the University of Science and Technology of China in 2010. He is now an Associate Professor of the School of Computer Science and Technology at the Huazhong University of Science and Technology. His research interests are in cloud storage, heterogeneous storage, network coding, and erasure coding.



Patrick P. C. Lee received the B.Eng. degree (first-class honors) in Information Engineering from the Chinese University of Hong Kong in 2001, the M.Phil. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2003, and the Ph.D. degree in Computer Science from Columbia University in 2008. He is now an Associate Professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in various applied/systems topics including storage systems, distributed systems and networks, operating systems, dependability, and security.