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

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

Department of Computer Science and Engineering, The Chinese University of Hong Kong

{*rhli, pclee*}@*cse.cuhk.edu.hk, yuchonghu@gmail.com*

Abstract—**To balance performance and storage efficiency, modern clustered file systems (CFSes) often first store data with random replication (i.e., distributing replicas across randomly selected nodes), followed by encoding the replicated data with erasure coding. We argue that random replication, while being commonly used, does not take into account erasure coding and hence will raise both performance and availability issues to the subsequent encoding operation. We propose *encoding-aware replication*, which carefully places the replicas so as to (i) avoid cross-rack downloads of data blocks during encoding, (ii) preserve availability without data relocation after encoding, and (iii) maintain load balancing as in random replication. We implement encoding-aware replication on HDFS, and show via testbed experiments that it achieves significant encoding throughput gains over random replication. We also show via discrete-event simulations that encoding-aware replication remains effective under various parameter choices in a large-scale setting. We further show that encoding-aware replication evenly distributes replicas as in random replication.**

I. 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 by load-balancing read requests across multiple replicas. On the other hand, erasure coding provably achieves higher fault tolerance than replication, while using much less redundancy [31]. For example, traditional designs of CFSes deploy 3-way replication [5, 14, 28], which incurs $3\times$ storage overhead. Azure reportedly uses erasure coding to reduce the storage overhead to $1.33\times$, leading to over 50% of operational cost saving for storage [17].

Recent studies [12, 17, 27] 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* [12]: 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 and minimizes storage overhead for old data via erasure coding. It simplifies deployment and error handling, and hides performance degradation [12].

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 will consume a substantial amount of bandwidth across racks. Cross-rack bandwidth is considered to be a scarce resource in CFSes [6, 9], and is often over-subscribed by many nodes [1, 15]. Thus, intensive cross-rack data transfers will degrade the performance of normal foreground operations. Second, relocation of encoded blocks may be needed to ensure the availability requirement (e.g., rack-level fault tolerance) is fulfilled. Although such relocation is rare in production [21], it is still undesirable, since it not only introduces additional cross-rack traffic, 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) [7], whose idea is to store replicas across randomly chosen nodes. RR is simple to realize and has been used by HDFS [28], Azure [5], and the DRAM-based storage system RAMCloud [22]. However, it does not take into account the relations among the replicas 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 later be encoded. 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 by equivalently solving a maximum matching problem. By doing so, EAR avoids downloading data blocks from other racks for encoding, and avoids relocation of encoded blocks after encoding. Thus, EAR reduces cross-rack traffic due to the encoding operation. In addition, EAR tries to randomly distribute replicas as in RR to maintain load balancing.

In summary, we make the following contributions:

- We present EAR, a new replica placement algorithm that addresses both performance and availability issues

of the encoding operation.

- We implement EAR on Facebook's HDFS implementation [11], with only few modifications to the source code of HDFS.
- We conduct testbed experiments on a 13-machine cluster. We observe significant encoding throughput gains of EAR over RR in different settings, and the gains can reach over 100% in some cases. Also, EAR improves write throughput by reducing network traffic. Furthermore, based on synthetic MapReduce workloads, we find that the replica placement of EAR does not compromise performance before encoding.
- We conduct discrete-event simulations based on CSIM 20 [8], 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 II presents the problem setting and issues of RR. Section III describes the design of EAR. Section IV presents the implementation details of EAR on HDFS. Section V presents our evaluation results. Section VI reviews related work, and finally Section VII concludes the paper.

II. PROBLEM

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

A. System Model

Clustered file system (CFS) architecture: We consider a 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. Cross-rack bandwidth is a scarce resource [6, 9] and often over-subscribed [1, 15], so we assume that cross-rack data transfer is the performance bottleneck in a CFS architecture. We 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 [14], HDFS [28], and Azure [5].

We motivate our study by examining the open-source HDFS implementation by Facebook [11], which supports erasure-coded storage based on HDFS-RAID [16]. Nevertheless, our discussion can be generalized for other CFSes.

Replication: Traditional CFS designs use r-way replication by storing r replicas for each block in different nodes, where $r = 3$ is commonly used [5, 14, 28]. One common replica placement policy is collectively called *random replication* (RR) [7], which is used by HDFS [28], Azure [5],
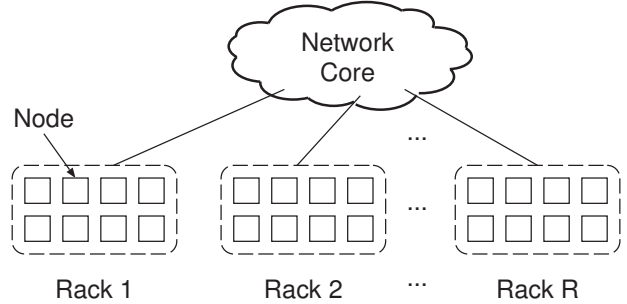

Figure 1. Example of a CFS architecture.

and RAMCloud [22]. 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 [28]: it uses 3-way 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.

Erasure coding: Erasure coding is a redundancy alternative that provably incurs less storage overhead than replication under the same fault tolerance [31]. We consider (n, k) erasure coding defined by two parameters n and k (where $k < n$). It transforms k original uncoded blocks (which we call *data blocks*) to create $n - k$ additional coded blocks (which we call *parity blocks*), such that any k out of the n data and parity blocks can reconstruct all k original data blocks. We call the collection of n data and parity blocks to be a *stripe*, and typical erasure coding schemes operate on each stripe independently. We assume *systematic* erasure coding, which keeps the k data blocks in a stripe. Examples of erasure coding schemes include Reed-Solomon codes [26] and Cauchy Reed-Solomon codes [3].

Asynchronous encoding: Erasure-coded data is usually generated *asynchronously* in the background (i.e., off the write path) [12, 17, 27], in which all blocks are first replicated when being written to a CFS, and the CFS later transforms the replicas into erasure-coded data. We call the transformation from replicas to erasure-coded data to be 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 other replicas.

Facebook's HDFS implementation [11] performs asynchronous encoding via a map-only MapReduce job, in which multiple map tasks run on different nodes simultaneously, and each map task performs encoding for a subset of stripes.

Thus, the encoding operation is parallelized at the stripe level. We provide more implementation details in Section IV.

B. 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 [21] (and we verify this feature in Facebook's HDFS implementation [11]). 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 [21], 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 with $(5, 4)$ erasure coding, 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 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 ($1 \leq i \leq R$) 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 $\frac{2 \times k}{R}$. If we pick a random node to perform encoding, the expected number of data blocks to be downloaded from different racks is $k - \frac{2 \times k}{R}$, which is almost k if R is large.

The same example also shows the availability issue. After we remove the remaining replicas (i.e., those crossed away

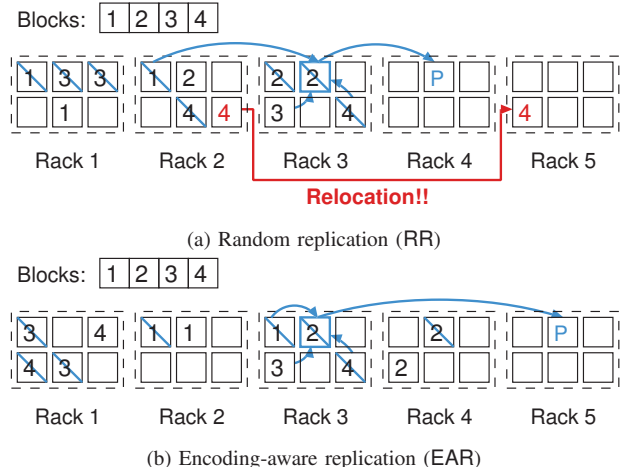(a) Random replication (RR)

(b) Encoding-aware replication (EAR)

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

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. We elaborate the design of EAR in Section III.

III. Design

In this section, we present the design of EAR. EAR imposes constraints on replica placement, so as to address both performance and availability issues of the encoding operation. 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:* Both node-level and rack-level fault tolerance requirements are fulfilled after the encoding operation, without the need of relocating any data or parity block.
- *Maintaining load balancing:* EAR tries to randomly place replicas as in RR for simplicity and load balancing [7], subject to the imposed constraints.

A. Eliminating Cross-Rack Downloads

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, each handling the encoding of a stripe, at a time. On the other hand, since stripes are encoded independently, our analysis focuses on a single stripe, and hence a single core rack.

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 using $(4,3)$ erasure coding. We also require to tolerate any $n-k = 1$ rack failure. Suppose that after we place the first replica of each data block in the core rack using the preliminary EAR, we place the remaining two replicas in a randomly chosen
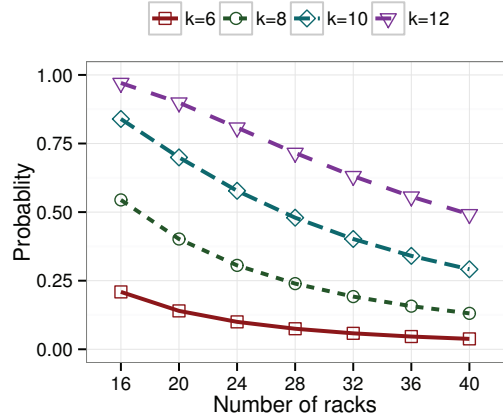

Figure 3. Probability that a stripe violates rack-level fault tolerance.

rack different from the core rack, as in HDFS [28]. 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.

As opposed to RR, the preliminary EAR has a smaller degree of freedom in placing replicas across racks. We argue 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 (n,k) erasure coding (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 a total of $(R-1)^k$ ways to place the replicas of the k data blocks. We also require to tolerate $n-k$ rack failures after the encoding operation as in Facebook [21].

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:

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

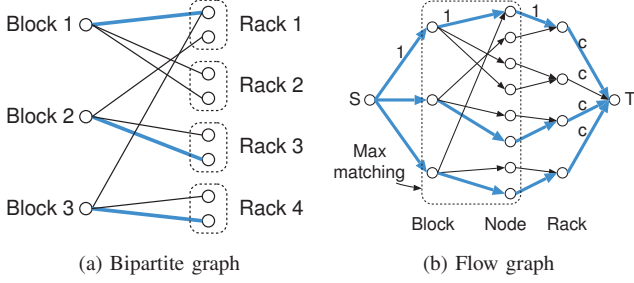Figure 3 plots f for different values of k and R using

(a) Bipartite graph   (b) Flow graph

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

Figure 5.   Illustration of EAR.

Equation (1). Note that $k = 10$ and $k = 12$ are chosen by Facebook [27] and Azure [17], 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 $R = 16$).

B. Preserving Availability

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 (n, k) erasure coding 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. Our (complete) 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 $(4, 3)$ erasure coding to tolerate a single node failure. We set $c = 1$, so as to tolerate $\lfloor \frac{4-3}{1} \rfloor = 1$ rack failure.

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
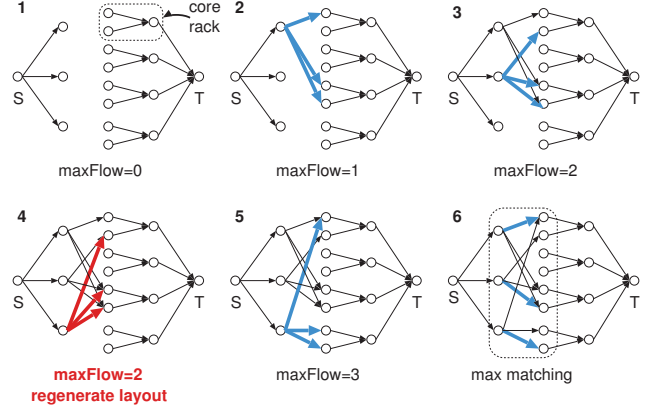
assigning parity blocks to the nodes of other racks that have fewer than c blocks of the stripe.

To determine if a maximum matching exists, we convert 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.

C. Algorithm

Combining the designs in Sections III-A and III-B, we propose an algorithm for EAR, which systematically places 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, when using 3-way replication, we follow HDFS [28] 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).

The following theorem quantifies that the expected number of iterations for generating a qualified replica layout in Step 5 is generally very small.

Theorem 1. *Consider a CFS with R racks, each containing a sufficiently large number of nodes. Under 3-way replication, the expected number of iterations (denoted by E_i) that EAR finds a qualified replica layout for the i-th data block, such that the maximum flow becomes i, is at most $\frac{R-1}{R-1-\lfloor (i-1)/c \rfloor}$, where $1 \le i \le k$.*

Proof (Sketch): Suppose that we have found a qualified replica layout for the $(i-1)$-th data block, which makes the maximum flow become $i-1$. Before finding a replica layout for the i-th data block (where $1 \le i \le k$), the number of racks (excluding the core rack) that have stored c blocks (call them *full racks*) is at most $\lfloor (i-1)/c \rfloor$. If (i) we place the second and third replicas in the remaining $R-1-\lfloor (i-1)/c \rfloor$ non-full racks and (ii) the nodes that will store the replicas of the i-th data block have not stored any replica of the previous $i-1$ data blocks, then the maximum flow will increase to i. Condition (i) holds with probability $1 - \frac{\lfloor (i-1)/c \rfloor}{R-1}$, and condition (ii) holds with probability almost one if each rack has a sufficiently large number of nodes. Thus, E_i is at most $[1 - \frac{\lfloor (i-1)/c \rfloor}{R-1}]^{-1}$. $\qquad\square$

Remarks: E_i increases with i and reaches the maximum at $i = k$, but it is generally very small. For example, suppose that a CFS has $R = 20$ racks and we set $c = 1$. Then for the k-th data block, E_i is at most 1.9 for $k = 10$ (used by Facebook [27]) and 2.375 for $k = 12$ (used by Azure [17]).

D. Discussion

To tolerate any $n - k$ rack failures [21], we must place n data and parity blocks of the same stripe in n different racks. 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 [24].

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 III-C) and find the maximum matching. Note that we require $R' \ge \frac{n}{c}$ (see Section III-B).
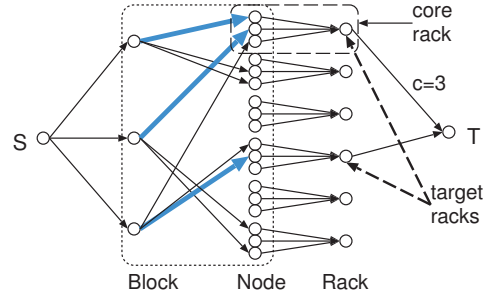

Figure 6. Revised flow graph with target racks.

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

IV. Implementation

We implement EAR as an extension to Facebook's HDFS [11]. In this section, we describe the implementation details.

A. Overview of Erasure Coding in HDFS

The original HDFS architecture [28] comprises a single *NameNode* and multiple *DataNodes*, such that the NameNode stores the metadata (e.g., locations) for HDFS blocks, while the DataNodes store HDFS blocks. Facebook's HDFS implements erasure coding based on a middleware layer called HDFS-RAID [16], which manages the erasure-coded blocks on HDFS. On top of HDFS, HDFS-RAID adds a new node called the *RaidNode*, which coordinates the encoding operation. The RaidNode also periodically checks for any lost or corrupted blocks, and activates recovery for those blocks. Currently, 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 [9], 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 [26] implemented by HDFS-RAID as our erasure coding scheme. Finally, the replicas of the data blocks are deleted.

B. Integration

Figure 7 depicts how we modify HDFS to integrate EAR. Our modifications are minor, and they can be summarized in three aspects.

Our first modification is that 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 Na-meNode to keep track of each stripe and the list of data block identifiers that belong to the stripe.

Our second modification is to 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 [9].

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, our third modification is to 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.

V. EVALUATIONS

In this section, we present evaluation results for EAR. Our evaluations comprise three parts: (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.
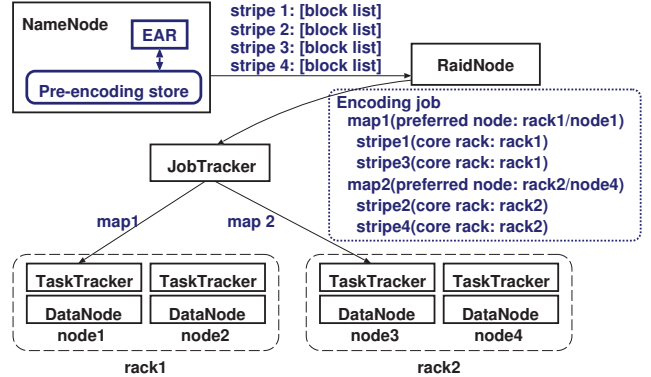

Figure 7. Integration of EAR into Facebook's HDFS.

A. Testbed Experiments

We conduct testbed experiments on a 13-machine HDFS cluster. 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. One limitation is that our testbed is of a small scale. Nevertheless, our testbed experiments provide insights into how EAR performs in real deployment.

To mimic a CFS architecture as in Figure 1, 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*) deploy one DataNode and one TaskTracker. We have validated that the network transfer over the 1Gb/s Ethernet switch is the bottleneck in our testbed.

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 II-A). For each encoding job, we configure the RaidNode to launch 12 map tasks. All our results are averaged over five runs.

Experiment A.1 (Raw encoding performance): We first study the raw encoding performance without write requests. We consider (n, k) erasure coding 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 8(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

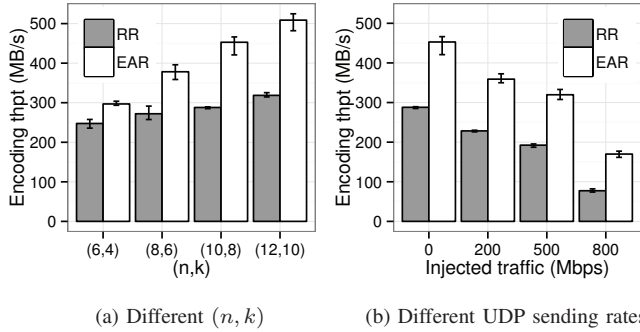(a) Different (n,k) (b) Different UDP sending rates

Figure 8. Experiment A.1: Raw encoding performance of encoding 96 stripes. Each bar shows the minimum and maximum values over five runs, represented by the endpoints of the error bars.
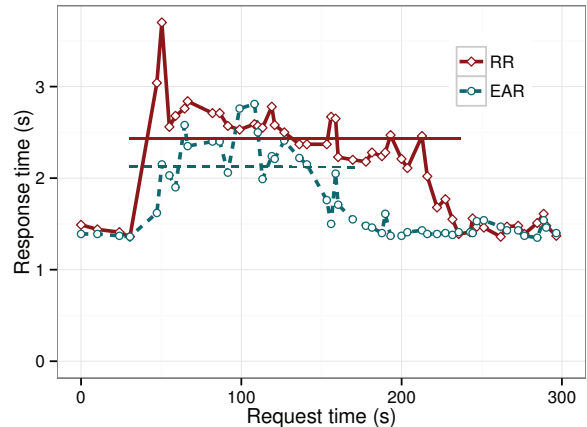

Figure 9. Experiment A.2: Impact of encoding on write performance under RR and EAR. The horizontal lines represent the average write response time during encoding and the duration of the whole encoding operation. For brevity, each data point represents the averaged write response time of three consecutive write requests.

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 [18]. 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 8(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. We expect that if a CFS is severely over-subscribed [1, 15], the benefits of EAR will be prominent.

Experiment A.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 of write requests, each writing 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 times 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 9 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 at 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 A.3 (Impact of EAR on MapReduce): We study how EAR influences the performance of MapReduce jobs before encoding starts. We use SWIM [30], 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 a combination of reading and processing input data from HDFS, shuffling data, and outputting final results to HDFS.

Figure 10 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.

B. 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 [8]. Figure 11 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
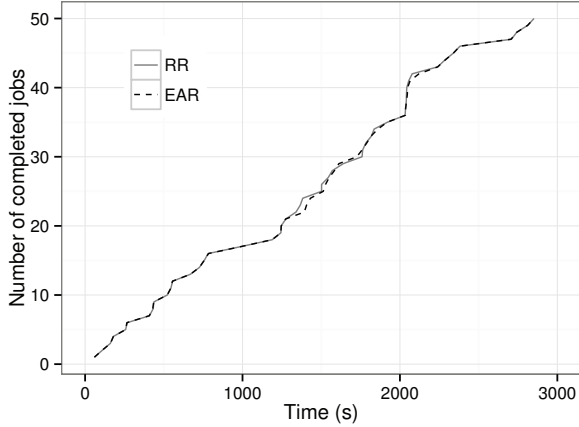
Figure 10. Experiment A.3: Impact of EAR on MapReduce performance. We observe similar performance trends between EAR and RR.
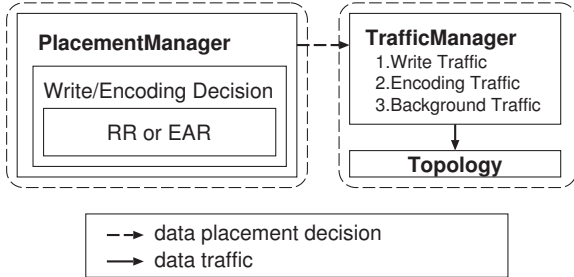

Figure 11. Simulator overview.

Figure 12. Experiment B.1: Simulator validation.

Table I
EXPERIMENT B.1: VALIDATION OF WRITE RESPONSE TIMES.

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

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.

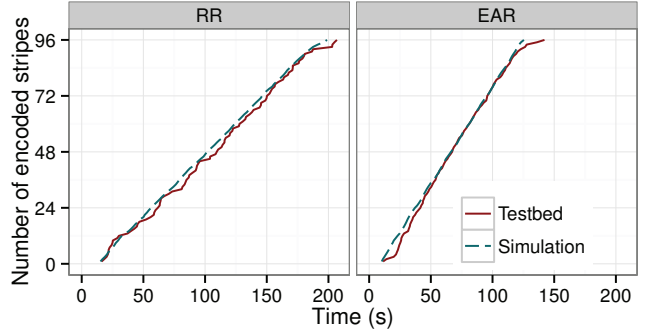**Experiment B.1 (Simulator validation):** We first validate that our simulator can accurately simulate the performance of both RR and EAR. We simulate our testbed with the same topology (12 racks with one DataNode each) and the same link bandwidth (1Gb/s Ethernet). Using the simulator, we replay the write and encode streams using the same setting as in Experiment A.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 12 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 I 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 less than 4.3%. Thus, our simulator also precisely captures write performance.

Experiment B.2 (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 $(n, k) = (14, 10)$ erasure coding that can tolerate 4-node or 4-rack failures, as in Facebook [21]. Although RR may require block relocation after encoding to preserve availability, we do not consider this operation, so
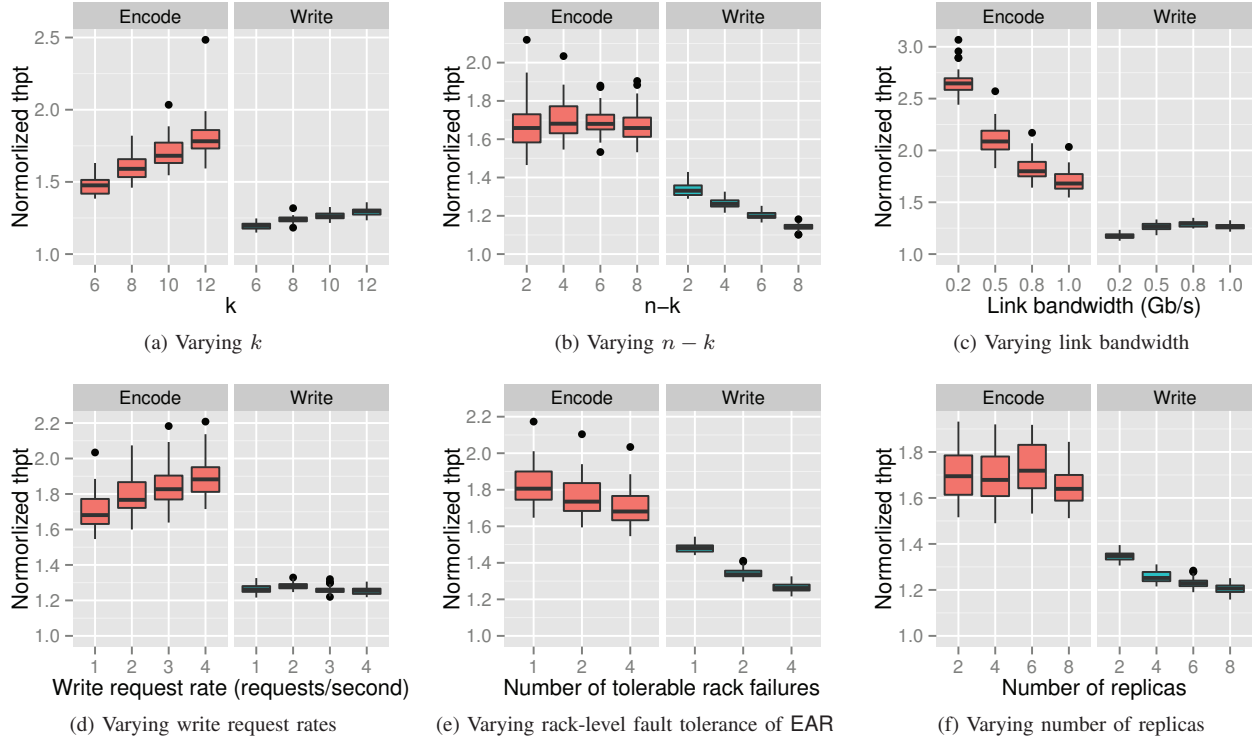
Figure 13. Experiment B.2: Impact of parameter choices on encoding and write performance under EAR and RR. Each plot denotes the normalized throughput of EAR over RR.

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 13(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 13(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, 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 13(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 13(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 III-B). Here, we keep RR to still provide tolerance against $n - k$ rack failures, while we vary the rack fault tolerance of EAR. Figure 13(e) shows the throughput results versus the number of rack failures tolerated in EAR. By tolerating fewer rack failures, EAR can
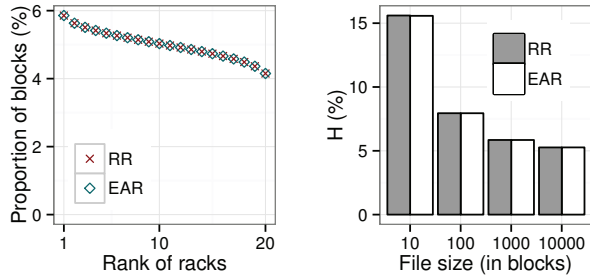
Figure 14. Experiment C.1: Storage load balancing.

Figure 15. Experiment C.2: Read load balancing.

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 13(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 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.

C. 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 [7]. 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, and the replicas are distributed across two racks as in HDFS [28]. For EAR, we choose $(14, 10)$ erasure coding. We obtain the averaged results over 1,000 runs.

Experiment C.1 (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 then count the number of replicas stored in each rack. Figure 14 shows the proportions of replicas of RR and EAR in each rack (sorted in descending order of proportions). We observe that 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 C.2 (Read load balancing): We also examine the distribution of read requests across racks. Suppose that the data blocks in File F 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 15 shows H versus the file size, which we vary from 10 to 10,000 blocks. Both RR and EAR have almost identical H.

VI. RELATED WORK

Erasure coding in CFSes: Researchers have extensively studied the applicability of deploying erasure coding in CFSes. Fan *et al.* [12] augments HDFS with asynchronous encoding to significantly reduce storage overhead. Zhang *et al.* [32] propose to apply erasure coding on the write path of HDFS, and study the performance impact on various MapReduce workloads. Li *et al.* [20] deploy regenerating codes [10] on HDFS to enable multiple-node failure recovery with minimum bandwidth. Silberstein *et al.* [29] propose lazy recovery for erasure-coded storage to reduce bandwidth due to frequent recovery executions. Li *et al.* [19] 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 [13], Azure [17], and Facebook [21, 27].

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 [23], Azure [17], and Facebook [27]. Piggybacked-RS codes [24, 25] 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 [21] 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 (except the work [32]) often assume asynchronous encoding, and focus on improving the applicability of erasure coding after the replicated data has been encoded. 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 [4, 7]. Scarlett [2] alleviates hotspots by carefully storing replicas based on workload patterns. Sinbad [6] 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.

VII. 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 argue that random replication (RR) brings both performance and availability issues to the subsequent encoding operation. We thus present encoding-aware replication (EAR) to take into account erasure coding. EAR imposes constraints to 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 evaluations 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 RR. In future work, we plan to study the scenarios with heterogeneous workloads and hardware resources. The source code of our EAR implementation is available at **http://ansrlab.cse.cuhk.edu.hk/software/ear**.

Acknowledgments

References

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

[2] 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.

[3] 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.

[4] 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.

[5] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McElvie, 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.

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

[7] 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.

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

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

[10] 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.

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

[12] 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.

[13] 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.

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

[15] 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.

[16] HDFS-RAID. http://wiki.apache.org/hadoop/HDFS-RAID.

[17] 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.

[18] Iperf. https://iperf.fr/.

[19] 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.

[20] 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*, 2014.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

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

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

[32] 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.