OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems

Xiaolu Li†, Runhui Li†, Patrick P. C. Lee†, and Yuchong Hu‡
†*The Chinese University of Hong Kong* ‡*Huazhong University of Science and Technology*

Abstract

Erasure coding becomes a practical redundancy technique for distributed storage systems to achieve fault tolerance with low storage overhead. Given its popularity, research studies have proposed theoretically proven erasure codes or efficient repair algorithms to make erasure coding more viable. However, integrating new erasure coding solutions into existing distributed storage systems is a challenging task and requires non-trivial re-engineering of the underlying storage workflows. We present OpenEC, a unified and configurable framework for readily deploying a variety of erasure coding solutions into existing distributed storage systems. OpenEC decouples erasure coding management from the storage workflows of distributed storage systems, and provides erasure coding designers with configurable controls of erasure coding operations through a directed-acyclic-graph-based programming abstraction. We prototype OpenEC on two versions of HDFS with limited code modifications. Experiments on a local cluster and Amazon EC2 show that OpenEC preserves both the operational performance and the properties of erasure coding solutions; OpenEC can also automatically optimize erasure coding operations to improve repair performance.

1 Introduction

Erasure coding provides a low-cost redundancy mechanism for fault-tolerant storage, and is now widely deployed in today's distributed storage systems (DSSs). Examples include enterprise-level DSSs [15,21,30] and many open-source DSSs [1,3,7,31,54,55]. Unlike replication that simply creates identical data copies for redundancy protection, erasure coding introduces much less storage overhead through the coding operations of data copies, while preserving the same degree of fault tolerance [53]. Modern DSSs mostly realize erasure coding based on the classical Reed-Solomon (RS) codes [43], yet RS codes have high performance penalty, especially in repairing lost data when failures happen. Thus, research studies have proposed new erasure coding solutions with improved performance, such as erasure codes with theoretical guarantees and efficient repair algorithms that are applicable to general erasure-coding-based storage (§6).

However, deploying new erasure coding solutions in DSSs is a daunting task. Existing studies often integrate new erasure coding solutions into specific DSSs by re-engineering the DSS workflows (e.g., the read/write paths). The tight coupling between erasure coding management and the DSS workflows makes new erasure coding solutions hard to be generalized for other DSSs and further enhanced. Some DSSs with built-in erasure coding features (e.g., HDFS with erasure coding [1, 5], Ceph [54], and Swift [7]) provide certain configuration capabilities, such as interfaces for implementing various erasure codes and controlling erasure-coded data placement, yet the interfaces are rather limited and it is non-trivial to extend the DSSs with more advanced erasure codes and repair algorithms (§2.2). How to fully realize the power of erasure coding in DSSs remains a challenging issue.

We present OpenEC, a unified and configurable framework for erasure coding management in DSSs, with the primary goal of bridging the gap between designing new erasure coding solutions and enabling the feasible deployment of such new solutions in DSSs. Inspired by software-defined storage [16,48,51], which aims for configurable storage management without being constrained by the underlying storage architecture, we apply this concept into erasure coding management. Our main idea is to decouple erasure coding management from the DSS workflows. Specifically, OpenEC runs as a middleware system between upper-layer applications and the underlying DSS, and is responsible for performing all erasure coding operations on behalf of the DSS. Such a design relaxes the stringent dependence on the erasure coding support of DSSs. More importantly, OpenEC takes the full responsibility of erasure coding management, and hence provides flexibility for erasure coding designers to (i) incorporate a variety of erasure coding solutions, (ii) configure the workflows of erasure coding operations, and (iii) decide the placement of both erasure-coded data and erasure coding operations across storage nodes. Our contributions are summarized as follows:

- We propose a new programming model for erasure coding implementation and deployment. Our model builds on an abstraction called an ECDAG, a directed acyclic graph that defines the workflows of erasure coding operations. We show how we feasibly realize a general erasure coding solution through the ECDAG abstraction.
- We design OpenEC, which translates an ECDAG into erasure coding operations atop a DSS. OpenEC supports encoding operations on or off the write path as well as various state-of-the-art repair operations. In particular, it can automatically optimize an ECDAG for hierarchical topologies to improve repair performance.

- We implement a prototype of OpenEC on HDFS-RAID [5] and Hadoop 3.0 HDFS (HDFS-3) [1]. Its integrations into HDFS-RAID and HDFS-3 only require limited code changes (with no more than 450 LoC).
- We evaluate OpenEC on a local cluster and Amazon EC2. OpenEC incurs negligible performance overhead in DSS operations, supports various state-of-the-art erasure codes and repair algorithms, and increases the repair throughput by at least 82% through automatically customizing an ECDAG for a hierarchical topology.

The source code of our OpenEC prototype is available at: **http://adslab.cse.cuhk.edu.hk/software/openec**.

2 Background and Motivation

2.1 Erasure Coding Basics

Consider a DSS that comprises multiple *storage nodes* and organizes data in units of *blocks*. We construct erasure coding as an (n,k) code with two configurable parameters n and k, where $k < n$. For every k fixed-size original blocks (called *data blocks*), an (n,k) code encodes them into $n-k$ redundant blocks of the same size (called *parity blocks*), such that any k out of the n erasure-coded blocks (including both data and parity blocks) can decode the k data blocks; that is, any $n-k$ block failures can be tolerated. We call the collection of n erasure-coded blocks a *coding group*. A DSS encodes different sets of k data blocks independently, and distributes the n erasure-coded blocks of each coding group across n storage nodes to protect against any $n-k$ storage node failures. In this paper, our discussion focuses on the coding operations (i.e., encoding or decoding) of a single coding group.

For performance reasons, a DSS implements coding operations in small-size units called *packets*, while the read/write units are in blocks; for example, our experiments set the default packet and block sizes as 128 KiB and 64 MiB, respectively). It divides a block into multiple packets, and encodes the packets at the same block offsets in a coding group together. Thus, instead of first reading the whole blocks to start coding operations, a DSS can perform packet-level coding operations, while reading the whole blocks, in a pipelined manner. To simplify our discussion, we use blocks as the units of coding operations, and only differentiate packets and blocks in our implementation (§4.5).

Given the prevalence of failures, repairs are frequent operations in DSSs [40]. We consider two types of repairs: (i) *degraded reads*, which decode the unavailable data blocks that are being requested, and (ii) *full-node recovery*, which decodes all lost blocks of a failed storage node. Since repairs trigger substantial traffic [40], achieving high repair performance is important in erasure coding deployment. RS codes [43] are the most popular erasure codes that are widely used in production [5, 7, 15, 31, 54, 55], but they incur high repair costs. Thus, many repair-friendly erasure codes have been proposed. Since single-failure repairs (i.e., repairing a single lost block of a coding group in degraded reads or a single failed node in full-node recovery) are the most common repair scenarios [21, 40], existing repair-friendly erasure codes aim to minimize the repair bandwidth or I/O in single-failure repairs. Examples are regenerating codes [14], including minimum-storage regenerating (MSR) and minimum-bandwidth regenerating (MBR) codes, as well as locally repairable codes (LRCs) [21, 23, 44, 49].

Our work focuses on practical erasure codes. In particular, we target *linear* codes, which include RS codes, MSR and MBR codes, as well as LRCs. Linear codes perform linear coding operations based on the Galois field arithmetic [17]. Mathematically, for an (n,k) code, let $d_0, \cdots, d_{k-1}$ be the k data blocks, and $p_0, \cdots, p_{n-k-1}$ be the $n-k$ parity blocks. Each parity block p_j $(0 \le j \le n-k-1)$ can be expressed as $p_j = \sum_{i=0}^{k-1} \gamma_{ji} d_i$, where γ_{ji} is some coding coefficient for computing p_j. Note that the linear operations are *additive associative* (i.e., independent of how additions are grouped).

Also, our work addresses *sub-packetization*, which is used in various designs of MSR and MBR codes [14, 18, 32, 39, 42, 45, 50, 52]. Sub-packetization divides each block into smaller-size sub-blocks, so that repairs can be done by retrieving sub-blocks rather than whole blocks.

Most DSSs assume that all erasure-coded blocks are *immutable* and do not support in-place updates. Thus, we focus on four basic operations: writes, normal reads, degraded reads, and full-node recovery (§4.2), while we address in-place updates in future work.

2.2 Limitations of Erasure Coding Management

Modern DSSs now support erasure coding, yet existing erasure coding management in such DSSs remains stringent and still faces practical limitations. To motivate our study, we review six state-of-the-art DSSs that currently realize erasure-coded storage: HDFS-RAID [5], HDFS-3 [1], QFS [31], Tahoe-LAFS [55], Ceph [54], and Swift [7]. HDFS-RAID is the erasure coding extension of HDFS [46] in the earlier version of Hadoop. Here, we focus on Facebook's HDFS-RAID implementation [3], which builds on Hadoop version 0.20. HDFS-3 builds on the newer Hadoop version 3.0, which includes erasure coding by design. QFS resembles HDFS and includes erasure coding by design. All HDFS-RAID, HDFS-3, and QFS organize data in fixed-size blocks. In contrast, Tahoe-LAFS, Ceph, and Swift organize data in variable-size objects and partition each object into equal-size data blocks for erasure coding.

(L1) Limited support for adding advanced erasure codes: Existing DSSs provide encoding/decoding interfaces for implementing new erasure codes. However, most DSSs do not provide interfaces for adding erasure codes with sub-packetization (e.g., MSR and MBR codes [14, 18, 32, 39, 42, 45, 50, 52]) and handling erasure-coded blocks at the granularity of sub-blocks, while only recently Ceph includes the sub-packetization feature in its master codebase [52]. Also,

recent erasure codes [19,38] address the hierarchical nature of DSSs to reduce cross-rack [19] (or cross-cluster [38]) repair traffic, yet realizing such hierarchy-aware erasure codes needs modifications to the DSS workflows.

(L2) Limited configurability for workflows of coding operations: Enabling configurable workflows of coding operations allows better resource usage within a DSS. Take repairs (degraded reads or full-node recovery) as an example. DSSs execute repairs at different entities upon the detection of failures. For a degraded read, it is executed at the client (in HDFS-RAID, HDFS-3, QFS, and Tahoe-LAFS), the proxy (in Swift), or a storage node (in Ceph); for full-node recovery, it is executed at either storage nodes (in HDFS-RAID, HDFS-3, QFS, Ceph, and Swift) or the client (in Tahoe-LAFS). Both degraded reads and full-node recovery operate in a *fetch-and-compute* manner, in which the entity that executes the repair will retrieve available blocks from other non-failed storage nodes and reconstruct the lost blocks. On the other hand, besides the fetch-and-compute approach, we cannot configure a DSS to adopt different repair workflows or distribute the repair loads across storage nodes. For example, recent repair algorithms [25,29] decompose a single-block repair operation into partial sub-block repair operations that are parallelized across storage nodes for better bandwidth usage, but existing DSSs do not support this feature by design.

(L3) Limited configurability for placement of coding operations: All DSSs we consider ensure that the n erasure-coded blocks of each coding group are stored in n distinct storage nodes, and most of them additionally allow configurable block placement. For example, both HDFS-RAID and HDFS-3 provide a base class for configuring block placement policies; QFS provides an *in-rack* placement option to store multiple blocks in a rack; Ceph uses *placement groups*, while Swift uses *object rings*, to control how erasure-coded blocks are placed in different storage nodes.

However, existing DSSs focus on how erasure-coded blocks are placed after encoding, but do not specify where to perform the coding operations. For example, in encoding operations, we may want to co-locate the computations of parity blocks at one storage node (rather than distribute the computations across different storage nodes) to limit the I/Os of retrieving data blocks. Also, the repair algorithms in [25,29] require some storage nodes that store available data blocks to first compute partially decoded blocks and send the results to other storage nodes for further decoding. In this case, we need to place the partial decoding operations at specific storage nodes. Such fine-grained placement of coding operations is currently not supported in existing DSSs.

2.3 Lessons Learned and Goals

The root cause of the limitations in §2.2 is that the current erasure coding management is tightly coupled with the DSS workflows. Realizing erasure coding in DSSs needs to address how coding operations are performed (i.e., the control flow) and how erasure-coded blocks are stored and accessed (i.e., the data flow). The current practice is that erasure coding designers only define an erasure code and its coding operations (e.g., the coding coefficients used in coding operations), while DSS developers require dedicated engineering efforts to integrate the coding operations into the read/write paths of DSSs without compromising the correctness of upper-layer applications. Such tight coupling makes the extensions of erasure coding features inflexible.

OpenEC decouples erasure coding management from the underlying DSS by providing a unified and configurable framework for erasure coding management, such that erasure coding designers can leverage OpenEC to realize new erasure coding solutions and configure the workflows of coding operations, without worrying how they are integrated into the DSS workflows. Specifically, OpenEC addresses the limitations in §2.2 with the following goals: (i) extensibility of new erasure codes; (ii) configurable workflows of coding operations; and (iii) configurable placement of both erasure-coded blocks and coding operations. To achieve these goals, OpenEC builds on a programming model for erasure coding management, as elaborated in the following sections.

3 Programming Model

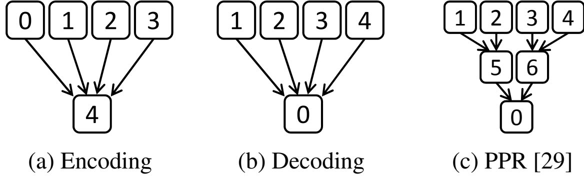
We propose a programming model that allows erasure coding designers to not only define an erasure code structure and its coding operations, but also configure how coding operations are performed in a DSS. We present a new erasure coding abstraction called an ECDAG (§3.1), followed by three primitives for constructing an ECDAG (§3.2). We then propose a programming interface for realizing an erasure code based on the ECDAG abstraction (§3.3).

3.1 ECDAG Overview

At a high level, an ECDAG is a directed acyclic graph that describes the workflows of coding operations of a coding group of an erasure code. Each vertex represents a block in the coding group, and the connections among vertices describe how vertices are related by linear combinations. To address the limitations in §2.2, we design ECDAGs to work for general linear codes (L1 addressed). Also, we can construct different ECDAGs to configure *how* and *where* coding operations are performed (L2 and L3 addressed, respectively).

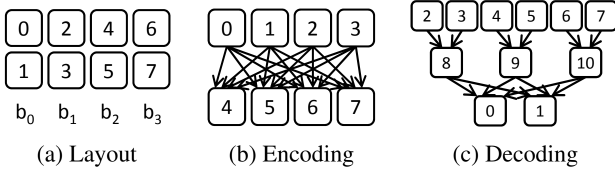
Consider a coding group of an (n,k) code with n erasure-coded blocks; to simplify our discussion, we do not consider sub-packetization first. We index the blocks from 0 to $n-1$, and let b_i denote the block with index i. Without loss of generality, we refer to $b_0,\cdots,b_{k-1}$ as k data blocks, and $b_k,\cdots,b_{n-1}$ as $n-k$ parity blocks. In some cases (see below), the coding operations may generate some intermediately computed blocks that will not be finally stored (as opposed to blocks $b_0,b_1,\cdots,b_{n-1}$). We call such blocks *virtual blocks*, and denote a virtual block by $b_{i'}$ for some $i' \geq n$.

In an ECDAG, let v_i ($i \geq 0$) be a vertex that maps to block



(a) Encoding (b) Decoding (c) PPR [29]

Figure 1: Example of an ECDAG for a $(5,4)$ code.



(a) Layout (b) Encoding (c) Decoding

Figure 2: Example for an ECDAG for a (4,2) code with $w = 2$.

b_i; we call a vertex $v_{i'}$ ($i' \geq n$) that maps to a virtual block $b_{i'}$ a *virtual vertex*. Let $e_{i,j}$ ($i, j \geq 0$) be a directed edge from v_i to v_j indicating that b_i is an input to the linear combination for computing b_j. Each edge is associated with a coding coefficient for the linear combination. If there exists an edge $e_{i,j}$, we say that v_j is the *parent* of v_i, while v_i is a *child* of v_j. A vertex can have any number of parents and children.

Both encoding and decoding operations are each associated with an ECDAG. The ECDAG for encoding is constructed at the beginning of the encoding operation to describe how data blocks are linearly combined to form each parity block. In contrast, the ECDAG for decoding is constructed on demand depending on what blocks are currently available.

For example, consider a $(5,4)$ code (i.e., (4+1)-RAID-5). Figure 1(a) shows the ECDAG for the encoding operation, which states that the parity block b_4 is a linear combination of the four data blocks b_0, b_1, b_2, and b_3. Suppose now that block b_0 is lost. Figure 1(b) shows the ECDAG for the decoding operation for b_0, which can be computed from other available blocks b_1, b_2, b_3, and b_4.

We can parallelize partial decoding operations as in PPR [29] by constructing another ECDAG for decoding b_0 (see Figure 1(c)), in which we first compute in parallel the partially decoded blocks b_5 and b_6 (both of which are virtual blocks) from b_1 and b_2 and from b_3 and b_4, respectively, followed by computing b_0 from b_5 and b_6. This shows that we can flexibly configure coding operations by constructing different ECDAGs. Note that PPR needs to compute b_5 and b_6 at the storage nodes where data blocks (e.g., b_2 and b_4, respectively) are stored (see [29] for details). We address this issue in §3.2.

We can also construct an ECDAG for erasure codes with sub-packetization. Let w be the number of sub-blocks per block ($w = 1$ means no sub-packetization). We index the sub-blocks of block b_0 from 0 to $w-1$, those of b_1 from w to $2w-1$, and so on. Each vertex v_i ($i \geq 0$) now corresponds to the sub-block with index i, while any vertex $v_{i'}$ for $i' \geq nw$ is a virtual vertex. For example, consider the $(4,2)$ MISER code [45] (an MSR code based on interference alignment), where $w = 2$. Figure 2(a) shows how the sub-blocks are indexed. Figure 2(b) shows the ECDAG for the encoding operation, in which the sub-blocks of parity blocks b_2 and b_3 are computed from the sub-blocks of data blocks b_0 and b_1. Suppose that block b_0 is lost. Figure 2(c) shows the ECDAG for decoding b_0 based on MISER codes [45], in which we first compute an encoded sub-block from each of other available blocks b_1, b_2, and b_3 (represented by the virtual vertices v_8, v_9, and v_{10}, respectively), followed by using the encoded sub-blocks to decode the lost sub-blocks of b_0.

```
void Join(int pidx, vector<int> cidxs, vector<int> coefs);
int BindX(vector<int> idxs);
void BindY(int pidx, int cidx);
```

Listing 1: Primitives for ECDAG construction.

3.2 ECDAG Primitives

An ECDAG can be constructed from three primitives: `Join`, `BindX`, and `BindY`. `Join` is used for constructing an ECDAG, while `BindX` and `BindY` control the placement of coding operations. Listing 1 shows their definitions in C++ format.

Join: It specifies how a parent vertex (with index `pidx`) is formed by the linear combinations of a list of child vertices (with indices in `cidxs`) and the corresponding coding coefficients (in `coefs`). For example, we deploy the $(6,4)$ RS code and encode four data blocks b_0, b_1, b_2, and b_3 into two new parity blocks b_4 and b_5. We can construct an ECDAG with `Join` as follows (see Figure 3(a)):

```
ECDAG* ecdag = new ECDAG();
ecdag->Join(4, {0,1,2,3}, {1,1,1,1});
ecdag->Join(5, {0,1,2,3}, {1,2,4,8});
```

BindX: It co-locates the coding operations of multiple vertices (with indices in `idxs`) that reside at the same level of an ECDAG (i.e., in the x-direction), so as to reduce I/O in coding operations. For example, in Figure 3(a), suppose that the data blocks being encoded are stored in different storage nodes. Without `BindX`, we need to compute b_4 and b_5 separately and retrieve each data block twice. Instead, we can call `BindX` on vertices v_4 and v_5 to create a new virtual vertex v_6 as follows (see Figure 3(b)):

```
int vidx = ecdag->BindX({4,5});
```

This indicates that blocks b_4 and b_5 are first computed together at the same storage node before being distributed to different storage nodes. Now we only need to retrieve each data block once. Note that the index of v_6 (i.e., 6) is generated randomly and returned as `vidx` by `BindX`.

BindY: It co-locates the coding operations of a parent vertex (with index `pidx`) and its child vertex (with index `cidx`) at different levels (i.e., in the y-direction). Consider the same example in Figure 3(b) after we call `BindX`. We can call `BindY` on vertices v_0 and v_6 as follows (see Figure 3(c)):

```
ecdag->BindY(vidx, 0);
```

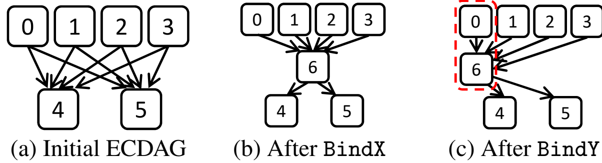


Figure 3: Construction of ECDAGs for the $(6,4)$ RS code.

```
class ECBase {
  int n, k, w;  // coding parameters
  vector<int> ecoefs; // encoding coefficients
public:
  ECDAG* Encode();
  ECDAG* Decode(vector<int> from, vector<int> to);
  vector<vector<int>> Place();
};
```

Listing 2: Erasure coding programming interface.

Thus, we compute parity blocks b_4 and b_5 at the same storage node that stores b_0, thereby saving the I/Os of retrieving b_0.

Note that BindY enables us to implement the repair algorithms (e.g., PPR [29] and repair pipelining [25]) that need to compute partially decoded blocks at the storage nodes that store the data blocks. For example, referring to Figure 1(c) for PPR, we can call BindY on v_2 and v_5, and on v_4 and v_6, to co-locate the computations of the partially decoded blocks b_4 and b_5 at the storage nodes that store b_2 and b_4, respectively.

Remarks: We provide flexibility for erasure coding designers to construct any ECDAG using the above three primitives, yet this also puts burdens on erasure coding designers to configure coding operations. Nevertheless, OpenEC can also automatically call BindX and BindY on some specific subgraph structures of an ECDAG (§4.4).

3.3 Erasure Coding Interfaces

We provide a programming interface for realizing an erasure code. Unlike the traditional approach that takes data blocks as input and generates parity blocks, we program an erasure code through the construction of ECDAGs. OpenEC then parses the ECDAGs to perform the actual coding operations and store the erasure-coded blocks.

Listing 2 shows the erasure coding programming interface as a base class ECBase. To realize an erasure code, we (as erasure coding designers) inherit ECBase and first define all necessary member variables (e.g., n, k, w, and encoding coefficients) in the constructor method as in traditional erasure code programming. Note that we can store encoding coefficients in a *generator matrix* [35] and compute decoding coefficients later based on the available blocks. We then implement three functions, namely Encode, Decode, and Place.

Encode: It constructs an ECDAG that describes the encoding operation. For example, to encode the $(6,4)$ RS code based on Figure 3(c), we can construct an ECDAG as in Listing 3.

Decode: It constructs an ECDAG that takes the available blocks (with indices in from) as input and decodes any lost blocks (with indices in to). For example, we can implement Decode for a single lost block as in Listing 4, in which the decoding coefficients are computed based on the available blocks in from. In general, Decode constructs an ECDAG for one of the two scenarios: (i) decoding one lost block, in which we can choose an efficient single-failure repair approach (e.g., see Figure 2(c) for the $(4,2)$ MISER code); or (ii) decoding multiple lost blocks, in which we can choose any k available blocks (e.g., the first k blocks in from) to compute the decoding coefficients and decode all lost blocks.

```
ECDAG* Encode() {
  ECDAG* ecdag = new ECDAG();
  ecdag->Join(4, {0,1,2,3}, {1,1,1,1});
  ecdag->Join(5, {0,1,2,3}, {1,2,4,8});
  int vidx = ecdag->BindX({4,5});
  ecdag->BindY(vidx, 0);
  return ecdag;
}
```

Listing 3: Encode function.

```
ECDAG* Decode(vector<int> from, vector<int> to) {
  ECDAG* ecdag = new ECDAG();
  vector<int> dcoefs;  // decoding coefficients
  // compute dcoefs based on the available blocks
  ecdag->Join(to[0], from, dcoefs);
  return ecdag;
}
```

Listing 4: Decode function.

```
vector<vector<int>> Place() {
  vector<vector<int>> groups;
  for (int i=0; i<n/2; ++i) groups[0].push_back(i);
  for (int i=n/2; i<n; ++i) groups[1].push_back(i);
  return groups;
}
```

Listing 5: Place function.

Place: It configures how erasure-coded blocks are placed with hierarchy awareness. In addition to storing erasure-coded blocks in different storage nodes, we can configure how the blocks are grouped (e.g., in the same rack in rack-based DSSs). This supports fine-grained block placement configurations as in existing DSSs (§2.2), and allows the realization of hierarchy-aware erasure codes [19, 38]. For example, we can divide n erasure-coded blocks into two groups via Place as in Listing 5. Note that BindX and BindY in ECDAG construction (§3.2) address the placement of coding operations, while Place addresses the placement of erasure-coded blocks.

4 OpenEC Design

We design OpenEC to provide erasure coding management for a DSS. We show its architecture (§4.1) and supported basic operations (§4.2). We then describe how it parses ECDAGs to realize coding operations (§4.3). We further show how it automatically optimizes coding operations (§4.4). We conclude this section with the implementation details (§4.5).

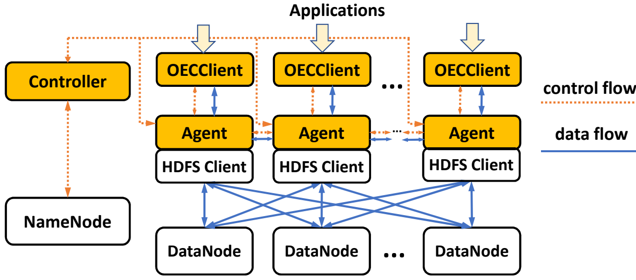


Figure 4: OpenEC architecture based on HDFS.

4.1 Architectural Overview

OpenEC runs as a middleware system atop a DSS. As a proof of concept, we design OpenEC atop two implementations of HDFS [46]: HDFS-RAID [5] and HDFS-3 [1][1]. HDFS (including both HDFS-RAID and HDFS-3) comprises a *NameNode* that coordinates the storage in units of *blocks* across multiple *DataNodes* (storage nodes). Figure 4 shows how OpenEC is integrated into HDFS. OpenEC comprises a centralized *controller*, which coordinates multiple *agents*. An application interacts with OpenEC via an *OECClient*.

Controller: The controller parses ECDAGs and instructs all agents how to perform coding operations and store erasure-coded blocks. It keeps erasure coding metadata and all ECDAGs in local disk for persistence. There are three types of metadata: (i) the information of blocks associated with each file; (ii) the information of blocks associated with each coding group; and (iii) the block locations. The controller interacts with the NameNode in two aspects. First, it accesses or updates the block locations of the NameNode to configure the placement of blocks. Second, it receives the reports of lost blocks from the NameNode and coordinates the repair operations among the agents.

We assume that the controller is reliable (i.e., no single-point-of-failure). Our measurements show that the controller can serve a request of parsing an ECDAG for coding operations in less than 0.3 ms in our local cluster (§5), and hence it incurs limited overhead to basic operations.

Agent: Each agent performs coding operations as instructed by the controller. It accesses the erasure-coded blocks in HDFS through the HDFS client interface. Note that agents can communicate among themselves to perform coding operations and exchange erasure-coded blocks. We currently deploy each agent at a DataNode, so that the agent can access the local storage of the DataNode without network transfers.

OECClient: Each OECClient is associated with an agent, and serves as an interface between an upper-layer application and the agent. It connects to the agent via Redis-based communication (§4.5). An application now accesses HDFS through an OECClient instead of an HDFS client.

[1] We also implement OpenEC atop QFS [31]. See [27] for details.

4.2 Basic Operations

OpenEC supports four basic operations: (i) writes; (ii) normal reads; (iii) degraded reads; and (iv) full-node recovery.

Writes: Note that HDFS-3 supports *online encoding* (i.e., clients perform encoding on the write path), while HDFS-RAID supports *offline encoding* (i.e., clients first write the data blocks in uncoded form, and the data blocks are later encoded in the background). OpenEC is currently designed to support both online and offline encoding. An OECClient specifies which encoding mode to use in a write request. For online encoding, OpenEC encodes data on a per-file basis. When an OECClient writes a file, its agent encodes every k data blocks into $n-k$ parity blocks and writes the n erasure-coded blocks to n DataNodes through the HDFS client. For offline encoding, an OECClient first writes file data via its agent to HDFS. When OpenEC receives an encoding request, the controller parses the specified ECDAG (§4.3) and instructs all agents to perform encoding, such that every k blocks are encoded into n erasure-coded blocks as a coding group.

Normal reads: An OECClient issues normal reads (under no failures) via its agent, which connects to the DataNodes that store the uncoded data blocks and retrieves the data blocks from the DataNodes.

Degraded reads: An OECClient issues degraded reads (under failures) via its agent, which connects to non-failed DataNodes and retrieves the available blocks for decoding the lost blocks based on the ECDAG specification.

Full-node recovery: The controller coordinates the full-node recovery operation. When it receives a report of lost blocks from the NameNode, it informs the agents to repair the lost blocks based on the ECDAG specification.

4.3 Parsing an ECDAG

OpenEC parses ECDAGs to perform coding operations in writes (online or offline encoding), degraded reads, and full-node recovery. Given an ECDAG, OpenEC decomposes a coding operation into multiple *tasks*, each of which is executed by an agent. Each task operates in blocks (or sub-blocks in sub-packetization). There are four types of tasks:

- **Load:** It loads a block into memory from the agent's input stream, which could be either the OECClient if the block is from upper-layer applications, or the HDFS client if the block is from HDFS.
- **Fetch:** It retrieves blocks from other agents.
- **Compute:** It computes a block based on the linear combination of blocks and coding coefficients.
- **Persist:** It either writes a block to HDFS via the HDFS client, or returns the block to an OECClient.

Parsing procedure: OpenEC performs *topological sorting* of an ECDAG (based on depth-first search) to identify the vertex sequence of coding operations. It then assigns tasks to each vertex based on the ECDAG structure. Depending on the

Vertices	Nodes	Tasks
v_0	C	**Load** b_0
v_1	C	**Load** b_1
v_2	C	**Load** b_2
v_3	C	**Load** b_3
v_6	C	**Compute** b_4 from $\{b_0, b_1, b_2, b_3\}$ with coding coefficients $\{1,1,1,1\}$; **Compute** b_5 from $\{b_0, b_1, b_2, b_3\}$ with coding coefficients $\{1,2,4,8\}$
v_4	C	–
v_5	C	–
–	C	**Persist** b_0; **Persist** b_1; **Persist** b_2; **Persist** b_3; **Persist** b_4; **Persist** b_5

(a) Online encoding

Vertices	Nodes	Tasks
v_0	N_0	**Load** b_0
v_1	N_1	**Load** b_1
v_2	N_2	**Load** b_2
v_3	N_3	**Load** b_3
v_6	N_0	**Fetch** b_1 from N_1; **Fetch** b_2 from N_2; **Fetch** b_3 from N_3; **Compute** b_4 from $\{b_0, b_1, b_2, b_3\}$ with coding coefficients $\{1,1,1,1\}$; **Compute** b_5 from $\{b_0, b_1, b_2, b_3\}$ with coding coefficients $\{1,2,4,8\}$
v_4	N_4	**Fetch** b_4 from N_0; **Persist** b_4
v_5	N_5	**Fetch** b_5 from N_0; **Persist** b_5

(b) Offline encoding

Table 1: Vertex sequence of coding operations, including the nodes that are responsible for processing the vertices as well as the tasks that are performed.

types of basic operations, OpenEC may perform coding operations on the client side (for online encoding and degraded reads) or distribute the coding operations across storage nodes (for offline encoding and full-node recovery).

OpenEC associates tasks with different types of vertices. At a high level, the Load task is associated with a vertex without any child; the Fetch task is associated with a parent vertex that has a child vertex; the Compute task is associated with a vertex with more than one child for the linear combination; the Persist task is associated with a vertex without any parent, while it is also associated with a vertex without any child in the case of online encoding (see the example below).

Example: We show the parsing procedure via an example. Suppose that we encode four data blocks (i.e., b_0, b_1, b_2, and b_3) to generate two parity blocks (i.e., b_4 and b_5) using the $(6,4)$ RS code, based on the ECDAG in Figure 3(c) and the `Encode` function in Listing 3. Table 1 shows the vertex sequence of tasks for both online and offline encoding.

For online encoding (see Table 1(a)), the client-side agent (denoted by C) performs all coding operations. It finally persists all data blocks and parity blocks into HDFS.

For offline encoding (see Table 1(b)), OpenEC distributes the coding operations across storage nodes. To elaborate, suppose that b_i is stored in storage node N_i, for $0 \leq i \leq 5$. First, since v_0, v_1, v_2, and v_3 have no child, OpenEC creates tasks for the agents in storage nodes N_0, N_1, N_2, and N_3 to load the blocks b_0, b_1, b_2, and b_3, respectively, from HDFS (via HDFS clients) into memory. Second, since vertex v_6 is created from `BindX` on vertices v_4 and v_5, OpenEC computes both b_4 and b_5 from the blocks in the child vertices (i.e., b_0, b_1, b_2, and b_3). Also, since `BindY` is called on v_6 and v_0, OpenEC assigns the tasks of v_6 to the agent in N_0. Finally, v_4 and v_5 retrieve blocks b_4 and b_5 from v_6, respectively. Since v_4 and v_5 have no parent and are the last vertices in the topological order, they persist the blocks to HDFS.

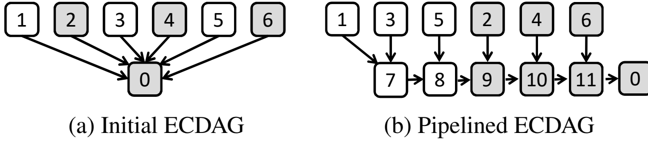
Note that OpenEC can parallelize the coding operations on the vertices that have no dependencies on others. For example, OpenEC can simultaneously execute the tasks for v_0, v_1, v_2, and v_3, and similarly the tasks for v_4 and v_5.

4.4 Automated Optimizations

In addition to letting erasure coding designers construct ECDAGs, OpenEC can automatically customize ECDAGs for performance optimizations to save manual configuration efforts. We address this in two aspects.

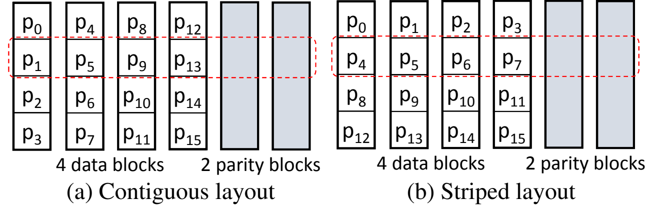
Automated BindX and BindY: OpenEC can automatically call `BindX` and `BindY` for some specific subgraph structures of an ECDAG. For `BindX`, OpenEC examines all parent vertices that have more than one child vertex in an ECDAG. If multiple parent vertices have the same set of child vertices, OpenEC calls `BindX` on those parent vertices (e.g., v_4 and v_5 in Figure 3(b)). For `BindY`, for any parent vertex (with one or more child vertices), OpenEC calls `BindY` on the parent vertex and any one of the child vertices (e.g., the parent vertex v_6 and the child vertex v_0 in Figure 3(c)).

Hierarchy awareness: OpenEC can further enhance the repair performance based on the physical DSS topology. One scenario is that a DSS hierarchically organizes storage nodes in racks [19] (or clusters [38]), such that the cross-rack bandwidth is much more constrained than the inner-rack bandwidth. OpenEC can transform an ECDAG into a *pipelined ECDAG*, so as to mitigate the cross-rack traffic. Our idea is based on repair pipelining [25], which pipelines partial coding operations across multiple storage nodes. We additionally perform all partial coding operations within a rack before sending the partial coding results to another rack. To illustrate, suppose that we deploy an (n,k) RS code with $k = 6$. We want to repair a lost block b_0 from six other available blocks b_1, b_2, b_3, b_4, b_5, and b_6, such that blocks b_1, b_3, and b_5 are in one rack, while blocks b_2, b_4, and b_6 are in another rack. We also want to store the reconstructed block b_0 at the same rack as b_2, b_4, and b_6. The conventional repair approach is to retrieve all six available blocks and construct an ECDAG as in Figure 5(a). Then we need to transfer three blocks (i.e., b_1, b_3, and b_5) across racks. Instead, OpenEC can automatically construct another ECDAG as in Figure 5(b), in which it first



(a) Initial ECDAG (b) Pipelined ECDAG

Figure 5: Example of constructing a pipelined ECDAG; vertices of the same color mean that their blocks are in the same rack.



(a) Contiguous layout (b) Striped layout

Figure 6: Block layouts for the $(6,4)$ RS code. Suppose that each block stores four packets. We partition file data into 16 packets, ordered as p_0, p_1, $\cdots$, and p_{15}. We place the packets across $k=4$ blocks. Packets at the same offset (e.g., in dashed boxes) are encoded together. In sub-packetization, each packet is further divided into sub-packets for encoding.

computes the partially decoded block b_8 (corresponding to vertex v_8) based on b_1, b_3, and b_5 in the same rack, followed by combining b_8 with b_0, b_2, and b_4 in another rack to reconstruct block b_0. In this case, we only need to transfer one block (i.e., b_8) across racks.

4.5 Implementation

We implement an OpenEC prototype in C++ with around 7K LoC. We use Intel's Intelligent Storage Acceleration Library (ISA-L) [6] to implement erasure coding functionalities. Here, we highlight several implementation details of OpenEC.

From blocks to packets: OpenEC performs coding operations in units of packets to improve performance, while the read/write operations are still in units of blocks (§2.1). By default, the packet size is 128 KiB. For encoding (both online and offline), OpenEC writes n erasure-coded packets to n DataNodes; in the case of sub-packetization, each packet is divided into sub-packets. If a DataNode receives an amount of packet data equal to the HDFS block size (64 MiB by default), it *seals* the block and stores additional packets in a different block. The n sealed erasure-coded blocks then form a coding group. Note that while OpenEC is sending packets to DataNodes, it can start encoding for the next group of packets. Thus, both the sending and encoding operations can be done in parallel. Similarly, OpenEC performs decoding (for degraded reads and full-node recovery) at the packet level.

As OpenEC performs packet-level coding operations, the block layouts differ in online and offline encoding. For online encoding, OpenEC adopts a *striped* layout as in HDFS-3 [4], as it stripes file data across blocks at the granularities of packets. For offline encoding, OpenEC adopts a *contiguous* layout, as the file data is first stored in a block before encoding. Figure 6 depicts both block layouts.

Internal communication: OpenEC uses Redis [8] for internal communications among the controller, agents, and OECClient. Each agent maintains a local in-memory key-value Redis store. The controller sends the task instructions of coding operations to an agent via the Redis client, and the task instructions are buffered at the agent for subsequent processing. Agent-to-agent communications are *pull-based* via the Fetch tasks (§4.3), such that the sender agent buffers the blocks to be sent in its local Redis store, and the receiver fetches the buffer via the Redis client. Each OECClient also communicates with its associated agent via Redis.

Integration: We integrate OpenEC into HDFS-RAID and HDFS-3 as follows. We realize a new block placement policy called `BlockPlacementPolicyOEC`, which redirects block placement requests to the controller to manage erasure-coded block placement. We also modify the `FSNamesystem` class in HDFS-RAID and the `BlockManager` class in HDFS-3 to redirect any lost block information to the controller so that OpenEC manages repair operations. Note that our integrations into HDFS-RAID and HDFS-3 only require limited modifications to their codebases, with around 300 LoC and 450 LoC, respectively[2].

5 Evaluation

We conduct testbed experiments on OpenEC. We summarize our major findings on OpenEC: (i) it preserves the performance of HDFS-RAID and HDFS-3 in erasure coding deployment (§5.2); (ii) it supports various state-of-the-art erasure coding solutions and preserves their properties, especially in network-bound environments (§5.3); (iii) it can automatically optimize the repair performance for a hierarchical topology (§5.4); and (iv) it achieves scalable performance in real cloud environments (§5.5).

5.1 Setup

Testbeds: We evaluate OpenEC on both a local cluster (§5.2-§5.4) and Amazon EC2 (§5.5). Our local cluster testbed comprises 16 machines, each of which has a quad-core 3.4 GHz Intel Core i5-3570, 16 GiB RAM, and a Seagate ST1000DM003 7200 RPM 1 TiB SATA hard disk. All machines are interconnected via a 10 Gb/s Ethernet switch. On the other hand, our Amazon EC2 testbed comprises 30 instances of type `m5.xlarge`, connected via a 10 Gb/s network, in the US East (North Virginia) region. Each instance has four vCPUs with Intel AVX-512 instruction sets and 16 GiB RAM. Both testbeds support optimized coding operations based on ISA-L.

Default setup: We set the HDFS block size as 64 MiB and the packet size for erasure coding as 128 KiB. We set HDFS-3

[2]We compare the amounts of code changes in OpenEC with those in our previously built prototypes CORE [26] and DoubleR [19], both of which modify HDFS-RAID to realize new erasure codes. Excluding the implementation of erasure codes (e.g., coding operations), CORE and DoubleR make around 2,300 LoC and 4,100 LoC of changes to the HDFS-RAID codebase for the integration of erasure codes, respectively.

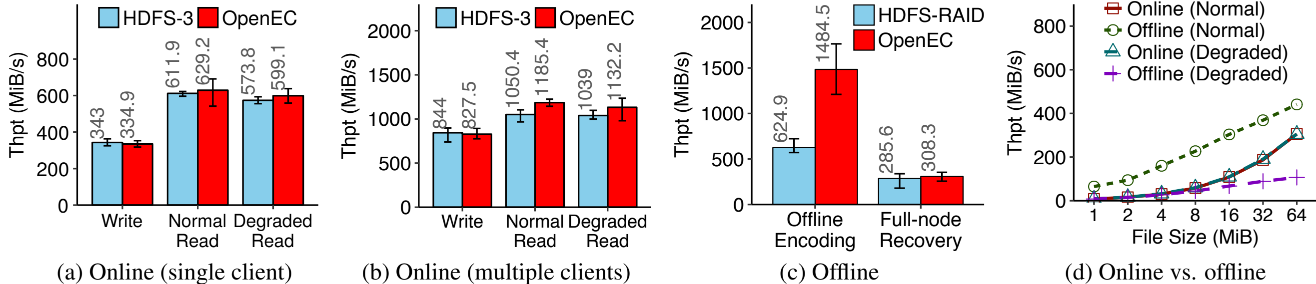


Figure 7: Performance of basic operations in online and offline encoding.

as the default DSS for OpenEC, except when we compare OpenEC with HDFS-RAID. Regarding the automated optimization features (§4.4), our experiments enable automated `BindX` and `BindY`, except when we evaluate the original performance of erasure codes without OpenEC optimization in §5.3 and when we evaluate `BindX` and `BindY` in §5.4. We also disable hierarchy-aware repairs until we evaluate this feature in §5.4. We assign a dedicated machine to serve both the OpenEC controller and the HDFS NameNode, while each remaining machine serves an OECClient, an OpenEC agent, an HDFS client, and an HDFS DataNode. We plot the average results over 10 runs, including the error bars showing the maximum and minimum of the 10 runs.

5.2 Performance of Basic Operations

We compare OpenEC with HDFS-RAID and HDFS-3 in terms of basic operations using our local cluster. As OpenEC adds another software layer between upper-layer applications and the underlying DSS, it may incur extra overhead. We show that such overhead (if any) is limited; in some cases, OpenEC even significantly improves performance. We also compare OpenEC with native coding performance and evaluate its performance for different block and packet sizes.

Single-client performance in online encoding: We first compare the single-client performance between HDFS-3 and OpenEC, both of which are configured with online encoding to generate erasure-coded data. Here, we use the $(9,6)$ RS code (as in QFS [31]). We first write a file of size 384 MiB (i.e., six times the block size), and issue a normal read to the file without failures. We also issue a degraded read to the file with one data block deleted. Figure 7(a) shows the throughput results of writes, normal reads, and degraded reads. Both OpenEC and HDFS-3 have similar performance: OpenEC's throughput is slightly less than HDFS-3's by 2.36% in writes, and is slightly higher than HDFS-3's by 2.83% and 4.41% in normal reads and degraded reads, respectively.

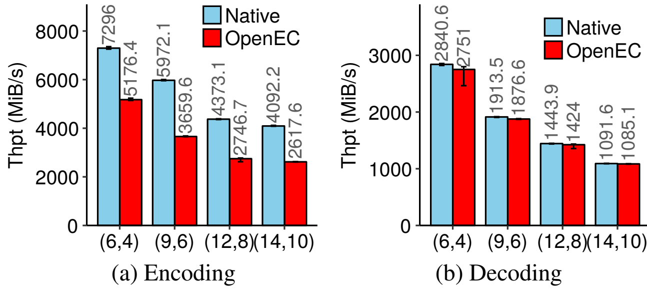
Multi-client performance in online encoding: We compare the multi-client performance between HDFS-3 and OpenEC. We run a total of five clients, each of which writes a file of size 384 MiB under the $(9,6)$ RS code. Figure 7(b) shows the aggregate throughput of all five clients in writes, normal reads, and degraded reads. OpenEC has lower aggregate throughput than HDFS-3 in writes by 1.95%, but higher aggregate throughput in normal reads and degraded reads by 12.9% and 8.97%, respectively. Nevertheless, considering the error bars in the figure, we do not see significant performance differences between OpenEC and HDFS-3.

Offline encoding: We compare the performance between HDFS-RAID and OpenEC in offline encoding. We now deploy OpenEC on HDFS-RAID for fair comparisons. We write 180 blocks, and use offline encoding to generate erasure-coded blocks using the $(9,6)$ RS code (i.e., a total of 30 coding groups). We then delete the blocks of one storage node and trigger full-node recovery. Here, we measure the offline encoding throughput (i.e., the amount of input data being encoded per unit time) and the full-node recovery throughput (i.e., the amount of lost data being recovered per unit time). Note that HDFS-RAID performs offline encoding and full-node recovery via MapReduce. To exclude the MapReduce startup overhead in our evaluation, we start an empty MapReduce job to measure its latency, and subtract this latency (which is around 20 s) in our evaluation of HDFS-RAID. Note that OpenEC does not use MapReduce in offline encoding and full-node recovery.

Figure 7(c) shows the results. Interestingly, OpenEC increases the offline encoding throughput of HDFS-RAID by 137%. We study the HDFS-RAID source code and find that the performance difference is mainly due to the extra step of HDFS-RAID in reading and re-writing all parity blocks into a single HDFS file after parity regeneration. For full-node recovery, OpenEC has slightly higher throughput than HDFS-RAID by 7.9%, yet the two systems have limited differences considering the error bars.

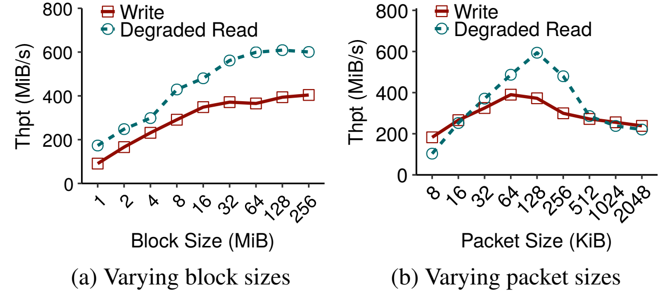
Online vs. offline encoding: We further compare online and offline encoding in OpenEC versus the file size, and study the performance difference between the striped layout (in online encoding) and the contiguous layout (in offline encoding). We deploy OpenEC atop HDFS-3, and show that it allows both online and offline encoding atop HDFS-3 (which currently supports online encoding only).

We consider the single-client performance, in which a client uses the $(12,8)$ RS code and writes a file of size ranging from 1 MiB to 64 MiB (assuming that the file size is divisible by eight). For online encoding, OpenEC stripes the file



(a) Encoding (b) Decoding

Figure 8: Comparisons with native coding operations.



(a) Varying block sizes (b) Varying packet sizes

Figure 9: Impact of block and packet sizes.

in packets across eight blocks and seals the blocks after the file write is completed (note that each block is less than the default block size 64 MiB); for offline encoding, OpenEC stores the file in a block and later encodes it with seven other blocks (§4.2). We compare their performance in a normal read (without failures) and a degraded read (with one data block deleted) to the file; in offline encoding, we delete the data block that stores the file in our degraded read evaluation.

Figure 7(d) shows the results. The throughput increases with the file size, since the data transfer performance becomes more dominant as the blocks become larger. We also see the performance differences in online and offline encoding. In online encoding, both normal reads and degraded reads show similar performance, in which the client issues reads to eight blocks in parallel. In offline encoding, its normal read throughput is much higher than that in online encoding (by 44-718%), as any slowdown in one of the parallel reads to online-encoded data can degrade the overall performance. However, the degraded read throughput in offline encoding is much less than that in online encoding especially for larger file sizes, as it needs to retrieve eight blocks (i.e., seven additional blocks over the original file) to recover the file. To validate our results, we conduct similar experiments using the original erasure coding implementations in HDFS-3 and HDFS-RAID (which realize online and offline encoding, respectively) and they show similar performance differences as in OpenEC (we omit the results here in the interest of space).

Comparisons with native coding operations: We compare the computational performance of the ECDAG-based coding operations with that of the native coding operations using ISA-L in HDFS-3. Figure 8(a) shows the encoding throughput for k 64-MiB blocks under (n,k) RS codes. ECDAG-based encoding has 29-38% lower throughput than native encoding, mainly because there is additional overhead for creating multiple compute tasks for computing the $n-k$ parity blocks. Figure 8(b) shows the decoding throughput for decoding one block, in which ECDAG-based decoding has only slightly less throughput (by 0.6-3.2%) than native decoding, as there is only one compute task for decoding a single block. Nevertheless, compared to the overall read/write operations (Figure 7), the computations of ECDAG-based coding are much faster and incur limited overhead.

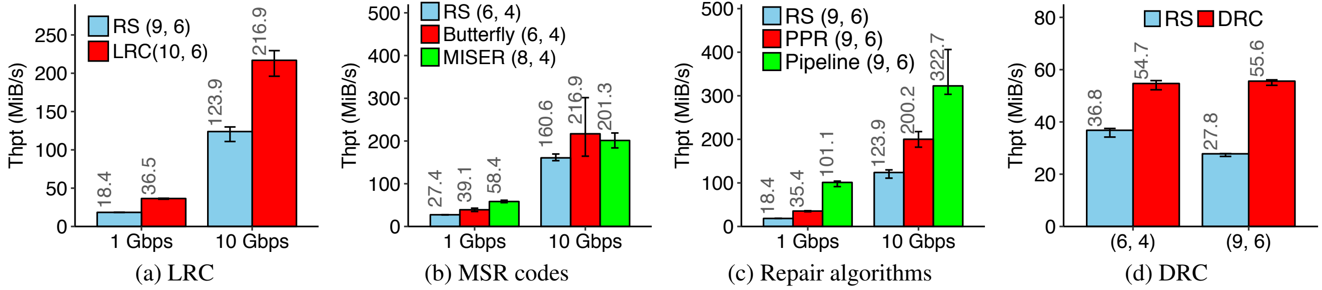
Impact of block and packet sizes: We study how the performance of OpenEC varies with block and packet sizes. We focus on the single-client throughput of online encoding and degraded reads under the $(9,6)$ RS codes as in §5.2. Figure 9(a) shows the throughput versus the block size, where the packet size is fixed as 128 KiB. The throughput of both operations increases with the block size as the disk and network bandwidths are better utilized, and stabilizes when the block size is at least 64 MiB. Figure 9(b) shows the throughput versus the packet size, where the block size is fixed as 64 MiB. The performance degrades if the packet size is too small since there are many function calls for retrieving individual packets, or if the packet size is too large since there is less parallelism. To achieve high performance, our default setup chooses the block size as 64 MiB and the packet size as 128 KiB.

5.3 Support of Erasure Coding Designs

We realize several state-of-the-art repair-friendly erasure coding solutions based on the ECDAG abstraction. Recall from §2.1 that existing repair-friendly codes are designed to minimize the repair bandwidth or I/O in single-failure repairs. Thus, we focus on evaluating their performance of repairing one lost block in a coding group under OpenEC. We configure two bandwidth settings in our local cluster: 1 Gb/s and 10 Gb/s. For the 1 Gb/s case, network transfer becomes the bottleneck (compared to coding computations and disk I/O), and we expect that the empirical performance conforms to the theoretical gains.

We use the conventional repair approach of RS codes as our baseline, in which it retrieves k blocks from k non-failed DataNodes to decode the lost block in a fetch-and-compute manner (§2.2). We compare the conventional repair approach with the following solutions:

- **LRC (Figure 10(a)):** We compare RS codes with Azure's LRC [21]. For RS codes, we set $(n,k) = (9,6)$; for LRC, we set $(n,k) = (10,6)$, in which there are two local parity blocks, each of which is encoded from a local group of three data blocks, and two global parity blocks that are encoded from all six data blocks.
- **MSR codes (Figure 10(b)):** We compare RS codes with MSR codes [14], which leverage sub-packetization to minimize the repair bandwidth. We focus on two variants of



(a) LRC (b) MSR codes (c) Repair algorithms (d) DRC

Figure 10: Support of erasure coding designs.

	LRC $(10,6)$ vs. RS $(9,6)$	Butterfly $(6,4)$ vs. RS $(6,4)$	MISER $(8,4)$ vs. RS $(6,4)$	PPR $(9,6)$ vs. RS $(9,6)$	Pipeline $(9,6)$ vs. RS $(9,6)$	DRC $(6,4)$ vs. RS $(6,4)$	DRC $(9,6)$ vs. RS $(9,6)$
Gain	2×	1.6×	2.28×	2×	6×	1.5×	2×

Table 2: Theoretical gains of state-of-the-art erasure codes or repair algorithms over the conventional repair of RS codes.

MSR codes: MISER codes [45] (which require $n \geq 2k$) and Butterfly codes [32] (which require $n = k+2$). We consider the $(6,4)$ RS code, the $(6,4)$ Butterfly code, and the $(8,4)$ MISER code.

- **Repair algorithms (Figure 10(c)):** We study how the repair algorithms, namely PPR [29] and repair pipelining [25], improve the repair performance of RS codes by parallelizing partial repair operations. We compare them with the conventional repair under the $(9,6)$ RS code.
- **Double Regenerating Codes (DRC) (Figure 10(d)):** We compare RS codes with DRC [19] in a hierarchical network setting. We divide our local cluster into three logical racks. We use the Linux `tc` command to limit the bandwidth between any two storage nodes at different logical racks as 1 Gb/s [44], while the bandwidth between any two storage nodes within the same logical rack remains 10 Gb/s. We compare RS codes and DRC under $(n,k) = (6,4)$ and $(n,k) = (9,6)$. In both cases, we distribute the erasure-coded blocks of each coding group evenly across different nodes in three racks (with $n/3$ erasure-coded blocks each).

Figure 10 shows the results; for our comparisons, Table 2 also shows the theoretical throughput gains of the erasure coding solutions over the conventional repair approach for RS codes. For the 1 Gb/s network, we observe that the empirical throughput gains of the erasure coding solutions are consistent (with only slight degradations) with the theoretical throughput gains. For the 10 Gb/s network, the empirical gains decrease since the coding computation and disk I/O overheads become more significant. For example, MISER codes have less throughput than Butterfly codes in the 10 Gb/s network; the throughput gain of MISER codes drops to 1.25×, while that of Butterfly codes drops to 1.35× (Figure 10(b)). The reason is that both MSR codes retrieve data from $n-1$ non-failed storage nodes for repairs, and MISER codes connect to more storage nodes than Butterfly codes (seven versus five) and incur higher disk I/O overhead. Overall, OpenEC preserves the properties of the erasure coding solutions.

5.4 Improvements with Automated Optimizations

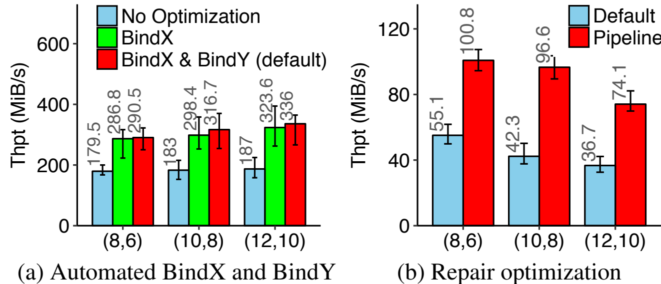
We now evaluate how OpenEC achieves performance gains via automated optimizations (§4.4) for a hierarchical topology. We again configure a three-rack logical topology in our local cluster as in our DRC experiments in §5.3.

We first compare the offline encoding performance for three configurations: (i) automated optimization is disabled, (ii) only automated `BindX` is enabled, and (iii) both automated `BindX` and `BindY` are enabled (our default setting). We consider the $(8,6)$, $(10,8)$, and $(12,10)$ RS codes. We measure the throughput of offline encoding by writing 30 coding groups of blocks into HDFS-3 via OpenEC, which evenly distributes the blocks across three racks. Figure 11(a) shows that enabling only `BindX` increases the throughput by 37-42%, while enabling both `BindX` and `BindY` increases the throughput by 38-44%.

We also evaluate how OpenEC automatically improves the repair performance via the construction of a pipelined ECDAG. We delete all blocks of one storage node and trigger full-node recovery on the same node. Figure 11(b) shows that the repair optimization increases the repair throughput of OpenEC by 82-128%.

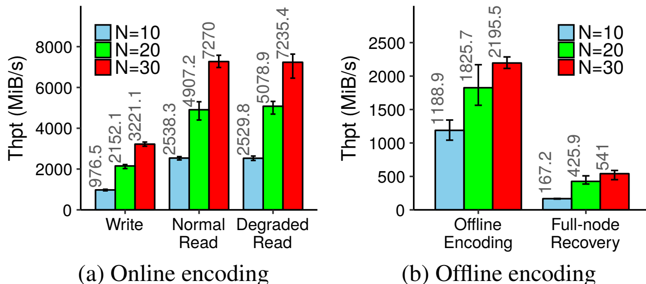
5.5 Performance in Amazon EC2

We finally evaluate OpenEC in Amazon EC2. We configure three settings with N instances, where $N = 10$, 20, and 30 (see §5.1 for the instance type). One instance hosts the OpenEC controller and the HDFS NameNode, and each of the remaining $N-1$ instances hosts an OECClient, an OpenEC agent, an HDFS client, and an HDFS DataNode. We consider the $(9,6)$ RS code, and all $N-1$ clients issue different basic operations as in §5.2. Figure 12 shows the results when OpenEC realizes online and offline encoding atop HDFS-3. We observe consistent throughput patterns as in our local cluster experiments in §5.2 (e.g., both normal reads and degraded reads have similar throughput). Also, the performance of OpenEC scales well with the number of instances.



(a) Automated BindX and BindY (b) Repair optimization

Figure 11: Automated optimization.



(a) Online encoding (b) Offline encoding

Figure 12: Performance in Amazon EC2.

6 Related work

New erasure coding solutions: RS codes [43] are widely deployed today (e.g., [5, 7, 15, 31, 54, 55]), mainly for two reasons. First, RS codes are *maximum distance separable (MDS)*, meaning that under the coding parameters (n,k), the fault tolerance against $n-k$ block failures is achieved with the minimum storage redundancy (i.e., n/k times the original data). Second, RS codes support general coding parameters n and k (provided that $k<n$). However, RS codes have high repair costs, and hence many new erasure coding solutions have been proposed to reduce the repair bandwidth or I/O.

One direction of research is to design new erasure codes. Minimum-storage regenerating (MSR) codes [14] minimize the repair bandwidth and preserve the MDS property. Follow-up studies design new MSR codes [18, 32, 39, 42, 45, 50, 52], some of which are evaluated in open-source DSSs (e.g., PM-RBT codes [39] are evaluated in HDFS, while Butterfly [32] and Clay [52] codes are evaluated in Ceph). Aside MSR codes, some MDS codes incur slightly more repair bandwidth than the minimum point but can be easily constructed with any (n,k) (e.g., [24, 41]), while some non-MDS erasure codes trade more storage redundancy than MDS codes for less repair I/O (e.g., [21–23, 33, 44, 49]). DRC [19] minimizes the cross-rack repair bandwidth in hierarchical topologies.

Another direction of research is to design efficient repair algorithms that apply to general erasure codes. Lazy repair [11, 47] reduces repair executions by deferring a repair until a threshold number of failures occurs. PPR [29] and repair pipelining [25] parallelize a single-failure repair across storage nodes. Proactive degraded reads [20] mitigate tail latencies via the load balancing of read requests.

Unlike the above studies, OpenEC targets a different perspective and focuses on unified and configurable erasure coding management. It supports different new erasure codes and repair algorithms in a unified framework.

Erasure coding programming: Several open-source libraries are available for erasure coding programming. Zfec [10] implements RS codes and is used by Tahoe-LAFS [55]. Jerasure [36] is a C library that supports various erasure codes. It is later extended with GF-Complete [34] to enable fast Galois Field arithmetic. ISA-L [6] is another C library that supports various erasure codes, and it optimizes Galois Field arithmetic for Intel hardware. Both Jerasure and ISA-L libraries are widely used in production (e.g., Ceph and Hadoop 3.0). PyEClib [9] is a Python library used by OpenStack Swift. It builds on liberasurecode [2], which unifies different erasure coding libraries including both Jerasure and ISA-L. OpenEC emphasizes the deployment of erasure codes in DSSs, and it can leverage the above libraries to implement erasure codes via the ECDAG abstraction.

Configurable storage: There is an increasing demand of providing flexibility for storage system management and configuring different storage policies based on application requirements. Existing approaches rely on either client-side customization [12, 13, 28, 37] or the coordination by a centralized controller under the software-defined storage (SDS) framework [16, 48, 51]. OpenEC borrows the same principle from SDS, but specifically focuses on configurable erasure coding management in distributed environments.

7 Conclusions and Future Work

This paper presents OpenEC, a new framework that provides unified and configurable erasure coding management for distributed storage. It leverages the ECDAG abstraction to define erasure codes and configure the workflows of coding operations. Our OpenEC prototype achieves effective performance atop HDFS in both local cluster and Amazon EC2 environments, while supporting a variety of state-of-the-art erasure codes and repair algorithms. Our work sheds light on how to facilitate erasure coding designers to deploy erasure coding solutions in a simple and flexible manner.

This paper currently focuses on HDFS, which organizes data in fixed-size blocks. Our technical report [27] also describes how we integrate OpenEC into QFS [31]. In future work, we study how OpenEC can be deployed in other DSSs, especially object-storage-based DSSs (e.g., Ceph and Swift) that organize data in variable-size objects.

Acknowledgments: We thank our shepherd, Brent Welch, and the anonymous reviewers for their comments. This work was supported by HKRGC (GRF 14216316, CRF C7036-15G) and NSFC (61872414, 61502191). Runhui Li is now with Sangfor Technologies and is the corresponding author.

References

[1] Apache Hadoop 3.0.0. https://hadoop.apache.org/docs/r3.0.0/.

[2] Erasure code API library written in c with pluggable erasure code backends. https://github.com/openstack/liberasurecode.

[3] Facebook's realtime distributed FS based on Apache Hadoop 0.20-append. https://github.com/facebookarchive/hadoop-20.

[4] HDFS erasure coding. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html.

[5] HDFS-RAID. https://wiki.apache.org/hadoop/HDFS-RAID.

[6] Intelligent storage acceleration library. https://github.com/01org/isa-l.

[7] OpenStack Swift. https://wiki.openstack.org/wiki/Swift.

[8] Redis. http://redis.io/.

[9] A simple Python interface for implementing erasure codes. https://github.com/openstack/pyeclib.

[10] zfec – an efficient, portable erasure coding tool. https://github.com/tahoe-lafs/zfec.

[11] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker. Total recall: System support for automated availability management. In *Proc. of USENIX NSDI*, 2004.

[12] F. Chen, M. P. Mesnier, and S. Hahn. Client-aware cloud storage. In *Proc. of IEEE MSST*, 2014.

[13] J. Y. Chung, C. Joe-Wong, S. Ha, J. W.-K. Hong, and M. Chiang. CYRUS: Towards client-defined cloud storage. In *Proc. of ACM EuroSys*, 2015.

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

[15] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, 2010.

[16] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López, Y. Moatti, and E. Rom. Crystal: Software-defined storage for multi-tenant object stores. In *Proc. of USENIX FAST*, 2017.

[17] K. M. Greenan, E. L. Miller, and T. J. E. Schwarz. Optimizing Galois field arithmetic for diverse processor architectures and applications. In *Proc. of IEEE MASCOTS*, 2008.

[18] Y. Hu, H. C. Chen, P. P. Lee, and Y. Tang. NCCloud: Applying network coding for the storage repair in a cloud-of-clouds. In *Proc. of USENIX FAST*, 2012.

[19] Y. Hu, X. Li, M. Zhang, P. P. C. Lee, X. Zhang, P. Zhou, and D. Feng. Optimal repair layering for erasure-coded data centers: From theory to practice. *ACM Trans. on Storage*, 13(4):33, 2017.

[20] Y. Hu, Y. Wang, B. Liu, D. Niu, and C. Huang. Latency reduction and load balancing in coded storage systems. In *Proc. of ACM SoCC*, 2017.

[21] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, et al. Erasure coding in Windows Azure storage. In *Proc. of USENIX ATC*, 2012.

[22] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proc. of USENIX FAST*, 2012.

[23] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg. On fault tolerance, locality, and optimality in locally repairable codes. In *Proc. of USENIX ATC*, 2018.

[24] K. Kralevska, D. Gligoroski, R. E. Jensen, and H. Øverby. Hashtag erasure codes: From theory to practice. *IEEE Trans. on Big Data*, 2017.

[25] R. Li, X. Li, P. P. C. Lee, and Q. Huang. Repair pipelining for erasure-coded storage. In *Proc. of USENIX ATC*, 2017.

[26] 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*, 64(7):1898–1911, 2015.

[27] X. Li, R. Li, P. P. C. Lee, and Y. Hu. OpenEC: Toward unified and configurable erasure coding management in distributed storage systems. Technical report, CUHK, 2019. http://www.cse.cuhk.edu.hk/~pclee/www/pubs/tech_openec.pdf.

[28] M. Mesnier, F. Chen, T. Luo, and J. B. Akers. Differentiated storage services. In *Proc. of ACM SOSP*, 2011.

[29] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi. Partial-parallel-repair (PPR): A distributed technique for repairing erasure coded storage. In *Proc. of ACM EuroSys*, 2016.

[30] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's warm blob storage system. In *Proc. of USENIX OSDI*, 2014.

[31] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast file system. In *Proc. of VLDB Endowment*, 2013.

[32] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Guyot, E. E. Gad, and Z. Bandic. Opening the chrysalis: On the real repair performance of MSR codes. In *Proc. of USENIX FAST*, 2016.

[33] D. S. Papailiopoulos, J. Luo, A. G. Dimakis, C. Huang, and J. Li. Simple regenerating codes: Network coding for cloud storage. In *Proc. of IEEE INFOCOM*, 2012.

[34] J. S. Plank, K. Greenan, E. Miller, and W. Houston. GF-Complete: A comprehensive open source library for galois field arithmetic. Technical Report UT-CS-13-703, University of Tennessee, 2013.

[35] J. S. Plank and C. Huang. Tutorial: Erasure coding for storage applications. Slides presented at USENIX FAST 2013, Feb 2013.

[36] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - version 1.2. Technical Report CS-08-627, University of Tennessee, 2008.

[37] R. Pontes, D. Burihabwa, F. Maia, J. Paulo, V. Schiavoni, P. Felber, H. Mercier, and R. Oliveira. SafeFS: A modular architecture for secure user-space file systems (one FUSE to rule them all). In *Proc. of ACM SYSTOR*, 2017.

[38] N. Prakash, V. Abdrashitov, and M. Médard. The storage versus repair-bandwidth trade-off for clustered storage systems. *IEEE Trans.on Information Theory*, 64(8):5783–5805, Aug 2018.

[39] K. 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.

[40] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Proc. of USENIX HotStorage*, 2013.

[41] 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.

[42] K. V. Rashmi, N. B. Shah, and P. V. Kumar. Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction. *IEEE Trans. on Information Theory*, 57(8):5227–5239, 2011.

[43] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[44] 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*, 2013.

[45] N. B. Shah, K. Rashmi, P. V. Kumar, and K. Ramchandran. Interference alignment in regenerating codes for distributed storage: Necessity and code constructions. *IEEE Trans. on Information Theory*, 58(4):2134–2158, 2012.

[46] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proc. of IEEE MSST*, 2010.

[47] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *Proc. of ACM SYSTOR*, 2014.

[48] I. A. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska. sRoute: Treating the storage stack like a network. In *Proc. of USENIX FAST*, 2016.

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

[50] I. Tamo, Z. Wang, and J. Bruck. Zigzag codes: MDS array codes with optimal rebuilding. *IEEE Trans. on Information Theory*, 59(3):1597–1616, 2013.

[51] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A software-defined storage architecture. In *Proc. of ACM SOSP*, 2013.

[52] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy, et al. Clay codes: Moulding MDS codes to yield an MSR code. In *Proc. of USENIX FAST*, 2018.

[53] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, 2002.

[54] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. of USENIX OSDI*, 2006.

[55] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *Proc. of ACM StorageSS*, 2008.

Appendix: OpenEC Integration into QFS

We implement OpenEC atop Quantcast File System (QFS) (v2.1.1) [31]. In the following, we describe the integration details and evaluation results.

Background of QFS: QFS organizes files in units of *chunks*. Similar to the HDFS architecture, QFS is composed of a single *MetaServer* and multiple *ChunkServers*, such that the MetaServer is responsible for managing chunk placement among ChunkServers, while each ChunkServer provides

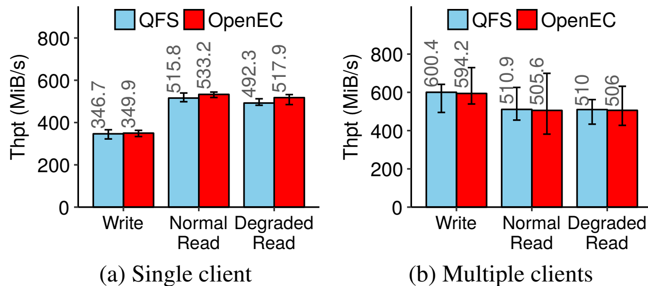


Figure 13: Performance comparison between QFS and OpenEC.

chunk storage. QFS currently supports online encoding under the $(9,6)$ RS code and stores the chunks with the striped layout. Specifically, when a QFS client writes a file, it first partitions the file into six data chunks and generates three additional parity chunks. It then writes the nine chunks across nine ChunkServers. We refer readers to [31] for the details of the QFS workflows.

Integration of OpenEC: We mainly modify the *Layout-Manager* in the MetaServer for OpenEC integration. The LayoutManager is responsible for chunk allocation and failure detection. We modify the LayoutManager in two parts. First, we redirect chunk placement requests to OpenEC's controller, which now takes the responsibility of chunk placement. Second, we report the information of any lost chunk to OpenEC's controller when the LayoutManager detects a lost chunk. Our integration modifies a total of 283 LOC in the QFS codebase. Please refer to our project website for the source code and integration details (`http://adslab.cse.cuhk.edu.hk/software/openec`).

Evaluation results: We compare the performance OpenEC with the vanilla QFS in terms of basic operations using our local cluster and report the results over 10 runs (see §5).

We first compare the single-client performance between QFS and OpenEC. We write a file of size 384 MiB, and issue a normal read request to the file without failures. We also issue a degraded read request to the file by deleting one of the chunks. Figure 13(a) shows that both QFS and OpenEC achieve similar performance: OpenEC's throughput is 0.9%, 3.2%, and 4.9% more than QFS in write, normal read, and degraded read operations, respectively.

We also compare the multi-client performance between QFS and OpenEC. We run five clients, each of which writes a file of 384 MiB. We measure the aggregate throughput of the five clients in writes, normal reads, and degraded reads. Figure 13(b) shows that both QFS and OpenEC achieve similar performance: OpenEC's throughput is 1%, 1% and 0.7% more than QFS in write, normal reads, and degraded reads, respectively. Note that the performance of both normal reads and degraded reads in the multi-client case is similar to that in the single-client case. We suspect that there is resource contention among the reads of multiple clients.