UMR-EC: A Unified and Multi-Rail Erasure Coding Library for High-Performance Distributed Storage Systems

Haiyang Shi, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K. (D.K.) Panda
Department of Computer Science and Engineering, The Ohio State University
{shi.876,lu.932,shankar.50,panda.2}@osu.edu

ABSTRACT

Distributed storage systems typically need data to be stored redundantly to guarantee data durability and reliability. While the conventional approach towards this objective is to store multiple replicas, today's unprecedented data growth rates encourage modern distributed storage systems to employ Erasure Coding (EC) techniques, which can achieve better storage efficiency. Various hardware-based EC schemes have been proposed in the community to leverage the advanced compute capabilities on modern data center and cloud environments. Currently, there is no unified and easy way for distributed storage systems to fully exploit multiple devices such as CPUs, GPUs, and network devices (i.e., multi-rail support) to perform EC operations in parallel; thus, leading to the under-utilization of the available compute power. In this paper, we first introduce an analytical model to analyze the design scope of efficient EC schemes in distributed storage systems. Guided by the performance model, we propose UMR-EC, a Unified and Multi-Rail Erasure Coding library that can fully exploit heterogeneous EC coders. Our proposed interface is complemented by asynchronous semantics with optimized metadata-free scheme and EC rate-aware task scheduling that can enable a highly-efficient I/O pipeline. To show the benefits and effectiveness of UMR-EC, we re-design HDFS 3.x write/read pipelines based on the guidelines observed in the proposed performance model. Our performance evaluations show that our proposed designs can outperform the write performance of replication schemes and the default HDFS EC coder by 3.7x - 6.1x and 2.4x - 3.3x, respectively, and can improve the performance of read with failure recoveries up to 5.1x compared with the default HDFS EC coder. Compared with the fastest available CPU coder (i.e., ISA-L), our proposed designs have an improvement of up to 66.0% and 19.4% for write and read with failure recoveries, respectively.

ACM Reference Format:
Haiyang Shi, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K. (D.K.) Panda. 2019. UMR-EC: A Unified and Multi-Rail Erasure Coding Library for High-Performance Distributed Storage Systems. In *The 28th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '19), June 22–29, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3307681.3325406

This research is supported in part by National Science Foundation grants CCF#1822987, CNS#1513120, IIS#1636846, and OAC#1664137.

HPDC '19, June 22–29, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6670-0/19/06…$15.00
https://doi.org/10.1145/3307681.3325406

1 INTRODUCTION

Replication has been a key technique of reliable distributed storage systems for years [8, 46, 49]. Multiple replicas stored at several locations in the system keep sufficient redundancy to tolerate individual failures. Since the data being generated increases rapidly every day, petabytes of data in today's distributed storage systems are becoming common. Such unprecedented data growth rates encourage modern distributed storage systems to employ efficient storage schemes. Some popular systems, such as Google Colossus [9], Facebook HDFS-RAID [7, 41], the Quantcast File System [30] and Microsoft Azure Storage System [12], are transforming to the use of Erasure Coding (EC) scheme, which offers high reliability and availability at a prominently low storage overhead [40, 48].

Prevalent EC techniques such as Reed-Solomon (RS) [31, 39] code and its variations have been widely used in many distributed storage systems (e.g., HDFS 3.x [10], Ceph [2], QFS [30], Google Colossus [9], Facebook f4 [28], and Baidu Atlas [17]). RS codes are based on *Galois Field* arithmetic [32] (termed as $GF(2^w)$), and they are *Maximum Distance Separable (MDS)*; thus enabling us to recover data from any k of the $(k + m)$ words in case of using an RS (k, m) coder. This EC-based resilient scheme can tolerate up to m node failures with a storage overhead of m/k. In contrast, data replication needs $m+1$ replicas, and the storage overhead is as high as m. For example, the RS(6,3) EC scheme has a storage overhead of 50% and delivers the same fault-tolerance as 4-way replication that incurs a 3x overhead. However, EC encoding and decoding are time-consuming operations, which prevents EC from being used as the primary fault-tolerance mechanism.

To overcome the high computational costs involved with RS erasure coding, two broad categories of coders have been proposed in the community to take advantage of modern hardware capabilities: (1) *EC-Onload*, where optimized host-based libraries such as Jerasure [34] and Intel ISA-L [14] are employed, and, (2) *EC-Offload*, wherein the EC computation tasks can be offloaded to accelerators (e.g., GPGPU [4]) or high-performance network devices (e.g., Host Channel Adapters (HCA) of Mellanox ConnectX-4 and later [24]). While these hardware-optimized libraries can potentially facilitate EC to be employed as a viable choice for fault-tolerance in modern distributed storage systems, currently there are two major issues to consider: (1) these optimized EC libraries are strictly designed and optimized for specific hardware, and have varied performance characteristics and APIs, and, (2) there is a significant mismatch between the semantics of existing EC library interfaces and those that are desired by the upper-layer distributed storage systems. These issues in turn make it hard for users to optimize their storage systems by

utilizing heterogeneous EC libraries in a unified manner, because it would require maintaining large volumes of metadata information necessary to store and retrieve the distributed data blocks. More importantly, due to hardware dependencies of these optimized EC libraries, none of the current-generation storage systems can exploit multiple devices like CPUs, GPUs, and network devices in parallel (i.e., *Multi-Rail Support*) on modern data centers and clouds. These drawbacks leave the available high-speed hardware significantly under-utilized and adversely affect the performance of the EC-based storage system.

In this paper, we first introduce a performance model to identify the challenges and opportunities in existing distributed storage systems. Guided by the performance model, we propose UMR-EC, a Unified and Multi-Rail Erasure Coding library, which provides upper-layer applications and frameworks with a unified way to exploit heterogeneous EC coders. UMR-EC comprises three fundamental building blocks for delivering high-speed encoding/decoding. The first building block is the multi-rail based EC scheme, which enables applications to exploit the heterogeneous EC capabilities of modern data centers in parallel. Secondly, the uniform and asynchronous interfaces of UMR-EC abstract all the functionalities required by the distributed storage system to access different types of hardware-specific coders, and can be used to design a heterogeneity-aware I/O pipeline that can match the high-throughput requirements in distributed storage systems. The third building block is the high-performance UMR-EC runtime, which enables metadata free scheme and EC rate-aware task scheduling for performing EC operations.

To demonstrate the effectiveness of our library, we co-design HDFS 3.x with UMR-EC (i.e., HDFS-UMR). We see that the asynchronous semantics of UMR-EC APIs can match well with the high-throughput requirements from HDFS. Our co-designed I/O pipelines with UMR-EC can outperform default HDFS with both replication and EC schemes. Performance evaluations show that HDFS-UMR can outperform the write performance of replication schemes and the default HDFS EC coder by 3.7x - 6.1x and 2.4x - 3.3x, respectively, and can improve the performance of read with failure recoveries by up to 5.1x compared with the default HDFS EC coder. Compared with the fastest available CPU coder (i.e., ISA-L), our designs have an improvement of up to 66.0% and 19.4% for write and read with failure recoveries, respectively.

Overall, this paper makes the following key contributions:

(1) We introduce an analytical model as a guide for designing efficient single-/multi-rail EC schemes and co-designing I/O pipelines of distributed storage systems.
(2) We propose a unified and multi-rail EC library (i.e., UMR-EC) which can almost achieve the sum of peak throughputs from multiple devices, such that it provides high-performance and high-productivity EC-based schemes for application frameworks on various advanced devices.
(3) We co-design HDFS 3.x with UMR-EC to enable more efficient HDFS write and read pipelines for Big Data workloads; with no additional metadata overhead for exploiting CPU, GPU, and HCA resources in parallel.
(4) We present extensive evaluations for UMR-EC and HDFS-UMR to prove that, through our designs, EC can be used as the primary choice for next-generation distributed storage systems.

To the best of our knowledge, this is the first work to propose a unified EC library that can support efficient multi-rail encoding and decoding across different hardware platforms.

2 MODELING AND GUIDANCE

In this section, we outline the major challenges and opportunities of exploiting multi-rail EC in distributed storage systems via latency performance modeling.

2.1 Performance Models of Write and Read

Many distributed storage systems (e.g., HDFS) have incorporated EC support in their write and read pipelines. Figures 1a and 1b present the high-level workflows for write and read operations in HDFS 3.x [1], respectively. During HDFS write, the HDFS client first computes the parity chunks, and then writes the data and parity chunks to a distributed set of nodes, as shown in Figure 1a. During HDFS read, specifically with node failures, the HDFS client determines how to read sufficient data or parity chunks to recover the original data. The write/read flows depicted in the figure are quite common in other distributed storage systems as well [2, 30].

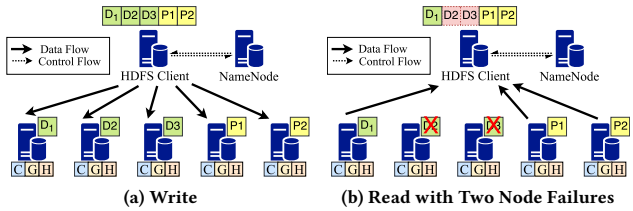


Figure 1: Erasure Coding in HDFS 3.x. The *C*, *G*, and *H* under each DataNode refer to CPU, GPU, and IB HCA, respectively.

2.1.1 Modeling Write Latency. In an EC-based distributed storage system, as shown in Figure 1a, a considerable portion of time is spent on encoding data chunks (denoted by T_{enc}), in addition to the time consumed by other computation (e.g., serialization) and communication (denoted respectively by T_{other} [2] and T_{comm}). T_{enc} is the cost we have to pay to leverage EC technologies in distributed storage systems. Suppose scheme EC(k, m) is employed in the system, the encoding operation splits a data block of size D into k data chunks, and generates m parity chunks of size D/k. Therefore, T_{comm} and T_{enc} can be represented as functions of value D. Finally, the latency of writing a data block of size D can be expressed as follows:

$$T_{\text{w}}(D) = T_{\text{enc}}(D) + T_{\text{comm}}(D) + T_{\text{other}} \quad (1)$$

2.1.2 Modeling Read Latency. Similar to write latency, the read pipeline (depicted in Figure 1b) can also be partitioned into decoding operations (denoted as T_{dec}), communication, and other computation. For EC(k, m) scheme, if any chunk corruptions occur,

[1] Hadoop 3.0.0 alpha2
[2] T_{other} may or may not relate to the size of data chunks.

k out of $k+m$ chunks will be necessary to recover the original data chunks. Such that T_{dec} is also a function of value D. Therefore, the model of read latency and decoding latency can also be denoted as:

$$T_{\text{r}}(D) = T_{\text{dec}}(D) + T_{\text{comm}}(D) + T_{\text{other}} \quad (2)$$

where $T_{\text{dec}}(D) = 0$ if there are no data corruptions.

2.1.3 Major Bottleneck Analysis. The EC-based scheme will incur many activities in the write and read pipelines, such as encoding/decoding, network transfer, and other computation. Without high-performance designs for the write/read pipelines, the storage systems may not be able to deliver the desired performance for the applications. To quantify the overheads of computation in existing distributed storage systems (e.g., HDFS), we conduct some experiments[3] to compare the write and read performance of replication and default EC (i.e., a Java-based Reed-Solomon (RS) implementation) in HDFS 3.x. The results are shown in Table 1.

Table 1: HDFS 3.x Write and Read Throughput Comparisons between Replication and EC. The higher the number, the better the throughput. The number in the parentheses indicates the average number of data blocks residing in the failed DataNodes. The boxes left blank mean that they are not able to be recovered from the failures.

Unit: MB/s	Write	Read				
		w/o failures	1 failure	2 failures	3 failures	4 failures
3-Rep	247.05	1344.18	997.31 (104)	653.17 (249)	-	-
RS(3, 2)	229.31	1609.83	850.92 (75)	467.71 (151)	-	-
4-Rep	211.43	1011.22	761.92 (139)	701.12 (330)	646.01 (520)	-
RS(6, 3)	200.05	1855.07	774.17 (70)	498.79 (137)	311.14 (206)	-
5-Rep	166.08	945.14	805.58 (178)	769.92 (353)	754.94 (616)	582.43 (856)
RS(10, 4)	170.56	1724.06	798.43 (61)	445.75 (125)	322.58 (191)	232.50 (252)

From Table 1, it can be seen that the default EC implementation (i.e., HDFS-EC) has a similar write performance as compared to replication across multiple configurations. Since we employ the fast interconnect (i.e., IB EDR, 100Gbps) and RAMDisk to eliminate bottlenecks in communication and I/O paths as much as possible, we can infer that the poor performance is caused by the EC operation overheads. For write experiments, we implement a no-overhead EC scheme (do nothing in EC operations to mimic the behavior of the theoretically fastest coder) in HDFS to predict the optimal performance for HDFS write. We observe that the no-overhead EC can achieve a throughput of 508 MB/s, which demonstrates that there is a considerable room for potential performance improvement of write operations.

With respect to HDFS read performance, we observe that the default EC scheme performs well in the case on no node failures. When node failures occur, we see that the performance of EC-based read degrades significantly, especially as the number of failures increases. Obviously, the major bottleneck for EC-based reads are the decoding operations. Therefore, the optimal performance for HDFS reads would be to achieve performance similar to the no failure scenario.

[3]These numbers are taken on Cluster B. The cluster specification is introduced in Section 5. InfiniBand and RAMDisk are used for these experiments for achieving the highest performance.

2.2 Challenges and Opportunities

Based on the proposed performance model in Section 2, we analyze challenges and opportunities in further improving the performance of write and read pipelines in distributed storage systems.

2.2.1 Leveraging Multiple Devices Simultaneously. As emphasized in 2.1.3, the major bottleneck in incorporating EC scheme into write/read pipelines is the time-consuming EC operations (i.e., $T_{\text{enc}}(D)$ and $T_{\text{dec}}(D)$). With the fact that modern data centers are equipped with multiple advanced hardware supporting EC operations, as shown in Figure 1, we will assess the challenges and opportunities in resolving the major bottleneck by leveraging multiple devices.

These advanced hardware devices (e.g., multi-core CPUs, GPUs, and InfiniBand/IB) provide a huge potential for reducing the overhead of time-consuming EC operations. A prominent example is the widely used Intel's Intelligent Storage Acceleration Library (ISA-L) [14] that accelerates EC-related linear algebra calculations through the use of advanced SIMD instruction sets like SSE, AVX and AVX2 [13, 15]. Similarly, a new feature called Erasure Coding Offloading (EC Offloading) [25] has been introduced in Mellanox InfiniBand (IB) ConnectX-4 and later adapters. With this feature, EC calculations can be offloaded to the Host Channel Adapter (HCA), which can significantly reduce CPU consumption without sacrificing performance.

Table 2: Example of Onload and Offload EC Coders

Approach	Erasure Coder	Specific Hardware Support	Peak Performance (encode / decode)
Onload	Jerasure	CPU	17.4/10.9 GB/s †
	ISA-L	CPU with SSE/AVX	31.2/29.7 GB/s †
Offload	Mellanox-EC	IB NIC with EC Offload	8.5/7.6 GB/s ‡
	Gibraltar	GPU	11.5/10.6 GB/s †

1) Note that Jerasure is compiled without SSE support, such that Jerasure represents onload erasure coder with common instruction sets while ISA-L with advanced instruction sets.
2) The peak performance labeled with † and ‡ is taken on Clusters A and B, respectively. The cluster specification is in Section 5. We cannot conduct experiments on the same cluster, because 1) Cluster A has CPUs, GPUs, but not the latest ConnectX-5 IB NICs, which are now the only NICs supporting EC offload under $GF(2^8)$, and 2) Cluster B has CPUs, the latest ConnectX-5 IB NICs, but not GPUs.

To better understand their performance characteristics and assess the opportunities to leverage multiple hardware devices, we design a multi-threaded EC benchmark that measures the peak encoding and decoding throughput of these popular EC coding libraries. Table 2 summarizes these results. In Table 2, we observe that: Onload and offload coders have varied performance characteristics and APIs, and, (2) CPU-optimized onload coders such as ISA-L can give the best performance, while, offload-based coders

(GPU/HCA) perform about 2.8 to 3 times worse than the fastest onload coders. However, we also observe that onload coders nearly utilize 100% of the CPU, and as expected the offload coders have much lower CPU utilization (near to 0%).

Given that multiple advanced hardware devices are offering the capability of performing EC operations, $T_{\text{enc}}(D)$ and $T_{\text{dec}}(D)$ in Equation 1 and 2 can be presented more precisely. Let C denote the collection of best EC coders implemented for these hardware devices (e.g., CPU, GPU, and IB HCA). Such that the best encoding and decoding latencies can be achieved by employing the most powerful hardware device in the system is:

$$T_{\text{enc}}(D) = \min_{c \in C} \left\{ T^c_{\text{enc}}(D) \right\} \tag{3}$$

$$T_{\text{dec}}(D) = \begin{cases} 0 & \text{if no corruptions,} \\ \min\limits_{c \in C} \left\{ T^c_{\text{dec}}(D) \right\} & \text{otherwise} \end{cases} \tag{4}$$

where T^c_{enc} and T^c_{dec} is the encoding latency and decoding latency of coder c ($c \in C$), respectively.

Definitely, with exploiting the most powerful hardware device in the distributed storage system, i.e., by substituting $T_{\text{enc}}(D)$ and $T_{\text{dec}}(D)$ with Equation 3 and 4 respectively, Equation 1 and 2 can be improved to some extent. From the overall storage system perspective, however, we infer that leveraging the available compute capabilities like CPUs, GPUs, and network devices in parallel, can achieve the potentially optimal performance; in contrast to employing just a single EC coder (e.g., ISA-L only). The approach of leveraging multiple available devices in parallel to increase parallelism is named **multi-rail** in this paper.

Let D_{c} denote the data size dispatched to EC coder c, where $c \in C$. Since T_{enc} and T_{dec} (if corruptions occur) have the same form of equations, we use T_{ec} to indicate both T_{enc} and T_{dec} in this section ($T_{\text{ec\_mr}}$ refers to latency of multi-rail approach). Such that the EC operation (including encoding and decoding) latency can be rewritten as follows:

$$T_{\text{ec\_mr}}(D) = \max_{c \in C} \left\{ T^c_{\text{ec}}(D_{\text{c}}) \right\} \tag{5}$$

where $D_{\text{c}} \subset D$, $\bigcup\limits_{c \in C} D_{\text{c}} = D$, and $\forall i, j\ D_{\text{i}} \cap D_{\text{j}} = \emptyset$.

For a system with CPU, GPU, and IB HCA coders, suppose D_{C}, D_{G}, D_{H} denote the workload scheduled to CPU, GPU, and IB HCA, respectively, and T^C_{ec}, T^G_{ec}, T^H_{ec} denote the execution time spent on CPU, GPU, and IB HCA, respectively. Then, $T_{\text{ec\_mr}}(D) = \max \left\{ T^C_{\text{ec}}(D_{\text{C}}), T^G_{\text{ec}}(D_{\text{G}}), T^H_{\text{ec}}(D_{\text{H}}) \right\}$. This means all available devices are fully exploited to increase the data parallelism in performing EC operations.

Equation 5 indicates that as long as the workload is dispatched to multiple coders such that they can work together with their peak throughput and complete tasks scheduled to them at the same time, the optimal latency can be obtained. Therefore, the optimal EC operation latency of multi-rail approach is:

$$T_{\text{ec\_mr}}(D) = \frac{D}{\sum\limits_{c \in C} \text{Thr}_c} < \frac{D}{\max\limits_{c \in C} \{\text{Thr}_c\}} = T_{\text{ec}}(D) \tag{6}$$

where Thr_c is the throughput of coder c ($c \in C$). The challenge is how to achieve the sum of peak throughput from multiple devices in a real system.

2.2.2 Full Overlap among EC, Communication and Computation in Write/Read Pipelines. By exploiting the multi-rail approach, we have already achieved optimal performance for EC operations (i.e., T_{enc} and T_{dec}). However, Equation 1 and 2 still reveal chances to further improve the performance of write and read pipeline. To this end, we perform an in-depth analysis of the HDFS write and read pipelines with existing EC coders. We find that there is a big mismatch between the semantics of the existing EC coders and the desired semantics of upper-layer pipeline designs, which prevents the EC operations in current schemes to be fully overlapped with other activities (computation and communication) in the read-/write pipelines. This leads us to rethink about the write and read performance model with non-blocking support.

If the non-blocking semantics are introduced into the performance model of EC operations, then the costly EC operations can be removed from the critical path of the write pipeline. Such that the write latency can be reduced to:

$$T_{\text{w\_nb}}(D) = \max \left\{ T_{\text{enc}}(D), T_{\text{comm}}(D) + T_{\text{other}} \right\} \tag{7}$$

Similarly, the read latency becomes:

$$T_{\text{r\_nb}}(D) = \max \left\{ T_{\text{dec}}(D), T_{\text{comm}}(D) + T_{\text{other}} \right\} \tag{8}$$

Given that the chunk corruptions are not supposed to happen every time, this approach becomes trivial for read pipeline. Suppose one corruption occurs for every 100 reads, then for the 99 healthy reads, $T_{\text{r\_nb}}(D) = \max \left\{ 0, T_{\text{comm}}(D) + T_{\text{other}} \right\} = T_{\text{comm}}(D) + T_{\text{other}} = T_{\text{r}}(D)$. Such that, the benefit obtained by the only one degraded read is amortized by 100 reads and thus becomes minor. Hence, it is not worthy to incorporate this design into read pipeline in a distributed storage systems.

To further improve overlapping, we find the entire write and read pipelines can be partitioned into three functional activities or stages (i.e., EC operations, communication, and other computation) based on Functional Partitioning (FP) (similar to Staged Event-Driven Architecture (SEDA)) [18, 50]. The FP-based approach enables overlapping to occur among EC operations, communication, and other computation through dedicated thread pools on separate processing units. This motivates us to think how to co-design write/read pipelines in existing distributed storage systems (e.g., HDFS) with FP principle to obtain the best overlapping. The detail of the FP-based design will be discussed in Section 4. By utilizing functional partitioning, the write and read latencies can be optimized further as follows:

$$\begin{aligned} T_{\text{w\_fp}}(D) &= \max \left\{ T_{\text{enc}}(D), T_{\text{comm}}(D), T_{\text{other}} \right\} \\ T_{\text{r\_fp}}(D) &= \max \left\{ T_{\text{dec}}(D), T_{\text{comm}}(D), T_{\text{other}} \right\} \end{aligned} \tag{9}$$

2.2.3 Summary. To summarize, the key challenges to enable distributed storage systems to thoroughly benefit from the various advanced devices and CPUs with EC support on modern data centers and cloud are:

(1) How to achieve the sum of peak throughput from multiple devices in a real system?
(2) How to introduce non-blocking semantics to EC operations with high-performance and high-productivity?
(3) How to co-design write/read pipelines with FP principle to achieve the best overlapping in real-world distributed storage systems?

The in-depth modeling analysis motivates us to rethink the EC coding architecture and the corresponding I/O pipeline designs in distributed storage systems. To address these challenges identified, we discuss our designs in the following two sections (Section 3 and Section 4).

3 UMR-EC DESIGN

In this section, we present the programming model and designs employed in UMR-EC as guided by the analysis in Section 2.

3.1 Programming Model

To program with various EC coders in a high-productivity manner, we highlight two major needs from the application perspective: (1) We need unified interfaces and data structures for different EC coders to hide all the platform-specific details, so that applications can easily use these coders in a unified manner. (2) To achieve the goal of efficiently utilizing heterogeneous hardware on modern data centers, asynchronous semantic needs to be introduced to make EC operations overlap with other computations and communications activities, such that the computation overhead of EC operations can be hidden in the write/read pipelines.

Motivated by the optimization opportunities discussed in Section 2.2, we propose our non-blocking programming model into EC operations. The proposed design separates one EC operation into **issue** and **completion** phases, which enable the upper-layer applications to proceed with other tasks, such as computations or communications, while waiting for the issued encoding/decoding requests to be completed. Furthermore, UMR-EC manages heterogeneous hardware in fine-grained strategy on behalf of upper-layer applications. In this way, the UMR-EC library provides flexibility and chances to achieve more overlapping and more efficiency in utilizing underlying hardware at the upper-layer applications.

As listed in Figure 2, we design two blocking APIs (i.e., encode and decode), two non-blocking APIs (i.e., iencode and idecode), two sets of tracking functions (i.e., test and wait), and the corresponding supporting data structures.

API	Description
ECFuture **iencode**(Matrix& data, Matrix& parities, size_t chunkSize); ECFuture **idecode**(Erasures& erasures, Matrix& data, Matrix& parities, size_t chunkSize);	Non-blocking encode and decode
void **wait**(); void **wait**(const ECFuture& future); void **wait**(vector<ECFuture>& futures);	Blocking wait on all or partial submitted requests to be completed
bool **test**(); bool **test**(const ECFuture& future); bool **test**(vector<ECFuture>& futures);	Non-blocking test whether all or partial submitted requests are completed

Figure 2: Proposed APIs

Our proposed APIs are beneficial to distributed storage systems in the following ways:

(1) The proposed approach offers opportunities to the distributed storage systems to overlap the costly EC computations in the existing write/read pipelines with slight modifications. One real-world example of co-designing HDFS 3.x with UMR-EC is given in Section 4.
(2) The *test* and *wait* APIs provide two different approaches to track and check the progress of submitted EC tasks. The upper-layer applications can choose a proper one to deal with request completions.
(3) The proposed APIs enable upper-layer applications to employ heterogeneous hardware platforms via two coder selection strategies: (a) **Single-rail** mode, wherein the user can post EC tasks to a single specific EC coder of choice, using either blocking or non-blocking APIs, and, (b) **Multi-rail** mode, wherein multiple heterogeneous coders are leveraged automatically and concurrently, based on the underlying hardware, through the use on non-blocking APIs. More importantly, if the application framework is designed with our proposed APIs, they can easily switch between these two modes by a configuration parameter.

3.2 Runtime Architecture

Our UMR-EC library fully implements the above-mentioned programming model with C/C++ and Java bindings. The *ErasureCoderManager* in the UMR-EC library provides pluggable interfaces for adding new coders to UMR-EC in a convenient manner. For example, to add a new Mellanox EC offloading coder into UMR-EC, we only need 58 LOC to achieve it. Specifically, in the multi-rail mode, the *ErasureCoderManager* will auto-select a set of EC coders based on available hardware and their performance characteristics, for maximizing performance for the upper-layer application. In the following two sub-sections, we highlight two key techniques in this layer, which bring performance benefits to multi-rail based EC operations.

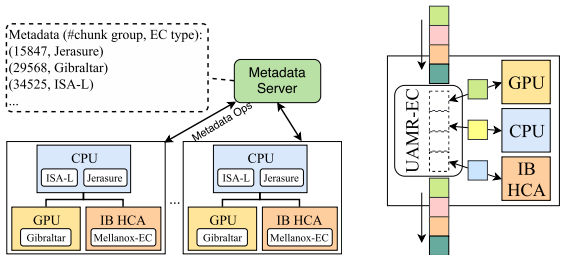
3.3 Metadata-Free Multi-Rail EC

As described before, current storage systems use well-known EC coding schemes such as Reed-Solomon [39]. However, these RS-based EC coders, which might be implemented with variant Galois Field arithmetics (e.g., $GF(2^w)$) or generator matrices, could generate different results though given identical input. This is known as the cross-compatibility issue among different EC coders. To exploit multi-rail EC support efficiently, there are two ways to handle the cross-compatibility challenge.

3.3.1 Metadata-Based Design. A generic approach is to track the EC type of each block or chunk groups by the assistance of a central or distributed *metadata server*. In this design, data chunks are split into chunk groups. Each chunk group is encoded and decoded using the same EC coder. Therefore, the metadata server maintains a per-chunk-group metadata information entry of the following format (#chunk group, EC type), as shown in Figure 3a. For instance, a metadata record (29568, Gibraltar) indicates that the chunk group with ID 29568 is encoded by the EC coder, Gibraltar, and will be decoded using the same EC coder. Now, in terms of performance, the choice of the number of chunks per group can become a trade-off between increased data parallelism and metadata storage overhead. Increasing the number of groups can help us exploit available compute resource, but this comes at the cost of higher metadata overhead. Also, in case the required hardware is unavailable during decoding, this design cannot easily resolve the

issue. However, this approach enables us to plug-in and leverage any new EC coder, irrespective of the type of EC scheme it employs.

3.3.2 Metadata-Free Design. The metadata-based approach is generic but with high overhead of maintaining and retrieving metadata. Therefore, we come up with a metadata-free design to eliminate the overhead. The requirement of obtaining metadata-free is to guarantee cross-compatibility among all available RS-based EC coders, which means that all involved EC coders generate equivalent output once given the same input. UMR-EC achieves cross-compatibility by unifying (1) *generator matrix*, and (2) *word size "w"* in all plugined RS-based EC coders without any negative effect on performance. With this key idea, we design a metadata-free EC engine, as shown in Figure 3b, that can encapsulate the different capabilities of underlying hardware into a single highly available EC service. This does not only enable zero metadata overhead, but also enables us to optimize the overlap of I/O operations within the read/write pipelines more efficiently. In addition to this, in case the required hardware is unavailable during runtime the UMR-EC can continue to provide highly available service by leveraging any of its compute resources. From the perspective of upper-layer applications, it can transparently leverage the underlying heterogeneity with the proposed unified APIs.



(a) Metadata-Based Approach **(b) Metadata-Free Approach**

Figure 3: Comparison of Multi-Rail Design Alternatives. A metadata record (29568, Gibraltar) indicates that the chunk group with ID 29568 is encoded by Gibraltar, and will be decoded with the same coder.

Note that some hardware devices (e.g., IB) have limitations on EC support like different word size support. In case there are devices which cannot be unified to support metadata-free approach, UMR-EC will perform a cross-compatibility check during the installation phase, and only selects the compatible devices to perform multi-rail EC operations. In the worst case, UMR-EC just enables single-rail EC, which is still more efficient than other EC libraries because of overlapping. This could be a limitation on metadata-free design, but note that the metadata-free approach is promising since we believe supporting the same generator matrix and word size by future devices will be a norm. For instance, in this paper, we successfully enabled this feature across two different clusters (see Section 5.1).

Section 5.2.1 presents detailed comparisons between metadata-based and metadata-free approaches and illustrates the efficiency of the metadata-free design.

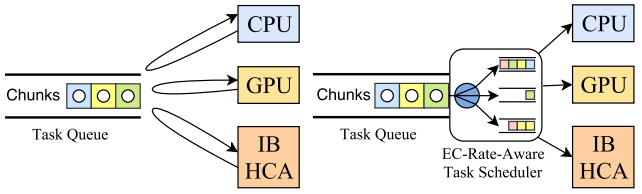
3.4 EC-Rate-Aware Task Scheduling

In the multi-rail mode, multiple underlying hardware will be carrying out EC computations simultaneously. However, the capability of a hardware device to perform EC operations is different to the others, and the performance is not consistent for the same hardware under variant situations and conditions. To maximally exploit underlying hardware, we need to design a light-weight and efficient (i.e., be able to achieve aggregated performance from multiple devices as mentioned in Equation 5) task scheduler to balance workloads.

Since the task scheduler will work in the critical path, it should be as light-weight as possible. Therefore, we compare the following two design alternatives for task scheduling.

3.4.1 First Come, First Served (FCFS) Task Scheduling. Figure 4a presents one implementation of FCFS task scheduling. In this approach, both onload and offload EC coders, are assigned to multiple worker threads in the same thread pool. Since they are in the same priority, the thread finishing one task can fetch another one directly from a global task queue, without any assistance from an explicit task scheduler thread. This approach is straightforward, but introduces possible contentions of retrieving tasks concurrently among multiple EC coders, and fails to balance workloads to achieve optimal performance.

3.4.2 EC-Rate-Aware Task Scheduling. We introduce an EC-rate-aware task scheduler, as depicted in Figure 4b, to dispatch tasks to different devices explicitly based on their current EC operation rates. This design is based on the assumption that the current EC operation rate can reflect the device EC capability as well as its current workload status. In this design, worker threads are split into thread groups based on the EC coders employed, such that threads using the same EC coders are grouped together. The task scheduler monitors the EC rates of different thread groups, and assigns a balanced amount of tasks into thread groups according to their normalized EC rates. The tasks assigned to the same thread group are placed into a sub task queue which can be pulled by the thread group exclusively. The advantages of the EC-rate-aware approach are achieving the awareness of workload characteristics and throughput of every EC coder, as well as being light-weight.



(a) FCFS Approach **(b) EC-Rate-Aware Approach (Proposed)**

Figure 4: Comparison of Scheduling Design Alternatives

To achieve higher efficiency, the thread pool for task scheduling is designed to be lock-free by two techniques: 1) Designing a lock-free task queue, which guarantees the expected order of multiple tasks with less overhead compared to the with-lock counterpart, and 2) Replacing mutex with spinlock + atomic, because the performance penalty of the use of mutex is considerable, since the context switching is really frequent and fast.

In Section 5.2.2, we will provide detailed comparisons between these two approaches and explain why the EC-rate-aware design can achieve higher aggregated throughput.

4 CO-DESIGN HDFS WITH UMR-EC

This section presents co-designed HDFS architecture with UMR-EC.

4.1 Writes

The default EC write pipeline of HDFS 3.x, as depicted in Figure 5a, consists of three basic phases: ① [**Collecting**] collects k data chunks, ② [**Encoding**] encodes on k data chunks to generate m parity chunks, and, ③ [**Streaming**] sends all chunks to corresponding DataNodes. This write pipeline coincides with Equation 1. Thus, the optimization approaches discussed in Section 2.2 can be applied to fully overlap between the three basic phases and to leverage heterogeneous compute resources in parallel.

Motivated by Equation 7, we propose one non-blocking write pipeline, as illustrated in Figure 5b, which achieves better performance by overlapping [**Encoding**] phase with [**Collecting**] phase and computation by utilizing asynchronous EC encoding provided in UMR-EC. Changes made to enhance the default HDFS write pipeline into a non-blocking pipeline, include: (1) replacing the deployed EC coder with the *ErasureCodingService*, and, (2) using UMR-EC's asynchronous APIs with a window-based approach to track outstanding encoding/decoding operations for completion.

To clarify the new non-blocking write pipeline, we introduce a new phase ④ [**Issuing**] that issues encoding requests. As shown in Figure 5b, after each [**Collecting**] phase, there is an [**Issuing**] phase instead of [**Encoding**] phase. As demonstrated in the figure, [**Encoding**] phase is able to overlap with [**Collecting**] phases and computation occurring in the main thread. After n [**Issuing**] phases, where n is a configurable window size, there will be a synchronization barrier which forces the write pipeline to stop and wait for all encodings to be completed. Finally, it performs a [**Streaming**] phase to send all chunks to DataNodes after the synchronization barrier.

With the non-blocking design, performing a huge [**Streaming**] phase at each synchronization barrier can become a potential performance bottleneck. To alleviate this problem and further improve write performance, we propose a Functional Partitioning (FP) [18] (similar to Staged Event-Driven Architecture (SEDA) [50]) based write pipeline, which is illustrated by Equation 9 and Figure 5c. Based on the functional partitioning principle, the entire write pipeline can be partitioned into three disjoint functional activities or stages, i.e., data preparing (i.e., [**Collecting**] phase), EC operations (i.e., [**Encoding**] phase), and communication (i.e., [**Streaming**] phase). As depicted in Figure 5c, in addition to main thread collecting data chunks and UMR-EC performing EC operations, there is one more dedicated communication thread pool (denoted as *Comm* in the figure). Rather than returning generated parity chunks to the main thread, UMR-EC in FP-based write pipeline will delegate the dedicated communication thread pool to send the parity chunks to remote DataNodes. Thus it enables better overlap among [**Collecting**] phases, [**Encoding**] phases, [**Streaming**] phases, and computation occurring in the main thread. To guarantee data consistency, the dedicated communication thread conducts [**Streaming**] phases with the same order as main thread posts requests.

4.2 Reads

The default HDFS 3.x read pipeline consists of two major phases: ① [**Fetching**] phase that aggregates data and/or parity chunks[4] from the remote DataNodes, and, ② [**Decoding**] phase that recovers any lost data. Figure 6a shows the default read pipeline of HDFS 3.x. In the default blocking read pipeline of HDFS, if any corrupted chunks are detected in [**Fetching**] phase, then a [**Decoding**] phase will follow directly after the [**Fetching**] phase. The latency model is illustrated by Equation 2. Thus, it is obvious that the read performance degrades significantly if data losses are frequent.

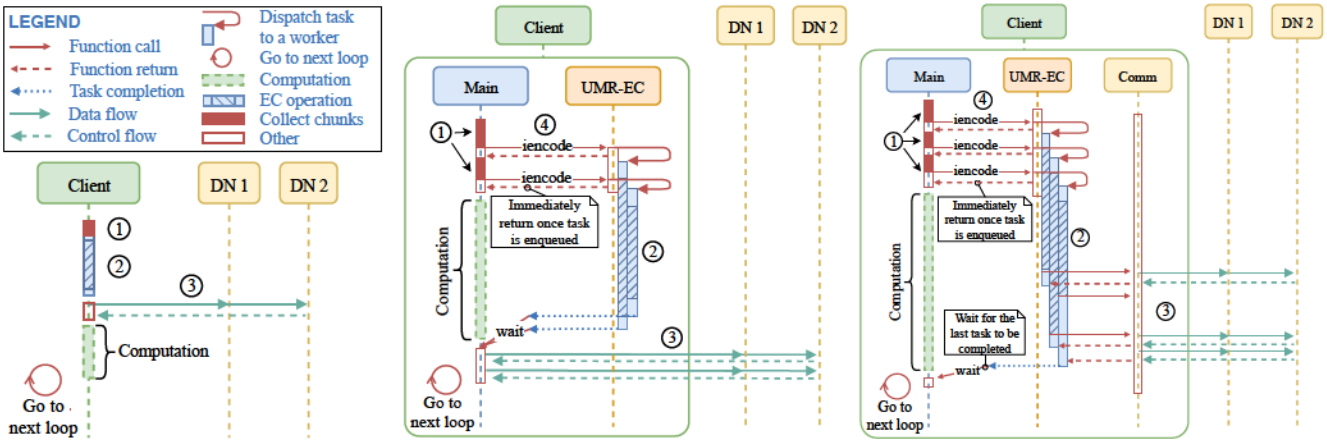
As already discussed in Section 2.2, chunk corruptions are not supposed to happen very often. Suppose one corruption occurs for every 100 reads, then for the 99 healthy reads, $T_{r\_nb}(D) = \max\{0, T_{comm}(D) + T_{other}\} = T_{comm}(D) + T_{other} = T_r(D)$. Such that, the benefit obtained by optimizing the only read with EC decoding is amortized by 100 reads and thus becomes minor. Hence, it is not profitable to incorporate this design into read pipeline in a distributed storage systems. However, the read latency can benefit from FP-based design approach, which is demonstrated by Equation 9. To clarify the new FP-based read pipeline, we introduce an additional phase, known as the ④ [**Issuing**] phase, that issues either non-blocking fetching requests or decoding requests. Our proposed FP-based design, implemented by exploiting the non-blocking semantics proposed in UMR-EC, splits the entire read pipeline into three functional components: 1) communication, 2) decoding, and 3) computation. With the FP-based design, the main thread focuses on computation jobs after issuing several fetching requests. The dedicated communication thread pool is wherein actual read operations happen. Once data corruptions occur, the communication thread pool will involve UMR-EC in recovering data chunks. As clarified in Equation 9, the FP-based approach enables better overlap among [**Fetching**], [**Decoding**], and computation occurring in the main thread. In the FP-based pipeline, synchronization is performed once after issuing n fetching operations, where n is a configurable window size. As we can see from this analysis, the read performance of the FP-based pipeline is better than the default one no matter data corruptions happen or not.

4.3 Multi-Instance-Aware Coordination

In an HDFS cluster, a node may have multiple writers/readers simultaneously writing/reading multiple files. Since each writer and reader will hold one UMR-EC instance to perform blocking or non-blocking EC operations, UMR-EC should be able to eliminate possible resource contentions among multiple UMR-EC instances running on the same node.

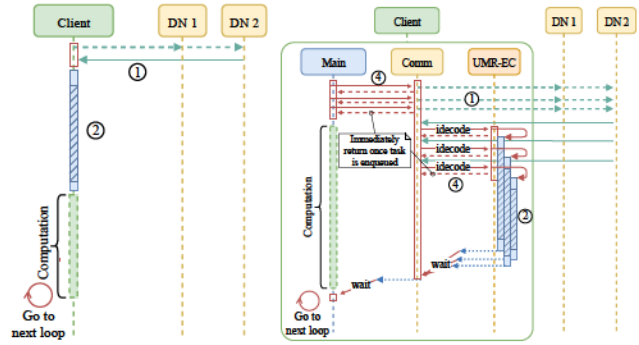
To achieve this goal, we propose multi-instance-aware coordination schemes in UMR-EC. The basic idea is that, by using shared memory, each UMR-EC instance can detect the existence of other instances running on the same node. Thus, the total number of running instances can be detected by all the UMR-EC instances on the same node. We need this information for coordination because we observe that the best number of threads to simultaneously perform EC operations on each device cannot be too high or too low. Within UMR-EC library, we design an architecture-aware tuning table mechanism to keep the pre-tuned best number of parallel

[4] The parity chunks will only be fetched if any data chunks corrupted.



(a) Blocking Write Pipeline (Def) (b) Non-Blocking Write Pipeline (Proposed) (c) FP-Based Write Pipeline (Proposed)

Figure 5: Default and Proposed Write Pipelines



(a) Blocking Read Pipeline (Def) (b) FP-Based Read Pipeline (Proposed)

Figure 6: Default and Proposed Read Pipelines

threads (i.e., best parallelism) for several modern CPUs, GPUs, and IB HCAs. Note that the table can be extended by running our tuning tools on a new hardware device, if it is missing in the table. If a matched record cannot be found in the tuning table, UMR-EC will choose the default numbers.

Once the UMR-EC library is loaded in HDFS, the proper number of EC operation threads will be calculated based on the best parallelism of available devices on the node and the total number of running instances. During runtime, all UMR-EC instances will adjust their parallelisms periodically and automatically based on the detected number of instances. With this multi-instance-aware coordination, our design can dynamically and transparently offer optimal performance for end applications.

5 EVALUATION

In this section, we conduct extensive evaluations, which are divided into the following categories:

(1) Evaluations with raw coders and design alternatives, to study the performance of the UMR-EC library

(2) Evaluations of 'HDFS+UMR-EC' co-designs with I/O intensive workloads, comprehensive workloads, and failure recovery tests

5.1 Experimental Setup

Two real-world clusters (i.e., A and B) are used in this paper as shown in Table 3. The version of Hadoop used in the evaluation is 3.0.0 alpha2. We conduct all the experiments on RAMDisk to minimize the performance influence from storage. Other necessary drivers and libraries are CUDA 8.0, Mellanox OFED 4.2, Jerasure 2.0, ISA-L 2.18.0, Gibraltar [5], and Mellanox-EC [6]. Note that Jerasure in our experiments is compiled without SSE support, such that Jerasure represents onload erasure coder with common instruction sets while ISA-L represents onload coder with advanced instruction sets.

Table 3: Specifications of Clusters

Specification	Cluster A	Cluster B
Processor	Intel Broadwell E5 v2680	Intel Broadwell E5-2697A v4
Frequency	2.4 GHZ	2.6 GHZ
GPU	K80 × 2	
RAM (DDR)	128 GB	256 GB
Interconnect	ConnectX-4 IB-EDR (100 Gbps)	ConnectX-5 IB-EDR (100 Gbps)
OS	CentOS 7.2	Red Hat 7.2
JDK	OpenJDK 1.8.0	OpenJDK 1.8.0
Scale	16 nodes	32 nodes
Policies	CPU + GPU	CPU + IB

Note that we are not able to perform multi-rail with CPU, GPU, and HCA simultaneously, because 1) Cluster A has CPUs, GPUs, but not the latest ConnectX-5 IB NICs, which are now the only NICs supporting EC offload under $GF(2^8)$, and 2) Cluster B has CPUs, the latest ConnectX-5 IB NICs, but not GPUs.

[5] Github: https://github.com/jaredjennings/libgibraltar, commit: c93f9d8c3be70ded173822cdca2e51900a3f5ed1

[6] Github: https://github.com/Mellanox/EC, commit: 00bf091aa14322baf4425f8a6d5d134e91fe2a5c

5.2 Raw Coder Benchmarks

In this section, we conduct benchmarks to compare design alternatives and to evaluate the performance of single-rail and multi-rail modes. The chunk size is fixed to 64KB, which is widely employed in real-world storage systems [10, 30]. Term *ISA* is short for ISA-L coder, *Jer* for Jerasure coder, *Gib* for Gibraltar coder, and *Mx* for Mellanox-EC coder. Meanwhile, we use name concatenations of utilized coders to represent multi-rail mode. For instance, term *Jer+Gib* represents the multi-rail mode with Jerasure and Gibraltar coders, etc.

5.2.1 Metadata-Free vs. Metadata-Based. As aforementioned in Section 3.3, there exist two approaches (i.e., Metadata-based approach and Metadata-free approach) to efficiently exploit heterogeneous hardware to perform EC operations. Figure 7 presents the performance comparison of both approaches. For performing the benchmarks of metadata-based approach, we implement a metadata server with Memcached to store the metadata information. In the benchmarks, we only deploy one metadata server and one client, since this configuration is the best case for the metadata-based approach as the metadata server will be a bottleneck if the number of clients increases. As shown in Figure 7, the number of chunks are varied from one until getting the peak throughout. Definitely, the one-chunk case presents the worst performance scenario because of the significant communication overhead. Meanwhile, the differences between 16-chunk and 32-chunk are trivial in all cases, such that we do not have to show other cases. To summarize, Figure 7a illustrates that EC encoding operations with metadata-free approach can be sped up by up to 4.2x and 1.3x compared to the worst case and the best case of metadata-based approach, respectively. Figure 7b shows that the speed-ups for decoding are up to 4.6x and 1.4x, respectively.

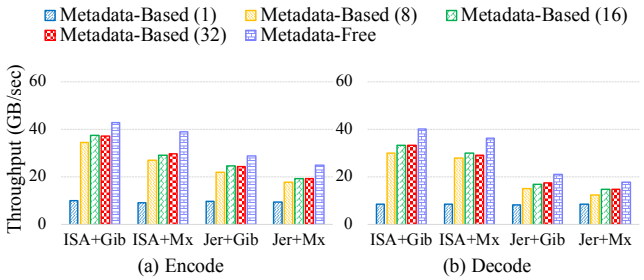


Figure 7: Performance Comparison of Metadata-Free Approach and Metadata-Based Approach (Cluster A). In the legends, the number in parenthesis refers to the number of chunks in a chunk group. All chunks in one chunk group share the same metadata.

5.2.2 EC-Rate-Aware Task Scheduling. To find out a proper approach maximizing the overlap of EC operations with computations and communications, we evaluate two design alternatives, i.e., EC-rate-aware task scheduling and First Come First Served (FCFS) task scheduling. The design details of both approaches are in Section 3.4. Figure 8 depicts the performance comparison of EC-rate-aware task scheduling and FCFS task scheduling with one node. The leftmost stacked bar in each bar group indicates the theoretical summation of the peak throughputs of exploited coders. The EC-rate-aware approach, which has a performance very close to the summation, outperforms the FCFS approach by up to 17.5% and 17.3% for encoding and decoding, respectively. This indicates that our proposed EC-rate-aware design is effective and sufficient enough to get the peak aggregated throughput.

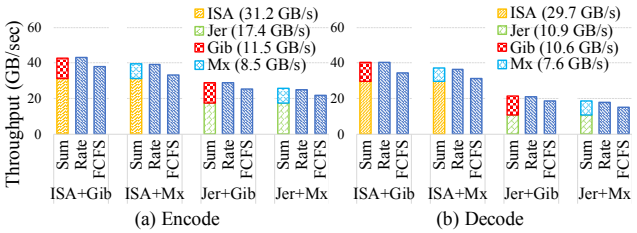


Figure 8: Performance Comparison of EC-Rate-Aware Task Scheduling and FCFS Task Scheduling (Cluster A). In the legends, the number after each coder name indicates the peak throughput in single-rail mode. Since the overhead introduced by the single-rail mode is trivial, the peak throughputs can be considered as the best throughputs which the coders can gain with well-tuned thread pool on our clusters.

To figure out the reason that slows down the performance of FCFS, we conduct some profiling experiments to analyze the bottlenecks. As highlighted by the red oval in Figure 9, the main factor is the tail latency. The tail latency issue is also revealed by Equations 7 and 8. As all workers in the FCFS task scheduling have the same priority, a task can be fetched by slower coders as well as faster coders with the same probability. However, faster coders could be 4x faster than slower coders (e.g., ISA vs. Mx). If the last several tasks are carried out by both faster coders and slower coders, the latency is bounded on the latencies of slower coders. On the other hand, the tasks in the EC-rate-aware approach are assigned according to the EC-rates of involved coders, and faster coders in this approach have a relatively higher priority than slower coders. Since the tasks are well-balanced among coders, the tail latency observed in FCFS approach is eliminated in EC-rate-aware one.

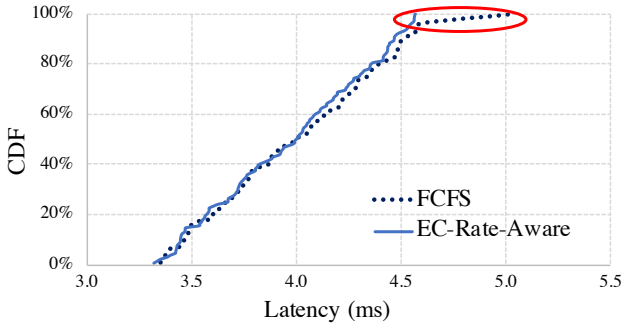


Figure 9: Cumulative Distribution Functions (CDFs) of EC-Rate-Aware Task Scheduling and FCFS Task Scheduling. The red oval highlights the tail latency of FCFS, which is the root cause of its performance degradation.

5.3 HDFS Micro-benchmarks

In this section, we choose three benchmarks and conduct them with the largest scale on each cluster: (1) I/O intensive workloads -

TestDFSIO Write benchmark, (2) Comprehensive workloads - Sort benchmark, and (3) Failure recovery benchmark. Since the default EC scheme in HDFS 3.0.0 alpha2 is RS(6, 3) with a chunk size of 64 KB, we choose four as the replication factor to conduct fair comparisons. Based on Table 1, the trends of performance comparisons between EC and replication are kept for other replication factors. To describe clearly, we introduce some terms: *No-Overhead EC* refers to an ideal EC coder which has no computation overhead, it is mimicked by doing nothing in encoding/decoding. *4-Rep* is short for 4-way replication. *HDFS-RS* refers to default EC coder (i.e., a Java-implemented Reed-Solomon coder) in HDFS 3.x. *NoFailure* refers to read without failures. *Def* indicates default approaches (e.g., 4-Rep, HDFS-RS) in HDFS 3.x. *UMR-SR* refers to UMR-EC single-rail mode, and *UMR-MR* refers to UMR-EC multi-rail mode.

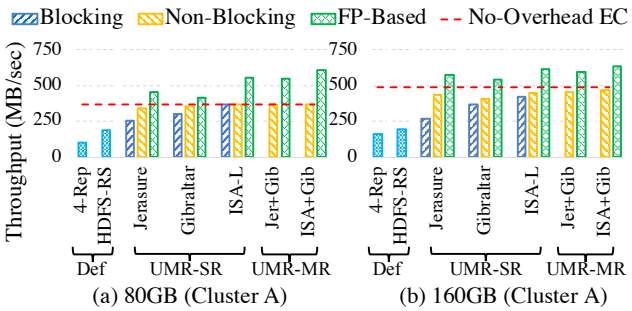


Figure 10: TestDFSIO Write on Cluster A (16 nodes).

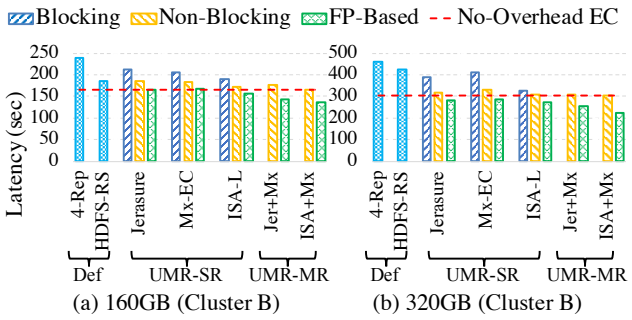


Figure 11: Sort on Cluster B (32 nodes).

5.3.1 I/O Intensive Workloads - TestDFSIO Write. The TestDFSIO Write tests are run with 16 maps/reduces on Cluster A. The results in Figure 10 demonstrate that the non-blocking approach offered by the UMR-EC library can fill up the gaps between No-Overhead EC and other EC coders by overlapping most of the overhead introduced by encoding. With the non-blocking single-rail and multi-rail APIs, the HDFS-UMR performs up to 3.7x and 2.4x faster than employing 4-Rep and HDFS-RS, respectively.

On the other hand, the most effective approach shown in the figure is the FP-based design. By overlapping data preparing, EC operations, and communication, the coders can achieve higher performance than No-Overhead EC by up to 1.66x. Compared to 4-Rep and HFDS-RS, the FP-based design has up to 6.1x and 3.3x improvements, respectively. In the meantime, the FP-based single-rail and multi-rail designs improve HDFS-UMR by up to 51.8% and 66.0%, respectively, compared to HDFS employed with the best EC coder on CPU, i.e., ISA-L. The results demonstrate that our FP-based design with UMR-EC can achieve optimal overlapping.

5.3.2 Comprehensive Workloads - Sort. To study how extra computations impact the performance of EC coders, we conduct a comprehensive benchmark (i.e., sort) to get extra computations involved in. The performance comparison between Figure 11a and 11b implies that the more workload the more benefit we can obtain from employing UMR-EC. It is because the multi-rail approach, which utilizes both onload and offload coders, reduces the performance degradation of onload coder by scheduling more workload to offload coders once there are heavy loads on CPUs.

The results from Cluster B clearly depict the trend of latency changes. As shown in Figure 11b, the FP-based multi-rail approach can significantly reduce the overhead and benefit from offload coders. It reduces latencies by up to 73.4% and 52.3% compared to 4-Rep and HDFS-RS, respectively. While compared with the best EC coder on CPU (i.e., ISA-L) in HDFS, the performance improvements achieved by exploiting FP-based single-rail and multi-rail are up to 21.9% and 45.5%, respectively.

5.3.3 Failure Recovery Evaluation. We conduct TestDFSIO read with failures to study the performance issue of DataNode failures. The recovery benchmark is conducted on Cluster A with 16 maps/reduces, and on Cluster B with 32 maps/reduces. As shown in Figure 12, the throughput performance of EC coders decreases dramatically along with the increasing of DataNode failures. This is a well-known issue of employing EC in modern distributed storage systems. However, the HDFS-UMR solves this issue by re-designing the read pipeline with the benefit of our UMR-EC library.

From the quantitative perspective, the FP-based multi-rail read pipeline improves the read performance up to 2.4x, 2.3x, and 2.6x for one DataNode failure, two DataNode failures, and three DataNode failures, respectively, compared to 4-Rep. Meanwhile, if compared with HDFS-RS, the improvements are up to 2.8x, 3.9x, and 5.1x for one DataNode failure, two DataNode failures, and three DataNode failures, respectively. In addition, the percentage improvements obtained by FP-based multi-rail approach against the best onload coder (i.e., ISA-L) are up to 17.9%, 13.8%, and 19.4% for one DataNode failure, two DataNode failures, and three DataNode failures, respectively. The numbers of two DataNode failures are not shown in Figure 12 because of space limitation.

6 RELATED WORK

Many studies in the community have been focusing on employing EC as an alternative fault-tolerance technique to replication in distributed storage systems.

Erasure Coding for Storage Systems Erasure codes have been extensively adopted in designing storage systems [1–3, 9–11, 16, 17, 28, 30, 49], as they offer higher reliability than replication methods with much lower storage overheads [48]. Towards reducing the recovery overhead in erasure codes, several studies introduce *local parities*, such as Local Reconstruction Codes [12] and Locally Repairable Codes [41]. In the meantime, some research works to reduce the network bandwidth usage are proposed, such as Partial-Parallel-Repair [27], Repair Pipelining [19], and [5, 11, 36, 37]. On the other hand, EC schemes are also being employed to design

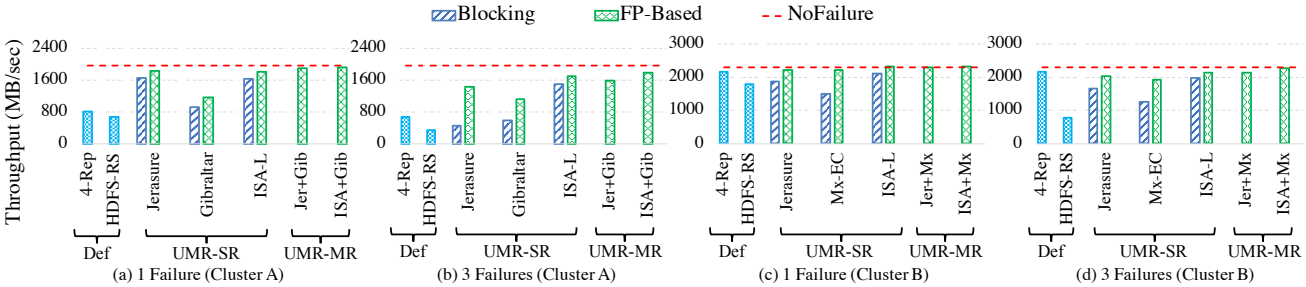


Figure 12: Recovery Evaluation with One and Three DataNode failures on Cluster A (16 nodes) and Cluster B (32 nodes).

resilient key-value store-based systems, including, Cocytus [52], EC-Cache [35], RDMA-Memcached with Online EC support [43], Hybris [6], and BCStore [20]. Recently, substantial research efforts, such as RS-based Hitchhiker [38], Locally Repairable Codes based design of HDFS-Xorbas [42], hybrid designs such as HACFS [51], and Facebook HDFS-RAID [41] have been proposed for designing large-scale Hadoop clusters, with focus on reducing network, disk and recovery overheads. This increased focus on EC for storage resilience serves as a motivation for this paper.

Multi-Rail Technologies The multi-rail concept has been widely used in designing high-performance communication runtimes and systems. Multi-rail MPI [21] and Multi-path RDMA [22, 26] are leveraging multi-rail designs to accelerate communication. On the other hand, prior studies such as [45, 47] demonstrate that utilizing multiple devices simultaneously is able to deliver faster computation performance. Our early study [44] sheds substantial light on multi-rail EC.

Hardware Acceleration and Optimizations for Erasure Coding Generic EC libraries such as Jerasure [33, 34], that support a wide variety of erasure codes, including Reed-Solomon [39], Cauchy-Reed-Solomon [31], etc., are widely being used today. However recent advancements in CPU architectures are enabling the design of advanced support for high-speed computations for EC [14, 23, 32], by taking advantage of instructions sets such as SSE, AVX, and AVX2. Along similar lines, to reduce CPU consumption and leverage the capabilities of high-performance network adapters, Mellanox ConnectX-4 (and later) adapters are enabling EC calculations to be offloaded to the HCA [25].

Initial work has been done to integrate the Intel ISA-L and Mellanox Offload coders into HDFS 3.x [24]. In addition to this, EC API libraries such as OpenStack's liberasurecode [29] also attempt to unify different EC libraries. However, they do not provide any support for asynchronous EC encode/decode semantics or a way to employ multiple EC-Onload and EC-Offload coders in parallel. In contrast, in this paper, we identify and overcome the bottlenecks and the limitations of these existing approaches. Based on our analysis, we introduce a unified high-performance and non-blocking UMR-EC library that can exploit the heterogeneous EC compute capabilities of various coders and hardware devices concurrently, to enable high-throughput I/O and storage-efficient resilience for distributed storage systems such as HDFS.

7 CONCLUSION

In this paper, we first introduce a performance model to analyze the design scope of efficient EC schemes in distributed storage systems. Guided by the performance model, we propose UMR-EC, a Unified and Multi-Rail Erasure Coding library that provides I/O-intensive applications with a unified way to exploit heterogeneous EC coders on modern data center and cloud environments. We design UMR-EC based on three main building blocks: (1) the support of multi-rail EC to exploit heterogeneous EC capabilities of modern data centers concurrently, (2) the uniform and asynchronous interfaces to abstract EC functionalities required for high-performance distributed storage systems, and, (3) the high-performance UMR-EC runtime, which enables metadata free scheme and EC rate-aware task scheduling for performing EC operations.

Through in-depth performance evaluations, we demonstrate that HDFS-UMR can outperform the write performance of replication schemes and the default HDFS EC coder by 3.7x - 6.1x and 2.4x - 3.3x, respectively, and can improve the performance of read with failure recoveries up to 5.1x compared with the default HDFS EC coder. Compared with the fastest CPU coder (i.e., ISA-L), HDFS-UMR has an improvement of up to 66.0% and 19.4% for write and read with failure recoveries, respectively. In the future, we plan to support other EC schemes such as FPGA-based coders and non-RS coders in UMR-EC.

REFERENCES

[1] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. 2002. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 1–14.
[2] Ceph. 2016. Ceph Erasure Coding. http://docs.ceph.com/docs/master/rados/operations/erasure-code/.
[3] Yu Lin Chen, Shuai Mu, Jinyang Li, Cheng Huang, Jin Li, Aaron Ogus, and Douglas Phillips. 2017. Giza: Erasure Coding Objects across Global Data Centers. In *Proc. USENIX Annu. Tech. Conf.(USENIX ATC)*.
[4] Matthew Curry, Anthony Skjellum, H Lee Ward, and Ron Brightwell. 2011. Gibraltar: A Reed-Solomon Coding Library for Storage Applications on Programmable Graphics Processors. In *Concurrency and Computation: Practice and Experience*, Vol. 23. 2477–2495.
[5] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. 2010. Network Coding for Distributed Storage Systems. *IEEE transactions on information theory* 56, 9 (2010), 4539–4551.
[6] Dan Dobre, Paolo Viotti, and Marko Vukolić. 2014. Hybris: Robust Hybrid Cloud Storage. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.
[7] Facebook. 2010. Facebook's Erasure Coded Hadoop Distributed File System (HDFS-RAID). https://github.com/facebookarchive/hadoop-20.
[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *ACM SIGOPS operating systems review*, Vol. 37. ACM, 29–43.

[9] Google. 2012. Colossus: Successor to the Google File System (GFS). https://www.systutorials.com/3202/colossus-successor-to-google-file-system-gfs/.

[10] Apache Hadoop. 2017. Apache Hadoop 3.0.0-alpha2. http://hadoop.apache.org/docs/r3.0.0-alpha2/.

[11] Yuchong Hu, Henry CH Chen, Patrick PC Lee, and Yang Tang. 2012. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds.. In *FAST*. 21.

[12] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. 2012. Erasure Coding in Windows Azure Storage. In *Usenix Annual Technical Conference*. Boston, MA, 15–26.

[13] Intel. 2011. Introduction to Intel® Advanced Vector Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-advanced-vector-extensions.

[14] Intel. 2016. Intel Intelligent Storage Acceleration Library (Intel ISA-L). https://software.intel.com/en-us/storage/ISA-L.

[15] Intel. 2016. Using Intel® Streaming SIMD Extensions and Intel® Integrated Performance Primitives to Accelerate Algorithms. https://software.intel.com/en-us/articles/.

[16] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. 2000. Oceanstore: An Architecture for Global-Scale Persistent Storage. In *ACM SIGARCH Computer Architecture News*, Vol. 28. ACM, 190–201.

[17] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidu's Key-value Storage System for Cloud Data. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*. IEEE, 1–14.

[18] Min Li, Sudharshan S Vazhkudai, Ali R Butt, Fei Meng, Xiaosong Ma, Youngjae Kim, Christian Engelmann, and Galen Shipman. 2010. Functional Partitioning to Optimize End-to-End Performance on Many-core Architectures. In *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*. IEEE, 1–12.

[19] Runhui Li, Xiaolu Li, Patrick PC Lee, and Qun Huang. 2017. Repair Pipelining for Erasure-coded Storage. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. 567–579.

[20] Shenglong Li, Quanlu Zhang, Zhi Yang, and Yafei Dai. 2017. BCStore: Bandwidth-Efficient In-memory KV-store with Batch Coding. In *Proc. of IEEE MSST*.

[21] Jiuxing Liu, Abhinav Vishnu, and Dhabaleswar K Panda. 2004. Building Multirail InfiniBand Clusters: MPI-Level Design and Performance Evaluation. In *SC'04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE, 33–33.

[22] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. 2018. Multi-Path Transport for RDMA in Datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 357–371.

[23] Aleksei Marov and Andrey Fedorov. 2016. Optimization of RAID Erasure Coding Algorithms for Intel Xeon Phi. In *Networking, Architecture and Storage (NAS), 2016 IEEE International Conference on*. IEEE, 1–4.

[24] Mellanox. 2016. HDFS Erasure Coding Offload Plugin. https://github.com/Mellanox/EC/tree/master/HDFS.

[25] Mellanox. 2016. Understanding Erasure Coding Offload. https://community.mellanox.com/docs/DOC-2414.

[26] Mellanox. 2018. Multi-Path RDMA. https://www.openfabrics.org/downloads/Media/Monterey_2015/Tuesday/tuesday_04.pdf.

[27] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. 2016. Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 30.

[28] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. 2014. f4: Facebook's Warm BLOB Storage System. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 383–398.

[29] OpenStack. 2014. liberasurecode. https://github.com/openstack/liberasurecode.

[30] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. 2013. The Quantcast File System. *Proceedings of the VLDB Endowment* 11 (2013), 1092–1101.

[31] James S Plank. 2005. Optimizing Cauchy Reed-Solomon Codes for Fault-tolerant Storage Applications. *University of Tennessee, Tech. Rep. CS-05-569* (2005).

[32] James S. Plank, Kevin M. Greenan, and Ethan L. Miller. 2013. Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*. USENIX Association, San Jose, CA, 298–306.

[33] James S Plank, Jianqiang Luo, Catherine D Schuman, Lihao Xu, Zooko Wilcox-O'Hearn, et al. 2009. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proccedings of the 7th Conference on File and Storage Technologies (FAST '09)*. USENIX Association, Berkeley, CA, USA, 253–265. http://dl.acm.org/citation.cfm?id=1525908.1525927

[34] James S Plank, Scott Simmerman, and Catherine D Schuman. 2008. Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications. (2008).

[35] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association.

[36] KV Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B Shah, and Kannan Ramchandran. 2015. Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth.. In *FAST*. 81–94.

[37] KV Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2013. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster.. In *HotStorage*.

[38] K.V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2014. A "Hitchhiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers. *Proceedings of the 2014 ACM Conference on SIGCOMM* 44, 4 (Aug. 2014), 331–342.

[39] Irving S Reed and Gustave Solomon. 1960. Polynomial Codes Over Certain Finite Fields. *J. Soc. Indust. Appl. Math.* 8, 2 (1960), 300–304.

[40] Rodrigo Rodrigues and Barbara Liskov. 2005. High Availability in DHTs: Erasure Coding vs. Replication. In *International Workshop on Peer-to-Peer Systems*. Springer, 226–239.

[41] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment* 6, 5 (March 2013), 325–336.

[42] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment* 6, 5 (March 2013), 325–336.

[43] Dipti Shankar, Xiaoyi Lu, and D. K. Panda. 2017. High-Performance and Resilient Key-Value Store with Online Erasure Coding for Big Data Workloads. In *Proceedings of the 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*.

[44] Haiyang Shi, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K Panda. 2018. High-Performance Multi-Rail Erasure Coding Library over Modern Data Center Architectures: Early Experiences. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 530–531.

[45] Rong Shi, Sreeram Potluri, Khaled Hamidouche, Xiaoyi Lu, Karen Tomko, and Dhabaleswar K Panda. 2013. A Scalable and Portable Approach to Accelerate Hybrid HPL on Heterogeneous CPU-GPU Clusters. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–8.

[46] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 1–10.

[47] Fengguang Song, Stanimire Tomov, and Jack Dongarra. 2012. Enabling and Scaling Matrix Computations on Heterogeneous Multi-core and Multi-GPU Systems. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 365–376.

[48] Hakim Weatherspoon and John D Kubiatowicz. 2002. Erasure Coding vs. Replication: A Quantitative Comparison. In *International Workshop on Peer-to-Peer Systems*. Springer, 328–337.

[49] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 307–320.

[50] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *ACM SIGOPS Operating Systems Review*, Vol. 35. ACM, 230–243.

[51] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. 2015. A Tale of Two Erasure Codes in HDFS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 213–226. https://www.usenix.org/conference/fast15/technical-sessions/presentation/xia

[52] Heng Zhang, Mingkai Dong, and Haibo Chen. 2016. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 167–180.