Revisiting Erasure Codes: A Configuration Perspective

Runzhou Han[†] Chao Shi[†] Tabassum Mahmud[†] Zeren Yang[△] Vladislav Esaulov[‡] Lipeng Wan[‡]
Yong Chen[§] Jim Wayda[§] Matthew Wolf [§] Mai Zheng[†]
[†]*Iowa State University* [△]*University of Wisconsin-Madison* [‡]*Georgia State University* [§]*Samsung*

ABSTRACT

Erasure coding (EC) plays a crucial role in the fault tolerance of modern distributed storage systems (DSS). Inspired by recent research on storage configuration, we study the configuration sensitivity of EC in real DSS in this paper. We systematically inject faults to trigger EC recovery under various configurations, and measure the impact on recovery time and storage overhead quantitatively. Our results show that configurations may affect the EC recovery time significantly (e.g., up to 426%). More interestingly, theoretically superior codes may perform worse in DSS under certain configurations. Also, there is a system checking period before EC recovery that accounts for 41% to 58% of the overall system recovery time, which has been largely ignored in previous studies. Finally, in terms of storage overhead, EC may introduce 32.3% to 72.0% more write amplification (WA) than the theoretical expectation, and we derive a formula to help estimate WA more precisely. Our work suggests the importance of considering the context of real DSS for EC research, and we hope the methodology and findings can contribute to a firmer footing for EC optimization in practice.

1 INTRODUCTION

Erasure coding (EC) is an essential fault-tolerance mechanism widely used in modern distributed storage systems (DSS) including Ceph [7], HDFS [47], Colossus [9], DAOS [10], and many others[16, 23, 51, 57]. Compared to traditional replication, EC can achieve the same level of fault tolerance with less storage overhead, trading off encoding and decoding computations for space efficiency. Such an advantage becomes critically important as the volume of data and the scale of storage systems keeps increasing rapidly.

Due to the prime importance, great efforts have been made to improve erasure codes. For example, a variety of regenerating codes (RGC) [31, 41, 44, 46, 54] and locally repairable codes (LRC) [17, 22, 23, 27, 30, 52] have been proposed to reduce the network and/or storage I/O cost, making them (theoretically) superior to the classic Reed-Solomon (RS) code [45].

Complementary to these efforts, we focus on the configuration sensitivity of existing erasure codes in practical DSS in this paper. Recent research show that configurations may affect the performance and/or reliability of various systems significantly [5, 8, 37]. For example, sub-optimal configurations may lead to data corruptions on local storage systems [13, 37], while carefully tuned configurations may improve the performance by 9X [5]. This raises the concern on how we should optimize EC in highly-configurable DSS in practice.

As one step toward addressing the challenge, we build a framework to support analyzing the impact of various configurations on EC-based DSS thoroughly. We decouple the storage devices from a target DSS via a remote storage protocol [40] to enable flexible control of device states, and orchestrate a set of activities (e.g., system configuration, fault injection, workload execution, logging) to trigger EC operations and measure the behaviors systematically.

We apply our methodology to investigate two popular erasure codes, Reed-Solomon (RS) and Clay, in the widely used Ceph distributed storage system. Our study covers a wide range of EC-related configurations, such as various caching schemes of DSS backend, concurrency and locality of failures, placement group numbers, stripe units, and EC parameters. To the best of our knowledge, many of these factors have not been adequately addressed in previous studies. Through this invetigation, we aim to address several research questions, such as: What configurations might affect EC recovery time, and to what extent? Is EC recovery time consistently the primary bottleneck? Furthermore, apart from recovery time, do these configurations impact storage overhead or write amplification?

Our experimental results show that configurations can substantially impact EC recovery time, with variations ranging from 101% to 426%. Interestingly, despite Clay codes being generally perceived as more efficient than RS codes, they may perform worse than RS codes under certain configurations. Additionally, we observed that once a failure is detected by the DSS, there is consistently a system checking period before EC recovery, which may constitute 41% to 58%

HOTSTORAGE '24, July 8–9, 2024, Santa Clara, CA, USA

ACM ISBN 979-8-4007-0630-1/24/07.
https://doi.org/10.1145/3655038.3665951

of the overall system recovery time, depending on the workload configuration. In essence, EC recovery may not always be the primary bottleneck of the overall DSS recovery cycle.

In terms of storage overhead, our investigation reveals that EC may introduce 32.3% to 72.0% more write amplification (WA) than what is theoretically expected (i.e., $\frac{n}{k}$ for RS(n,k)). To facilitate the understanding, we have derived a formula based on our experiments to more accurately estimate the actual WA. In summary, our results underscore the significance of contextualizing real DSS environments when designing and assessing erasure codes. We anticipate that our methodology and discoveries will inspire subsequent configuration-aware optimizations for erasure codes and EC-based DSS systems at large.

The rest of the paper is organized as follows: §2 introduces the background; §3 describes the study methodology; §4 shows experimental results; §5 discusses related work; §6 concludes the paper with future work.

Table 1: Main Configurations of Ceph EC Pool

Config. Parameters	Options
Ceph storage backend	BlueStore, FileStore
BlueStore cache	meta_ratio, kv_ratio, autotune, etc.
Ceph interface	RADOS, RGW, RBD, CephFS
Num. of PGs in pool	customized, autoscale
EC plugin	Jerasure, ISA, Clay, LRC, SHEC
EC technique	reed_sol_van, cauchy_orig, etc.
EC failure domain	device (OSD), host, rack, etc.
EC device class	HDD, SSD
EC parameter	k, m (equal to n-k), d, stripe_unit, etc.

2 BACKGROUND

2.1 Erasure Coding

Erasure coding achieves fault tolerance via encoded redundancy. For example, given an RS code RS(n,k) with two parameters n and k where k<n, it splits a data object into k blocks (called data chunks/blocks), and generates n-k (or m) blocks (called parity chunks/blocks) by encoding the original k blocks. By distributing the n blocks to different failure domains, any n-k block failures can be tolerated and be re-constructed from the k surviving chunks (i.e., decoding or repair) with the storage overhead of $\frac{n}{k}$ (i.e., theoretical amplification factor). Besides the classic RS code, many other erasure codes have been proposed with different tradeoffs in terms of storage efficiency, repair bandwidth, etc [1, 11, 22, 23, 30, 41, 45, 54]. Nevertheless, how to measure their effectiveness in practical systems thoroughly is still an open challenge [41].

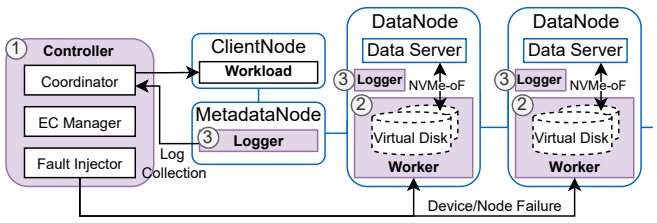


Figure 1: ECFault Overview

2.2 Distributed Storage Systems

Distributed storage systems are designed to manage data at scale [7, 9, 10, 16, 23, 47, 51, 57]. They typically consist of a set of nodes with different functionalities (e.g., metadata service, object storage). We use Ceph, one of the most widely used DSS, as a concrete example to introduce the architecture and relevant configurations that may affect erasure coding.

A basic Ceph cluster consists of one monitor/manager node (MON/MGR) and multiple OSD hosts (each may host multiple OSD devices). Objects in the storage system reside in a logical concept named *pool*. For better object management, objects in a Ceph pool are further divided into *placement groups* (PGs). PGs reside on one or more OSD devices and can be overlapped on OSDs. Ceph supports multiple EC plugins, including RS codes (via Jerasure [34] or ISA [33] libraries), Clay codes, etc. Each object in Ceph is divided into k data chunks and encoded into n-k parity chunks for EC. Table 1 summarizes the main configurations related to an erasure-coded pool in Ceph. We focus on an important subset as a starting point based on our domain knowledge, including (1) caching (e.g., `meta_ratio` of BlueStore) and object distribution (e.g., number of PGs) which may affect the system I/O path, (2) EC parameters (i.e., n and k) and basic encoding size (i.e., `stripe_unit`) which can affect EC behaviors directly (e.g., Clay's subpacketization may be sensitive to EC chunk size [43]), (3) failure modes which are crucial for triggering EC recovery operations (more details in §4).

3 METHODOLOGY

Unfortunately, none of the existing tools can meet our needs due to limitations in compatibility, dependencies, etc (See §5 for more discussion). Therefore, we design and implement a framework called *ECFault* to support a systematic study. As shown in Figure 1, given a typical DSS with three types of nodes (i.e., DataNode, MetadataNode, and ClientNode), ECFault can be integrated with the target DSS via three major components (i.e., *Controller*, *Worker*, *Logger*):

- **Controller**. This component controls the overall configuration and execution of EC experiments on the target DSS. It consists of three sub-modules: (a) *EC Manager*

manages all EC-related configurations in an experimental profile. For example, in case of Ceph experiments, the profile specifies a variety of parameters including EC plugins (e.g., Jerasure), EC parameters (e.g., k and n), basic encoding unit size (`stripe_unit`), and other relevant system features that may affect EC operations such as number of placement groups in pool (`pg_num`). (b) *Fault Injector* sends fault injection requests to DataNodes in a white-box way to explore EC operations systematically and properly. That is, the fault never goes beyond the guaranteed fault tolerance capacity (i.e., not exceeding $n-k$ failures within the defined failure domain) based on the profile defined in EC Manager. (c) *Coordinator* orchestrates all the activities in the target DSS including workloads execution, fault injection, and log collection.

- **Worker**. This component works on individual nodes of the target DSS for two purposes: (a) Virtual disk provisioning to the DSS storage service through a remote storage protocol to enable easy control of storage states; and (b) DSS manipulation, which receives and applies a variety of faults specified by the Global Controller to trigger the EC operations in the target DSS under desired workloads and configurations.
- **Logger**. This component collects various logs (e.g., I/O events and statistics, DSS failure logs, EC recovery logs) throughout the experiment cycle to facilitate fine-grained measurements and in-depth analysis of potential anomalies and bottlenecks.

Next, we elaborate on a few key implementation details:

3.1 Virtual Disk Provisioning via NVMe-oF

EC operations are closely related to the durable states of storage devices in DSS. Managing the device states efficiently is essential for evaluating erasure codes in DSS. To this end, we decouple the storage server nodes from the underlying storage devices by provisioning virtual storage devices to the target DSS through remote storage protocols [40]. Specifically, we leverage `nvmetcli` to create a set of virtual NVMe disks on each DataNode, and connect them to the DataNode operating system as local devices through NVMe-oF [40]. We provision virtual NVMe subsystems because modern DSS are increasingly optimized for NVMe SSDs for high performance. Therefore, NVMe-oF based disk provisioning enables ECFault to keep up with the technology trend of NVMe-optimized storage systems. Compared to using physical disks, hosting virtual disks makes the control and management of device states more flexible and efficient.

3.2 EC-aware Fault Injection

With the auto-provisioned disks and system information, Coordinator manipulates the DSS states based on EC configurations. Specifically, it sends fault injection requests to Workers to emulate the failure modes reported in the literature [15, 25], and thus trigger diverse EC recovery operations for measurement. The current prototype provides two fault levels: node and device. In terms of node failure, Fault Injector will send requests to Workers to shutdown specified physical or virtual machines. To emulate device failures, Fault Injector sends requests to Workers to remove the NVMe subsystems of specified virtual disks via `nvmetcli`. Moreover, the fault injection is topology-aware. For example, concurrent device failures can be either co-located on the same storage node or distributed on different nodes, enabling us to explore erasure coding recovery under different failure patterns.

3.3 System Log Collection

On each DSS server, ECFault collects both general I/O information (via `iostat` [24]) and DSS-specific logs. To reduce the network traffic of log collection, the Loggers parse the raw log files on individual nodes locally first, classify log entries based on keywords (e.g., decoding, failure, recovery, etc.), and only send the most relevant ones to the Coordinator for global sorting and merging. The log messaging between the Coordinator and Loggers is implemented via Kafka [28].

4 CASE STUDY: CEPH

We apply our methodology (§3) to analyze Ceph, a representative DSS supporting EC plugins. We focus on three research questions below:

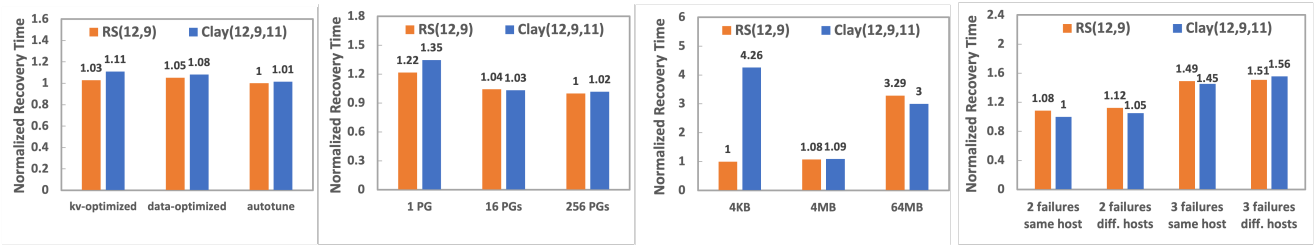
- **Q1:** What configurations can affect EC recovery time? To what extent? (§4.2)
- **Q2:** Is EC recovery time always the bottleneck? (§4.3)
- **Q3:** What is the impact on write amplification? (§4.4)

4.1 Experimental Methodology

We built a Ceph (v17.2.6 Quincy) cluster on AWS EC2 [3] for experiments, which included 31 virtual machine (VM) instances of type `m5.xlarge` with 25GB network bandwidth. Ubuntu 20.04 LTS (Linux-5.15.0-1039-aws kernel) was installed on each VM. One VM served as the MON/MGR host while the rest were OSD hosts. Each OSD host was attached with two 100GB General Purpose SSD (NVMe) volumes (6TB in total). ECFault was co-located with the Ceph cluster as shown in Figure 1. We studied two classic erasure codes including Reed-Solomon (RS) and Clay. Unless otherwise specified, we set EC to RS(12,9) and Clay(12,9,11), applied a workload of 10,000*64MB object writes (comparable to previous work [41, 54]), and measured the average recovery time of three runs.

4.2 Impact on EC Recovery Time

In this section, we first discuss the impact of three configurations (i.e., *Backend Cache*, *Placement Group*, and *Stripe Unit*)



(a) Backend Cache (b) Placement Group (c) Stripe Unit (d) Failure Mode

Figure 2: Impact of Configurations on EC Recovery Time.

under a single OSD host failure, and then present the impact of different failure modes.

Backend Cache. Ceph's BlueStore backend supports three types of cache (i.e., KV store, metadata, and data), each of which can be configured with a ratio (total 100%). Table 2 summarizes the three configurations (i.e., C1, C2, and C3) for this set of experiments.

Table 2: Three Caching Configurations.

ID	Caching Scheme	KV-ratio	Metadata-ratio	Data-ratio
C1	`kv-optimized`	70%	20%	10%
C2	`data-optimized`	20%	20%	60%
C3	`autotune` (init value)	45%	45%	10%

As shown in Figure 2a, RS(12,9) configured with `autotune` generates the best performance (i.e., lowest recovery time), which verifies that the cache resizing algorithm of BlueStore is effective. Meanwhile, Clay(12,9,11) with `kv-optimized` leads to the worst performance (i.e., 11% more time to recover compared to RS with `autotune`).

Placement Group. The number of placement groups (i.e., `pg_num`) is a critical configuration as it can affect the object distribution (and thus I/O performance) directly. We set `pg_num` to three values (i.e., 1, 16, 256) respectively. As shown in Figure 2b, Clay(12,9,11) with `pg_num=1` has the worst performance (i.e., 135% compared to RS with `pg_num=256`). We also observe that a larger `pg_num` leads to a faster recovery for both codes, which is likely because objects are distributed more evenly among OSDs with a larger `pg_num`.

Stripe Unit. This configuration (i.e., `stripe_unit`) affects the size of the basic encoding/decoding unit of erasure codes. We set `stripe_unit` to three values (i.e., 4KB, 4MB, 64MB) with `pg_num=256` in this experiment. As shown in Figure 2c, both codes are highly sensitive to `stripe_unit`. For example, RS with `stripe_unit=64MB` is 3.29 times slower than RS with `stripe_unit=4KB`. Moreover, Clay with `stripe_unit=4KB` can be 4.26 times slower than the best case. This is mainly because the subpacketization of Clay code can incur high overhead when the stripe unit is small. Note that both codes show relatively high recovery time when `stripe_unit=64MB`. This is because a larger `stripe_unit` can lead to more undersized chunks, which will be zero-padded to `stripe_unit` and generate additional I/O traffic for encoding/decoding (more details in §4.4).

Failure Mode. In this experiment, we set the failure domain to OSD and add one more SSD to each OSD host (i.e., three OSDs in total on each host) to enable more concurrent failure modes. As shown on the x-axis of Figure 2d, we compare four scenarios (i.e., two/three concurrent OSD failures on the same or different hosts). All other configurations are the same (e.g., `pg_num=256`). We can see that both codes require more time to recover when the number of concurrent failures increases, which is expected. More interestingly, the locality of three OSD failures may affect the relative performance of RS and Clay: when they occur at the same host, Clay may recover faster than RS; but when they occur at different hosts, RS may be faster. While it is known that the main advantage of Clay over RS (e.g., reduction of repair network traffic) may decrease when the failure count increases [54], we are surprised to see that the benefit may disappear with only three concurrent failures (on different hosts), which suggests the importance of considering the failure locality when measuring EC.

SUMMARY. Overall, we observe that configurations may affect the EC recovery time by up to 426%. Moreover, different from the general beliefs [54], Clay is not necessarily better than the classic RS, depending on the configurations.

4.3 Breakdown Analysis of Recovery

Figure 3 shows a fine-grained timeline of one entire system recovery cycle, which covers the period from when an OSD failure is detected ('0') to when the recovery finished ('1128s'). We divide the system recovery period into two parts: (1) *System Checking Period*, which involves massive heartbeats between MGR and other hosts, checking OSD resources, calculating surviving chunks, among others; (2) *EC Recovery*

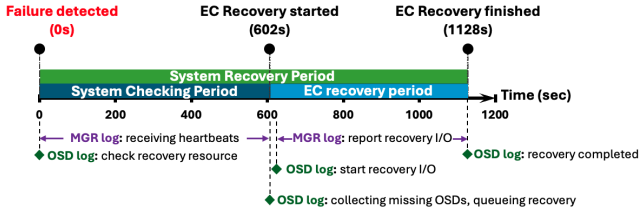


Figure 3: Timeline of System Recovery

Period, which involves the actual EC I/O operations and CPU utilization for decoding and recovery. The *System Checking Period* accounts for 53.7% of the overall system recovery time. Moreover, we find that the percentage is dependent on the workload size. We adjust the workload size to be the same as previous work [41, 54] and observe that 41% to 58% of the overall recovery time is for checking. This implies that optimizing the EC recovery period, which has been the focus of most existing efforts, might not be enough in practice.

4.4 Impact on Write Amplification

In this section, we discuss the impact of EC parameters (n, k) and `stripe_unit` on write amplification (WA), which is an undesirable phenomenon that can affect storage capacity, device lifetime, system performance, etc. negatively [29, 39]. WA can occur at different layers of the storage stack, and we focus on measuring the actual WA at the OSD level below.

Table 3: Write amplification of RS codes.

ID	Code(n,k)	$\frac{n}{k}$	Actual WA Factor	Diff. %
J1	RS(12,9)	1.33	1.76	+32.3%
J2	RS(15,12)	1.25	2.15	+72.0%

Table 3 shows the WA of two RS codes with different (n,k) but the same level of fault tolerance (i.e., 3 concurrent failures) and other configurations. We calculated the theoretical WA of each code (i.e., $\frac{n}{k}$), and measured the actual storage usage at the OSD level. We define the *Actual WA Factor* as the actual storage usage divided by the write size of the workload. We can see that the *Actual WA Factor* is always larger than $\frac{n}{k}$ (e.g., 72.0% more in case RS(15,12)), which is mainly because: (1) zero-padding on undersized data chunks; and (2) additional metadata for EC (e.g., mapping among EC chunks). This result suggests that $\frac{n}{k}$, which has been widely used for calculating EC storage overhead, may not be accurate enough to reflect the actual write amplification.

Note that the gap between the theoretical WA and the actual WA may change significantly depending on (n,k) (e.g., from 32.3% to 72.0%), which suggests the importance of considering EC parameters when estimating WA in practice. Moreover, based on our understanding of EC related configurations in Ceph, an object in an erasure coded pool will be first divided into k data chunks of size S_{object}/k, where S_{object} is the object size. If a chunk is undersized, it will be padded to `stripe_unit`. On the other hand, if a chunk is oversized, it will be further divided into $S_{object}/(k * S_{unit})$ (where S_{unit} is `stripe_unit`) encoding units, and each unit will be padded to `stripe_unit`. In other words, `stripe_unit` is another important configuration that can affect WA in practice. Based on such division-and-padding policy, we derive a formula to describe the average storage consumption for a EC chunk $\overline{S_{chunk}}$ as follows:

$$\overline{S_{chunk}} = S_{unit} * \lceil \frac{\overline{S_{object}}}{k * S_{unit}} \rceil$$

This formula captures the fact that due to the division-and-padding policy, there is always a gap between theoretical and practical WA on each EC chunk, even without considering the extra metadata overhead. The actual WA can be further estimated as follows:

$$r_{wa} = \frac{n * \overline{S_{chunk}} + S_{meta}}{\overline{S_{object}}}$$

where S_{meta} stands for the metadata size of the code stripe for the object. Note that due to the complexity of DSS metadata, the value of S_{meta} may not be readily available. But the rest of the formula can still be calculated based on the object size S_{object}, EC parameter (n, k), and `stripe_unit` (S_{unit}), which can serve as a more accurate lower bound of WA for the EC pool compared to $\frac{n}{k}$. We have validated the effectiveness of the formula for WA estimation through a set of experiments with a variety of object size, EC parameter (n, k), and `stripe_unit`, and we leave the further refinement and theoretical proof as future work.

5 RELATED WORKS

Erasure Coding. Various erasure codes have been proposed with different tradeoffs [27, 41, 42, 44, 46, 54], but unfortunately they are mostly evaluated with limited real-world configurations. In terms of EC measurement, OpenEC [32] provides a unified framework for integrating EC solutions into DSS, which is complementary to our effort.

Reliability of Distributed Storage Systems. Great efforts have been made to improve DSS reliability (e.g., [2, 4, 14, 18, 20, 26, 35, 36, 38, 50, 58–61]). Two most relevant projects are CORDS [14] and PFault [4], which used FUSE and iSCSI to inject faults to DSS respectively. While excellent for their original goals, they are largely incompatible with modern NVMe based DSS. For example, the FUSE-based CORDS cannot support the customized storage backends of DSS (e.g., BlueStore in Ceph). Moreover, they are agnostic to the various configurations that can affect erasure coding. In addition,

Amazon commercializes a Fault Injection Simulator (FIS) [12], which differs from our method in multiple ways: (1) it relies on running utility programs [49] inside the target system to simulate faults, which can change the DSS and affect the fidelity; (2) lacks of fined-grained EC configuration support and orchestration; (3) relies on other AWS services (e.g., EC2, CloudWatch). Therefore, we view them as complementary.

I/O Profiling Tools. Many tools have been proposed for measuring I/O performance [6, 19, 21, 48, 53, 55, 56]. For example, Darshan [48] is a valuable tool for characterizing I/O behavior and performance analysis in HPC systems. However, these tools cannot be used for our purpose directly due to a number of limitations including no fault injection capability to trigger EC recovery operations, unaware of EC related configurations, incompatible with customized storage backends, etc. On the other hand, they can potentially be integrated with our methodology to help collect more I/O metrics.

6 DISCUSSION AND FUTURE WORK

Inspired by recent research on system configurations, we studied the impact of various configurations on EC-based DSS and demonstrated the gaps between theory and practice. Our work suggests many opportunities for follow-up research. For example, we only studied a subset of configurations on a single DSS, which can be extended to cover more configurations and DSS to generate more comprehensive insights. Also, the quantitative analysis on configuration sensitivity could potentially help create more intelligent mechanisms for tuning EC-based DSS automatically. Additionally, we hope to develop ECFault into an open-source artifact to facilitate optimizing EC-based DSS in general.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their invaluable feedback. This work was supported in part by National Science Foundation (NSF) under grands CNS-1855565 and CNS-1943204, and a Global Research Outreach (GRO) Award (2022) from Samsung Advanced Institute of Technology (SAIT) and Samsung Research America (SRA). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] 2012. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. In *10th USENIX Conference on File and Storage Technologies (FAST)*.

[2] Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. Protocol-Aware Recovery for Consensus-Based Storage. In *16th USENIX Conference on File and Storage Technologies (FAST)*.

[3] AWS-EC2. https://aws.amazon.com/ec2/?nc2=h_ql_prod_fs_ec2.

[4] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, and Yong Chen. 2018. PFault: A general framework for analyzing the reliability of high-performance parallel file systems. In *Proceedings of the 2018 International Conference on Supercomputing (ICS)*.

[5] Zhen Cao, Geoff Kuenning, and Erez Zadok. 2020. Carver: Finding Important Parameters for Storage System Tuning. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*.

[6] Benjamin Carver, Runzhou Han, Jingyuan Zhang, Mai Zheng, and Yue Cheng. 2024. λFS: A Scalable and Elastic Distributed File System Metadata Service using Serverless Functions. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*.

[7] Ceph. https://ceph.com/en/. (accessed April 3, 2024).

[8] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. 2020. Understanding and discovering software configuration dependencies in cloud and datacenter systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[9] Colossus. https://cloud.google.com/blog/products/storage-data-transfer/a-peek-behind-colossus-googles-file-system.

[10] DAOS. https://ethereum.org/en/dao/.

[11] 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. https://doi.org/10.1109/TIT.2010.2054295

[12] AWS FIS. https://aws.amazon.com/fis/. (accessed April, 2024).

[13] Windows 10 2004/20H2: Microsoft fixes chkdsk issue in update KB4592438. https://borncity.com/win/2020/12/21/windows-10-2004-20h2-microsoft-fixes-chkdsk-issue-in-update-kb4592438/.

[14] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 149–166. https://www.usenix.org/conference/fast17/technical-sessions/presentation/ganesan

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*.

[16] GlusterFS. https://www.gluster.org.

[17] Parikshit Gopalan, Cheng Huang, Huseyin Simitci, and Sergey Yekhanin. 2012. On the Locality of Codeword Symbols. *IEEE Transactions on Information Theory* 58, 11 (2012), 6925–6934. https://doi.org/10.1109/TIT.2012.2208937

[18] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. 2011. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. https://www.usenix.org/conference/nsdi11/fate-and-destini-framework-cloud-recovery-testing

[19] Runzhou Han, Suren Byna, Houjun Tang, Bin Dong, and Mai Zheng. 2022. PROV-IO: An I/O-Centric Provenance Framework for Scientific Data on HPC Systems. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*.

[20] Runzhou Han, Om Rameshwar Gatla, Mai Zheng, Jinrui Cao, Di Zhang, Dong Dai, Yong Chen, and Jonathan Cook. 2022. A Study of Failure Recovery and Logging of High-Performance Parallel File Systems. *ACM Transactions on Storage (TOS)* (2022).

[21] Runzhou Han, Mai Zheng, Suren Byna, Houjun Tang, Bin Dong, Dong Dai, Yong Chen, Dongkyun Kim, Joseph Hassoun, and David Thorsley. 2024. PROV-IO$^+$+: A Cross-Platform Provenance Framework for Scientific Data on HPC Systems. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[22] Cheng Huang, Minghua Chen, and Jin Li. 2007. Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems. In *Sixth IEEE International Symposium on Network Computing and Applications (NCA 2007)*. 79–86. https://doi.org/10.1109/NCA.2007.37

[23] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. 2.

[24] iostat. https://linux.die.net/man/1/iostat.

[25] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. 2019. Evaluating File System Reliability on Solid State Drives. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*.

[26] Jepsen. https://jepsen.io.

[27] Saurabh Kadekodi, Shashwat Silas, David Clausen, and Arif Merchant. 2023. Practical Design Considerations for Wide Locally Recoverable Codes (LRCs). In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 1–16. https://www.usenix.org/conference/fast23/presentation/kadekodi

[28] Apache Kafka. https://kafka.apache.org.

[29] Sungjoon Koh, Jie Zhang, Miryeong Kwon, Jungyeon Yoon, David Donofrio, Nam Sung Kim, and Myoungsoo Jung. 2019. Exploring Fault-Tolerant Erasure Codes for Scalable All-Flash Array Clusters. *IEEE Transactions on Parallel and Distributed Systems* 30, 6 (2019), 1312–1330. https://doi.org/10.1109/TPDS.2018.2884722

[30] Oleg Kolosov, Gala Yadgar, Matan Liram, Itzhak Tamo, and Alexander Barg. 2018. On Fault Tolerance, Locality, and Optimality in Locally Repairable Codes. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 865–877. https://www.usenix.org/conference/atc18/presentation/kolosov

[31] Xiaolu Li, Keyun Cheng, Kaicheng Tang, Patrick P. C. Lee, Yuchong Hu, Dan Feng, Jie Li, and Ting-Yi Wu. 2023. ParaRC: Embracing Sub-Packetization for Repair Parallelization in MSR-Coded Storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 17–32. https://www.usenix.org/conference/fast23/presentation/li-xiaolu

[32] Xiaolu Li, Runhui Li, Patrick P. C. Lee, and Yuchong Hu. 2019. OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 331–344. https://www.usenix.org/conference/fast19/presentation/li

[33] Intel Intelligent Storage Acceleration Library. https://www.intel.com/content/www/us/en/developer/tools/isa-l/overview.html. (accessed April, 2024).

[34] Jerasure: Erasure Coding Library. https://jerasure.org. (accessed April, 2024).

[35] Yifei Liu, Manish Adkar, Gerard Holzmann, Geoff Kuenning, Pei Liu, Scott A. Smolka, Wei Su, and Erez Zadok. 2024. Metis: File System Model Checking via Versatile Input and State Exploration. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*.

[36] Yifei Liu, Gautam Ahuja, Geoff Kuenning, Scott Smolka, and Erez Zadok. 2023. Input and Output Coverage Needed in File System Testing. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*.

[37] Tabassum Mahmud, Om Rameshwar Gatla, Duo Zhang, Carson Love, Ryan Bumann, and Mai Zheng. 2023. ConfD: Analyzing Configuration Dependencies of File Systems for Fun and Profit. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*.

[38] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. 2020. A Study of SSD Reliability in Large Scale Enterprise Storage Deployments. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*.

[39] Jayashree Mohan, Rohan Kadekodi, and Vijay Chidambaram. 2017. Analyzing IO Amplification in Linux File Systems. *ArXiv* abs/1707.08514 (2017). https://api.semanticscholar.org/CorpusID:10285032

[40] NVMe-oF. https://nvmexpress.org/developers/nvme-of-specification/.

[41] Lluis Pamies-Juarez, Filip Blagojević, Robert Mateescu, Cyril Gyuot, Eyal En Gad, and Zvonimir Bandić. 2016. Opening the Chrysalis: On the Real Repair Performance of MSR Codes. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 81–94. https://www.usenix.org/conference/fast16/technical-sessions/presentation/pamies-juarez

[42] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn. 2009. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *7th USENIX Conference on File and Storage Technologies (FAST 09)*. https://www.usenix.org/conference/fast-09/performance-evaluation-and-examination-open-source-erasure-coding-libraries

[43] Clay Code Plugin. https://docs.ceph.com/en/quincy/rados/operations/erasure-code-clay/.

[44] K. V. 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 *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 81–94.

[45] Irving S. Reed and Gustave Solomon. 1960. Polynomial Codes Over Certain Finite Fields. *Journal of The Society for Industrial and Applied Mathematics* 8 (1960), 300–304.

[46] Nihar B. Shah, K. V. Rashmi, P. Vijay Kumar, and Kannan Ramchandran. 2012. Interference Alignment in Regenerating Codes for Distributed Storage: Necessity and Code Constructions. *IEEE Transactions on Information Theory* 58, 4 (2012), 2134–2158. https://doi.org/10.1109/TIT.2011.2178588

[47] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, et al. 2010. The Hadoop Distributed File System. In *Proceedings of 26th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*.

[48] Shane Snyder, Philip Carns, Kevin Harms, Robert Ross, Glenn K. Lockwood, and Nicholas J. Wright. 2016. Modular HPC I/O Characterization with Darshan. In *2016 5th Workshop on Extreme-Scale Programming Tools*.

[49] stress-ng. https://wiki.ubuntu.com/Kernel/Reference/stress-ng.

[50] Wei Su, Yifei Liu, Gomathi Ganesan, Gerard Holzmann, Scott Smolka, Erez Zadok, and Geoff Kuenning. 2021. Model-Checking Support for File System Development. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*.

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

[52] Itzhak Tamo and Alexander Barg. 2014. A Family of Optimal Locally Recoverable Codes. *IEEE Transactions on Information Theory* 60, 8 (2014), 4661–4676. https://doi.org/10.1109/TIT.2014.2321280

[53] A. Uselton, M. Howison, N. J. Wright, D. Skinner, N. Keen, J. Shalf, K. L. Karavanic, and L. Oliker. 2010. Parallel I/O Performance: From Events to Ensembles. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 1–11.

[54] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P. Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, Syed Hussain, and Siddhartha Nandi. 2018. Clay Codes: Moulding MDS Codes to Yield an MSR Code. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. 139–154. https://www.usenix.org/conference/fast18/presentation/vajha

[55] Jeffrey Vetter and Carsten Chambreau. 2004. mpiP: Lightweight, Scalable MPI Profiling. http://mpip.sourceforge.net.

[56] K. Vijayakumar, F. Mueller, X. Ma, and P. C. Roth. 2009. Scalable I/O Tracing and Analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*. ACM, 26–31.

[57] WekaIO. https://www.weka.io.

[58] Erci Xu, Mai Zheng, Feng Qin, Jiesheng Wu, and Yikang Xu. 2018. Understanding SSD Reliability in Large-Scale Cloud Systems. In *Proceedings of the 3rd ACM/IEEE Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems (PDSW-DISCS) at ACM/IEEE Supercomputing (SC)*.

[59] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, and Jiesheng Wu. 2019. Lessons and Actions: What We Learned from 10K SSD-Related Storage System Failures. In *Procedings of USENIX Annual Technical Conference (ATC)*.

[60] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 249–265.

[61] Di Zhang, Dong Dai, Runzhou Han, and Mai Zheng. 2021. SentiLog: Anomaly Detecting on Parallel File Systems via Log-based Sentiment Analysis. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*.