THE UNIVERSITY OF CHICAGO

MULTI-LEVEL ERASURE CODED STORAGE DESIGN AND ITS RELATIONSHIP TO DEEP LEARNING WORKLOADS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
MENG WANG

CHICAGO, ILLINOIS
DECEMBER 2024

To my parents.

"The strong pass of the obstacles is like a wall of iron,

yet with firm strides, we are conquering its summit."

— Mao Zedong

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Pursuing a PhD is a long and challenging journey, and I could not have reached this point without the help of many people. I am deeply grateful to everyone who has supported me along the way.

First and foremost, I would like to thank my advisor, Haryadi Gunawi, who introduced me to the world of systems research and graciously accepted me as his PhD student. I still remember the afternoon when I was working on the Linux kernel, and Haryadi called me to his office, offering me the opportunity to begin my PhD in the upcoming autumn. This chance to start my PhD has been invaluable, and I truly appreciate it. The PhD journey is not an easy one, and I know I made my share of mistakes along the way. I am grateful to Haryadi for his patience, for constantly guiding me on how to improve, and for teaching me many essential skills. He was patient during my unproductive days, explored new areas alongside me, connected me with industry experts and professors, advocated for my funding, and much more. I sincerely thank him for everything.

I would also like to thank the rest of my dissertation committee. John Bent introduced me to erasure coding, shared deep insights into storage from a real-world industry perspective, connected me with experts from the HPC world, and has always been supportive and encouraging. John was also my manager during two internships at Seagate, where he was a fantastic mentor. Junchen Jiang, who taught my first PhD course on modern network systems, kindly served on my master's exam, candidacy, and dissertation committees, for which I am deeply appreciative. I also extend my gratitude to Anjus George, who collaborated with me on failure analysis and multi-level erasure coding, always offering support, resources, and availability.

Special thanks go to Mingzhe Hao, my mentor when I joined the group as a master student. Mingzhe taught me not only about systems research, experimental techniques, system hacking, and storage knowledge but also about the importance of continuous self-improvement and mentoring with kindness and patience. I am fortunate to have had him as a mentor, and he remains a role model to this day. My only regret is not fully leveraging his guidance while he was here, as I was shy and introverted. Thankfully, I'll soon have the opportunity to work with him again in industry,

where he will be my manager, and I look forward to learning even more from him in this new phase.

I also want to acknowledge my collaborators: Serkay Olmez, who taught me mathematical modeling; Garrett Wilson Ransom and Gary Grider, who shared insights into large-scale data center deployment in the HPC world; Jun Li, who advised me on erasure coding research; Vijay Chidambaram, who offered valuable guidance on storage for deep learning systems and my GPU emulator; and Swaminathan Sundararaman, who shared his knowledge on the challenges and considerations of deep learning systems in industry.

I also want to acknowledge my mentors during my industry internships: John Bent, who was an exceptional manager at Seagate; Norbert Egi, who taught me a great deal during my internship at Futurewei; and Qiaobin Fu, who was incredibly supportive during my internship at Google.

My heartfelt thanks go to the UCARE members I worked with, including Cesar Stuardo, Daniar Kurniawan, Huan Ke, Huaicheng Li, Jeffrey Lukman, Maharani Irawan, Martin Putra, Mingzhe Hao, Ray Andrew, Riza Suminto, Roy Huang, and Ruidan Li. You all made UCARE feel like a warm and supportive family.

I would also like to thank my friends, Chengpeng Xue, Guangyu Xue, Lei Chen, Neng Huang, Renyu Zhang, Xiao Xu, Xu Zhang, Yuhao Zhou, Zhuokai Zhao and many others. I owe special gratitude to Lei Chen, who has been incredibly supportive during my difficult times. Lei, you are always my hero.

Finally, I want to thank my parents. They have always supported my decision to pursue a PhD, and it was their encouragement that pushed me toward this path when I was still uncertain about my career. They worked hard to support me both financially and emotionally.

ABSTRACT

Large-scale data centers store vast amounts of user data across numerous disks, necessitating redundancy mechanisms like erasure coding (EC) to protect against disk failures. As storage systems scale in size, complexity, and layering, disk failure frequency and rebuild times increase. For managing tens or hundreds of thousands of disks, traditional single-level erasure coding (SLEC) does not scale well, as it struggles to balance repair overhead with rack- and enclosure-level failure tolerance. Multi-level erasure coding (MLEC), which applies EC at both network and local levels, has been deployed in large-scale systems. However, no in-depth study has addressed its design considerations at scale, leaving many research questions unaddressed. This dissertation provides a comprehensive analysis of MLEC at scale, focusing on its design considerations and relationship to deep learning (DL) workloads.

We begin by presenting a detailed analysis of MLEC's design space across multiple dimensions, including code parameter selection, chunk placement schemes, and repair methods. We quantify their performance and durability, identifying which MLEC schemes and repair methods best tolerate independent and correlated failures and reduce repair network traffic by orders of magnitude. Evaluation methods include simulation, splitting, dynamic programming, and mathematical modeling. We also compare MLEC's performance and durability with other EC schemes like SLEC and LRC, showing that MLEC can provide high durability with higher encoding throughput and less repair network traffic over both SLEC and LRC.

We then discuss the relationship between MLEC and DL workloads. As DL workloads become increasingly data-intensive, training datasets often exceed local storage, requiring access to remote erasure-coded storage. To cost-effectively evaluate MLEC's ability to meet the throughput demands of DL workloads, we develop an emulation-based approach. We introduce GPEmu, a GPU emulator designed for efficient evaluation of DL systems without physical GPUs. GPEmu supports over 30 DL models and 6 GPU models, providing capabilities for time emulation, memory emulation, distributed system support, and GPU sharing. We also develop MLECEmu, which simulates

the read throughput of erasure-coded disk arrays with I/O-throttled in-memory file systems. Using these tools, our end-to-end experiments show that MLEC storage can enhance GPU utilization with wider stripes, and our optimized MLEC repair methods reduce training performance degradation during catastrophic local failure repairs.

While MLEC storage provides high aggregated intra-cluster read throughput for DL workloads, the network bandwidth between the GPU cluster and the MLEC storage cluster can become a bottleneck during training, as inter-cluster bandwidth is typically more constrained than intra-cluster bandwidth. Since many samples significantly reduce in size during preprocessing, we explore selective offloading of preprocessing tasks to remote MLEC storage to mitigate data traffic. Our case study evaluates this approach's potential benefits and challenges. Based on our findings, we propose SOPHON, a framework that selectively offloads preprocessing tasks at a fine granularity to reduce data traffic. SOPHON uses online profiling and adaptive algorithms to optimize preprocessing for each sample in each training scenario. Evaluations using GPEmu and MLECEmu show that SOPHON reduces data traffic and training time by 1.2x to 2.2x compared to existing solutions.

CHAPTER 1

INTRODUCTION

Large-scale data centers store a huge amount of user data in a massive number of disks and require redundancy approaches such as erasure coding (EC) to protect them from disk failures [62, 63, 64, 73, 130, 131, 144, 157]. The sheer size, scale, complexity, and layering of distributed mass-capacity storage keep growing and have never stopped. Figure 1.1 shows that the number of disks managed in Backblaze and US DOE laboratories keeps increasing and the per-disk capacity also keeps growing for both max available capacity and average capacity of sold disks. Such extreme scales lead to more frequent disk failures and longer time to rebuild a failed disk.

For managing tens or hundreds of thousands of disks, the classical single-level erasure coding (SLEC) no longer scales and cannot provide a good balance between minimizing repair overhead and maximizing rack/enclosure-level failure tolerance. Multi-level erasure coding (MLEC), which performs EC at *both* network and local levels, becomes a popular choice for several reasons. **(a)** MLEC is a hybrid of the network- and local-level SLEC schemes, hence gaining the benefits of the two worlds. Local SLEC only performs EC inside a rack/enclosure and thus cannot tolerate rack/enclosure failures [100, 120, 164]. Network SLEC tolerates rack failures but requires cross-rack network traffic for repairing lost chunks [124, 125, 158]. MLEC, on the other hand, can repair most disk failures locally without interfering user network traffic while at the same time being able to tolerate rack failures. **(b)** MLEC's performance scales better. With tens of thousands of disks, deploying more parity chunks and wider stripes to achieve higher durability will lead to higher encoding computation overhead. However, the 2-level nature of MLEC can provide high durability with less encoding overhead than SLEC. **(c)** MLEC is stackable and easy to deploy/scale in practice. Since storage vendors sell RBODs (reliable bunch of disks) with EC controllers inside, larger-scale customers can build network-level EC on top of the local RBODs. **(d)** MLEC is more configurable. Customers can choose how many parities and what kind of chunk placement scheme to use at each level that fit their goals and constraints.

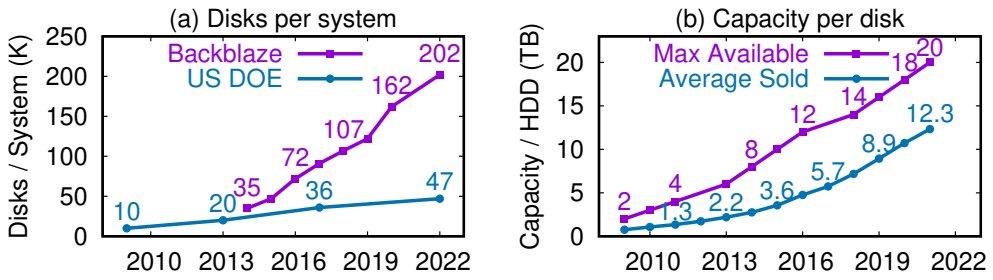


Figure 1.1: **Storage scaling over the years.** *The figures show (a) the increasing number of disks managed in Backblaze and US DOE laboratories and (b) per-disk capacity over the last 15 years*

MLEC has seen large deployments in the field, including in HPC data centers in national laboratories [90], enterprise-grade storage softwares [42], and commercial storage systems [1]. However, based on literature study and personal communications, there is no in-depth study of design considerations for MLEC at scale. Many research questions remain unanswered. What are the possible chunk placement schemes for MLEC at scale? What are their pros/cons in terms of performance and durability? What are the types of failure modes an MLEC system can face? Can we introduce advanced repair methods that are optimized for every specific scheme and failure mode? What are the implementation requirements for advanced repairs? Though other works analyze hierarchical RAID for small-scale systems [55, 122, 141, 142, 152], we have not seen any work answering the questions above or studying design considerations of MLEC for large-scale systems.

Furthermore, with the rapid advancement of deep learning (DL), DL training has become one of the primary workloads in modern cloud systems, requiring vast amounts of training data. This data, often too large to be stored locally, is typically housed in remote data centers and accessed during training [66, 103, 148, 162, 167]. To ensure data reliability, these data centers employ erasure coding. This raises a critical question in the context of MLEC: How does multi-level erasure-coded storage perform when serving DL workloads? Answering this question involves evaluating the performance of MLEC in the context of DL workloads. However, DL training requires costly GPUs, and MLEC storage relies on extensive disk arrays—both of which are limited resources in academic clouds. Is there a cost-effective approach to evaluate MLEC storage for DL workloads, enabling us to identify performance bottlenecks inexpensively and quickly prototype

optimizations?

This dissertation addresses these key research challenges surrounding MLEC. Specifically, **this work provides the most comprehensive design considerations and analysis of MLEC for large-scale data centers, and introduces an emulation-based approach for evaluating the performance of MLEC storage under deep learning workloads, without relying on physical GPUs or disk arrays. This approach enables fast and cost-effective identification of system bottlenecks and optimization strategies.**

In the rest of this chapter, we introduce our comprehensive design considerations and analysis of MLEC (Section 1.1), present our emulation-based approach for cost-effective evaluation of MLEC storage performance against deep learning workloads (Section 1.2), and explain how the emulators can be used to evaluate a novel selective preprocessing offloading approach that reduces network traffic between deep learning training workloads and remote MLEC storage (Section 1.3).

1.1 Design Considerations and Analysis of MLEC at Scale

There are many design choices in various dimensions (including code parameter selections, chunk placement schemes, and repair methods) when configuring MLEC for large-scale data centers. Different design choices offer various trade-offs between data durability, implementation complexity, and system performance. Configuring or extending an MLEC design at extreme scale can be costly without a clear understanding of the implications of proposed changes.

Therefore, we first provide, to the best of our knowledge, the most comprehensive design considerations and analysis of MLEC at scale [149], covering diverse design choices, offering multiple evaluation methods, considering various failure scenarios, and analyzing numerous trade-offs.

We introduce the design space of MLEC across multiple dimensions, including code parameter selections, chunk placement schemes, and repair methods—ranging from simple and practical approaches to more optimized ones that leverage multi-level EC but require cross-level transparency.

To evaluate different design choices, we employ a variety of strategies, including simulation, splitting, dynamic programming, and mathematical modeling. We build MLECSim, to the best of our knowledge, the first sophisticated MLEC simulator (approximately 13,000 lines of code (LOC)) capable of measuring MLEC performance and durability at scale (over 50,000 disks). Key features include simulating disk failures (based on distributions, rules, or real traces), supporting multi-level clustered/declustered placements, expressing failure tolerance, and executing complex repair strategies.

Based on these evaluation strategies, we quantify the performance (e.g., encoding throughput, repair network traffic, repair time) and durability (under both independent and correlated failures) of MLEC under various design choices. We demonstrate which MLEC schemes and repair methods offer the best tolerance against independent/correlated failures and reduce repair network traffic by orders of magnitude.

We also compare the performance and durability of MLEC with other EC schemes, such as SLEC and LRC [89], and show that MLEC can provide high durability with higher encoding throughput and less repair network traffic over both SLEC and LRC.

We hope that our analysis—along with the released simulation code [27] and evaluation artifact [26]—will enable engineers and operators of extreme-scale EC systems to gain a comprehensive understanding of the advantages and disadvantages of various MLEC schemes.

1.2 Cost-Effective Evaluation of MLEC Storage Against DL Workloads

With the rapid growth of deep learning, data centers increasingly store large datasets used for training models. These datasets, often scaling to tens or even hundreds of terabytes [33, 167], are typically too large for local storage and must be housed in remote data centers and ingested during training [66, 103, 148, 162, 167]. Data centers need to provide enough data throughput to feed the deep learning workloads; otherwise, data transfer can become a bottleneck and slow down training.

Meanwhile, to ensure data reliability, data centers commonly use erasure coding [69, 89, 128].

Therefore, in the context of MLEC storage, it is essential to evaluate its performance against deep learning workloads, identify bottlenecks, and explore potential optimization opportunities.

However, evaluating MLEC in deep learning workloads poses significant challenges. Deep learning training requires expensive GPUs, which are often inaccessible to the research community due to their high cost. Additionally, MLEC storage requires large, self-configurable disk arrays that are difficult to obtain from academic cloud providers. Even for industry engineers, accessing these resources for evaluation is challenging, as they are costly and typically reserved for real-world workloads.

To address this, we develop an emulation-based approach that enables cost-effective evaluation of MLEC storage in deep learning workloads using an end-to-end setup. Specifically, we introduce GPEmu, a GPU emulator that allows for the cheap evaluation of deep learning workloads without the need for real GPUs, and MLECEmu, an emulator that simulates the MLEC storage system without requiring physical disk arrays.

Our goal with GPEMU and MLECEmu is to provide a cost-effective way to identify system bottlenecks in deep learning and MLEC storage through emulation, offering valuable insights into system performance and enabling the prototyping of new optimizations.

1.2.1 GPEmu: A GPU Emulator for Cheaper Evaluation of DL System Research

We first introduce GPEmu, a GPU emulator designed for faster and more cost-effective prototyping and evaluation of deep learning systems research. GPEmu is motivated by the insight that, for many deep learning research projects [54, 104, 105, 112, 113, 113, 145, 162, 163, 173], real, physical GPUs are *not required*. For example, when evaluating and optimizing MLEC storage for deep learning training workloads, the focus is often on increasing GPU utilization by improving layers above the GPU, such as data loading and preprocessing. In such cases, rather than the *results* of GPU computations, the key factor is the *performance* of the GPU.

We argue that the research community needs a GPU emulator capable of replicating GPU behavior without the need for physical GPUs. Such an emulator would significantly enhance prototyping efficiency and reduce research costs. While there are some existing emulators, such as MLPerf Storage [28] or ad-hoc emulators developed for specific research [95, 167], they lack essential components like memory emulation and data preprocessing support. Furthermore, they only accommodate a limited number of configurations and domains.

We present GPEMU, a GPU emulator designed for faster and cheaper prototyping and evaluation of deep learning systems research. The design of GPEMU is versatile, supporting over 30 deep learning models and six different GPU configurations. GPEMU is easy to use: users can run typical deep learning frameworks (*e.g.*, PyTorch, TensorFlow) on top of GPEMU in both single-node and distributed setups (such as Kubernetes). GPEMU focuses on deep learning training workloads.

We introduce four key features in GPEmu: **(1)** time emulation, using a sleep-based approach to emulate GPU computations, host-to-GPU data transfers, and GPU-driven data preprocessing; **(2)** memory emulation, which replicates both GPU memory usage and pinned memory; **(3)** distributed system support, enabling multi-GPU single-node training, multi-node training, and multi-job scheduling; and **(4)** GPU sharing support, which emulates time-sharing for both single-node and Kubernetes setups.

To demonstrate the benefits of GPEMU, we leverage GPEMU to reproduce the main experimental results from several papers [54, 104, 105, 112, 113, 113, 145, 162, 163, 173] across diverse setups and levels of complexity. Additionally, we show how GPEMU can be used to introduce and evaluate a set of new micro-optimizations.

1.2.2 MLECEmu: An Emulator for Cheaper Evaluation of MLEC Storage

In addition to GPEmu, we introduce MLECEmu, an emulator designed for the fast evaluation of multi-level erasure coded storage systems. MLECEmu is motivated by the challenge that configuring MLEC storage typically requires a large number of disks, which academic cloud environments

do not usually provide. For instance, typical academic clouds, such as Chameleon Cloud [10, 97], often offer only one disk per machine. As a result, configuring even a simple MLEC system with 50 disks would require 50 machines, making it prohibitively expensive and inefficient.

To address this limitation, we present MLECEmu, an emulator for fast evaluation of MLEC storage. MLECEmu configures an emulated MLEC storage system using real HDFS [133] on top of ZFS [126]. Instead of relying on physical disks, MLECEmu uses multiple `tmpfs` devices in memory with throttled throughput to simulate the performance characteristics of real disks. It also emulates reduced throughput during disk failure repairs by limiting the emulated disk and rack bandwidth, with repair duration projected using MLECSim. Through these emulations, MLECEmu enables the evaluation of a configured MLEC storage system for both normal operations and degraded states during disk failure repairs.

By combining GPEmu and MLECEmu, we can evaluate MLEC storage performance for deep learning workloads without the need for physical GPUs or disk arrays. Using these tools, we demonstrate that MLEC storage can enhance GPU utilization in deep learning training through the use of wider stripes. Additionally, we show that our optimized MLEC repair methods can reduce the duration of trainin performance degradation in DL training caused by repairs under catastrophic local failures.

1.3 Reducing Cross-Cluster Data Traffic Between DL Training and Remote MLEC Storage

While MLEC storage can provide abundant aggregated intra-cluster bandwidth serving deep learning workloads when with wide stripes thanks to distributed reading from multiple disks and racks, the inter-cluster bandwidth is usually more constrained than intra-cluster bandwidth [48, 80]. Therefore, the cross-cluster network bandwidth can become a critical bottleneck for DL workloads and limiting GPU utilization [66, 103, 148, 167]

Several strategies have been proposed to address data fetch bottlenecks, with most approaches

focusing on selectively caching data in local storage or memory [66, 103, 105, 113, 116, 162, 167]. However, these methods are limited by local capacity, especially as datasets grow. Other approaches store preprocessed datasets in remote storage for reuse across epochs [91, 93, 106], but this can reduce training accuracy by omitting random transformations during preprocessing, which are essential for learning.

Previous work [54, 145, 169, 170, 171] has explored offloading preprocessing to reduce CPU bottlenecks by shifting tasks to extra CPU nodes, without addressing data fetch traffic. These methods also uniformly offload preprocessing, ignoring that many samples shrink significantly during intermediate stages.

We utilize preprocessing offloading differently: by observing that many data samples undergo significant size reductions at intermediate stages of the preprocessing pipeline, we propose to **selectively offload** parts of preprocessing for certain samples to the remote storage server. This approach leverages the higher intra-cluster bandwidth compared to inter-cluster bandwidth [48, 80, 88, 149]. By preprocessing larger samples within the storage cluster and transmitting smaller, partially processed data, our method reduces data traffic and improves training efficiency.

We introduce SOPHON (Selectively Offloading Preprocessing with Hybrid Operations Near-storage), a framework designed to selectively offload DL preprocessing tasks to remote storage servers with the goal of reducing data transfer traffic [151]. SOPHON has two key components: (1) A two-stage profiler that collects essential metrics for making offloading decisions. Offloading is activated only when the workload is identified as I/O-bound during profiling. (2) A decision engine that determines which samples to offload and identifies the specific operations to offload for each chosen sample, striking a balance between reducing traffic and managing CPU overhead. Together, these components enable SOPHON to provide tailored offloading strategies to meet the unique needs and constraints of each training scenario.

We evaluated SOPHON with deep learning training workloads which reads data from remote MLEC storage, using GPEMU and MLECEmu introduced Section 1.2. Our evaluation results

demonstrate that SOPHON can effectively enhance training efficiency, achieving a 1.2-2.2x reduction in training time over existing solutions.

1.4 Thesis Organization

The rest of this dissertation is organized as follows:

- In Chapter 2, we introduce the background of redundancy techniques used to ensure data center reliability and provide an overview of existing erasure coding studies and their limitations. We also discuss existing emulation and simulation techniques for cost-effective evaluation of system research, as well as the relationship between deep learning (DL) training and storage systems, motivating our study of evaluating MLEC storage under DL workloads.

- In Chapter 3, we present a comprehensive analysis and design consideration of multi-level erasure coding (MLEC) for large-scale data centers. We explore various code parameters, chunk placement schemes, and repair methods, analyzing the trade-offs between performance and durability and comparing MLEC with other EC schemes.

- In Chapter 4, we introduce an emulation-based approach to evaluate MLEC storage under DL workloads. We describe GPEMU, a GPU emulator for evaluating DL workloads without using real GPUs, and MLECEmu, an MLEC storage emulator that emulates MLEC storage without requiring extensive real disk arrays. We also show how these tools can be used together to evaluate MLEC storage under DL workloads and identify system bottlenecks.

- In Chapter 5, we introduce SOPHON, which uses selective preprocessing offloading to reduce cross-cluster data traffic between DL training and remote MLEC storage, evaluated using GPEMU and MLECEmu.

- In Chapter 6, we briefly mention our other work on availability and reliability of storage systems.

- In Chapter 7, we conclude the dissertation and discuss potential future directions for MLEC storage and its relationship to DL workloads.

CHAPTER 2

BACKGROUND AND MOTIVATIONS

In this chapter, we briefly introduce the background of redundancy techniques used to ensure data center reliability and provide an overview of existing erasure coding techniques in Section 2.1. Additionally, we discuss the remote I/O bottleneck problem in deep learning in Section 2.2, which motivates our study of evaluating MLEC storage under DL workloads. In Section 2.3, we review existing emulation and simulation methods for cost-effective system evaluation and highlight their limitations, motivating our development of GPEMU and MLECEmu. Finally, we introduce preprocessing offloading in Section 2.4, which is central to our optimization for reducing traffic between MLEC storage and DL workloads.

2.1 Existing Redundancy Approaches

2.1.1 Replication

The most straightforward way to protect data durability is replication [133], which makes multiple copies of the same data and distributes them across different disks. When a disk fails, its data can be recovered by reading from the copies in other disks. However, the price is the high storage overhead. For example, the commonly used 3-way replication requires 3x storage space as the original data.

2.1.2 RAID and Erasure Coding

Since replication introduces high storage overhead, nowadays many systems use parities to protect their data [120]. For example, the conventional RAID5 [38] systems use one parity per stripe and can tolerate any single device failure, and RAID6 [39] systems can tolerate up to 2 device failures by using 2 parities. In order to be more flexible and tolerate additional failures, many storage systems have started to use *erasure coding* (EC) [155].

In EC systems, a stripe of data is split into k *data chunks*, based on which p *parity chunks* are computed. These $(k+p)$ chunks are distributed into different storage devices. When one device fails, the lost chunks can be reconstructed from computation on the surviving chunks. Any $(k+p)$ erasure can tolerate up to p device failures without data loss. The conventional RAID5 systems utilizes $(k+1)$ erasure and RAID6 systems uses $(k+2)$ erasure.

2.1.3 Chunk Placement: Clustered vs. Declustered EC

When distributing chunks into storage devices, there are two typical kinds of data placement schemes: clustered and declustered.

In order to better illustrate the difference between clustered and declustered erasure, let's start with an example. Let's assume we want to do $(9+1)$ erasure among 100 drives.

In clustered erasure, we divide the 100 drives into 10 groups, each containing $9+1=10$ drives. Every stripe is assigned to a specific group, and the $(9+1)$ chunks are distributed among the 10 drives. A stripe either has no chunk in a group, or has all the chunks residing in the group. When a drive fails, we read from 9 surviving drives to reconstruct the lost data and write to a new spare drive. Only 10 drives participate in the rebuild, and the rebuild rate is bottlenecked by the single drive's read/write rate.

When using declustered erasure [12, 53, 86], every stripe is distributed across 10 pseudorandom drives. Each drive has some spare space where no data nor parity is stored. When a drive fails, it has a huge number of lost chunks on it. For each lost chunk, 9 pseudorandom surviving drives are read to reconstruct it, and the reconstructed chunk is written to the spare space on one pseudorandom drive. Since there are a huge number of lost chunks, the total build work is spread across all 99 surviving drives, and therefore the rebuild rate is much faster than that of clustered erasure.

The price for the faster rebuild is the less tolerance against concurrent failures. For example, if drive 1 and drive 99 fail simultaneously, in clustered erasure, the system is recoverable since the two failed drives belong to two different drive groups. In declustered erasure, however, due

to its pseudorandom placement scheme, some stripes have chunks on both failed drives, and these 2-chunk-failure stripes can no longer be restored using $(9+1)$ erasure.

To address the issue of low burst tolerance in conventional declustered parity placement, recent research has introduced a placement method known as Single-Overlap Declustered Parity ("SODp") [96]. Unlike conventional Dp's random chunk placement, SODp organizes deterministic declustered "stripesets," and each incoming stripe can only be mapped to one of these stripesets. A stripeset refers to a specific set of disks chosen for a one-to-one correspondence with a stripe.

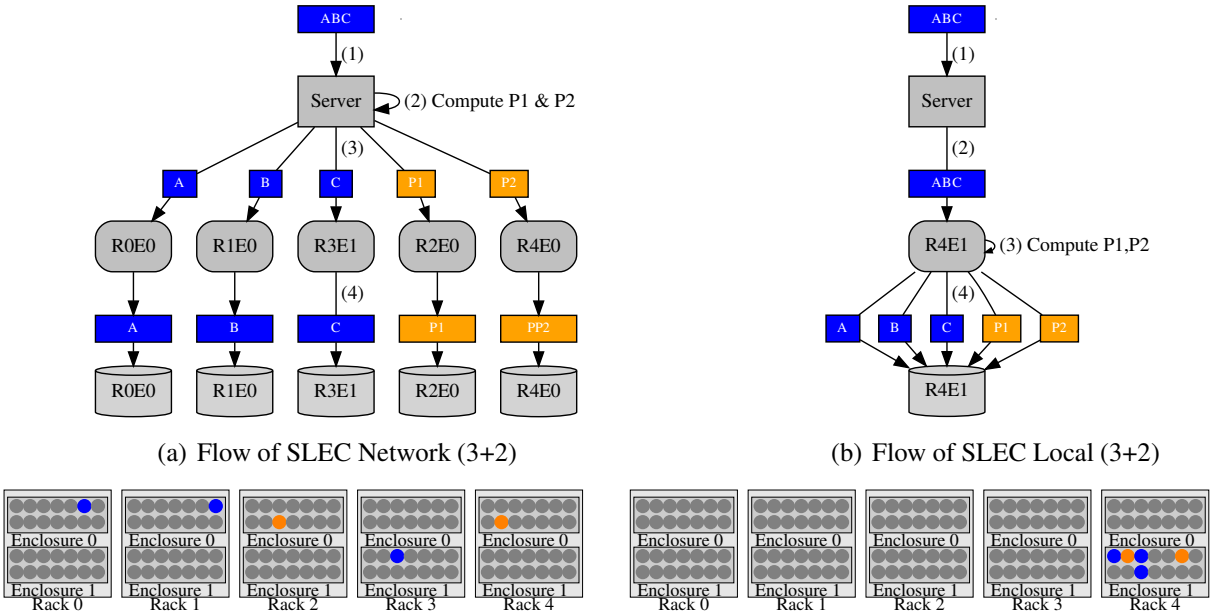
2.1.4 Local vs. Network SLEC

Many distributed storage systems today use declustered erasure and are built with similar hardware. Clients send and receive data from servers which, in turn, send and receive data between themselves and distributed storage *enclosures*. Storage enclosures are typically between four and five rack units high and contain around 100 drives. These enclosures contain *controllers* which either act merely as a path to the individual devices or add a degree of virtualization.

There are two existing single-level erasure coding (SLEC) approaches: *network-only SLEC* and local-only SLEC. To describe them, we imagine a data center that has five racks of storage, each holding two enclosures, each holding fourteen storage devices.

As exemplified in Figure 2.1(a), a *network-only SLEC* server splits received data into three data chunks, computes two parity chunks, and then distributes those chunks across enclosures by sending them to the controllers, which forward the data to the requested devices [49, 72, 129, 133]. Because racks and enclosures are well-known failure domains, typical network-only systems ensure a layout such that no stripe has more than p chunks per rack or enclosure.

In Figure 2.1(c), we show a typical layout in which each chunk is stored in a different rack and no one rack has more than one chunk. In this way, the system minimizes the number of chunks lost in any one stripe when a rack fails. Note that we assume that all approaches use declustered erasure such that the placement shown in the figures is only for one particular stripe of data. Other



(a) Flow of SLEC Network (3+2)

(b) Flow of SLEC Local (3+2)

(c) Resulting Layout of SLEC Network (3+2)

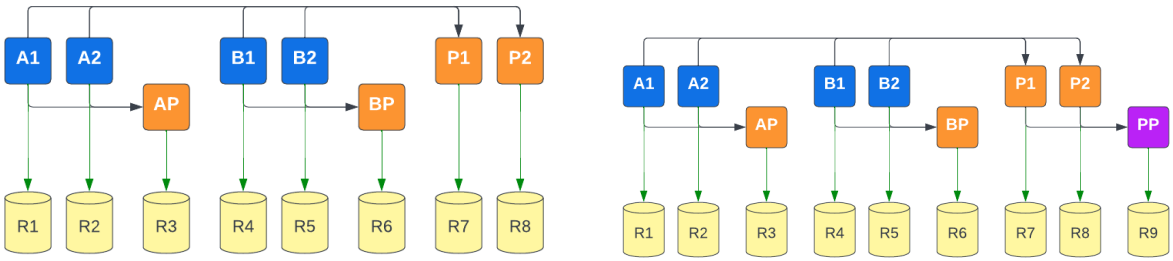
(d) Resulting Layout of SLEC Local (3+2)

Figure 2.1: **Logical Erasure Workflows and Physical Layouts of SLEC.** *The upper figures show the logical flow and parity generation when new data is stored. Rounded rectangles represent the controllers within each enclosure, and cylinders represent their drives. The lower figures show the resulting physical layout in an exemplary data center. All figures use colors to indicate the type of data being stored: blue for the original user data, and orange for the parity computed from that data. For the purpose of elucidation, not all elements are shown in each figure. For example, servers are not shown in the physical layouts, and not all enclosures are shown in the logical flows. Each is configured such that all have the same capacity overhead (40%).*

stripes will have different layouts (i.e. will be stored on different devices). The enclosures used in network-only architectures are often referred to as *JBOD* (just a bunch of disks).

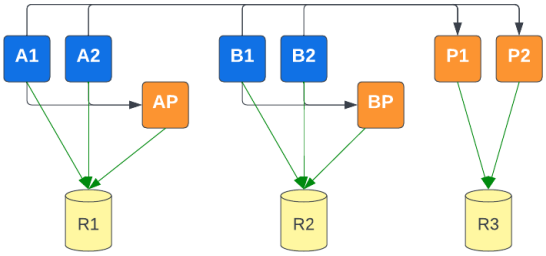
The second SLEC approach, *local-only SLEC* uses *RBOD* (reliable bunch of disks) instead of JBOD [59, 84, 114]. An RBOD, which is typically created using erasure firmware running on the controller, appears to the upper-level system as a very large drive. However, internally it is protecting data using erasure. As shown in Figures 2.1(b) and 2.1(d), a client sends data to a server, and that server passes the data directly to an RBOD controller, which then splits the data, computes the parity, and stores the five resulting chunks onto five of its drives.

Note that the work performed is fundamentally the same in both SLEC approaches with only two small differences. One is whether a server or a controller does the splitting, computing, and



(a) Azure-LRC (8,4,2)

(b) Optimal LRC (9,4,2)



(c) Combined Locality (8,4,2,3)

Figure 2.2: **Locally repairable coding (LRC).** *The figures illustrate the workflows of different LRC approaches including (a) Azure-LRC (b) optimal-LRC (c) Combined Locality.*

distributing, and the second is whether the distribution is done across enclosures or within a single one.

Also note that local-only SLEC does not protect data from rack or enclosure failures. Network-only SLEC does tolerate rack failures, but the repair introduces heavy network traffic.

2.1.5 Locally Repairable Coding

In $(k+p)$ erasure coding, repairing any lost chunk requires reading k surviving chunks from other disks. When k is large, the repair becomes IO-intensive and computation-intensive. Locally Repairable Coding (LRC) deals with this problem by adding local parities.

For example, in (n,k,r) Azure-LRC [89], k data chunks are divided into $\lceil k/r \rceil$ local groups. Each local group contains r data chunks, and one local parity is computed. Meanwhile, it encodes $n-k-\lceil k/r \rceil$ global parity chunks using all data chunks. This results in n chunks in total per stripe. When a data chunk or local parity is lost, it can be reconstructed by reading only r surviving chunks

in the same local group. Figure 2.2(a) shows an example of LRC $(8,4,2)$, which has 4 data chunks, 2 local parities, and 2 global parities.

Note that the repair of global parities in Azure LRC still requires reading k surviving chunks. Such LRCs are referred to as *data-LRCs* [101]. On the other hand, *full-LRCs* can repair any chunk (including the global parity chunk) using r surviving chunks.

Optimal-LRC [140] is a typical full-LRC, where all data chunks and global parity chunks are divided into local groups of size r, and each local group has one local parity. For example, Figure 2.2(b) shows optimal-LRC (9,4,2).

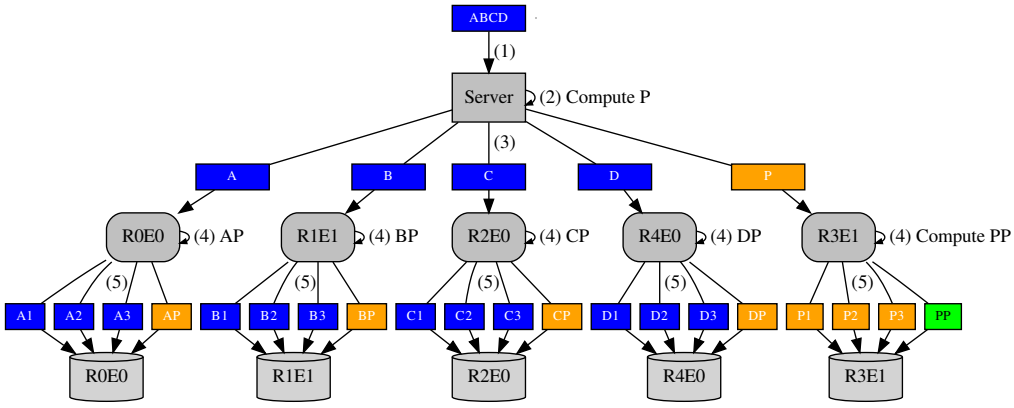
Note that Azure-LRC distributes each chunk of a stripe into a different rack. By doing this, it can tolerate rack failures. However, the repair of any lost chunk will introduce cross-rack network traffic.

Azure-LRC+1 [101], in the evaluation, first tried to place a local group in a rack to reduce cross-rack repair network traffic.

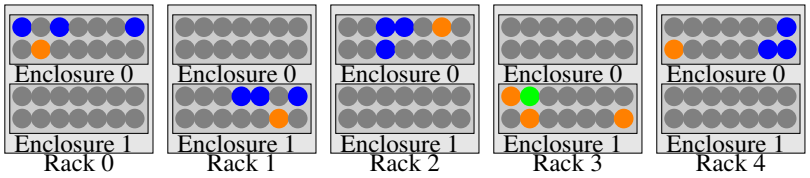
Combined locality (CL) [88] first systematically explores how to deploy LRC in a rack-hierarchical structure in order to reduce network traffic. In CL (n,k,r,z), they added a new parameter z to denote how many stripes to place a stripe. For example, Figure 2.2(c) shows CL (8,4,2,3) which distributes the 8 chunks into 3 racks. In this example, repairing data chunks or local parities requires 0 network traffic. Repairing the global parity chunk, however, requires 4 chunks of cross-rack network traffic cost.

2.1.6 MLEC and Hierarchical RAID

Multi-level erasure coding (MLEC) architecture, shown in Figure 2.3(a), combines two single-level erasure coding (SLEC) approaches. In MLEC, a server receiving data splits the data into chunks, computes parity chunks, and distributes them across enclosures in a rack-aware layout. Each enclosure, an RBOD, further splits the data it receives, computes additional parity, and stores these *subchunks* across its internal drives.



(a) Flow of MLEC (4+1)/(3+1)



(b) Resulting Layout of MLEC (4+1)/(3+1)

Figure 2.3: **MLEC Architecture.** *The upper figures show the logical flow and parity generation when new data is stored. The lower figures show the resulting physical layout in an exemplary data center.*

MLEC has seen large-scale deployments in HPC data centers at national laboratories [90], enterprise-grade storage software [42], and commercial storage systems [1]. However, an in-depth study of design considerations for MLEC at scale has not yet been conducted. While previous works analyze hierarchical RAID (also known as nested RAID or 2D RAID, which combines two RAID levels) for small-scale systems [55, 122, 141, 142, 152], they do not consider hierarchical RAID in the context of large-scale data centers.

Thus, many important research questions remain unanswered: What are the possible chunk placement schemes for MLEC at scale? What are their pros and cons in terms of performance and durability? What types of failure modes might an MLEC system face? Can we introduce advanced repair methods optimized for specific schemes and failure modes? What are the implementation requirements for such advanced repairs?

None of these questions have been addressed in existing works. Therefore, we aim to answer these questions in Chapter 3.

2.2 Remote I/O Bottlenecks in DL Training

Deep Learning has emerged as a transformative technology across diverse domains, including computer vision, natural language processing, and audio processing [65, 77, 83, 102, 134]. To support this growing demand, cloud platforms now offer specialized services for facilitating DL training at scale [6, 8, 18].

DL training is characterized by its intense computational demands, requiring vast datasets, significant CPU resources for preprocessing, and powerful GPUs for executing complex neural network models [60, 66, 103, 105, 112, 113, 121, 148, 159, 160, 161, 167, 173]. As the size of DL datasets now frequently scales to tens or even hundreds of terabytes [33, 167], the data volume often exceeds the local storage capacity of cloud compute nodes. This has led to a common architecture where DL training occurs on compute nodes that fetch data from remote storage clusters such as distributed file systems or object stores [4, 7, 17, 133].

The use of remote storage introduces a critical performance bottleneck in DL training. As GPUs become faster, the data transfer speeds must match this pace to avoid stalls, underutilized GPUs, and delayed training results [66, 103, 105, 113, 116, 162, 167]. If the data feed is insufficient, the entire DL pipeline suffers, resulting in inefficient use of resources.

To mitigate this, it is essential to ensure that the remote storage architecture can provide sufficient aggregated I/O bandwidth to meet the demands of DL workloads. This involves optimizing the data organization, management, and storage architecture to deliver the required bandwidth for high-throughput data access. In the context of Multi-Level Erasure Coding (MLEC) storage, a key question arises: Can distributed MLEC storage, with its specific file placement and design, provide enough aggregated bandwidth to efficiently feed DL workloads?

If MLEC storage fails to deliver the necessary bandwidth, the goal becomes identifying and resolving the system's bottlenecks to improve GPU utilization and overall performance. Optimizing the MLEC system design can potentially alleviate I/O bottlenecks, leading to faster and more efficient training.

However, addressing this issue is costly, particularly in academic settings where GPUs and large disk arrays—both of which are essential for DL workloads and MLEC storage—are scarce and expensive. Thus, a cost-effective method is needed to evaluate MLEC storage against DL workloads, allowing for the identification of system bottlenecks and the rapid prototyping of optimizations.

Motivated by these challenges, in Chapter 4, we introduce an emulation-based approach to evaluate MLEC storage in the context of DL workloads. This method provides a practical solution to overcome resource limitations in academic environments while offering insights for identifying system bottlenecks and optimizing performance.

2.3 Emulators and Simulators: Existing Work and the Need for New Tools

In Chapter 4, we introduce two emulators: GPEMU, a GPU emulator designed to evaluate deep learning system research without requiring real GPUs, and MLECEmu, a storage emulator that replicates the behavior of MLEC storage without the need for real disk arrays. In this section, we review existing emulation and simulation techniques and explain why developing new tools like GPEMU and MLECEmu is necessary.

2.3.1 Existing Emulation Tools

Emulators replicate specific components of a system to allow them to function on different platforms. A wide variety of emulation tools exist for system components other than GPUs. For example, FEMU [107] and RAMSSD [40] emulate SSDs, FAME [111] and HME [67] emulate memory, and Emulab [85] and CloudLab [15, 68] are popular for network emulation.

However, there are limited options for GPU emulation, particularly in the context of deep learning (DL) workloads. For instance, Silod [167] profiles model computation time on expensive V100 GPUs and emulates it on cheaper K80 GPUs for cluster scheduling evaluation, but still requires real GPUs. MLPerf Storage [28] and DLCache [95] offer simple host-side GPU emulation, focusing

on compute time but lacking critical features for full-stack DL system evaluation. These emulators support limited configurations and experiments, highlighting the need for a more versatile GPU emulator.

There are also no existing emulators for erasure-coded storage systems. While tools like FEMU [107] and RAMSSD [40] emulate SSDs, no emulator currently supports the complex behaviors of erasure-coded storage systems, particularly MLEC.

2.3.2 *Simulators for GPU and Erasure-Coded Systems*

Simulators model the behavior of real-world systems for performance analysis and prediction. Numerous GPU simulators exist, such as MacSim [99], GPGPU-Sim [56], MGPU-Sim [139], and Accel-Sim [98], as well as GPU cluster scheduling simulators used in prior work [79, 104, 112, 138].

These simulators can analyze GPU performance metrics, such as memory usage and clock cycles, but they lack support for end-to-end DL system experiments that incorporate all layers, including storage, CPUs, host memory, and network. In contrast, GPEMU is designed to facilitate full-stack experimentation, encompassing a broader range of system components.

While there are simulators for single-level erasure-coded storage systems [96, 166], none exist for multi-level erasure coding (MLEC). The only MLEC simulator, MLECSim, was developed by us to simulate MLEC performance and durability. However, MLECSim does not support end-to-end experiments, motivating the development of MLECEmu, which enables such experiments by emulating MLEC storage in full-stack settings.

2.3.3 *The Need for* GPEMU *and MLECEmu*

The limitations of existing emulators and simulators underscore the need for new tools tailored to the unique demands of DL workloads and MLEC storage. GPEMU addresses the gaps in GPU emulation by providing extensive support for a wide range of configurations and features essential

for deep learning system research. Similarly, MLECEmu fills the void in erasure-coded storage emulation, enabling detailed performance evaluation of MLEC storage without requiring real disk arrays.

By combining these two emulators, researchers can conduct full-stack experiments that cover both GPU and storage layers, identify system bottlenecks, and prototype optimizations in a cost-effective and resource-efficient manner.

2.4 Preprocessing Offloading

Preprocessing is a crucial stage in deep learning (DL) training, involving tasks such as data augmentation, normalization, and transformation to prepare raw data for model consumption. This step is CPU-intensive and often becomes a significant bottleneck, especially in large-scale DL workloads with continuously processed data. As a result, previous studies [54, 145, 169, 170, 171] have explored offloading preprocessing tasks to additional CPU nodes to alleviate this bottleneck. However, these works primarily focus on reducing CPU load and overlook the substantial data fetch traffic between remote storage and compute nodes. Additionally, they apply a uniform, coarse-grained offloading strategy that treats all data samples equally, failing to account for the varying data sizes at different preprocessing stages.

In Chapter 5, from the storage perspective, we propose a different approach to preprocessing offloading: by observing that many data samples undergo size reductions during intermediate stages of preprocessing, we propose to **selectively offload** specific preprocessing tasks to the remote storage server. Given that intra-cluster bandwidth (within storage) is generally much higher than inter-cluster bandwidth (between storage and compute) [48, 80, 88, 149], our approach preprocesses larger data samples within the storage cluster itself. By preprocessing large data within the storage cluster and transmitting only smaller, partially processed data to compute nodes, we reduce data traffic, optimize network usage, and improve training efficiency.

CHAPTER 3

DESIGN CONSIDERATIONS AND ANALYSIS OF MULTI-LEVEL ERASURE CODING IN LARGE-SCALE DATA CENTERS

3.1 Introduction

MLEC has seen large deployments in the field, including in HPC data centers in national laboratories [90], enterprise-grade storage softwares [42], and commercial storage systems [1]. However, based on literature study and personal communications, there is no in-depth study of design considerations for MLEC at scale. Many research questions remain unanswered. What are the possible chunk placement schemes for MLEC at scale? What are their pros/cons in terms of performance and durability? What are the types of failure modes an MLEC system can face? Can we introduce advanced repair methods that are optimized for every specific scheme and failure mode? What are the implementation requirements for advanced repairs? Though other works analyze hierarchical RAID for small-scale systems [55, 122, 141, 142, 152], we have not seen any work answering the questions above or studying design considerations of MLEC for large-scale systems.

In this chapter, we provide, to the best of our knowledge, the most comprehensive design considerations and analysis of MLEC at scale that addresses the questions above. More specifically, we present the following contributions.

1. We introduce the design space of MLEC in multiple dimensions, including various code parameter selections, chunk placement schemes, and various repair methods from a simple and practical one to a more optimized one that leverages the multi-level EC but requires cross-level transparency.

2. We quantify their performance (encoding throughput, repair network traffic, repair time, etc.) and durability (under independent and correlated failures). We show which MLEC schemes and repair methods can provide the best tolerance against independent/correlated failures

and reduce repair network traffic by orders of magnitude.

3. To achieve all of the above, we use various evaluation strategies including simulation, splitting, dynamic programming, and mathematical modeling. We build, to the best of our knowledge, the first sophisticated MLEC simulator (in almost 13,000 lines of code (LOC)) that allows us to measure MLEC performance and durability at scale (over 50,000 disks), with many capabilities such as simulating disk failures (based on distributions, rules, or real traces), combining multi-level clustered/declustered placements, expressing failure tolerance, and executing complex repairs.

4. We also compare the performance and durability of MLEC with other EC schemes such as SLEC and LRC [89] and show that MLEC can provide high durability with higher encoding throughput and less repair network traffic over both SLEC and LRC.

In this chapter, we first introduce the MLEC design in Section 3.2, followed by our evaluation methodologies in Section 3.3. We then present an in-depth analysis of the performance and durability implications of various MLEC schemes and repair methods in Section 3.4, and evaluate MLEC in comparison with SLEC and LRC in Section 3.5. In Section 3.6, we discuss how our findings can guide large-scale storage architects in selecting optimal configurations tailored to their specific environments and requirements. Finally, we conclude the chapter in Section 3.7.

3.2 MLEC Design

We begin by describing the MLEC design, starting from the logical and physical views (§3.2.1-3.2.2) to the failure modes and possible repair methods (§3.2.3-3.2.6). To ease readers in finding definitions and descriptions, we use bold text for findings, the first mentions of figure references, and important terms.

3.2.1 MLEC Basics and Logical View

We begin with showing the *logical view* of the MLEC architecture by comparing it with basic SLEC architectures in **Figure 3.1**. For simplicity, not all physical elements are shown yet. We use the **(k+p)** notation to describe an SLEC setup with k data and p parity chunks. For MLEC, we use the $(k_n+p_n)/(k_l+p_l)$ **notation** where n and l respectively stand for "network" and "local."

Network SLEC: Figure 3.1a shows a simple (2+1) *network* SLEC with three enclosures. When **user data** arrive (a_1 and a_2), the storage server builds the parity chunk (a_{12}) and sends each of these chunks to a separate rack, and in each rack the chunk might go to a different enclosure (*e.g.*, a_1 might go to rack R_1 enclosure E_1, a_2 to R_2E_3, and a_{12} to R_3E_2. But for simplicity, a_1, a_2, a_{12} all go to E_1 in the figure). An **enclosure** is a collection of disks stored within the same **rack**. A "disk" can be an HDD, SSD, or other types of drives. Network-level SLEC can tolerate rack/enclosure-level failures but requires cross-rack network traffic for every repair.

Local SLEC: Figure 3.1b shows a (2+1) *local* SLEC with one enclosure. The storage server picks an enclosure and simply forwards the entire user data stripe to the enclosure-level controller, which then builds the parity chunk and writes all the chunks to different disks in the enclosure. Local SLEC can tolerate disk failures but not rack/enclosure-level failures.

MLEC: Figure 3.1c shows a (2+1)/(2+1) MLEC architecture, which is a *combination of network and local* SLEC architectures. Note that the k_n and k_l do *not* have to be the same, but we use k_n=k_l=2 here for simplicity. Upon receiving a full data stripe (four chunks, from a_1 to a_4), the storage server splits it to two **network data chunks** (a_1a_2 and a_3a_4) and build one **network parity chunk** ($a_{12}a_{34}$). Note here, a network-level chunk contains two local-level chunks. The server then distributes them across different enclosures in separate racks. Each enclosure takes the data and splits it to **local data chunks** (*e.g.*, a_1a_2 is split to a_1 and a_2 chunks), computes the **local parity chunk** (*e.g.*, a_{12} in R_1E_1), and sends them to three different disks in the enclosure.

Overall, the local-level MLEC manages the **local stripes** (*e.g.*, a_1–a_2–a_{12} is a local stripe with three local chunks). Likewise, the network-level MLEC is responsible for managing the **network**

stripes where each network stripe contains multiple local stripes (*e.g.*, $a_1a_2a_{12}$–$a_3a_4a_{34}$–$a_{13}a_{24}a_P$ is a network stripe containing three local stripes). a_P is the parity of a_{13} and a_{24}.

Clustered vs. declustered parity (Cp vs. Dp): Now let's look at the local placement, where one can deploy the conventional clustered or declustered parity placement. In the clustered parity ("**Cp**") placement, every (k+p) disks will form a **pool**. In **Figure 3.1d**, the 6 disks in one enclosure form *two* (2+1) **local-Cp** pools. Here, a local stripe must go to a specific pool, *i.e.*, a stripe either has no chunk in the pool, or has all the chunks residing in the pool. When a disk in a local-Cp pool fails, the local repairer reads from the *k* (two) surviving disks to reconstruct the lost data and write to a new spare disk. The rebuild time is *bottlenecked* both by the read bandwidth of only the participating disks (two disks here) and by the single disk's write bandwidth (to the one spare disk).

In order to improve the rebuild rate, declustered parity ("**Dp**") placement was proposed. Here we briefly introduce the design idea of declustered parity. More details can be found in literature [51, 52, 86, 96, 115, 135]. In declustered parity, a **local-Dp** pool should have *(much) more than* (k+p) disks. For example, in **Figure 3.1e**, *all* the 6 disks in the enclosure form only one local-Dp pool. Here, the data, parities, and spare space are pseudorandomly spread (declustered) across all the disks. When a disk fails, all the surviving disks in the large pool participate in both reading and writing which leads to faster repair rate. Later on, the admin can bring in a new disk and rebalances the data in the background.

Declustered parity placement can also be applied to network SLEC. In *network-Dp* SLEC, the entire system is treated as a pool, and each chunk in the stripe is placed pseudorandomly in a separate rack. When a disk fails, all the surviving disks in the system can participate in the repair, utilizing the network bandwidth of all the racks in the system to speed up the repair.

While declustered parity offers improved repair speed because of its pseudorandom chunk placement, it is susceptible to lower tolerance against correlated failure bursts. For instance, in a 6-disk pool using a (2+1) declustered parity scheme, if two disks fail simultaneously, there must

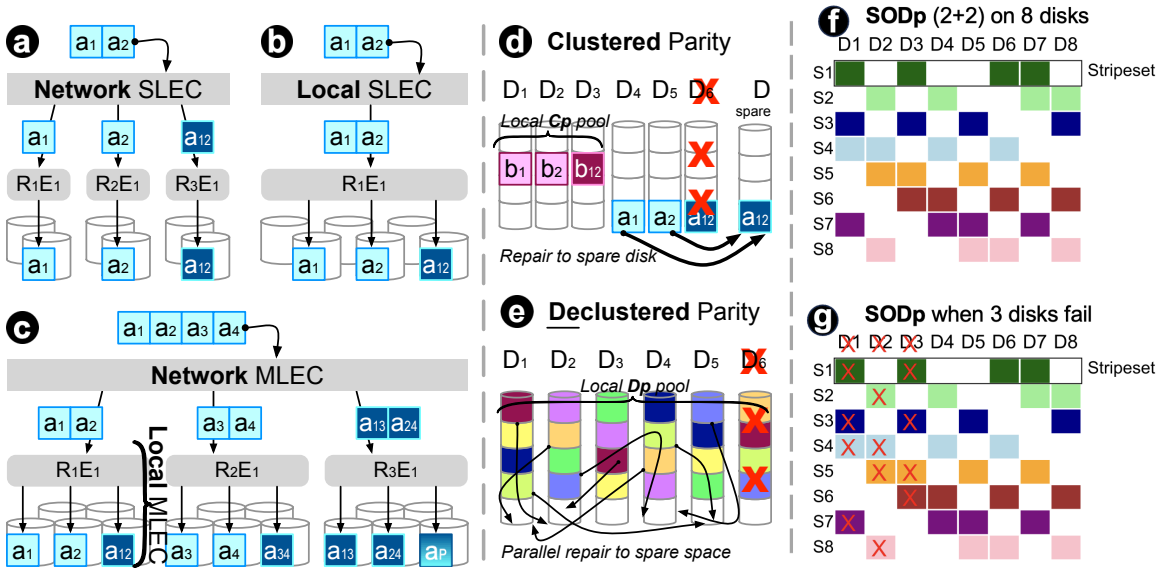


Figure 3.1: **SLEC vs. MLEC logical view (§3.2.1).** *The figures show (a) a network SLEC, (b) a local SLEC, and (c) an MLEC. Light-colored boxes (e.g., a_1, a_2) are data chunks and dark-colored boxes (e.g., a_{12}, a_{24}) are parity chunks. "R" and "E" respectively denote racks and enclosures. Figures (d) and (e) differentiates local clustered and declustered parity placements. Figure (f) and (g) illustrates how SODp works. Note that figures a-e assumes (2+1) in 6 disks for simplicity, while figures f-g assumes (2+2) in 8 disks to better illustrate the features of SODp.*

be a stripe that contains chunks on both failed disks. Consequently, this stripe cannot be repaired, resulting in data loss in the event of any two disk failures.

Single-overlap declustered parity (SODp): To address the issue of low burst tolerance in conventional declustered parity placement, recent research has introduced a placement method known as Single-Overlap Declustered Parity ("**SODp**") [96]. Unlike conventional Dp's random chunk placement, SODp organizes deterministic declustered "stripesets," and each incoming stripe can only be mapped to one of these stripesets. A stripeset refers to a specific set of disks chosen for a one-to-one correspondence with a stripe. **Figure 3.1f** illustrates the SODp placement for (2+2) on a 8-disk pool. Here 8 stripesets are formed, and an incoming stripe can only be put onto one of the stripesets.

SODp can still reap the advantages of parallel reconstruction. For instance, as illustrated in **Figure 3.1f**, when Disk 1 encounters a failure, its associated stripesets—S1, S3, S4, and S7—span all other disks within the pool. Consequently, all remaining disks can actively contribute to the

repair process. Moreover, SODp exhibits enhanced resilience against failure bursts. In scenarios like **Figure 3.1g**, where three disks (D1, D2, and D3) fail simultaneously, there is no data loss since no stripe contains three failed chunks.

3.2.2 MLEC Schemes and Physical View

Given the two levels (network and local) and the three chunk/parity placements (clustered, declustered, and SODp), we can permute them into six basic placement schemes (or "MLEC schemes" for short). It's worth noting that the SODp paper [96] exclusively introduces the algorithm for constructing SODp in the context of local SLEC, without addressing its application in network SLEC, which presents non-trivial challenges. Consequently, when we study SODp for MLEC, we will focus on the design covering local-level SODp.

Below we define each MLEC scheme, using **Figure 3.2** to illustrate the physical views. Again, for simplicity, we use a (2+1)/(2+1) MLEC, *i.e.*, k_n=2, p_n=1, k_l=2, and k_l=1. Hence, we show 3 racks (R_1 to R_3), where each rack contains 2 enclosures (E_1 and E_2) and each enclosure contains 6 disks (D_1 to D_6).

Clustered-clustered ($^C/_C$) scheme: In **Figure 3.2a**, this simplest scheme performs clustered parity at both network and local levels. A (2+1) local stripe is mapped to a local-Cp pool, containing adjacent k_l+p_l disks; for example, $a_1a_2a_{12}$ and $b_1b_2b_{12}$ are mapped to two different local pools, each with three consecutive disks. Moving up to the network level, every k_n+p_n enclosures at the same position across the three racks form a **network pool** for (2+1) network stripes. For example, for the network stripe $a_1 \ldots a_P$, they all have to reside in the same local-Cp pool position in the three E_1 enclosures across the three racks.

Clustered-declustered ($^C/_D$) scheme: In **Figure 3.2b**, this scheme performs a network clustered and local declustered placements. Starting from the top, all the data and parity chunks of a network stripe still have to reside within the same local-Dp pool position within the same enclosure position in each rack. For example, $a_1 \ldots a_P$ are mapped to the first local-Dp pool in enclosure E_1

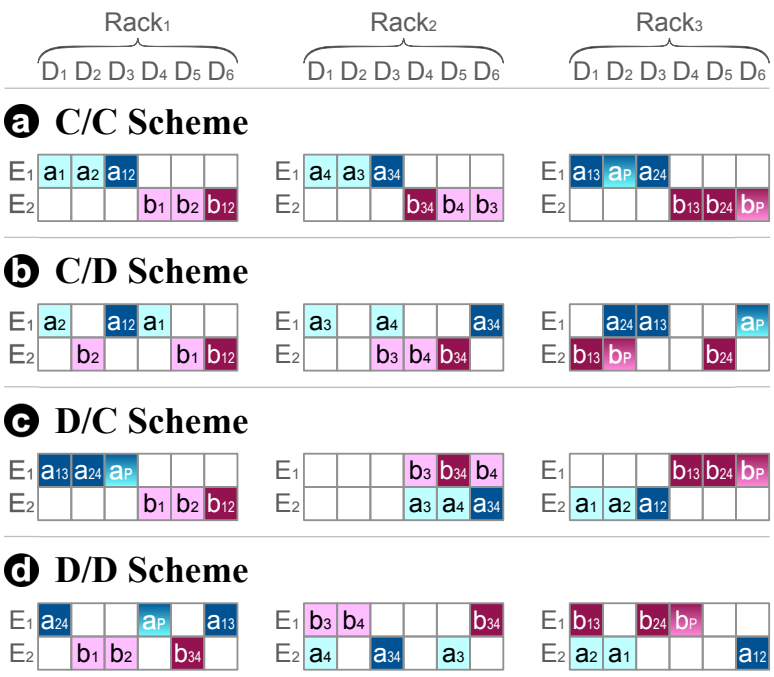


Figure 3.2: **Four MLEC schemes and their physical views (§3.2.2).** *The figure shows three racks, each with two enclosures, each with six disks. For simplicity, we only show chunk per disk (no disk cylinders); e.g., a_1 chunk is in Rack R_1, Enclosure E_1, Disk D_1. We are not showing $^{C}/_{S}$ and $^{D}/_{S}$ here, because they will look similar to $^{C}/_{D}$ and $^{D}/_{D}$ in this setup. The detailed difference between Dp and SODp can be found in Section 3.2.1*

across the three racks. Locally, as explained before, the local-Dp pool has 6 disks. The chunks of a local stripe are pseudorandomly spread across the 6 disks (*e.g.*, a_2, a_1, and a_{12} chunks are mapped to disks D_1, D_3, and D_4, respectively), but the chunks in the stripe cannot go to the same disk (to tolerate disk-level failure).

Clustered-SODp ($^{C}/_{S}$) scheme: The $^{C}/_{S}$ scheme is very similar to $^{C}/_{D}$, given that SODP serves as a variation of conventional declustered parity. As a result, we will not provide a separate illustration for $^{C}/_{S}$, as it would closely resemble **Figure 3.2b**. It's important to note that the primary distinction between $^{C}/_{S}$ and $^{C}/_{D}$ lies in the local stripe placement. In $^{C}/_{D}$, the chunks of a local stripe are pseudorandomly distributed across all local disks, while in $^{C}/_{S}$, a local stripe is assigned to one of the deterministic stripesets.

Declustere-clustered ($^{D}/_{C}$) scheme: Reversing the previous scheme, now we have the network level performing declustered parity. That is, the local-stripes of a *network* stripe will be pseudo-

randomly spread across the enclosures within the network pool, but they cannot go to the same rack (to tolerate rack-level failure). For simplicity, **Figure 3.2c** shows a network pool containing only six enclosures across the three racks. The network stripe $a_1 \ldots a_P$ is split to three local stripes stored in different enclosure positions in the three racks (*e.g.*, the local stripe $a_1 a_2 a_{12}$ is mapped to enclosure E_2 in rack R_3). At the local level, $^{D}/_{C}$ follows $^{C}/_{C}$, *i.e.*, a local stripe goes to a local-Cp pool.

Declustered-declusted ($^{D}/_{D}$) scheme: In this scheme, we have declustered placements in both network and local levels. For example, in **Figure 3.2d**, just like in the previous figure/scheme, the local stripe $a_1 a_2 a_{12}$ is mapped to enclosure E_2 in rack R_3, but now the chunks of this local stripe are scattered across the 6 disks in the local-Dp pool.

Declustered-SODp ($^{D}/_{S}$) scheme: Finally, we introduce $^{D}/_{S}$ scheme. Again, $^{D}/_{S}$ is very similar to $^{D}/_{D}$ in **Figure 3.2d**, despite specific details of the local stripe placement. Consequently, we will not provide a distinct figure illustration for $^{D}/_{S}$.

In large-scale deployments, in network clustered ($^{C}/_{*}$)[1] schemes, every k_n+p_n inter-rack local pools in the same enclosure position will form a network pool, hence the total rack count must be a multiple of k_n+p_n. However, in network declustered ($^{D}/_{*}$) schemes, a network pool usually contains much more than but does not have to be a multiple of k_n+p_n racks. Likewise, in local clustered ($^{*}/_{C}$) schemes, a local pool contains k_l+p_l disks, and hence an enclosure must have a multiple of k_l+p_l disks. However, in local declustered ($^{*}/_{D}$) and local SODp ($^{*}/_{S}$) schemes, a local pool usually contains much more than but does not have to be a multiple of k_l+p_l disks.

3.2.3 Failure Modes

Given the more complex chunk placements, MLEC can face various failure modes, as listed in **Table 3.1**, which we will use heavily throughout the paper. With the listed definitions, we now can derive the data loss conditions, which vary across the MLEC schemes. A **data loss** is defined

1. $*$ **is a don't-care symbol.**

Local level failures
• **A failed chunk**: A lost (but may be recoverable) chunk due to a disk failure.
• **An affected local stripe**: a local stripe with any number of chunk failures.
• **A locally-recoverable local stripe**: A local stripe containing 1 to p_l failed chunks.
• **A lost local stripe**: A local stripe containing p_l+1 or more failed chunks but may still be recoverable from the network level.
• **A catastrophic (locally-unrecoverable) local pool**: A local pool with 1 or more lost local stripes, *e.g.*, in (10+2)/(17+3) MLEC, a local pool with 4 disk failures is not recoverable locally and requires network repair.
Network level failures
• **An affected network stripe**: a network-wide stripe with any number of lost local stripes.
• **A recoverable network stripe**: A network stripe containing 1 to p_n lost local stripes.
• **A lost network stripe (a data loss)**: A network stripe with p_n+1 or more lost local stripes.

Table 3.1: **MLEC failure modes (§3.2.3).**

as the loss of a network stripe, more specifically the loss of p_n+1 local stripes. In network-Cp ($^{C}/_{*}$) schemes, only p_n+1 catastrophic local pools in the same network pool can cause a network stripe to have p_n+1 lost local stripes. Since a $^{C}/_{*}$ system can have many network-level pools, p_n+1 catastrophic local pools that are scattered in multiple network-level pools will not cause data loss. In network-Dp ($^{D}/_{*}$) schemes, since there is only one network pool in the system, any arbitrary p_n+1 catastrophic local pools may lead to a lost network stripe with p_n+1 lost local stripes. However, the probability for such a lost network stripe to happen can be extremely low, depending on the actual chunk placement (which is pseudorandom) in network-Dp ($^{D}/_{*}$) schemes.

3.2.4 Repair Methods

When it comes to repair, the network pool level is more important. Repairing locally-recoverable pools is straightforward (similar to SLEC repairs in Figures 3.1d-e). The challenge is to recover[2] a **catastrophic (locally-unrecoverable) local pool** (defined in Table 3.1). For this, we introduce four possible local-pool repair methods applicable to all the MLEC schemes, as illustrated in **Figure**

2. We use recover/repair/rebuild/reconstruct interchangeably.

3.3, from the simple to optimum ones, along with their pros and cons.

Repair All (R_{ALL}): In **Figure 3.3a**, the failures of disks D_1 and D_3 in rack R_1 caused a catastrophic local pool failure, thus chunks a_1 and a_2 need to be reconstructed. "Repair All" (R_{ALL}) is a method that simply *rebuilds the entire local pool* (*e.g.*, disks D_1 to D_6 in rack R_1) from the other healthy local pools in other racks (R_2 and R_3) via a network-level parity calculation. As the downside, it unnecessarily leads to a much higher amount of network traffic. However, R_{ALL} is common in deployment (*e.g.*, in MarFS [90]) because it is considerably the *easiest* to implement. That is, the network repairer does not need to know the layouts of the local part of the MLEC. The network-level sysadmins can use black-box/off-the-shelf RBODs (*e.g.*, CORVAULT [11], ZFS pools [58], and PowerEdge RAID [13]).

Repair Failed Chunks Only (R_{FCO}): Unlike R_{ALL}, R_{FCO} only rebuilds the failed chunks. For example, in **Figure 3.3b**, a_1 in rack R_1 is rebuilt by network parity calculation on a_3 and a_{13} from the other two racks (and similarly for a_2). By doing this, R_{FCO} reduces the network repair traffic. Although it seems simple, R_{FCO} is less straightforward to implement. It requires the local/enclosure-level repairer to report which chunks have failed and coordinates with the network-level repairer. This is one reason why repair methods like R_{FCO} are not supported by many existing RAID systems. For example, ZFS [58] handles its own block device mappings internally, and does not readily expose the mapping information. Therefore, when a ZFS local pool fails, it is often impossible for the network-level repairer to know which chunks have failed. Thus, although R_{FCO} is more efficient than R_{ALL}, R_{FCO} requires a proper API design and metadata management across the local and network levels.

Repair Hybrid (R_{HYB}): R_{HYB} repairs the failed chunks in a hybrid way using both network and local repairs. To show this optimization, **Figure 3.3c** shows a slightly different layout, where two failed chunks a_2 and b_2 are stored in the same failed disk D_3 in rack R_1(plus another failed chunk a_1 in disk D_1). Here, R_{HYB} basically analyzes every failed chunk and asks whether a network repair is needed or not. The loss of a_1 and a_2 form a *lost local stripe*, hence they *cannot* be

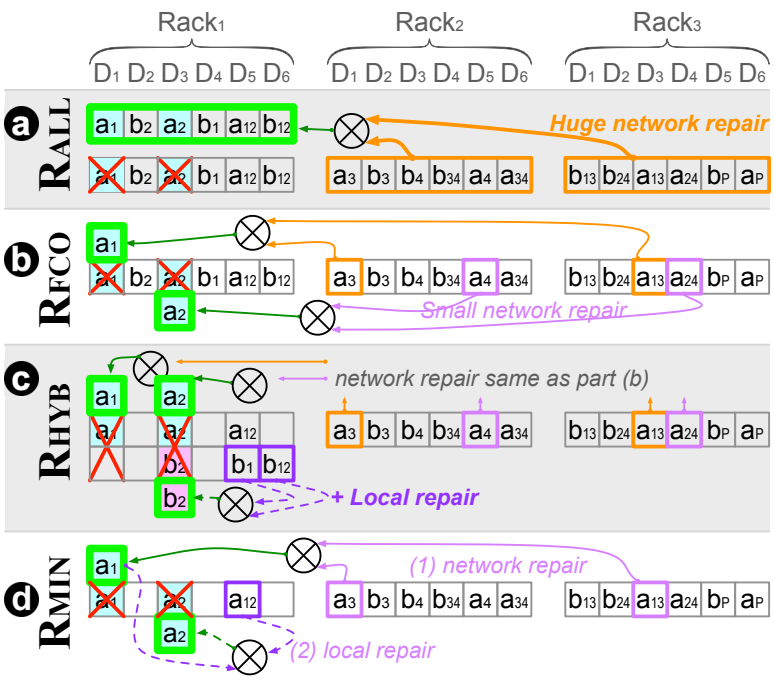


Figure 3.3: **Four repair methods, R_{ALL} to R_{MIN} (§3.2.6).** *For simplicity, we only show a (2+1)/(2+1) $^{C}/_{D}$ MLEC scheme. Also, not all the figures use the same chunk locations.*

repaired locally and require the network repair. However, for b_2, it can be rebuilt locally because b_1 and b_{12} are still available in the surviving disks (*i.e.*, a *locally-recoverable stripe*). As we measure and explain later in Section 3.4.2, R_{HYB} works well for local-Dp ($^{*}/_{D}$) schemes.

Repair Minimum (R_{MIN}): Taking the insight from the previous method, this last method incurs the minimum amount of network repair traffic. It does so in two stages. First, it finds all the lost local stripes and reads the minimum number of chunks over the network such that the stripes transition to locally-recoverable local stripe. Second, all the locally-recoverable local stripes can be rebuilt locally. For example in **Figure 3.3d**, the lost local stripe $a_1a_2a_{12}$ is partially repaired by rebuilding a_1 first (by xor-ing a_3 and a_{13}) over the network. Now, a_2 is still lost but can be rebuilt locally by xor-ing a_{12} and the recently-rebuilt a_1.

3.2.5 Encoding

For encoding in MLEC, there are two options:

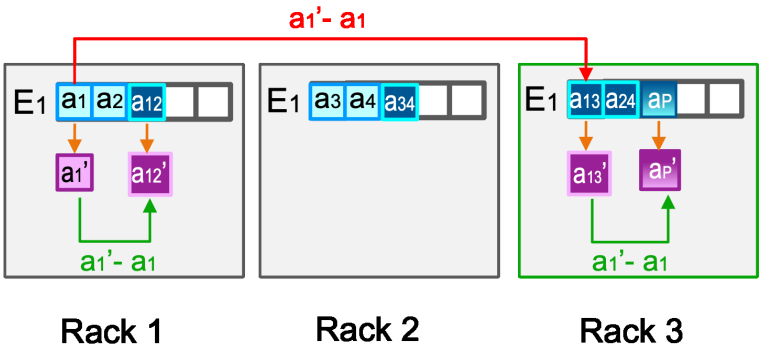


Figure 3.4: **Parity updates in MLEC.** *The figure shows how MLEC updates local parity, global parity, and double parity.*

1. The network-level server computes all local parities, network parities, and double parities and then distributes all chunks to local pools.

2. The network-level server only computes network parities, distributes the chunks to local pools, and lets local controllers compute the local parities and double parities.

Our experiments indicate that both methods achieve similar encoding throughput when using the same number of CPUs. However, the first method introduces more cross-rack network traffic during chunk distribution since it also distributes local parities and double parities across racks. In contrast, the second method computes them locally, resulting in less cross-rack network traffic overhead. Additionally, many local pools, such as RBODs, are already equipped with local controllers, making the second choice more practical. Therefore, we have adopted the second choice in all our designs and analyses.

3.2.6 Updates

Existing MLEC systems, such as MarFS [90], do not support in-place updates, as they target workloads characterized by bulk ingestion and retrieval, where read-modify-write operations are not required. However, for completeness, we describe how parity updates during read-modify-write operations on data chunks in MLEC can be performed with minimal cross-rack data transfers.

Figure 3.4 depicts how to update the parities in MLEC. It can be divided into three steps:

1. **Local parity update:** We first update the local parities in the same rack. For example, when a data chunk a_1 is updated to a_1', we compute the delta chunk $a_1' - a_1$, and update the local parity $a_{12}' = a_{12} + \alpha_L(a_1' - a_1)$, where α_L is the local-level encoding coefficient for computing a_{12}.

2. **Network parity update:** We next update the network parities across racks. For example, in Figure 3.4 we transfer the delta chunk $a_1' - a_1$ across rack and compute the network parity $a_{13}' = a_{13} + \alpha_N(a_1' - a_1)$, where α_N is the network-level encoding coefficient for computing a_{13}.

3. **Double parity update:** Finally we update the double parities within the rack based on updates of the network parity. For example, in Figure 3.4, we use the delta chunk for network parity (computed in prior step) $a_{13}' - a_{13}$ to compute the double parity: $a_P' = a_P + \alpha_L(a_{13}' - a_{13})$

Among all these steps, only Step 2 requires cross-rack network traffic. Therefore, MLEC only requires p_l cross-rack transferred chunk for updating parities. The disk IO cost, however, can be as much as $2(p_l+1)(p_n+1)$ in order to finish the read-modify-write updates for all parities.

3.3 Methodology

To perform in-depth analysis for various MLEC schemes and repair methods, we use the four evaluation strategies: simulation, splitting, dynamic programming, and mathematical modelling. In this section, we introduce these four evaluation strategies, followed by the description of our target system setup.

3.3.1 Simulation

To quantify performance and reliability, one can start with a mathematical model. However, our work introduces complex repair methods that are hard to model. For this reason, we build a sophisticated MLEC simulator (in almost 13 kLOC) with many capabilities such as simulating disk failures (based on distributions, rules, or real traces), combining multi-level (de)clustered placements, expressing failure tolerance, and executing complex repairs.

The high-level logic of the event-driven simulator works as follows:

1. We first geenerate failure times (following a specific distribution or from a real-world trace) for all the drives in the system
2. We then select the failure times that are less than the mission time (e.g. one year).
3. For each disk failure:
    (a) We calculate the repair time for drive by following the erasure coding policy and considering the current state of the system
    (b) We check if the drive failure causes the rack to fail.
        i. If the rack fails, we calculate the repair time for the rack.
        ii. We also check if the rack failures causes the system to fail. If so, fail the system, return fail.
    (c) We update system states when detecting a disk failure, starting a repair, done repairing high-prioirty stripes, and finished repairing the entire disk.
    (d) We generate new failures for those drives that have been repaired
4. If the system survives after the mission time, return success.
5. Repeat process 1-4 for M times, get N failures and M-N successes. The the probability of data is N/M.

The details on how to handle differnet events (e.g. failure detection, repair initialization, high-prioirity repair, etc) depends on the specific erasure coding design, which adds more complexity to the simulator.

We use this simulator to measure repair traffic, repair time, and system durability in Sections 3.4.1, 3.4.1, 3.4.2, 3.4.2, 3.5.1, 3.5.2.

3.3.2 Splitting (multi-stage simulation)

A uniqueness of our work is the focus on extreme-scale deployments (*e.g.*, >10k disks), which require protection from a large number of parities. However, to estimate high durability, a simulation must run a large number of iterations to capture even one system data loss event (it will take years even with a 200-core simulation). To reach rare events faster, we adopt the splitting method [74, 119]. First, we simulate the durability of a single local pool using regular simulation and collect local pool failure samples. Second, we systematically inject catastrophic local pool failures from the samples at MLEC level. We use this to evaluate high durability in Sections 3.4.2, 3.5.1, and 3.5.2.

3.3.3 Dynamic programming

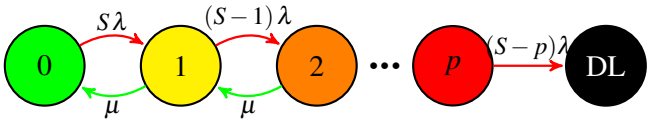
While splitting is efficient to measure high durability under independent failures, it is hard to do the same under correlated failure bursts. This is because each local pool's durability is not deterministic but correlated to other local pools' durability. Thus, we use dynamic programming, specifically by using it to count the number of all the possible disk failure layouts under a certain correlated failure burst scenario, and then count how many such failure layouts could cause a data loss in MLEC. We use this strategy for measuring the probability of data loss (PDL) under correlated failure bursts in Sections 3.4.1, 3.5.1 and 3.5.2.

3.3.4 *Mathematical model*

For thoroughness, we also build a mathematical model to verify our simulation strategies, but we can only do so for the simplest repair method (R_{ALL}). We choose to use Markov Chain model as it's commonly used to analyze durability of SLEC systems [50, 69, 78]. To model the MLEC system, we first took existing SLEC durability models that use Markov chain models and probability theory [50, 69, 78, 143], and then we iteratively apply the model to network-level MLEC by treating a local pool like a disk.

Here we give a description on how we build the model.

We model the data durability using a Markov chain [143, 153]:



An erasure coded system has $S = k + p$ storage devices, where k and p are respectively the number of data and parity chunks. Storage devices fail and are repaired at the respective constant rates of λ and μ. As drives fail more quickly than they are repaired, a stripe of data becomes increasingly at-risk. With p failed drives, the stripe can survive no further failures. At $p+1$ failures, data has been lost (DL).

We assume that the reliability of each individual storage device d can be described by an exponential function:

$$R_d(t) \sim e^{-\lambda t}, \tag{3.1}$$

where λ is the failure rate. EC across S devices with p parities results in the Mean Time To Data Loss(MTTDL) given by:

$$\text{MTTDL}_s = \left(\frac{\mu}{\lambda}\right)^k \frac{(S-p-1)!}{\lambda S!}, \tag{3.2}$$

where μ is the repair rate. With the common assumption that the data loss times are exponentially

distributed [82, 123], we have:

$$\mathrm{R}(t) \quad \sim \quad e^{-t/\mathrm{MTTDL_s}} \equiv e^{-\lambda_s t}, \tag{3.3}$$

such that system level data loss rate is:

$$\lambda_s = \frac{1}{\mathrm{MTTDL_s}} = \left(\frac{\lambda}{\mu}\right)^k \frac{\lambda S!}{(S-p-1)!}. \tag{3.4}$$

The data durability of such a system is typically measured in number of nines (NoN); defined as follows:

$$\mathrm{NoN} = -\log\left(1-\mathrm{R}(t)\right) \quad \simeq \quad \log\left(\mathrm{MTTDL_s}\right), \tag{3.5}$$

in which we set $t = 1\,\mathrm{year}$ and measure MTTDL in years. For MLEC, a second layer of EC is implemented using the erasure blocks already computed at the first layer. We consider S_n of such systems and create a second layer of EC with p_n second layer parities. We can then use Eq. 3.2 substituting the network level EC parameters: $\{S_n, p_n, \lambda_s, \mu_n\}$ (the subscript "n" stands for network) to get:

$$\mathrm{MTTDL^J} \quad = \quad \left(\frac{\mu_n}{\lambda_s}\right)^{p_n} \frac{(S_n-p_n-1)!}{\lambda_s S_n!}, \tag{3.6}$$

in which we used the superscript J to show that this is the joint MTTDL for MLEC.

Using Eq. 3.5, we calculate the NoN for MLEC as:

$$\mathrm{NoN^J} = \log\left(\mathrm{MTTDL^J}\right)$$

Note that the model assumes the use of the $\mathrm{R_{ALL}}$ repair method, a detection time of 0, the exponential distribution of data loss times, and an infinite number of stripes in the local pool. Therefore, the model is only used in special cases with these assumptions enabled to verify the

correctness of our simulator. More complicated cases with different assumptions are evaluated using the simulator.

3.3.5 Setup

We use the following setups, which mimic real large-scale deployments [34, 35, 90].

Datacenter setup: We simulate 57,600 disks across 60 racks, with 8 enclosures per rack, 120 disks per enclosure, 20 TB per disk, and a chunk size of 128 KB.

MLEC configuration: We use a **(10+2)/(17+3)** MLEC. For $*/C$ schemes, a local-CP pool contains exactly 20 disks. For $*/D$ schemes, given the local declustered approach, a local-DP pool contains 120 disks.

Available repair bandwidth: The *raw* bandwidth is set to be 200 MB/s for per-disk I/Os and 10 Gbps per rack for cross-rack network, respectively. However, for repairs, disk and network traffics are both capped at 20% of their respective raw bandwidth. Thus, we use the term "available repair bandwidth" to reflect the resulting data repair bandwidth that is subject to this policy.

Fault simulation: To simulate a catastrophic local pool failure, we generate p_l+1 disk failures simultaneously. However, to measure long-term durability (*e.g.*, over one year), we generate random disk failures independently following an exponential distribution with an annual failure rate (AFR) of 1%.

Failure detection time: We also follow previous works [88, 89] that use 30 minutes to detect each failure and trigger the repair.

3.4 MLEC Analysis

We now present an in-depth analysis of performance and durability implications of various MLEC schemes in Section 3.4.1 and repair methods in Section 3.4.2.

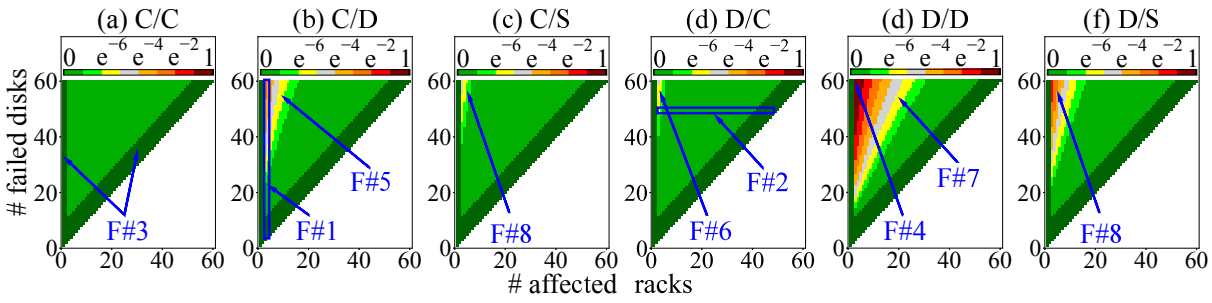


Figure 3.5: **PDL under correlated failures (§3.4.1).** *The square color represents the PDL of the MLEC scheme when a total of y simultaneous disk failures are randomly scattered across x racks.*

3.4.1 Analysis of MLEC Schemes

First, we analyze the six MLEC schemes ($^{C}/_{C}$ to $^{D}/_{S}$), particularly the impact of their chunk/parity placements on the probability of data loss under correlated failures (§3.4.1), repair speed (§3.4.1), and probability of catastrophic local failure (§3.4.1).

PDL under Correlated Failures

We begin with analyzing the impact of different MLEC schemes on the probability of data loss (PDL) under *a wide range* of failure burst topologies, from scattered disk failures across many racks to highly correlated failures localized in a single rack. **PDL** is defined as the probability that a storage system unrecoverably loses any data, and **failure bursts** refer to failures that are temporally correlated and happen concurrently at the same time or within a small time window [69].

Figure 3.5 shows heatmaps of the PDL, where greenish squares (near the value of 0) represent high durability and reddish squares (near the value of 1) represent low durability. We vary the number of disk failures (in the y-axis) and spread them across one or more racks (in the x-axis). For example, $\{y = 60, x = 60\}$ implies that every rack has a single disk failure (*i.e.*, scattered failures), while $\{y = 60, x = 1\}$ implies that only one rack experiences 60 disk failures. The six subfigures show the impact of these various failures on $^{C}/_{C}$, $^{C}/_{D}$, $^{C}/_{S}$, $^{D}/_{C}$, $^{D}/_{D}$, and $^{D}/_{S}$ schemes, respectively.

Below we present our findings "**F#1-8**" in Figure 3.5. Findings #1-4 are applicable to all six schemes, but for presentation clarity, we spread them across the subfigures. Findings #5-8, however, are unique to the labeled schemes, respectively. We believe Findings #2-#8 were never reported before.

Finding #1: *When a failure burst happens in at least p_n+1 racks, the more failed disks in those racks, the higher the PDL.* For example, in the highlighted vertical area in Figure 3.5b, p_n is 2, but we have 3 racks ($x = 3$) experiencing failures. If the number of failed disks goes up (in the y-axis), it will cause more local stripe failures. As a result, the PDL will go up (greenish to reddish squares), and data loss will happen if p_n+1 or more local stripe failures happen within a network stripe (as the network stripe cannot recover).

Finding #2: *MLEC is more robust to scattered failures.* As highlighted in the horizontal area in Figure 3.5d, when a fixed number of disk failures happen concurrently, the more racks they are scattered to, the lower the PDL is. This is because each rack is more likely to have fewer disk failures, which is more tolerable by the local-level EC of MLEC.

Finding #3: *Full local failures can be recovered by the network-level EC up to some limit.* In Figure 3.5a, zero data loss can be guaranteed (PDL = 0) when the number of affected racks is smaller than or equal to 2. This is because a network stripe can survive any p_n rack failures, and p_n is 2 here. The PDL is also 0 when no more than x+8 disk failures are scattered in x racks. This is because a local (17+3) stripe can tolerate any 3 failures. Thus, x+8 disk failures in x racks can cause at most 2 lost local stripes in the same network stripe, which can be tolerated by the network-level (10+2) EC.

Finding #4: *MLEC is susceptible to data loss under highly localized failure bursts.* Deriving from previous findings, MLEC suffers the most when failures bursts happen in exactly p_n+1 racks, depicted in Figure 3.5e. In all the figures, we can also see that PDL is the highest when 60 disk failures happen concurrently in 3 racks.

Finding #5: *$^{C}/_{D}$ has lower tolerance than $^{C}/_{C}$ under more localized failure bursts.* By com-

paring the area pointed to by Finding #5 in Figure 3.5b and the same area in Figure 3.5a, we can see that $^{C}/_{C}$ has better tolerance than $^{C}/_{D}$ when failure bursts happen in more than p_n racks. This is because a local-Dp pool can only tolerate p_l arbitrary concurrent disk failures out of many (let's say D_l) disks. On the other hand, a local-Cp pool can tolerate up to p_l disk failures out of (k_l+p_l) disks, where usually D_l>k_l+p_l. Therefore, when the same number of random disks in a rack fail concurrently, it's more possible to cause local pool failures in $^{C}/_{D}$ than in $^{C}/_{C}$, and more frequent local pool failures in turn make system-wide data loss more likely.

Finding #6: *Likewise, $^{D}/_{C}$ has lower tolerance than $^{C}/_{C}$.* Similar to above, when p_n+1 or more racks have failures, $^{D}/_{C}$ performs worse than $^{C}/_{C}$, yet for a different reason. Whenever more than p_n racks have local pool failures, no matter which local pool in the rack fails, $^{D}/_{C}$ can lose data. On the other hand, $^{C}/_{C}$ experiences data loss only when more than p_n local pools in the same network-level pool fail.

Finding #7: *$^{D}/_{D}$ has the worst data durability under failure bursts*. In Figure 3.5, $^{D}/_{D}$ has the most reddish squares (higher PDL) among all the schemes. This is because the local-Dp scheme makes local pool failures more likely, and local pool failures in the network-Dp scheme are more likely to cause data loss. Even when failures are roughly scattered, unlike other schemes, $^{D}/_{D}$ has a lower chance of survival, because its local-Dp pools are more possible to fail compared to local-Cp pools in $^{C}/_{C}$ and $^{D}/_{C}$. Furthermore, $^{D}/_{D}$ can lose data when any arbitrary p_n+1 local pools in separate racks fail, while $^{C}/_{C}$ and $^{C}/_{D}$ only lose data when p_n+1 local pools in the same network-level pool fail, which is less likely to happen.

Finding #8: *$^{*}/_{S}$ has the higher tolerance than $^{*}/_{D}$, but has lower tolerance than $^{*}/_{C}$.* The burst tolerance of $^{*}/_{S}$ falls betwen $^{*}/_{C}$ and $^{*}/_{D}$. This is attributed to the local-SODp scheme, which positions a local stripe onto a set of deterministic stripesets rather than using purely random placement. As a result, a local-SODp pool is less susceptible to failures in comparison to a local-Dp pool, reducing the risk to network-level erasure. However, in many cases, when p_l+1 disks fail, it can still lead to the failure of a local-SODp pool, making it more susceptible to failure compared to a

	Single disk failure		Local pool failure	
MLEC Schemes	Disk size (TB)	Avail. Repair BW (MB/s)	Pool size (TB)	Avail. Repair BW (MB/s)
$^{C}/_{C}$	20	40	400	250
$^{C}/_{D}$	20	264	2400	250
$^{C}/_{S}$	20	160	2400	250
$^{D}/_{C}$	20	40	400	1363
$^{D}/_{D}$	20	264	2400	1363
$^{D}/_{S}$	20	160	2400	1363

Table 3.2: **Repair size and available repair bandwidth (§3.4.1).** *Under (a) single disk failure and (b) catastrophic local failure.*

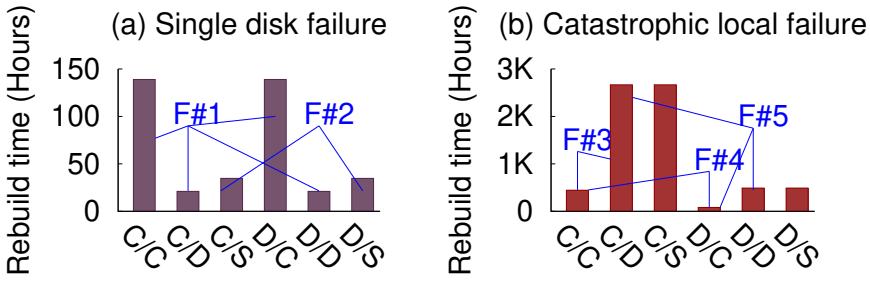


Figure 3.6: **Repair time (§3.4.1).** *Under (a) a single disk failure and (b) a catastrophic local failure.*

local-Cp pool.

In conclusion, different MLEC chunk placement schemes provide different tolerance against correlated failure bursts. Among them, $^{C}/_{C}$ performs the best while $^{D}/_{D}$ has the largest probability of data loss. However, $^{C}/_{C}$'s repair rate unfortunately is not as fast as the other schemes, as we will dissect in the next section.

Repair Speed

We now analyze how long data repair will take under various MLEC schemes. Specifically, we analyze the time of repairing (a) a single disk and (b) a catastrophic local failure. *This section starts with the simplest method, Repair-All (R_{ALL})*, but later Section 3.4.2 uses all the repair methods.

Figure 3.6 shows the rebuild time of the six MLEC schemes under both failure conditions. To explain the rebuild time in the figure, **Table 3.2** provides further information on the amount of

data to rebuild and the available repair bandwidth, which also depends on how many disks/local pools/racks participate in the repair as explained in earlier in the "R_{ALL}" paragraph of Section 3.2.6. We now elaborate findings "**F#1-5**" in Figure 3.6.

Finding #1: *In repairing a single disk failure, local declustered placement in $^{C}/_{D}$ and $^{D}/_{D}$ makes rebuilding fast.* Figure 3.6a shows that $^{C}/_{D}$ and $^{D}/_{D}$ are 6x faster compared to $^{C}/_{C}$ and $^{D}/_{C}$. This is because, when a disk in a 120-disk local-Dp pool fails, the local repairer can in parallel read the healthy chunks from and write the reconstructed chunks to spare spaces on *all the* 119 surviving local disks. On the other hand, the local-Cp repairer reads from 19 surviving disks and rebuilds to *only* 1 spare disk. Since it only has at most 20 disks to participate in the local repair, its available repair bandwidth is lower. In Table 3.2, the single disk repair bandwidth in $^{C}/_{C}$ and $^{D}/_{C}$ is around 6x lower than that of $^{C}/_{D}$ and $^{D}/_{D}$.

Finding #2: *In repairing a single disk failure, local SODp placement in $^{C}/_{S}$ and $^{D}/_{S}$ makes rebuilding faster than $^{C}/_{C}$ and $^{D}/_{C}$, but slightly slower than $^{C}/_{D}$ and $^{D}/_{D}$.* Figure 3.6a shows that both $^{C}/_{S}$ and $^{D}/_{S}$ are 4x faster compared to $^{C}/_{C}$ and $^{D}/_{C}$. This enhancement is attributed to the use of declustered stripe placement in local-SODp pools, which facilitates parallel rebuilding and boosts repair bandwidth. However, it's important to note that SODp's declustering is not perfectly balanced, which means it cannot achieve optimal parallel rebuilding. In a SODp pool, when a disk fails, some healthy disks may share more stripes with the failed disk than others, requiring greater participation in the repair process and potentially becoming bottlenecks. As a result, the repair bandwidth in a SODp pool is slightly lower than that in a Dp pool, causing $^{C}/_{S}$ and $^{D}/_{S}$ to be slightly slower than $^{C}/_{D}$ and $^{D}/_{D}$ when repairing a single disk.

Finding #3: *In repairing a catastrophic local failure, $^{C}/_{D}$ and $^{C}/_{S}$ takes the longest time due to its larger local pool size.* While in Figure 3.6a, $^{C}/_{D}$ and $^{C}/_{S}$ are faster than $^{C}/_{C}$ in rebuilding a single disk failure, it is the contrary under a catastrophic local failure, as shown in Figure 3.6b. This is because $^{C}/_{D}$ and $^{C}/_{S}$ have a local-Dp or local-SODp pool whose sizes are much larger than that of a local-Cp pool, leading to more data to reconstruct across the network with limited network

bandwidth. As detailed in Table 3.2, the local-Dp ($*/D$) and local-SODp ($*/S$) pool size is 2400 TB while the local-Cp ($*/C$) pool size is only 400 TB.

Finding #4: *In repairing a catastrophic local failure, $^{D}/_{C}$ is the fastest scheme.* The speed-up comes from the network-level declustered chunk placement. When reconstructing the target local pool, the repairer needs to read chunks from other local pools. Thanks to the network-level declustering, the chunks are spread across all the other 59 racks and the rebuilt data can be written to spare spaces in all the 60 racks, including the rack that contains the failed local pool. This in turn gives a 5x repair rate compared to $^{C}/_{C}$, which has only 12 (from 10+2) racks participating in the repair at most. As shown in Table 3.2, $^{C}/_{C}$'s local pool repair bandwidth is 250 MB/s while $^{D}/_{C}$'s can be as high as 1,363 MB/s.

Finding #5: *$^{D}/_{D}$ and $^{D}/_{S}$ are faster than $^{C}/_{D}$ and $^{C}/_{S}$, but slower than $^{D}/_{C}$ and slightly slower than $^{C}/_{C}$.* Still in Figure 3.6b, compared to $^{C}/_{D}$ and $^{C}/_{S}$ (the slowest of all), $^{D}/_{D}$ and $^{D}/_{S}$ are around 5x faster due to the network declustering. However, with its local pool that is 6x larger in size compared to that of $^{D}/_{C}$, $^{D}/_{D}$ and $^{D}/_{S}$ are around 6x slower than $^{D}/_{C}$. As a result, $^{D}/_{D}$ and $^{D}/_{S}$ take a bit longer than $^{C}/_{C}$ as the repair overhead caused by the large local-Dp pool dominates here. However, in a cluster with more racks and smaller local-Dp pool size, or in a cluster with "unlimited" network bandwidth, $^{D}/_{D}$ and $^{D}/_{S}$ could be faster than $^{C}/_{C}$ in repairing a catastrophic local pool. This is because $^{D}/_{D}$ and $^{D}/_{S}$ theoretically can read from and write to all local pools.

In conclusion, under a single disk failure, $^{C}/_{D}$ and $^{D}/_{D}$ have the fastest repair rate, but $^{D}/_{C}$ is the fastest under a catastrophic local failure. However, we emphasize the tradeoffs again that all these speed-ups are obtained at the cost of a higher probability of data loss against correlated failure bursts (as explained in Section 3.4.1 where $^{C}/_{C}$ has the lowest PDL). We also have shown other tradeoffs where single disk repair is faster (*e.g.*, $^{C}/_{D}$ is faster than $^{C}/_{C}$ in Figure 3.6a) but at the cost of slower local pool repair (*e.g.*, $^{C}/_{D}$ is slower than $^{C}/_{C}$ in Figure 3.6b).

Local Failure Probability

Prior sections have established that a catastrophic local pool failure is an *Achilles' heel* of MLEC because it will lead to a huge amount of network traffic with limited available bandwidth. Even worse, when p_n+1 catastrophic local failures happen concurrently, the system will lose data. Thus, we now ask what the probability of catastrophic local failure is in different MLEC schemes.

Figure 3.7 shows that *fortunately the probability of the catastrophic local failure is low in all MLEC schemes.* For example, the probability with $^{C}/_{C}$ and $^{D}/_{C}$ is lower than 0.001% per year. Even better, the probability is almost 0.00001% with $^{C}/_{D}$ and $^{D}/_{D}$. There are two main reasons why the latter is better by orders of magnitude. First, a local-Dp pool has higher durability than a local-Cp pool. This is not only because the local-Dp pool has a higher disk repair rate, but also because it has less high-priority stripes (i.e. stripes that have multiple failed chunks), which can be prioritized and repaired quickly. Accordingly, a local-Dp pool is more durable in our setup. Second, since the local-Dp pool size (120) is larger than the local-Cp pool size (20), the system has fewer local-Dp pools. Thus, the probability of an arbitrary local-Dp pool failing is lower.

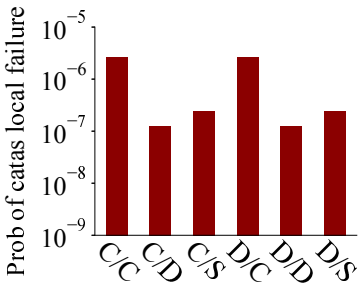


Figure 3.7: **Prob. of catastrophic local failure.**

Additionally, we observe that the catastrophic local failure probability with $^{C}/_{S}$ and $^{D}/_{S}$ is slightly higher than with $^{C}/_{D}$ and $^{D}/_{D}$. This discrepancy arises from the fact that when multiple disks fail, a local-SODp pool typically has more high-priority stripes to repair than a local-Dp pool. This results in longer high-priority repair time and ultimately leads to lower durability. With a large parity number (>2), this durability loss even surpasses the durability gain achieved through SODp's burst tolerance, causing local-SODp pools to exhibit lower durability than local-Dp pools.

3.4.2 Analysis of Repair Methods

Previous section brings the good news that the probability of the catastrophic local failure is low in general. However, as alluded to in Section 3.4.1, the repair will take a large amount of time under a straightforward repair method, R_{ALL}. Thus, in the following sections, we extend the evaluation to include R_{FCO}, R_{HYB} and R_{MIN}'s performance in repairing a catastrophic local pool failure.

Cross-Rack Repair Traffic

We begin with quantifying the cross-rack network traffic of the four repair methods (R_{ALL} to R_{MIN}) in all the six MLEC schemes ($^{C}/_{C}$ to $^{D}/_{S}$), as shown in **Figure 3.4.2** with the findings **F#1-5** described below.

Finding #1: *R_{ALL} is the simplest to implement but results in the most network traffic.* As elaborated in Section 3.2.6, R_{ALL} does not require the network-level and local-level repairers to be aware of each other, and hence simple to implement. As an implication, however, R_{ALL} needs to repair the entire local pool, as opposed to just the failed data in the local pool, which in turn leads to unnecessary work. R_{ALL} also results in much more network traffic in local-Dp ($*/_{D}$) and local-SODp ($*/_{S}$) than in local-Cp ($*/_{C}$) schemes (26,400 vs. 4400 TBs in the figure). This is because local-Dp and local-SODp have a larger local pool to reconstruct (120 disks) than local-Cp (20 disks).

Finding #2: *R_{FCO} significantly improves upon R_{ALL}.* This is because, instead of repairing the entire pool, R_{FCO} only repairs the failed chunks. The reduction is more apparent for local-Dp ($*/_{D}$) schemes due to larger local pool sizes (*e.g.*, from 26,400 to 880 TB in $^{C}/_{D}$). As alluded to in Section 3.2.6, an MLEC deployment with two different vendors might need to implement some APIs to pass failure information across the two layers.

Finding #3: *R_{HYB} (a hybrid local and network repair) further reduces cross-rack repair traffic for local-Dp ($*/_{D}$) and local-SODp ($*/_{S}$) schemes.* For $^{C}/_{D}$ and $^{D}/_{D}$, R_{HYB} only transfers 3.1 TB, much fewer than 880 TB in R_{FCO}. This is because in a local-Dp pool, when p_l+1 disks fail, only a

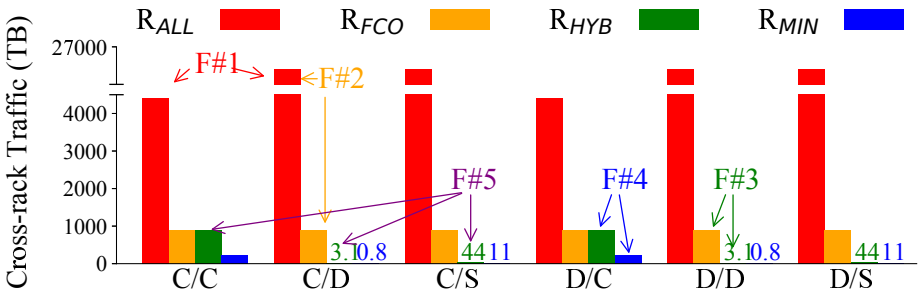


Figure 3.8: **Cross-rack network traffic (§3.4.2).** *The figure shows the cross-rack traffic (in TB) generated by the four different repair methods (R_{ALL} to R_{MIN}) on six MLEC schemes.*

small fraction of affected local stripes are lost local stripes which require network-level repair, and other affected local stripes that have less than p_l+1 chunk failures can be repaired locally. R_{HYB} also reduces cross-rack repair traffic for $^{C}/_{S}$ and $^{D}/_{S}$, from 880TB to 44TB. The reduction is smaller than that for $^{C}/_{D}$ and $^{D}/_{D}$, because a failed local-SODp pool in $^{*}/_{D}$ typically have more lost local stripes than a local Dp pool. For local-Cp ($^{*}/_{C}$) schemes, the figure shows that R_{HYB} does not give advantage over R_{FCO}, because here we inject all the p_l+1 disk failures at the same time. Later in Section 3.4.2, we simulate different timings of failures.

Finding #4: *R_{MIN} provides the minimum cross-rack traffic among the four repair methods.* For all MLEC schemes, R_{MIN} reduces network traffic by 4x or more compared to R_{HYB}, thanks to the opportunistic 2-stage method in R_{MIN}. In this case, for example, instead of repairing all 4 failed chunks from the network level, R_{MIN} only repairs one failed chunk from each 4-chunk-failure stripe (the first stage) and after that rebuilds the three remaining failed chunks locally (the second stage).

Repair Time

As the last section focuses on the network traffic size, we now measure the repair time and describe findings **F#1-3** in **Figure 3.9**. The first two repair methods, R_{ALL} and R_{FCO}, only employ network-level repair, while the last two, R_{HYB} and R_{MIN}, leverage local repairs, depending on the MLEC schemes, as we explain below.

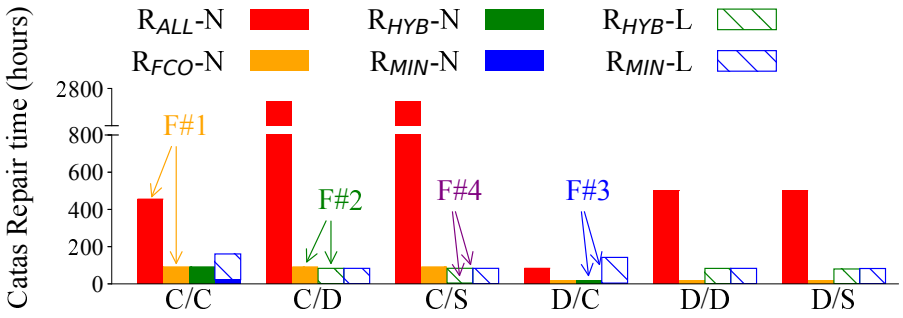


Figure 3.9: **Repair time (§3.4.2).** *The figure shows the network-level (-N) and local (-L) repair time with solid and striped bars, respectively. When the solid bars are not visible, the numbers are very small.*

Finding #1: *R_{ALL} imposes the longest time and R_{FCO} reduces the network-level repair time by 5-30x.* R_{ALL} reconstructs the entire local pool, leading to a slow repair (with network throttling) in all the MLEC schemes. However, R_{FCO} only reconstructs the 4 failed disks, resulting in much faster repair time.

Finding #2: *R_{HYB} reduces network repair time, but induces local repair time.* This is true for $^{C}/_{D}$, $^{D}/_{D}$, $^{C}/_{S}$, and $^{D}/_{S}$, wherein after the affected stripes are partially reconstructed via network repair, the local pool exits the catastrophic state and can be repaired locally. The local repair however can only read from the disks in the local pool (*i.e.*, less parallelism compared to reading from network-wide disks in other racks). For example, on $^{C}/_{D}$, R_{HYB} takes a similar amount of time as R_{FCO} to fully recover the data.

Finding #3: *While R_{MIN} transfers the minimum amount of data over the network, it can take even longer to repair the local pool.* In all the MLEC schemes, as R_{MIN} quickly repairs a less amount of data from the network level, it makes the failed local pool exit the catastrophic state faster, but could take longer time to fully repair all the failed disks. We would like to note that although the total repair time is longer due to local repair, R_{MIN} reduces network contention for foreground I/Os and improves the durability of the system (since the network-level EC is able to tolerate more catastrophic local failures after the fast network-level repair).

Data Durability

Continuing what we built in the last section, we now analyze how different repair time from different methods would affect long-term data durability, as shown in **Figure 3.10** along with findings **F#1-4**. **Data durability** in one year is measured in *the number of nines* which is defined as $-\log_{10}(\text{PDL})$, *e.g.*, 99.999% durability means 5 nines. In addition to the basic setups (§3.3), we also prioritize repairing local stripes with more failed chunks in local-Dp ($*/D$) schemes, and network stripes with more affected local stripes in network-Dp ($D/*$) schemes.

Finding #1: *Compared to R_{ALL}, R_{FCO} increases the durability by 0.9-6.6 nines.* R_{ALL}'s slow repair impacts durability, but R_{FCO}'s faster repair increases the durability. The increase is as high as 6.6 nines in D/D due to two reasons. First, the repair time reduction is large in D/D (Section 3.4.2). Second, when D/D has p_n+1 catastrophic local pools, it might not have any lost network stripe because each catastrophic local-Dp pool only has a small number of lost local stripes, and the probability of a lost network stripe with p_n+1 lost local stripes is as low as 0.03% due to the network-Dp placement. However, R_{ALL} is not able to detect this information as it treats the entire local-Dp pool as lost. But again in R_{FCO}, the network repairer has the knowledge of which exact chunks are lost, and hence it can tolerate the cases where there are p_n+1 catastrophic local pools but no lost network stripes.

Finding #2: *R_{HYB} further increases the durability by 0.6-4.1 nines.* This is because R_{HYB}'s faster repair only rebuilds the lost local stripes. The increase is more apparent in C/D and D/D because each local-Dp pool only has a small portion of lost local stripes (due to the declustered placement). Note that R_{HYB} also increases the durability of local-Cp ($*/C$) schemes although R_{HYB} did not show repair traffic/time reduction in earlier sections. This is because previously we injected all the p_l+1 disk failures at the same time to reflect a catastrophic local pool failure. However, to measure long-term durability (Section 3.3), we let disks fail independently following exponential distribution, and thus disks usually fail at different times. Therefore, some disks could have been partially repaired when a catastrophic local pool failure happens. In such cases, only part of the

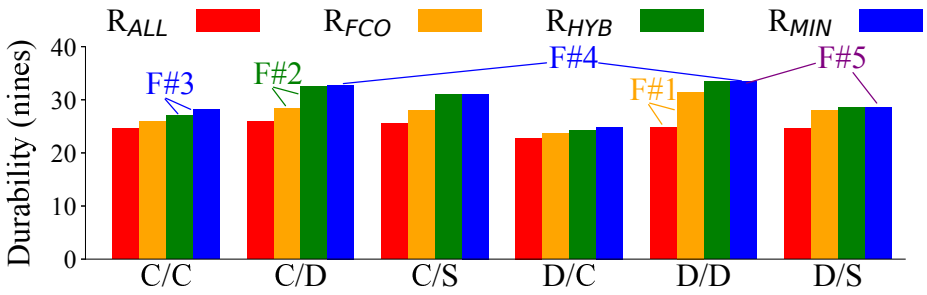


Figure 3.10: **Durability (§3.4.2).** *The figure shows the durability of MLEC schemes under various repair methods.*

affected local stripes are lost local stripes, and R_{HYB} can still deliver the traffic reduction for $*/C$ schemes.

Finding #3: *R_{MIN} further increases the durability by 0.1-1.2 nines.* R_{MIN} further increases the durability since it further decreases the repair time, especially for C/C. However, the increase is small in C/D and D/D. This is because their network repair is already fast (because the local declustered placement results in much fewer lost local stripes that require network repair), and thus the repair is bottlenecked by the time to detect the failure and trigger the repair.

Finding #4: *After all the optimizations, C/D and D/D provide the best durability while D/C provides the worst.* Among all MLEC schemes, C/D and D/D provide the best durability, because a local-Dp pool has higher durability than a local-Cp pool, thanks to the priority reconstruction in local-Dp. Moreover, a catastrophic local-Dp pool has fewer lost local stripes to repair compared to a catastrophic local-Cp pool, resulting in less network repair time, especially under R_{MIN}. Note that although D/D has a higher chance to have p_n+1 catastrophic local pools in a network pool compared to C/D, its disadvantage in terms of the durability is compensated as there are cases when D/D has p_n+1 catastrophic local pools but no lost network stripes, as mentioned in Finding #1 above. On the other hand, D/C provides the worst durability as it can lose data when any p_n+1 arbitrary local-Cp pools in separate racks fail catastrophically, and its benefit of fast repair from network-Dp is bottlenecked by the failure detection time.

Finding #5:: **/S provides lower durability than */D, especially after repair optimizations and when network-level is Dp.* This lower long-term durability in */S is attributed to three key reasons. *Firstly*, as discussed in Section 3.4.1, local-SODp pools tend to exhibit lower durability than local-Dp pools. While this difference may not be substantial for local pools, it becomes more pronounced and is amplified multiple times in the multi-level nature. *Secondly*, in the event of catastrophic local failures, a local-SODp pool has more lost local stripes compared to a local-Dp pool. These lost local stripes cannot be repaired locally and necessitate network repair. Consequently, this extended network repair time under R_{HYB} and R_{MIN} leads to lower durability. *Thirdly*, under network-Dp placement, the increased number of lost local stripes in a local-SODp pool significantly raises the probability of encountering a lost network stripe with $p_n + 1$ lost local stripes, reaching as high as 76%. This is much worse than $^{D}/_{D}$ placement, where the probability is only 0.03%. This heightened likelihood of lost network stripes further diminishes the durability of $^{D}/_{S}$ placements.

3.5 vs. Other EC Schemes

We now evaluate MLEC with SLEC and LRC [89], in terms of their encoding throughput, durability, failure burst tolerance, and repair network traffic.

3.5.1 vs. SLEC

Encoding Throughput

Figure 3.11 shows the single-core encoding throughput (the heatmap color) under various k and p configurations, in the x- and y-axis, respectively. We perform the measurement using the Intel ISA-L tool [22] on a single core of Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz.

Generally, *EC with larger values of k and p has lower encoding throughput*. When more parities (higher p) need to be computed, more computation overheads are introduced. On the other hand, with wider stripes (larger k), the encoding process might not be able to fit the required input

data into CPU cache, which can degrade the encoding throughput [88]. This is a reason why SLEC is hard to scale. With MLEC, we can limit the number of parities and stripe width and attain better durability as we show next.

Durability vs. Throughput

EC designers face performance-durability tradeoffs, for example more parities give higher durability, but lower encoding throughput. **Figure 3.12** quantifies this tradeoff with two main findings, **F#1-2**. Here we show two notable MLEC schemes, (a) $^{C}/_{C}$ and (b) $^{C}/_{D}$, vs. a local (de)clustered SLEC ("Loc-Cp-S" or "Loc-Dp-S", respectively) and a network (de)clustered SLEC ("Net-Cp-S" or "Net-Dp-S"), whose layouts were already presented in Section 3.2.1). We don't compare local SODp here because it's already shown to have lower long-term durability than local-Dp in Section 3.4.1. For MLEC's repair method, we use R_{MIN}, the most optimized one. For fairness, all the points in the figure come from MLEC/SLEC configurations with a capacity (parity space) overhead of roughly 30%.

Finding #1: *For both MLEC and SLEC, higher durability leads to lower encoding throughput.* To achieve higher durability, both MLEC and SLEC need more parities, and to maintain the same capacity overhead, they will need a wider stripe.

Finding #2: *MLEC can provide high durability while maintaining higher encoding throughput.* At low durability (*e.g.*, <20 nines), MLEC's throughput is lower than SLEC, *however* at high durability (*e.g.*, >20 nines), MLEC can maintain almost the same throughput while at the same time

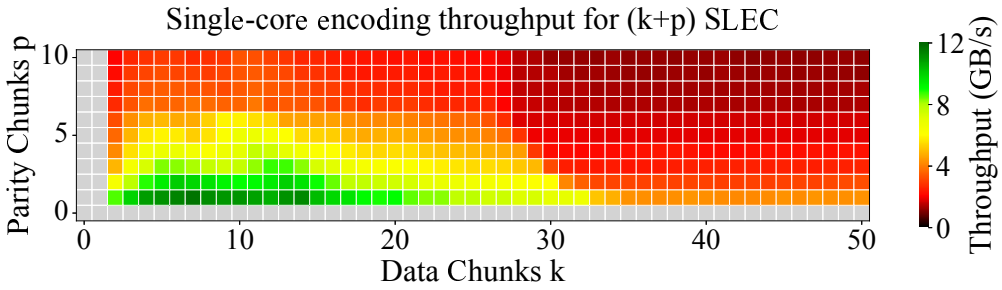


Figure 3.11: **Encoding throughput for various (k+p) (§3.5.1).**

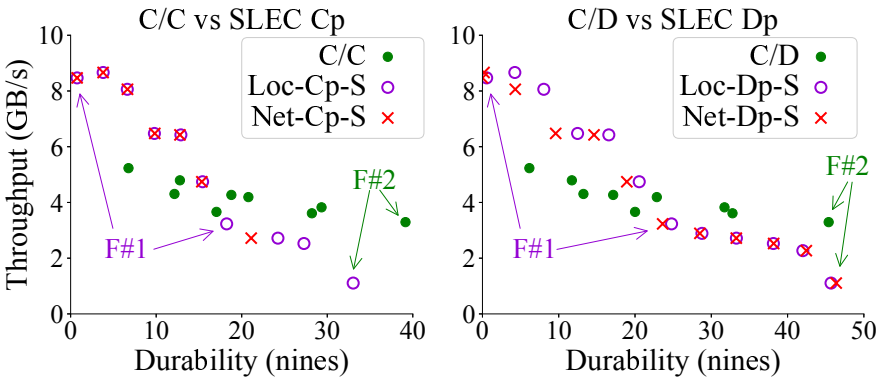


Figure 3.12: **MLEC vs. SLEC durability/throughput tradeoff (§3.5.1).** *A dot represents a specific configuration. For example, the green MLEC $^{C}/_{C}$ and $^{C}/_{D}$ dots come from various configurations such as (5+1)/(5+1), (10+2)/(17+3), and many others. For fairness, all the dots have a configuration with around 30% parity space overhead.*

increasing its durability dramatically, thanks to the two-level protection (p_l and p_n) and our repair optimizations. For example, at the two points pointed by F#2 in Figure 3.12a, a (17+3)/(17+3) C/C can reach 39-nine durability with 3GB/s throughput while a local (28+12) SLEC reaches 33-nine durability with only 1GB/s throughput. Increasing throughput can be done with more CPU cores, but would lead to higher hardware cost, and potentially extra overhead caused by imperfect parallelism.

Durability vs. Update Cost

Here we analyze another tradeoff: the tradeoff between update cost and durability. We compare two kinds of update cost: the cost of disk IO when updating a data chunk, and the cost of cross-rack network traffic required to update a data chunk. **Figure 3.13** quantifies this tradeoff. Again we show two notable MLEC schemes, (a) $^{C}/_{C}$ and (b) $^{C}/_{D}$, vs. a local (de)clustered SLEC ("Loc-Cp-S" or "Loc-Dp-S", respectively) and a network (de)clustered SLEC ("Net-Cp-S" or "Net-Dp-S"). We have two findings:

Finding #1: *MLEC requires same, or sometimes even more disk IO cost for data updates.* As Figure 3.13 a and b show, with the same durability, MLEC introduces same or even more disk IO when updating a chunk. This is because MLEC not only needs to update local parities, but also

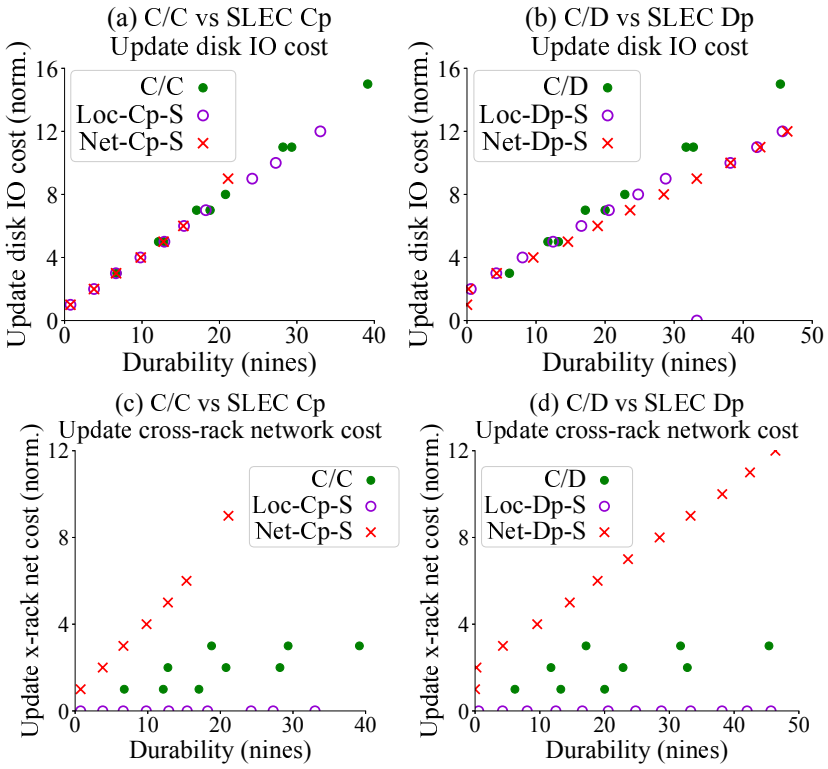


Figure 3.13: **MLEC vs. SLEC durability/update tradeoff (§3.5.1).** *Figures a and b are comparing update cost of disk IO, and figures c and d are comparing update cost of cross-rack network traffic overhead. A dot represents a specific configuration. For example, the green MLEC* $^{C}/_{C}$ *and* $^{C}/_{D}$ *dots come from various configurations such as (5+1)/(5+1), (10+2)/(17+3), and many others. For fairness, all the dots have a configuration with around 30% parity space overhead.*

needs to update network parities, and even double parities (the network parity of a local parity).

Finding #2: *MLEC incurs lower cross-rack network traffic update costs in comparison to network-SLEC.* Although MLEC's data updates may involve increased disk IO, its cross-rack network traffic expenses are significantly reduced when compared to network-SLEC. This cost reduction is a result of MLEC's unique update strategy, where updates for network parity and double parities within the same rack only necessitate the transfer of a single chunk across racks. This chunk is used to update the network parity, and subsequently, the double parities can be computed locally. As a result, MLEC only requires p_n cross-rack network costs to update a single data chunk, and thus requires less update cost than network SLEC when aiming for the same level of durability. It's important to note that local SLEC doesn't incur any network costs but comes at the tradeoff of

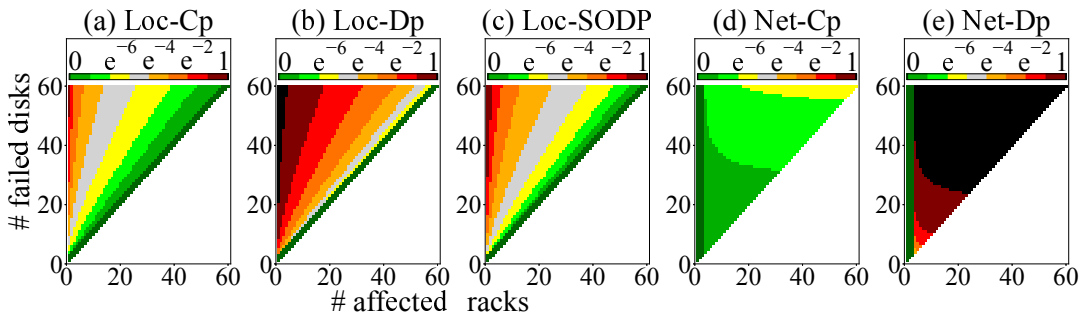


Figure 3.14: **PDL of SLEC under correlated failures (§3.5.1).** *Whose patterns (but not the actual values) can be compared with Figure 3.5. A total of y simultaneous disk failures are randomly scattered across x racks. The square color represents the PDL.*

not being able to tolerate rack failures.

Failure Burst Tolerance

Similar to the MLEC's failure burst analysis earlier in Figure 3.5 in Section 3.4.1, **Figure 3.14** shows the PDL of an (7+3) SLEC under correlated failure bursts with four possible chunk placements (local-Cp, local-Dp, network-Cp, and network-Dp). We describe their pros/cons. Again, by combining the two levels, MLEC hides each of the SLEC's limitations and gains the benefits of the two worlds.

Local SLEC is more susceptible to localized failure bursts. In **Figure 3.14a**, local-Cp SLEC can survive highly scattered failures when no more than $y{=}x{+}p$ failures are scattered in x racks. However, it is susceptible to localized failure bursts since any $p{+}1$ disk failures in the same local-Cp pool can cause data loss. In **Figure 3.14b**, local-Dp SLEC performs even worse under localized failure bursts (high y, low x), since it has a larger local pool and thus has a higher chance to have $p{+}1$ disk failures in the pool, which can lead to data loss. In **Figure 3.14c**, local-SODp falls between local-Cp and local-Dp in terms of performance. This positioning is attributed to its reduced randomness in declustered placement, allowing it to withstand more failure bursts compared to local-Dp.

Network SLEC is more susceptible to scattered failure bursts. In **Figure 3.14c**, network-Cp SLEC performs well under localized failure bursts, and the PDL is 0 when no more than p racks have failures. However, it can lose data under scattered failures (high y, high x), which are more likely to have $p+1$ or more disk failures in the same network-Cp pool. In **Figure 3.14d**, network-Dp SLEC performs even worse under scattered failures, since it can lose data when any arbitrary $p+1$ disks in separate racks fail.

Repair Network Traffic

Lastly, we analyze the repair network traffic. We find that network SLEC requires a huge amount of cross-rack repair network traffic, which can increase with more parities and wider stripes for higher durability. A (7+3) network SLEC requires *hundreds of TB repair network traffic every day*, which can largely interfere user network traffic. On the other hand, MLEC only requires *a few TB repair network traffic every thousand of years*, thanks to both the local protection and our repair optimizations.

3.5.2 vs. LRC

We now perform the same evaluation but with LRC, a popular EC approach that has been extensively studied in recent years [88, 89, 92, 101, 128, 140]. We start with describing the layout differences.

MLEC vs. LRC Layouts

A (k,l,r) LRC, in the first stage, divides k data chunks into l local groups and computes one local parity in each group, and in the second stage, computes r global parities from all the k data chunks [89]. **Figure 3.15** shows a (4,2,2) LRC. We treat LRC as a *one-level placement* EC, but with *two-stage* encoding. Unlike LRC, MLEC might fit better deployments where the two levels are managed by different organizations; for example, a large institution buys RBODs (local EC pools)

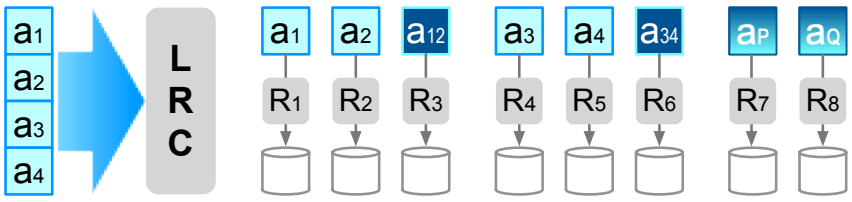


Figure 3.15: **A (4,2,2) LRC (§3.5.2).** *To be compared with MLEC layout in Figure 3.1c. Here, a_P and a_Q are the first and second parities of a_1 to a_4 using specific LRC encoding formulas [89].*

from storage vendors and on top of them manages the network-level EC. LRC on the other hand might be more desirable to deployments that have direct access to and can manage all the disks.

Although MLEC and LRC both perform two stages of coding, they differ in several ways. **(a)** For the top-level encoding, each of MLEC's network parity is computed from only part of data chunks in the network stripe (*e.g.*, back in Figure 3.2.1c, a_{13} is computed from a_1 and a_3), but each of LRC's global parity is computed from all the data chunks in the stripe (*e.g.*, in Figure 3.15, a_P and a_Q are computed based on a_1 to a_4). **(b)** For the bottom-level encoding, MLEC can have multiple parities in each local stripe (*e.g.*, an $*/(4+2)$ MLEC has 2 local parities in a local stripe), but LRC always has one single parity in each local group (*e.g.*, a_{12} in Figure 3.15). **(c)** Regarding the double parity, MLEC always computes double parities from network parities (*e.g.*, back in Figure 3.2.1c, a_P is based on a_{13} and a_{24}), but many LRCs don't do the same (although some of its variants do [101, 140]). **(d)** On chunk placement, MLEC puts one local stripe in a local pool on a separate rack, and can choose de/clustered placement for inter-pool and clustered/declustered/SODp for intra-pool, resulting in six possible schemes. In contrast, LRC usually puts every chunk in a separate rack in a declustered way [89, 92]. Although it's possible for LRC to put one local group in the same rack [88], we are not aware of any existing LRC systems that adopt this.

Durability vs. Throughput

Due to these differences, MLEC and LRC provide different tradeoffs in performance and durability, which we evaluate here. **Figure 3.16** quantifies the durability and throughput tradeoff in MLEC and LRC. Here, we only show declustered LRC ("LRC-Dp") as the most common configuration;

we never found "LRC-Cp." For the MLEC scheme, we only show $^{C}/_{D}$ as it gives the best durability among all the other schemes. Just like previously, we compare various configurations all having capacity (parity space) overhead of around 30%.

Finding #1: *MLEC can provide high durability with higher encoding throughput.* This is because LRC only has one parity in each local group, and thus depends on more global parities to provide higher durability, but again more global parities can degrade the encoding throughput. Moreover, LRC-Dp places chunks in a one-level declustered way just similar to network-Dp SLEC, making it have a similar durability pattern as network-Dp SLEC.

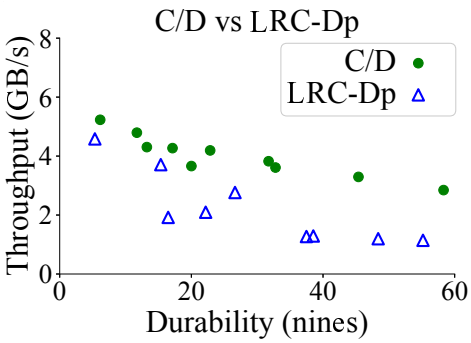


Figure 3.16: **MLEC vs. LRC durability and throughput tradeoff (§3.5.2).**

Finding #2: *The multi-level nature of MLEC allows fewer parities in each level, which then helps MLEC alleviate durability loss due to failure detection time.* Note that when multiple disks fail, a declustered (Dp) pool usually has a very small number of high-priority stripes (*i.e.*, stripes that have multiple failed chunks). Such stripes are prioritized and repaired fast, leading to high durability. This durability benefit increases with more parities and larger pool size (due to an even smaller number of high-priority stripes). *However*, because there is a 30-minute failure detection time (commonly used [88, 89, 92]), the increased durability diminishes and is not fully reflected. This is the reason why both $^{C}/_{D}$ and LRC-Dp lose some durability. But, since $^{C}/_{D}$ performs declustered parity in a local pool (which is smaller than LRC-Dp's pool) and has fewer parities at each level, the declining durability is less severe compared to LRC-Dp.

Furthermore, our multi-level-aware repair optimizations further improve $^{C}/_{D}$'s durability, helping it reach high durability with higher encoding throughput compared to LRC-Dp. We also note that if failure detection time is reduced significantly (*e.g.*, to 1 minute), LRC-Dp's durability could be similar or slightly better than MLEC, which we will explore in the future. However, such fast failure detection is *not* realistic as disks often fail transiently and should not be rebuilt hastily.

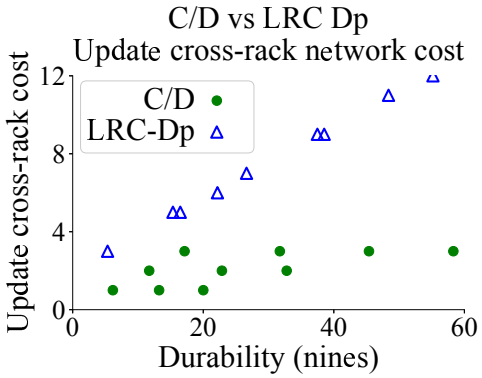


Figure 3.17: **MLEC vs. LRC durability and update cost tradeoff (§3.5.2).**

Durability vs. Update Cost

We also analyze the tradeoff between durability and update cost. Here we focus on comparing the udpate cost of cross-rack network traffic, since LRC is typically designed for network-level erasure.

Figure 3.17 quantifies the durability and update cost tradeoff in MLEC $^{C}/_{D}$ and LRC-Dp. As we can see, LRC incurs significantly higher cross-rack network traffic when updating a chunk. This is because LRC needs to update all global parities across different racks, and LRC needs to have more global parities in order to achieve high durability. On the other hand, MLEC eliminates this overhead by using a smaller p_n while maintaining same high durability.

Failure Burst Tolerance

Just like in Section 3.5.1, we now measure the PDL of LRC under correlated failure bursts, as shown in **Figure 3.18**, using a (14, 2, 4) LRC-Dp. We pick this LRC configuration as it has a similar throughput rate as our (10+2)/(17+3) MLEC configuration, based on previous section's findings. However, we emphasize that their actual values in the figure should *not* be directly compared with the MLEC's PDL results shown earlier in Figure 3.5, as it would be an unfair comparison since one could always increase the number of parities and stripe width to get better durability/PDL. We already compare MLEC vs. LRC fairly in the previous section.

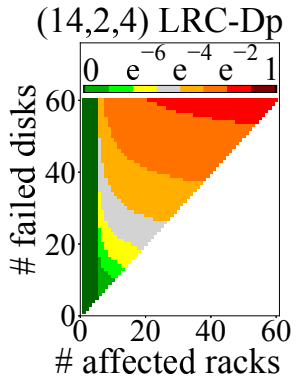


Figure 3.18: **PDL pattern of LRC under correlated failures.** *Whose patterns can be compared with Figures 3.5 and 3.14.*

LRC-Dp is susceptible to highly scattered failure bursts. Since LRC-Dp places chunks in a one-level declustered way across racks, its failure burst tolerance pattern is similar to network-Dp SLEC (shown earlier in Figure 3.14d), and can lose data under highly scattered failure bursts. However, as discussed earlier in Section 3.4.1, MLEC in general is robust to highly scattered failures.

Repair Network Traffic

Finally, we analyze the repair network traffic of LRC-Dp. We find that LRC-Dp's repair network traffic is less than network SLEC since most failures can be repaired using the local group with less chunks. However, every repair still needs to read and write over the network, which can still lead to lots of repair network traffic. Therefore, LRC-Dp still requires tens or even hundreds TB of cross-rack network traffic every day. On the other hand, MLEC requires minimal network traffic on average, since most failures are repaired locally.

3.6 Discussions

3.6.1 Takeaways

We believe our findings will provide guidance for large-scale storage architects to choose ideal configurations for their particular environments and requirements. Here are some examples:

1. For institutions without large storage devops teams, they can buy RBODs from storage vendors, build MLEC on top easily, and choose RepairALL with some sacrifice in performance and durability.

2. Those with more flexibility can optimize their MLEC with our advanced repair techniques such as RepairMIN.

3. Systems detecting frequent occurrences of correlated failure bursts should utilize C/C to get better failure burst tolerance.

4. Systems with rare failure bursts should use C/D or D/D to get higher durability under independent failures.

5. Systems with lower durability requirements should choose SLEC for better performance.

6. Systems that prioritize high durability (e.g. certain HPC systems where any lost chunk can make PBs of correlated data useless) should choose MLEC to minimize overheads.

We also hope our paper will encourage more research on MLEC for ML/HPC/cloud systems. For example, when offloading analytics to computational storage in HPC systems, efficiently mapping logical objects to physical blocks in erasure-coded systems poses a challenge. MLEC adds complexity to this problem due to its layering, which can be explored in future work.

3.6.2 Reproducibility of the Study

This project has been ongoing for over 1.5 years involving a group consisting of a theorist, a large-scale system administrator (>20k disks), a developer/maintainer, an architect, and academics. The diversity of roles ensures that the assumptions in our work closely match real-world scenarios.

The group has verified many important failure cases, possible/practical repair methods, and actual system architectures.

We together scrutinized the correctness of every scheme and method. We went back and forth to fix errors/misassumptions. Our multiple methodologies verify each other. For example, when the simulator result didn't conform with the theoretical model, the theorist and simulator developer went back and forth in multiple iterations to resolve discrepancies.

Our results are fully reproducible. We have released our source code [27] on Github and a detailed evaluation artifact [26] on Chameleon Trovi.

3.7 Conclusion

We have provided comprehensive design considerations and analysis of MLEC at scale. MLEC can be designed in multiple dimensions. We have quantified their performance and durability with various evaluation strategies, and have shown which MLEC schemes and repair methods can provide the best failure tolerance and greatly reduce repair network traffic. We have also shown that MLEC can provide high durability with higher encoding throughput and less repair network traffic over other EC schemes.

CHAPTER 4

AN EMULATION APPROACH FOR COST-EFFECTIVE EVALUATION OF MULTI-LEVEL ERASURE CODED STORAGE AGAINST DEEP LEARNING WORKLOADS

With the rapid development of deep learning, one prevalent type of workload in large-scale erasure-coded data centers is deep learning. Deep learning training requires vast amounts of data to ensure model accuracy, and this data is often too large to fit into local storage, necessitating storage and retrieval from remote erasure-coded systems [66, 103, 148, 162, 167]. As a result, it is critical to evaluate the performance and identify bottlenecks in MLEC storage systems when serving deep learning workloads.

However, conducting such evaluations poses two significant challenges for the academic community. First, deep learning workloads typically require GPUs, which are scarce in academic clouds and costly to rent from commercial providers. Second, MLEC storage requires a substantial number of disks to configure, while academic clouds often lack flexible, configurable disk arrays on individual machines. For example, in Chameleon Cloud [10, 97], most machines have only a single disk. Although a small number of machines come with pre-configured 7+1 RAID disk arrays, these arrays are not easily configurable for custom erasure coding and are insufficient for wide-stripe erasure coding setups.

To overcome these challenges in evaluating MLEC storage for deep learning workloads, we introduce an emulation-based evaluation approach. We have developed two emulation tools: GPEMU, a GPU emulator that allows for efficient prototyping and evaluation of deep learning systems without the need for real GPUs, and MLECEmu, a storage emulator that can cheaply emulate the performance of multi-level erasure-coded storage systems without using physical disk arrays.

GPEMU is based on the observation that for many deep learning research projects, including the evaluation of MLEC against deep learning workloads, real GPUs are not necessary. In such

experiments, the focus is on GPU performance rather than the actual computation results, and GPU performance is predictable. Similarly, MLECEmu is motivated by the fact that when using MLEC storage for deep learning training, the throughput of the storage system is the critical factor. This can be effectively emulated by throttling in-memory `tmpfs` loop devices.

The two emulators are designed to ensure broad applicability. GPEMU provides extensive support for emulating a wide range of models, GPUs, and batch sizes, with features such as time emulation, memory emulation, distributed system support, and GPU sharing support. MLECEmu supports various code parameters and chunk placement schemes.

With these two efficient tools, users can effectively identify system bottlenecks during emulations of MLEC storage for deep learning workloads, providing valuable insights into system performance. Furthermore, the emulators enable users to quickly demonstrate the benefits of new system optimizations across a wide range of research areas.

In this chapter, we introduce the design of GPEMU in Section 4.1. We demonstrate the capabilities of GPEMU by reproducing the main results from nine recent publications in Section 4.2 and prototyping three new micro-optimizations in Section 4.3. Next, we present the design of MLECEmu in Section 4.4, followed by a demonstration of how GPEMU and MLECEmu can be used together to evaluate the performance of MLEC storage for deep learning workloads in Section 4.5. We conclude this chapter in Section 4.6.

4.1 GPEMU Design Features

We first introduce the design of GPEMU. To prototype and evaluate without GPUs, people resort to many simulation approaches [56, 79, 98, 104, 112, 138] but impedes end-to-end/full-stack experiments. Others try "emulation" approaches as summarized in **Table 4.1**. Silod [167] for example, profiled model compute time on the expensive V100 GPU and emulated it on a cluster of cheaper K80 GPUs to evaluate GPU cluster scheduling. MLPerf [28] and DLCache [95] provide a host-side emulator that only performs compute time emulation. As hinted in the table, these approaches

Features	G TTT MM DDD S F CTP CP GNJ S	#DL	#GPU	#BS
Silod [167]	. x.. .. xxx .	8	1	N/A
DLCache [95]	x x..	2	1	5
MLPerf [28]	x x.. .. x.. .	3	2	1
GPEMU	x xxx xx xxx x	36	6	256

Table 4.1: **Features and configurations supported by prior GPU emulators and GPEMU (§2).** *The features' abbreviations read from top to bottom. Labeled by "x", supported features are compared across five categories: (1) GPU-Free (**GF**); (2) Time emulation: Compute (**TC**), Data Transfer (**TT**), and Preprocessing (**TP**); (3) Memory emulation: GPU Memory Consumption (**MC**) and Pinned memory (**MP**); (4) Distributed system support: Multi-GPU training (**DG**), Multi-Node (**DN**) training, and Multi-Job scheduling (**DJ**); (5) GPU Sharing Support (**SS**). Furthermore, we evaluate the configurations that they support, including the number of DL models (**#DL**), GPU models (**#GPU**), and batch sizes (**#BS**).*

lack vital emulation components and support only limited configurations and experiments.

To the best of our knowledge, GPEMU is the first advanced GPU emulator that provides various time emulation (compute time, data transfer time, preprocessing time), memory emulation (GPU memory consumption, pinned memory), distributed system support (multi-GPU training, multi-node training, multi-job scheduling), and GPU sharing support. In addition, to date GPEMU is the first emulator that supports the largest number of DL models, GPU models and DL configs, as shown in the right hand side of Table 4.1. Below we detail the GPEMU design and various emulation challenges that we have addressed.

4.1.1 Time Emulation

Time emulation is a feature where the emulator "fakes" the GPU-side operation with a simple sleep timer. While this sounds simple, providing an accurate time emulation requires understanding of the various **steps** of DL workloads: **(1)** Reading samples; **(2)** Preprocessing; **(3)** Data Transfer (from host memory to GPU memory); **(4)** Forward Propagation; **(5)** Backpropagation [75]. It's also worth noting that some frameworks, such as DALI [30], allow for the preprocessing step to be shifted from the CPU to the GPU, altering the sequence of steps 2 and 3.

To emulate DL workloads without actual GPUs, we replace GPU-related steps (steps #3–5,

and Step 2 if GPU-based) with simple `sleep(T)` calls, where T represents the projected time for each specific step. This method effectively emulates the wait time for data transfer between host memory and GPU memory, as well as the completion of GPU preprocessing and model computation. We encapsulate these time emulation methods into a Python package, providing convenient APIs such as `emuForward(config)` for users to easily emulate the corresponding operation. The config parameter encompasses workload-specific details such as the DL model, batch size, and GPU model. These factors are pivotal as they influence the predicted operation time.

We also need to support both synchronous (`sync`) and asynchronous (`async`) modes. In `sync` mode, the sleep-based emulation blocks until the sleep duration has elapsed. In `async` mode, the emulation is non-blocking and can be used to simulate operation pipelining optimizations found in PyTorch [36] and TensorFlow [116]. For PyTorch, this is achieved using the `asyncio` library. In TensorFlow, the `sleep` is integrated into the computation graph. It's crucial in `async` mode to maintain dependencies between operation emulations to prevent unexpected reordering or pruning in asynchronous execution. For instance, we ensure that the forward propagation emulation (Step 4) only starts after the input tensor has been transferred to the GPU (Step 3).

Finally, accurately predicting the emulated sleep time for each step, tailored to the user's specific configuration (model, batch size, and GPU type), is essential. The subsequent subsections delve into this aspect for each step.

2.1.1 COMPUTE/PROPAGATION TIME: We first predict GPU compute time for forward and backward propagations (Steps 4 and 5), a critical aspect of DL workloads. Fortunately, they are notably predictable with minimal variability, since the compute process involves a predefined sequence of tensor operations and has no conditional branches [81, 159]. For instance, as illustrated in **Figure 4.1a**, training ResNet50 on a NVIDIA V100 GPU for 1000 batches with a batch size of 128 shows highly consistent forward and backward propagation durations. The only exceptions are the first 1-2 batches, which typically take longer due to initialization.

To predict per-batch GPU compute time, we profile various models, GPU types, and batch

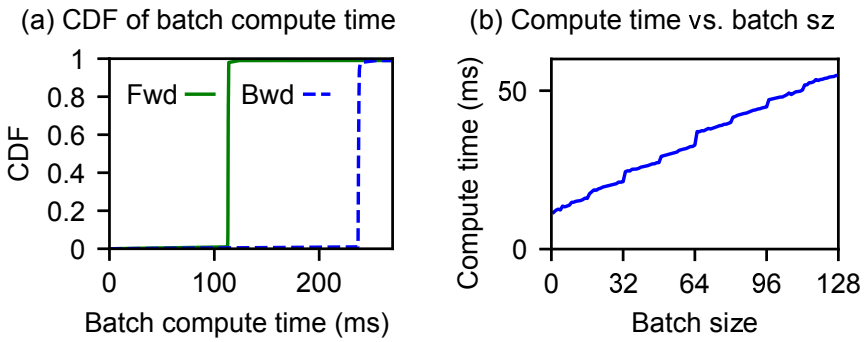


Figure 4.1: **Compute time's pattern (§2.1.1).** *The figures show that (a) compute time is consistent within the same setup, (b) compute time doesn't always linearly correlate with batch size.*

sizes. We then calculate the average compute time, excluding outliers during initialization, and store this data in a database. When users emulate training for a specific configuration, GPEMU retrieves the profiled compute time from the database for that setting.

Some might think GPU compute time linearly correlates with batch size, allowing for straightforward projections from just two profiled batch sizes. However, this holds true primarily for compute-heavy models like ResNet50. For compute-light models like AlexNet, the relationship is more complex and less predictable. In the case of AlexNet on a P100 GPU, as shown in **Figure 4.1b**, the correlation appears as a piecewise linear pattern rather than a simple linear one. Consequently, we employ linear projection methods solely for compute-heavy models with clear linear relationships. In contrast, for compute-light models, we extensively profile across a broader range of batch sizes. Additionally, we offer users a profiling tool for their custom settings (§4.1.6).

2.1.2 DATA TRANSFER TIME: We also predict the time required to transfer the input tensor from host to GPU memory. This aspect is often overlooked by other emulation tools [28, 95], yet its importance varies depending on the model and GPU. For instance, as indicated in **Figure 4.2**, in larger models like ResNet50 where GPU compute time predominantly dictates performance, the transfer time can be relatively negligible. However, for computation-light models such as AlexNet, the input transfer time becomes more significant, reaching as much as 20% of the GPU compute time.

Like GPU compute time, the host-to-GPU data transfer time shows minimal variability for a fixed-sized tensor on a specific GPU, and it typically has a linear relationship with the data volume. Therefore, we predict the input transfer time based on this linear relationship.

2.1.3 PREPROCESSING TIME: Most existing DL frameworks perform data preprocessing (Step 2) on CPUs, thus no emulation is needed (the preprocessing still runs on CPUs). However, some DL libraries, such as DALI and FFCV, support offloading preprocessing to the GPU [30, 105], and GPEMU also supports this. The challenge is that *GPU-side preprocessing time is often not constant*. For instance, in **Figure 4.3**, we display the CDF of two preprocessing operations, normalization and decoding, from FFCV and DALI, respectively. Operations like normalization exhibit smaller variability, as their computational complexity remains consistent across batches. However, other operations like decoding vary significantly by batch, depending on factors like file format, compression quality, and image complexity.

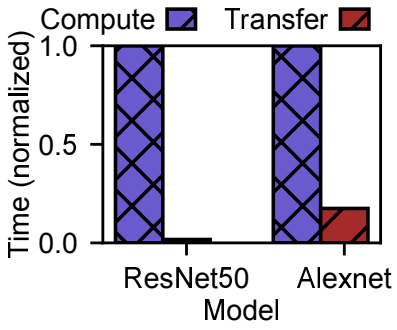


Figure 4.2: **Amount of data transfer time (§2.1.2).**

To accurately emulate GPU-side preprocessing time, we begin by profiling the time distribution for various operations across different batch sizes, storing this data in a database. During emulation, rather than relying on a fixed average value, we randomly draw preprocessing times from this distribution. It's worth noting that our profiling for preprocessing is dataset-based rather than model-based, as many models often share the same preprocessing operations when training on the same dataset.

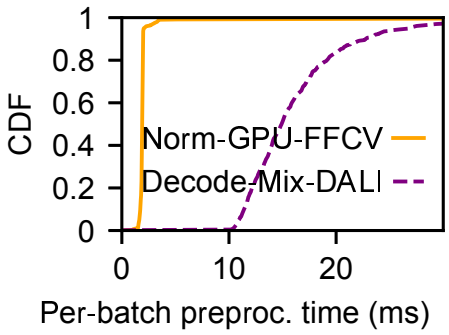


Figure 4.3: **CDF of preprocessing time (§2.1.3).**

Additionally, we've observed that the time cost of certain operations, such as decoding in DALI, also changes with the number of CPUs used. This is because DALI's decoding involves a

mix of CPU and GPU processing. Since the mix is within the DALI native code and challenging to decouple, we emulate both CPU and GPU processing in this operation. To correctly emulate its time cost, we profile its time distribution for various CPU counts. To account for CPU consumption by decoding, we use busy-waiting loops during the emulation.

4.1.2 Memory Emulation

Next, we discuss the need for memory emulation. We divide this into two features: tracking GPU memory consumption and pinning host-side memory.

2.2.1 GPU MEMORY CONSUMPTION: Not all DL workloads can run on a specific GPU; a job's GPU memory requirement might exceed the GPU's memory capacity. To *precisely* emulate if a DL workload can fit into a GPU's memory capacity, it's crucial to understand three types of memory usage in DL jobs: (1) compute peak, (2) model persistent, and (3) preprocessing memory usages.

Compute peak memory usage means the maximum memory to hold all intermediate results produced by model computation during the propagation phase. Persistent memory usage pertains to storing model parameters over time and plays a role in emulating GPU sharing (§4.1.4). For these two types of usages, we analyze how a DL model uses memory during propagations. Fortunately, they follow a repeating pattern, as seen in **Figure 4.4a**. This happens because each batch goes through the same calculations and memory (de)allocations. We also found that both of these usages remain consistent across different GPUs and demonstrate a strong linear correlation with batch size. To mimic this predictable behavior, we profile both usages after processing a few batches across different models. This data serves as the basis for predicting memory usage for configurations that we have not profiled yet.

For preprocessing memory usage (if done on the GPU side like in DALI), we also analyze its behavior as illustrated in **Figure 4.4b**. The memory consumption keeps growing and reaches its maximum after a significant number of batches and never decreases afterward. This is because DALI does not free up the preprocessing memory during training to avoid the costly overhead of

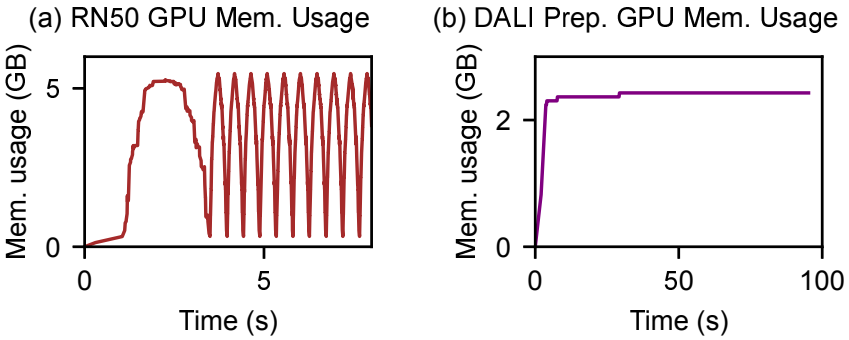


Figure 4.4: **GPU memory consumption over time (§2.2.1).** *The figures show the GPU memory consumption of propagation and GPU-driven preprocessing during DL training.*

freeing and reallocating memory. The maximum memory usage is typically reached when a batch of images that require extensive preprocessing is processed. To accurately mimic this behavior, we profile the highest memory usage for preprocessing after a few epochs, ensuring that it reaches its peak.

2.2.2 PINNED MEMORY: To ensure accurate memory emulation, we need to emulate pinned memory at the host that is managed by CUDA. In real GPU runs, Direct Memory Access (DMA) is vital for efficient data transfer from the host to GPU memory. During this data transfer process, CUDA first allocates a pinned (page-locked) host memory region, copies the host data into this pinned region, and subsequently transfers the data from the pinned region to the GPU memory [20].

Failure to properly emulate the use of pinned memory *can lead to inconsistent performance* compared to real GPU runs. For instance, we observed differences in AlexNet's epoch time with various PyTorch setups as shown in **Figure 4.5**. When the emulation does not involve pinned memory (the orange bar), the epoch time can be up to 20% longer than in real GPU runs (vs. the red bar). After investigating PyTorch internals, we discovered that this additional time arises from Python garbage collections (GC) within the main training loop when the pageable input data is dereferenced. This overhead does not occur in real GPU runs because GC takes place in the background workers after the pageable input data is copied to pinned memory. CUDA

refrains from freeing the pinned memory during training to avoid the overhead of reallocation, which eliminates the need for Python GC within the main training loop.

For these reasons, we need to support pinned memory emulation. Since the host-side pinned memory was initially managed by CUDA, which does not run on CPU nodes, we developed our own pinned memory manager. Our manager employs the `mlock()` system call function to allocate genuine page-locked memory on the host. We meticulously mimic CUDA's memory management to allocate the same amount of space and avoid deallocation. As indicated by the green bar in **Figure 4.5**, our pinned memory emulation successfully resolves the GC stall problem.

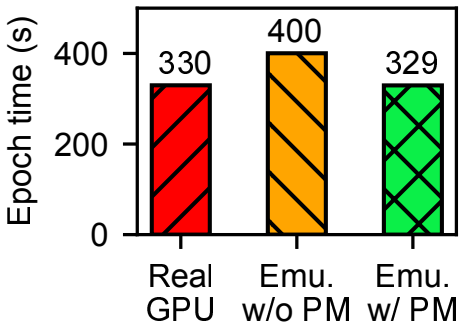


Figure 4.5: **Impact of emulating pinned memory (§2.2.2).**

4.1.3 Distributed System Support

Next we discuss how to emulate DL workloads in distributed setups, covering multi-GPU, multi-node, and multi-job (cluster) scheduling.

2.3.1 MULTIPLE-GPU (SINGLE-NODE) TRAINING: When emulating multiple GPUs within the same machine, we opt for PyTorch's DataParallel (**DP**) module due to its simplicity and convenience. DP divides a batch of input data into multiple smaller chunks, with each chunk sent to a different GPU for training. Since our time and memory emulations are robust and flexible, the timing and memory usage can be adapted directly with the smaller chunk size via the input configuration (*e.g.*, 2x smaller chunk results in ∼2x faster compute time).

2.3.2 MULTI-NODE TRAINING: When emulating distributed DL training across multiple nodes, we leverage PyTorch's Distributed DataParallel (**DDP**) module [108]. Initially, we ran our time and memory emulations on each node. However, we found that this naive approach caused each node's training to operate without proper synchronization with other nodes. This discrepancy arises because the gradient synchronization, originally performed within the backward function, has been

replaced with our sleep operation, leading to a lack of coordinated training across nodes.

A straightforward solution would be to manually initiate gradient synchronization based on CPUs once the backward emulation is completed. However, we observed that after implementing this solution, the batch training time in our emulation becomes notably longer compared to real GPU training. This can be attributed to two primary reasons. First, gradient synchronization requires computing the average of gradients across all nodes using AllReduce operations, which is considerably less efficient on CPUs compared to real GPUs. Second, the DDP framework in PyTorch inherently overlaps gradient reduction with the backward pass, thus masking the communication overhead.

Accurately emulating this overlap is challenging, as gradient reduction occurs when a bucket of gradients is ready. Achieving precise emulation would require a finer-grained profiling of time emulation for operations during the backward pass, which is more complicated. Therefore, we opted for a simpler solution: we perform an `AllReduce` operation on a basic tensor after the backward pass is completed, which synchronizes the training across all nodes while incurring minimal time overhead. While this solution may not precisely replicate the network communication aspect, it allows us to make multi-node distributed training emulation functional. This approach is already quite effective for emulating evaluations that are less concerned about gradient communications, as later shown in Section 4.2.4.

2.3.3 MULTI-JOB (CLUSTER) SCHEDULING: We also offer support for emulating multi-job (cluster) GPU scheduling, catering to both custom schedulers and Kubernetes environments, which brings different challenges.

First, custom schedulers like Synergy [112] rely on predefined cluster configuration files containing detailed resource information for the GPU cluster. We enable GPU scheduling emulation by providing emulated GPU availability information within these configuration files. This allows the scheduler to continue scheduling DL jobs using its original scheduling algorithm. Each DL job runs within a container that already integrates GPEMU and executes the emulation with the

allocated GPUs' configurations.

In contrast, Kubernetes scheduling operates differently. It discovers the availability of custom resources such as GPU on nodes through the custom device plugin [14]. To accommodate this, we introduced a new resource type called "`emuGPU`" (emulated GPU) and developed a custom device plugin specifically for it. Each `emuGPU`'s information is represented as a file on the worker node. The `emuGPU` device plugin runs on each worker node, identifies the corresponding files, and reports this information to the Kubernetes scheduler. Based on the reported data, the Kubernetes scheduler can schedule `emuGPU`s using the same algorithm it employs for real GPUs. When a job is scheduled, the device plugin receives the `emuGPU` allocation decision from the Kubernetes scheduler and mounts the allocated `emuGPU` files to the job's container. The job can subsequently access the emulated GPU information from these files and emulate DL workloads accordingly.

4.1.4 Sharing Support

DL workloads often need to time-share the GPU, a technique commonly used in both academia [81, 156, 163] and industry [45, 46]. They typically time-slice the GPU at the granularity of a batch or an inference request. Hence, we build time-sharing support in GPEMU for both single-node setups and Kubernetes environments.

We start with single-node configurations. The emulation poses two key challenges. First, we must effectively coordinate the time emulation of various DL applications sharing the GPU to ensure that only one application uses the GPU at any given moment. Second, we need to emulate GPU memory usage during this sharing process. If the combined usage of the model persistent memory of waiting applications and the compute peak memory of the running application surpass the GPU's memory capacity, the application will crash.

Based on these considerations, we've developed an *emulation sharing manager*. DL applications first register with their model persistent memory usage. Registered applications seeking GPU access send time emulation requests to the manager via RabbitMQ [37]. When dealing with

a shared GPU, the manager handles one request at a time, sleeping for the requested duration and returning a completion response. The request is rejected if the combined memory usage exceeds the GPU memory capacity.

For Kubernetes setups, to mimic real GPU time sharing on Kubernetes [46], we create multiple replica files for each `emuGPU`. This allows our device plugin to allocate them to multiple DL job containers. Additionally, we deploy a RabbitMQ broker and a sharing manager for each node as Kubernetes daemonsets, facilitating communication and emulating GPU time-sharing management.

4.1.5 Extensibility

Although we have covered a large variety of popular models and GPUs, we believe it is important to continuously extend GPEMU to support an even broader range of deep learning models and GPU architectures. Fortunately, adding supports for new models or GPUs is convenient with GPEMU. We offer users a Profiler tool and an extensible library to profile and use new models and GPUs with ease. The profiling process typically requires only 1-2 hours for a new model-GPU pair, and this is a one-time effort. Once profiled, the data can be added to our library for repeated use, benefiting not only the initial user but also others for future experiments.

The challenge of profiling more configurations is not unique to GPEMU, but same for other simulators and emulators, including those in the storage or networking worlds. For example, SSD simulators (e.g. SSDSim[87]) and emulators (e.g. FEMU[107]) also need to profile new SSD models.

We believe our Profiler is also a contribution, as it comes with many features to profile various metrics such as model compute time, CPU-GPU data transfer time, and GPU memory consumption. The features can also be extended in the future to profile more fine-grained metrics like layer-level compute time and inter-GPU communication time. With this Profiler, we hope the community can work with us together to cover more models and GPUs.

(a)	LOC
Emulation lib	916
K8s device plugin	488
Profiler	1482
PyTorch integration	79
TF integration	20
DALI integration	46
Total	3031

(b) Reimpl.	LOC
CoorDL	2756
LADL	96
Muri	298
Micro-Opt.	
SFF	45
Async Batch	1171
File grouping	695
Total	5061

Table 4.2: **Implementation efforts (§2.5).** *The left table shows the LOCs for implementing* GPEMU. *The right table shows the LOCs for re-implementing work from existing papers and for implementing our new micro-optimizations.*

4.1.6 Implementation Efforts

Our entire effort is quantified in **Table 4.2a**, a total of **3031 LOC** for the GPEMU implementation, comprising an all-in-one emulation library (in Python) for time emulation (§4.1.1), memory emulation (§4.1.2), multi-GPU and multi-node training (§4.1.3.1, §4.1.3.2), and sharing support (§4.1.4), a Kubernetes device plugin for multi-job cluster scheduling and sharing support (§4.1.3.3, §4.1.4), and a Profiler for profiling data used for time and memory emulation (§4.1.1, §4.1.2). Finally, by adding 20-80 LOC "hooks," we easily integrated these features into various platforms such as PyTorch, TensorFlow, and DALI.

Furthermore, for showing many case studies in the next two sections, we wrote another **5061 LOC** (**Table 4.2b**) for re-implementing existing work from scratch (either because they are not available or not fully functional) and for adding new micro-optimizations. For example, we reimplemented CoorDL (§4.2.4), LADL (§4.2.4), and Muri (§4.2.6). We also wrote three micro-optimizations: small-file first (SFF) caching policy (§4.3.1), asynchronous batch reading (§4.3.2), and random-class file grouping (§4.3.3).

We have open-sourced our code, data, and other artifacts on Github ([19]). All our experiment results are reproducible, and we will submit our experiments for reproducibility evaluation if the paper gets accepted. We hope we and the community can work together to extend GPEMU.

	T C	T T	T P	M C	M P	D G	D N	D J	S S
§4.2.1 DataStall [113], VLDB '21	x	x	.	x	x	x	x	.	.
§4.2.2 TF-DS [54], SoCC '23	x	x	.	x	.	x	.	.	.
§4.2.2 FastFlow [145], VLDB '23	x	x	.	x	.	.	.	.	.
§4.2.3 FFCV [105], CVPR '23	x	x	x	x	x	.	.	.	.
§4.2.4 LADL [162], HiPC '19	x	x	.	x	x	x	x	.	.
§4.2.5 Synergy [112], OSDI '22	x	x	.	x	x	x	.	x	.
§4.2.5 Allox [104], EuroSys '20	x	x	.	x	x	.	.	x	.
§4.2.6 Salus [163], MLSys '20	x	x	.	x	x	.	.	.	x
§4.2.6 Muri [173], SIGCOMM '22	x	x	.	x	x	.	.	.	x

Table 4.3: **GPEMU features for paper reproductions (§3).** *The features' short names are identical to the names in Table 4.1.*

4.2 Case Studies of GPEMU's Supported Research

To demonstrate GPEMU's versatility and capabilities, we reproduced experiments from nine papers [54, 104, 105, 112, 113, 113, 145, 162, 163, 173], each utilizing different GPEMU features. **Table 4.3** provides a comprehensive overview of GPEMU's design features used (labeled by "x") in each reproduction. These experiments encompass a wide range of complexities, from single-node training to multi-node distributed training, and even a cluster configuration of 13 nodes for GPU scheduling emulation.

The papers also span a variety of analyses and optimization techniques pertinent to deep learning systems, including data stall analysis in DL training (§4.2.1), optimizations in single-node training data loaders (§4.2.3), disaggregating data preprocessing tasks across remote workers (§4.2.2), caching optimizations in distributed DL training (§4.2.4), resource allocation and scheduling optimization within GPU clusters (§4.2.5), and GPU sharing along with its associated optimizations (§4.2.6).

The advantage of GPEMU over existing emulators can be seen by joining **Table 4.1** with **Table 4.3**. If an emulator in **Table 4.1** doesn't support the features needed in **Table 4.3**, then it means this emulator cannot evaluate the corresponding paper. For example, MLPerf [28] doesn't support the multi-job scheduling (DJ) feature and thus cannot evaluate Synergy [112] and Allox [104] papers in **Table 4.3**.

A primary usage of GPEMU is for faster prototyping and evaluation of systems related research in the area of deep learning, given the scarcity of GPU resources. Therefore, our evaluation focused on validating that GPEMU can effectively replicate the patterns observed in the original papers by comparing our results with their figures. Additionally, we also evaluated GPEMU's accuracy by comparing GPEMU 's results with actual GPU runs for some of the experiments. For example, we presented the accuracy of epoch time compared to real GPU runs in **Figures 4.5** and **4.9** with satisfying results. While we are pleased with the accuracy achieved, we emphasize that our intent is not to advocate for GPEMU as a complete replacement for actual GPU usage.

In the following subsections, we illustrate how GPEMU can support different types of deep learning system research.

4.2.1 Data Stall Analysis

A common use case for our GPEMU is analyzing system behaviors with varying DL training workloads. Here, we reproduce the **DataStall**$_{\text{VLDB22}}$ [113] paper ("DS" for short). The DS paper extensively studied the impact of input data pipelines on training durations for popular deep learning models. *They found that data stalls, referring to time spent waiting for data fetching and CPU preprocessing, accounted for the majority of training time in numerous instances.* We demonstrate how GPEMU can be used to reproduce this conclusion without the need for real GPUs.

The first step in the DS paper was measuring the fetch stall percentage (defined as the time spent waiting for data fetching divided by the total training time) when only 35% of data can be cached in memory. In **Figure 4.6a**, we successfully reproduced these measurements for 6 models with GPEMU. We made two key observations from the figure. First, *the pattern is consistent*. Fetch stalls are common, with varying percentages (y-axis) across different models. AlexNet, with its lightweight computations, spends more time waiting for data, resulting in the largest fetch stall. Conversely, Vgg11, with heavier computations, shows better performance as most data fetch time is masked by GPU computation time due to prefetching. Second, *the exact values differ, which is*

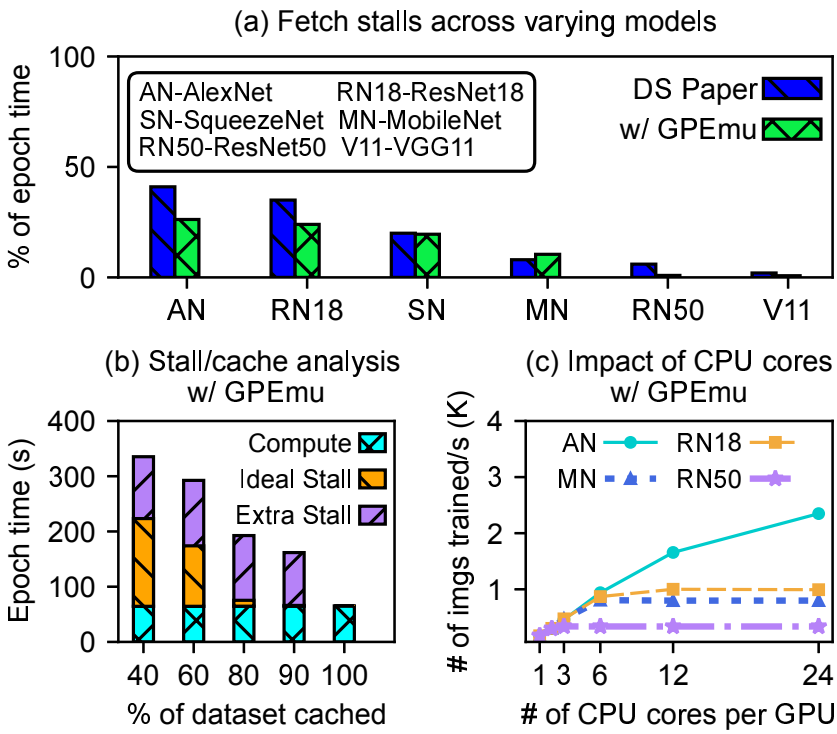


Figure 4.6: **Data stall analysis with GPEMU (§3.1).** *The figures show that we successfully reproduced the analysis experiments in the DS paper [113] using* GPEMU.

expected due to differences in end-to-end setups. For instance, the data stall percentage for AlexNet in our experiment is 26%, compared to 40% in the DS paper. This variation can be attributed to factors like different dataset sizes (we used a subset for fast prototyping), fluctuating page cache size during training (due to PyTorch memory usage), and differing data loaders (PyTorch default vs. DALI).

The DS paper next assessed training time impacts relative to cached data amounts. In **Figure 4.6b**, our reproduction of this experiment with GPEMU shows that lower cache percentages lead to longer training times from increased fetch stalls. We omitted the DS paper result [113, Fig.4] to avoid overcrowding the graph. The first bar indicates that with only 40% of data cached in host memory, training slows by a factor of 5.2 compared to full dataset caching in host memory. This figure also reveals the inefficiency of the OS page cache in DL training due to thrashing. For example, with 40% data cached, the ideal cache hit rate should be 40%, resulting in a fetch

stall of 158 seconds per epoch. But with the OS page cache, the fetch stall can reach up to 270 seconds. While our emulation results align with the DS paper patterns, exact values differ due to setup variations.

Finally, the DS paper evaluates the relation between CPU numbers per GPU and training speed in terms of images per second [113, Fig. 5]. In **Figure 4.6c**, we successfully reproduced the same relationship using GPEMU for four models. Each model shows distinct CPU requirements per GPU to minimize prep stalls (*i.e.*, delays due to CPU data preprocessing). For instance, ResNet50 (the purple line), a computation-intensive model, needs only 3 CPUs per GPU to eliminate prep stalls and fully utilize GPU. In contrast, AlexNet (cyan line), with lighter computations, requires 24 CPUs per GPU to avoid prep stalls and optimize GPU use.

4.2.2 Preprocessing Disaggregation

To alleviate data stalls, extensive research has focused on optimizing the data preprocessing stage, which is often a bottleneck in DL training across various scenarios. A common solution is preprocessing disaggregation [54, 76, 145, 169, 171]. In this subsection, we reproduce two notable studies with GPEMU: **tf.data service**$_{\text{SoCC23}}$ [54] and **FastFlow**$_{\text{VLDB23}}$ [145].

The first system, tf.data service (TF-DS for short) [54], is an open-source data preprocessing disaggregation framework based on TensorFlow's tf.data module. Unlike traditional methods that reply on local CPUs for data preprocessing— which may not feed GPUs quickly enough—TF-DS disaggregates these tasks to remote worker machine CPUs. This method allows for scaling out data preprocessing independently of expensive GPUs, by adding more worker machines.

We reproduced the results from the TF-DS paper using GPEMU. Our experiments ran a single training client with 4 CPUs, emulating GAN model training on the CUB-200-2011 dataset on 4 RTX-6000 GPUs in TensorFlow, integrated with GPEMU. The baseline involved local data preprocessing on the client machine. We then offloaded preprocessing to a variable number of remote workers, each with 4 CPUs. Training time speedups relative to the baseline are shown in

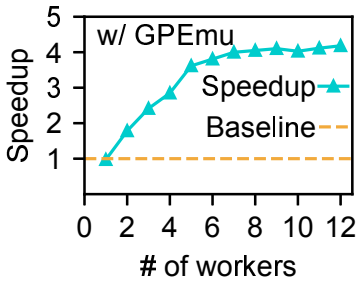


Figure 4.7: **TF-DS [54] training speedup with GPEMU (§3.2).** *The figure shows that we managed to demonstrate TF-DS's benefits with* GPEMU.

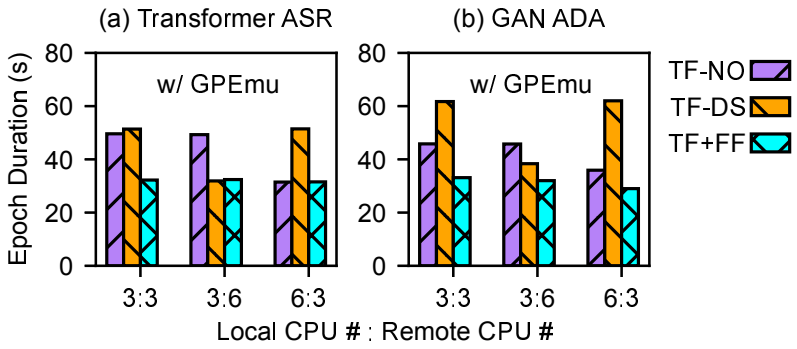


Figure 4.8: **FastFlow's benefits with GPEMU (§3.2).** *These figures show that we successfully reproduced experiments in the FastFlow paper [145] using* GPEMU.

Figure 4.7.

We have two observations from this figure. First, the speedup increases with more workers, demonstrating TF-DS's effectiveness. Second, beyond the deployment of 7 or more workers, adding more workers yields minimal training time improvements. This is because data preprocessing is already fast enough, and the bottleneck becomes the (emulated) GPU compute. These observations align closely with those reported in the TF-DS paper [54, Fig. 9], although their experiment involved more workers on a larger scale.

Next, we reproduce FastFlow [145], built atop TF-DS. FastFlow enhances the disaggregation mechanism by using both local and remote CPUs for data preprocessing. It dynamically divides the preprocessing pipeline between local and remote workers based on performance metrics, aiming to optimize training time.

We obtained the FastFlow source code from their GitHub [16] and reproduced experiments for Transformer ASR and GAN ADA workloads using GPEMU. Our setups varied local-to-remote CPU ratios (3:3, 3:6, 6:3). **Figures 4.8a and b** compare epoch times across three policies: TF-NO (no preprocessing disaggregation), TF-DS, and TF+FF (FastFlow). Notably, FastFlow consistently outperforms the others in all scenarios. TF-NO struggles with limited local CPUs, while TF-DS lags behind with few remote CPUs. Our emulation results closely mirror the FastFlow paper [145, Fig. 6].

4.2.3 Data Loader Optimization

Another approach to reducing data stall is optimizing the data loading phase in DL training. For example, **FFCV**$_{\text{CVPR23}}$ [105] introduces techniques like efficient file storage formats, optimized process-level caching, and just-in-time compiled data preprocessing. Here we demonstrate FFCV's benefits using GPEMU.

In **Figure 4.9**, we present results from three setups: a real RTX-6000 GPU (red bars), original FFCV paper (blue bars), and GPEMU (green bars). Our emulation *successfully shows the performance improvement from FFCV*. For example, it cut training time by 80% compared to PyTorch's default ImageFolder data loader (leftmost green bar vs. rightmost green bar). The emulation results closely match those from real GPU experiments and align with the FFCV paper. Note, the y-axis is normalized to the maximum training time per configuration, as we utilized a smaller dataset for faster research. Once again, the use of GPEMU serves the purpose of fast prototyping.

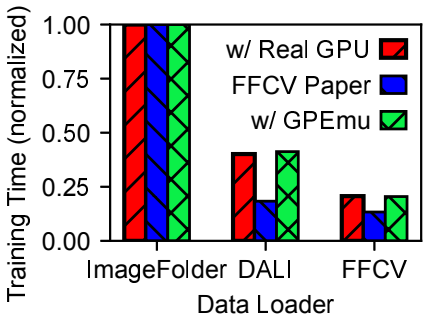


Figure 4.9: **FFCV's [105] benefits shown with GPEMU (§3.3).**

4.2.4 Distributed Training Optimization

In the previous subsections, we focused on using GPEMU to reproduce single-node training analyses and optimizations. Now, we show how GPEMU *can also be used to reproduce optimizations for distributed training*, like those related to distributed caching, which aims to reduce data stalls in distributed training [109, 113, 136, 162, 174]. We reproduce two papers: CoorDL from the DataStall$_{\text{VLDB21}}$ ("DS") paper [113], and the locality-aware data loading ("LADL") paper from HiPC 2019 [162].

Let's start with **CoorDL** [113], which introduces partitioned caching. In this approach, different compute nodes cache distinct parts of the dataset and coordinate to speed up data loading in distributed training. With partitioned caching, a local cache miss leads to fetching the missing data from remote caches on other nodes via network communication. CoorDL applies this technique to their MinIO caching algorithm, demonstrating significant advantages.

Now, we show using GPEMU to assess CoorDL's benefits. Initially, we tried to obtain CoorDL's source code from its GitHub repository [25], but faced challenges in compiling due to dependency issues with an older version of DALI. As a result, we *re-implemented* CoorDL's MinIO and partitioned caching from scratch in PyTorch.

To reproduce CoorDL's results, we used GPEMU for distributed training across various compute nodes, each emulating 8 GPUs. We experimented with different local cache percentages (65% and 40%). The normalized training speedups are shown in **Figure 4.10**. In it, orange bars represent the baseline, blue bars are CoorDL results from the DS paper, and green bars indicate CoorDL with GPEMU. The DS paper's results show CoorDL's speedups increasing with more nodes, attributed to partitioned caching. Our emulation also reveals this pattern, though values may vary due to different setups. For instance, at 65% local cache with 1 node, CoorDL with GPEMU achieves a 3x speedup, mainly due to MinIO caching. With 2 nodes, the speedup reaches 6x as the combined cache covers the entire dataset.

Next, let's move to "**locality-aware data loading**" (or "**LADL**") [162]. Distributed caching

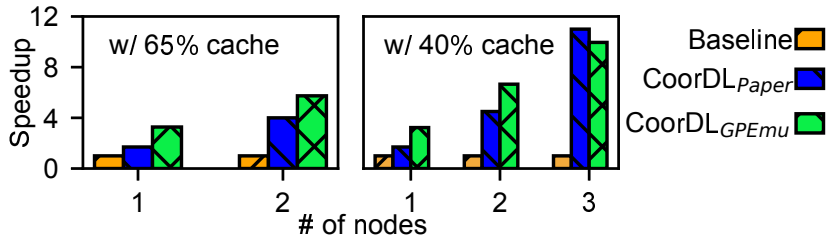


Figure 4.10: **CoorDL's benefits with GPEMU (§3.4).** *The figures show that with* GPEMU *we successfully demonstrated the effectiveness of CoorDL from the DS paper [113].*

techniques like CoorDL offer impressive gains with few nodes and adequate bandwidth but face scalability challenges in large-scale distributed training. These challenges are mainly due to increased network traffic, potentially causing bandwidth bottlenecks. LADL addresses this by altering the sampling algorithm to prefer locally cached samples, reducing data fetching from remote cache. Though this slightly reduces sampling randomness, it significantly decreases network traffic, thus enhancing scalability for distributed training.

Here we use GPEMU to showcase LADL's advantages. With the original source code unavailable and no response to our request, we *re-implemented* it in PyTorch based on our earlier CoorDL implementation. While the original paper used a 16-128 node cluster to show distributed cache scalability challenges, our resources are more limited. However, as GPEMU is designed for rapid prototyping, we emulated results on a 4-node cluster. To mimic network bandwidth bottlenecks typical in large-scale training, we limited our cluster's network bandwidth to 3Gbps.

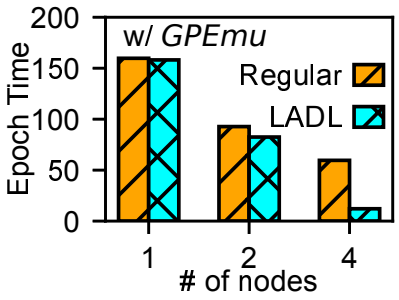


Figure 4.11: **LADL's [162] benefits demonstrated with GPEMU (§3.4).**

The emulation results are depicted in **Figure 4.11**, with orange bars for regular distributed cache and cyan bars for that with LADL. The results clearly demonstrate LADL's improvment on distributed training scalability. For instance, for 4-node training with data all cached (either locally or remotely), regular distributed cache still takes 60 sec per epoch due to network bandwidth

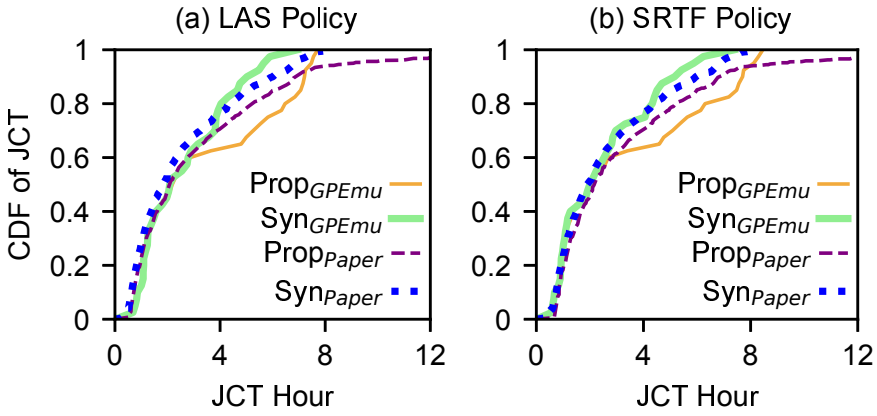


Figure 4.12: **Synergy [112] JCT reduction with GPEMU (§3.5).** *The figures show that* GPEMU *successfully demonstrated Synergy's JCT reduction over GPU-proportional resource allocation.*

limitations. However, LADL reduces the epoch time to just 12 sec, thanks to the reduced network bandwidth demands. These results align with the original paper.

4.2.5 GPU Scheduling

Next, we explore using GPEMU in GPU scheduling. Numerous techniques have been proposed for enhancing GPU scheduling in clusters [104, 112, 117, 160, 167, 168]. We reproduce two: **Synergy**$_{\text{OSDI22}}$ [112] and **Allox**$_{\text{Eurosys20}}$ [104]. Synergy is selected for its custom scheduler's easy compatibility with GPEMU, and we also reproduce Allox to demonstrate GPEMU's ability to support Kubernetes scheduling.

We first reproduce some of the **Synergy** experiments [112]. Synergy is a scheduler that evaluates each job's resource needs through optimistic profiling and allocates GPUs, CPUs, and memory accordingly. This approach effectively prevents data stalls and maximizes GPU utilization.

To reproduce the Synergy paper, we obtained its source code from their GitHub repository [43] and integrated it with GPEMU, using GPEMU for resource demand profiling and GPU scheduling emulation. Synergy was designed for jobs that utilize DNN-aware cache systems like MinIO [113]. We opted for our implementations of MinIO instead of the original, as it faced compilation issues due to dependency conflicts.

We conducted experiments scheduling 40 DL jobs on a 4-node cluster. Each node had 24 CPUs, 100GB DRAM, and 8 emulated GPUs. Our evaluation compared Synergy against GPU-proportional resource allocation using the Least Attained Service (LAS) and Shortest Remaining Time First (SRTF) policies respectively. **Figure 4.12** presents the CDF of job completion time (JCT) for both policies, with results from both the original paper and GPEMU. Consistent with the original paper, our emulation shows Synergy's effectiveness in reducing JCT. For instance, in **Figure 4.12a**, Synergy decreases average JCT by 22% and 95th percentile JCT by 25% (solid orange vs. green lines) for the LAS policy.

Next, we reproduce **Allox** [104] to showcase our Kubernetes scheduling support capabilities. Allox is designed for DL jobs that have interchangeable resource configurations, such as CPU versus GPU. It strategically schedules jobs across various compute resources, selecting the optimal configuration for each while maintaining fairness.

We acquired the Allox source code from their GitHub [3] and integrated it with GPEMU based on our Kubernetes support. In our experiments, we set up a cluster with one master node, four `emuGPU` worker nodes, and eight CPU worker nodes. We scheduled 40 PyTorch jobs among four users. The emulated GPU vs. CPU worker speedup ranged from 1.3 to 8.2 for these jobs. **Figure 4.13** shows the normalized average JCT for three scheduling policies: Dominant Resource Fairness with First Come First Serve (DRFF), Dominant Resource Fairness with Shortest Job First (DRFS), and Allox. Consistent with the original paper, our emulation results highlight Allox's effectiveness. For example, it reduces average JCT by 42% compared to DRFF (leftmost green bar vs. rightmost green bar).

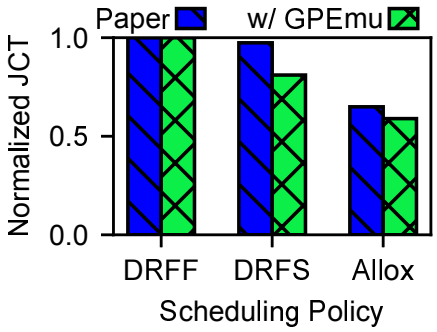


Figure 4.13: **Allox's [104] JCT reduction with GPEMU (§3.5).**

Other than Synergy and Allox, we believe other scheduling papers such as Gavel, HiveD, SiloD [117, 167, 168] can also be reproduced with our GPEMU features. Papers like OASis and Themis

[57, 110] could be reproduced too, but they require integrating GPEMU features into other ML frameworks like MXNet [61] and other scheduling frameworks like YARN [146].

4.2.6 GPU Sharing

Finally, we reproduce papers on GPU sharing. There's been considerable research into enabling GPU sharing among DL workloads and optimizing GPU job scheduling based on this sharing [81, 159, 160, 163, 173]. We focus on two notable studies: **Salus**$_{\text{MLSys20}}$ [163] and **Muri**$_{\text{SIGCOMM22}}$ [173]. Salus is chosen for its focus on enabling fine-grained GPU sharing in DL applications, and Muri for its insights on GPU sharing's impact on GPU job scheduling.

We begin with reproducing **Salus [163]**, which offers an efficient execution service for fine-grained GPU sharing through iteration-level computation scheduling, thereby enabling rapid job switching between different DL applications.

Figure 4.14: **Salus's [163] GPU sharing with GPEMU (§3.6).**

With our GPU sharing emulation support, *we successfully emulated iteration-level job switching akin to Salus*. We integrated GPEMU into PyTorch and reproduced [163, Fig. 9] by running three ResNet50 training jobs on the same machine, fairly sharing an emulated P100 GPU via our manager. The emulated training throughput is shown in **Figure 4.14**. Initially, Job 1 achieves 220 images/sec, fully utilizing the GPU. With the start of Job 2 at 30s, the per-job throughput halves (dashed blue line). This division intensifies with Job 3 at 60s, resulting in each job's share dropping to a third. As jobs conclude in reverse order, this trend reverses. The pattern matches the one in [163, Fig. 9].

Next, we reproduce **Muri [173]**. Muri is a DL job scheduler for multi-resource clusters based on GPU time sharing. It interleaves DL jobs bottlenecked by different resources like GPU, CPU, storage, and network. Muri groups these jobs onto the same machine and interleaves their GPU

executions to optimize resource utilization.

To reproduce Muri, we obtained its code from their GitHub [29]. However, the available code for testbed deployments was primarily pseudocode, with the original industry code not publicly available. Consequently, *we developed our simplified version of Muri*. For fast reproduction, our version focused on scheduling on a single machine, interleaving just CPU and GPU resources. We used our GPU sharing emulation for GPU interleaving.

In our experiments, we used Muri with GPEMU to schedule 10 emulated training jobs, from CPU-bound ones like AlexNet to GPU-bound like ResNet50. We compared two algorithms: SRTF and Muri-S (Muri with SRTF). **Figure 4.15a** illustrates the job queue length changes over time under each policy. Muri outperforms SRTF, reducing queue length faster by running more jobs concurrently and using GPUs more efficiently, in line with [173, Fig. 8]. **Figure 4.15b** shows normalized makespan for both. Our emulation shows Muri's 1.35x speedup in makespan compared to SRTF, validating its effectiveness and aligning with the original paper.

4.3 Micro-Optimizations Evaluated Using GPEMU

Besides reproducing existing work, we showcase the power of GPEMU by prototyping and evaluating new micro-optimizations without using real GPUs. Below, we present several storage-stack optimizations that can improve DL training epoch time, such as small-file first (SFF) caching policy (§4.3.1), asynchronous batched data (§4.3.2), and random-class file grouping (§4.3.3).

4.3.1 Cache Small Files

In this section, we explore the impact of tailoring caching algorithms by considering the characteristics of DL training datasets and HDDs, and show the benefits with GPEMU. We took inspiration from the datastall paper where they propose the "MinIO" algorithm [113]. MinIO is designed around the unique data access pattern in DL training, where data is read in epochs, and every epoch reads every item in the dataset exactly once in a random order. Consequently, if a certain

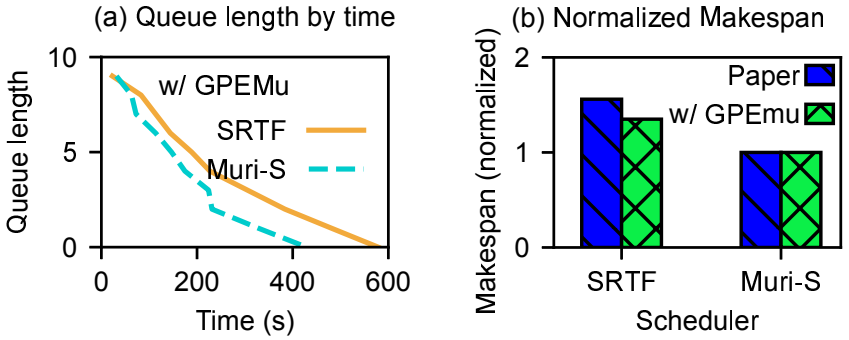


Figure 4.15: **Muri's benefits shown using GPEMU (§3.6).** *The figures show that, utilizing* GPEMU*, we successfully reproduced the experiments in the Muri paper [173].*

percentage (X%) of data can be cached, the maximum attainable cache hit rate will also be X%. Based on this observation, MinIO never evicts data items once the cache is filled with randomly fetched data items. This strategy effectively mitigates the thrashing issue associated with prior cache eviction policies, where valuable items could be evicted prematurely.

We implemented MinIO by ourselves in PyTorch and evaluated it using GPEMU. **Figure 4.16a** shows that we can run MinIO with GPEMU and demonstrate its benefits, where the purple bars represent the performance of the OS page cache, while the orange bars depict the results achieved with MinIO. With the same cache percentage, MinIO outperforms the OS page cache. For example, when cache size is 40%, MinIO achieves an epoch time of only 394 seconds, compared to the 537 seconds with the OS page cache.

However, we found *a limitation in MinIO: it treats all data items as equal and populates the cache with random items*. Yet, the storage reads can exhibit preferences for certain files over others within the same dataset. For example, **Figure 4.16b** illustrates the cumulative distribution function (CDF) of file sizes in the ImageNet dataset [127]. Evidently, file sizes exhibit significant variability, with orders of magnitude separating the smallest from the largest files. On the other hand, storage systems, especially HDDs, favor large file reads over small ones. This preference arises from the fact that large file reads are more sequential and entail fewer seek and rotation overheads. As illustrated in **Figure 4.16c**, when reading files randomly from disk, larger files

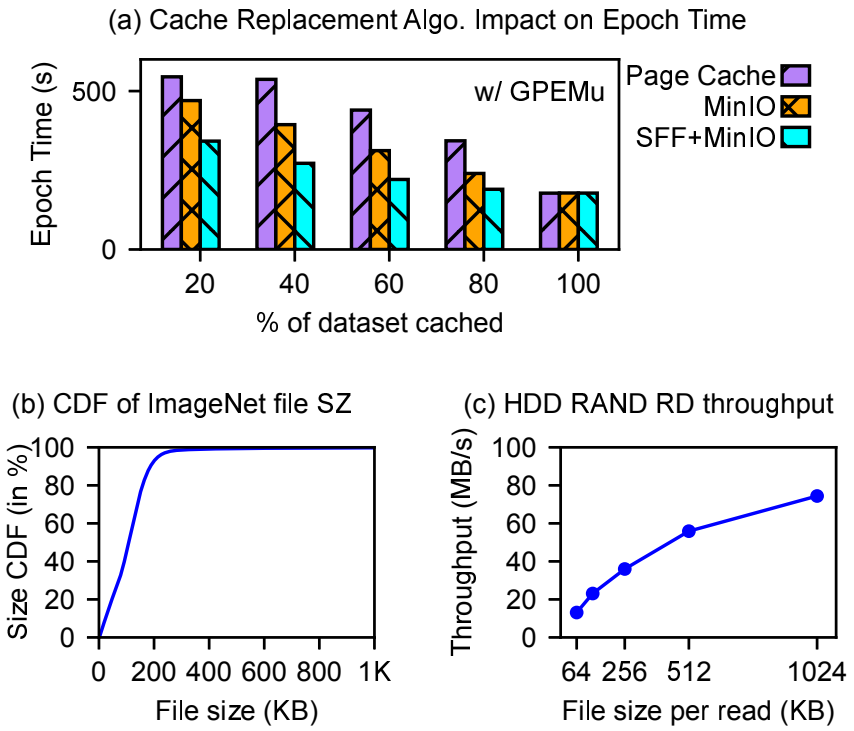


Figure 4.16: **SFF's benefit with GPEMU (§4.1).** *The figures show (a) SFF's benefit demonstrated using* GPEMU*, (b) variable Imagenet file sizes, (c) better throughput with larger file reads.*

generally achieve higher read throughput compared to smaller ones.

To address this limitation, we introduce the **Small Files First (SFF)** caching strategy, *which gives precedence to small files when populating the MinIO cache*. Specifically, we begin by reading the metadata for all files in the dataset information prior to training. We then sort them by file size and find a size threshold (`Th`) such that all files smaller than `Th` could exactly fill the cache. Subsequently, we pass this threshold to MinIO and instruct it to cache only files smaller than the threshold during the first epoch. By adopting this approach, we ensure that a greater number of small files are cached, while only large files are read from storage.

We implemented SFF into MinIO, and evaluated it with GPEMU. The cyan bars in **Figure 4.16a** represent the training performance achieved by MinIO+SFF. The figure shows that with the same cache size, we can achieve better performance compared to MinIO alone. Notably, the greatest improvements are observed when the cache percentage is relatively small. For instance,

with a cache percentage of 20%, we reduce the training time by 28% compared to MinIO. As the cache percentage increases, the benefits of SFF become less pronounced, primarily because the file sizes cached in MinIO align more closely with the SFF strategy.

4.3.2 Async Batch

Another way to reduce the overhead of random reads is by leveraging I/O reordering, a technique employed by the Linux I/O scheduler [23]. The I/O scheduler reorders I/O requests based on their logical block addresses (LBA) in order to reduce seek time and rotational latency, hence increasing the overall throughput. The more requests the scheduler can process together, the greater the reduction is.

Unfortunately, we found that PyTorch's data loader does *not* fully benefit from I/O reordering. More specifically, PyTorch's official ImageNet training script [21] employs four workers to load data, with each worker handling one batch. Each worker reads *one* image at a time, waiting for the read to complete. Consequently, *at most* four read requests are simultaneously sent to the OS.

PyTorch's design is simple to implement; every worker can read and preprocess each image synchronously (without having to worry about concurrency issues). Overall, it also uses minimal memory; each worker only has a single batch in memory during loading. However, it is not efficient because the OS does not have enough in-flight I/Os to reorder to improve the I/O performance and the LBA gaps between the four target I/Os remain large, causing high seek and rotational overhead.

Ideally we should send more concurrent requests to the OS so that it may benefit more from reordering. To do this, we send all the requests in the entire batch. While this sounds simple, one minor complication is that we must move from the blocking/synchronous style to an asynchronous I/O design, otherwise the OS cannot see all the requests. To achieve this, we (1) utilize `io_uring` [47] asynchronous system call, (2) organize the sending and receiving of asynchronous I/Os using input and completion queues, and (3) carefully employ spinlocks and atomics to handle concurrency issues when using `io_uring`. Furthermore, since the OS is limited by its maximum queue

depth of 2048 requests [2], when submitting more than this limit, we must pre-sort the requests by LBA at the application level before sending them to the OS.

Within this "*asynchronous-batch*" design, we found more room for optimization. The number of concurrent I/O requests is currently bottlenecked by the training batch size, and in some scenarios the batch size can be as small as 2 [31], limiting the benefits of I/O reordering. To overcome this limitation, we introduce "*superbatch*, " which is a multiple (*e.g.*, 2x) of the regular batch size. During data loading, we asynchronously send a superbatch of requests to the OS, maximizing the advantages of I/O reordering. It is important to note that superbatch only affects data loading and does not alter the training batch size or other training steps.

We evaluated the benefits of asynchronous (super)batch reading with GPEMU. **Figure 4.17** shows the epoch time for emulating AlexNet training with 4 workers and batch size of 16 using different data reading policies. Employing the asynchronous batch reading alone without superbatch (yellow bar) reduces epoch time by 19% compared to the original PyTorch data loader (purple bar). After applying the superbatch approach, the reduction can be up to 50% (the cyan bars). Increasing the superbatch size too big yields minimal additional benefits, as I/O reordering is already efficiently utilized. Excessively large superbatch is also not recommended as it increases the memory pressure, which can adversely affect the epoch time.

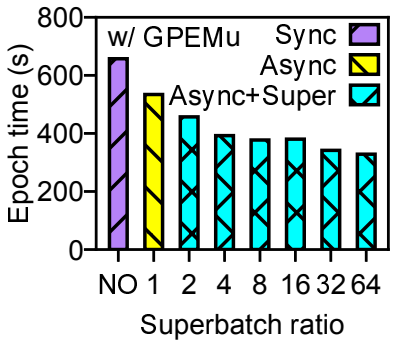


Figure 4.17: **Async batch's benefit demonstrated with GPEMU (§4.2).**

4.3.3 File Grouping

The last two micro-optimizations have focused on reducing the penalty of random file reads for disk-based systems without altering the original random sampling order. In this subsection, we propose another optimization, termed "file grouping," which slightly reduces sampling randomness

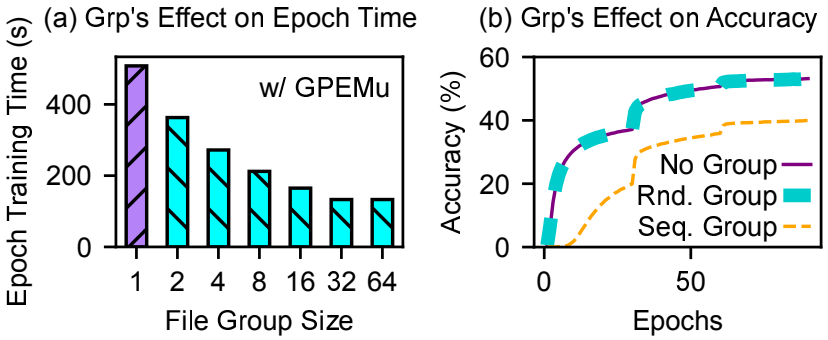


Figure 4.18: **File grouping's effect (§4.3).** *The figures show (a) successful showcasing of file grouping's performance benefits using* GPEMU*, and (b) file grouping's impact on accuracy.*

in exchange for enhanced I/O performance.

The file grouping approach works as follows: (a) Prior to training, we group every X random small files into one large file, storing location and labels in the metadata. For a dataset with M files, this results in M/X large files; (b) During training, instead of reading N random small files per batch, we read N/X random large files.

With GPEMU, we evaluated file grouping's impact on AlexNet training time with various group sizes, as is shown in **Figure 4.18a**. For example, without grouping (group size = 1), the training time reaches 498 seconds. With 8 images per group, the training time can be reduced by 2.4x. Interestingly, grouping the images more than 32 images will not yield better performance, implying that the disk seek overhead is already minimal compared to the data transfer time.

One concern about file grouping is its potential impact on model accuracy. Could the model become biased and lose accuracy? We experimented with *random-class grouping*, where each group contains a mix of random "classes" (e.g., "cat", "dog", "snake" in the ImageNet dataset) and found no loss in accuracy (compared to no grouping) as shown in **Figure 4.18b**. We validated the accuracy of random-class grouping across three different tasks (image classification, object detection, audio classification), using three datasets (ImageNet, COCO, Speech Commands) and five models (AlexNet, ResNet18, ResNet50, Faster R-CNN, M5), consistently observing the same pattern.

However, employing a naive *"sequential" grouping* (*i.e.*, grouping the unzipped ImageNet dataset images sequentially) significantly reduced accuracy, shown in the orange line in **Figure 4.18b**. This decline is likely because sequential groups contain images from the same category, like all "cat"' pictures, leading to biased learning. This bias can cause catastrophic forgetting [70], where the model temporarily excels in the current category but may forget the previously learned ones.

4.4 MLECEmu Design

In this section, we introduce the design of MLECEmu, an emulator developed to provide end-to-end emulation of multi-level erasure-coded (MLEC) storage systems for workloads such as deep learning training. MLECEmu operates using a real erasure-coded storage stack, including systems like HDFS [133] and ZFS [126], but instead of physical disks, it utilizes `tmpfs` loop devices in memory with throttled throughput to simulate the behavior of physical disks. Below, we detail the key components of MLECEmu's design.

4.4.1 Implementation of MLEC Storage Stack

MLECEmu builds an emulated MLEC storage system using real implementations of HDFS for network-level erasure coding and ZFS for local-level erasure coding. There are several reasons for choosing HDFS and ZFS. First, both systems are widely adopted: HDFS is a key technology in big data and distributed computing, while ZFS is extensively used in national research laboratories. Second, both HDFS and ZFS support single-level erasure coding configurations, either clustered or declustered, which simplifies our development. Third, ZFS can be configured as a Linux kernel module, interfacing seamlessly with HDFS like a traditional Linux filesystem.

In HDFS, when the number of nodes equals the network stripe width, it performs clustered parity; when there are more nodes than the stripe width, it performs declustered parity by spreading blocks across all nodes. ZFS similarly supports both clustered parity (using `raidz`) and declustered

parity (using `draid`). This setup allows MLECEmu to flexibly configure various MLEC chunk placements, including $^{C}/_{C}$, $^{C}/_{D}$, $^{D}/_{C}$, and $^{D}/_{D}$.

The implementation of MLEC with the R_{ALL} repair method in HDFS over ZFS is straightforward: HDFS treats each ZFS pool (ZPool) as a standalone volume, simulating a disk. Incoming data stripes are divided into network chunks (called blocks in HDFS), and HDFS applies its inherent erasure coding mechanisms. Each HDFS block, whether data or parity, is then stored in a different ZPool, where ZFS further performs erasure coding using its configured policies.

4.4.2 Disk Throughput Emulation

To emulate disks without relying on physical hardware, we create multiple loop devices [24], which are pseudo-devices that allow a file to be accessed as a block device. These loop devices are mounted using `tmpfs`, which mimics a disk in memory while providing controlled, throttled bandwidth. We use the Linux `cgroup` tool [9] to limit the bandwidth of each `tmpfs` loop device, effectively simulating the throughput of physical disks.

With these emulated disks, we create a ZFS pool (ZPool) using multiple loop devices. When evaluating MLEC storage for deep learning workloads, the key performance metric is the storage system's throughput, which is directly influenced by the bandwidth of each emulated disk, controlled using the `cgroup` throttling method described above.

4.4.3 Emulating Repair Time and System Degradation

As discussed in Section 3.2.6, we introduced four repair methods. When directly configuring HDFS over ZFS, the default repair method is R_{ALL}. If a ZPool fails, HDFS reconstructs all data from the failed ZPool onto other ZPools using network-level erasure coding, following the R_{ALL} repair method.

However, implementing advanced repair methods presents several challenges: (1) During catastrophic local failures, ZFS suspends all I/O operations, preventing any repair attempts; (2) ZFS

must signal HDFS to initiate network-level repairs for specific data chunks, which requires complex internal checks and API designs; and (3) HDFS must support the repair of smaller, local chunks, which differ from the typical network-level chunks.

Due to the complexity of these implementations, we defer the real implementation of advanced repair methods to future work. Fortunately, MLECEmu provides a mechanism to emulate the performance of these advanced repair methods. We simulate catastrophic failures by taking the ZPool offline. To prevent HDFS from automatically performing repairs using R_{ALL}, we configure a long repair delay. Instead of carrying out real repairs, MLECEmu emulates the system degradation that would result from different repair methods.

Specifically, we project the disk and rack I/O bandwidth degradation that would occur during the repair process based on the repair method and the projected duration calculated by our MLEC simulator (introduced in Section 3.3). We then throttle the disk and rack bandwidth accordingly for the repair duration, effectively simulating the performance impact of catastrophic failures and their subsequent repairs.

4.4.4 Limitations and Future Work

While our emulation effectively mimics disk throughput, it does not currently support emulating disk seek time, which is common for hard drives and varies dynamically in real workloads. We believe seek time could be emulated by integrating MLECEmu with existing disk simulators. This enhancement is left for future work and is discussed in Section 7.2.

4.5 Evaluating MLEC Storage for DL Workloads Using GPEMU and MLECEmu

In this section, we demonstrate how GPEMU and MLECEmu can be used together to evaluate MLEC storage in the context of deep learning (DL) workloads. We also identify system bot-

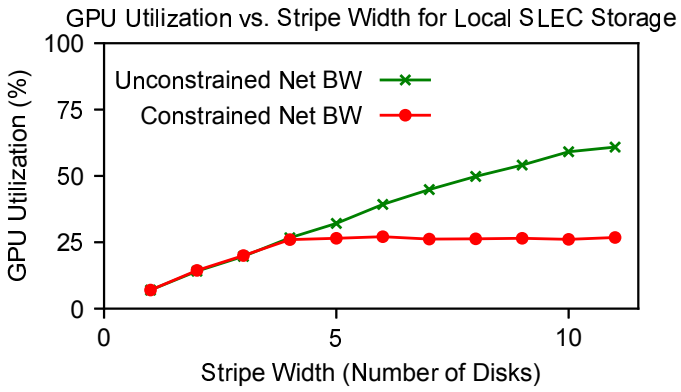


Figure 4.19: **GPU Utilization vs. Stripe Width for Local SLEC Storage.** *The figure demonstrates that wider stripes in SLEC storage can lead to higher GPU utilization in deep learning workloads, as distributed reads from more disks are enabled. However, limited inter-rack bandwidth can become a bottleneck.*

tlenecks and assess GPU utilization under different conditions, such as varying stripe widths, (un)constrained network bandwidths, and under catastrophic failures.

4.5.1 Evaluating GPU Utilization with Varying Stripe Widths in SLEC

We begin by evaluating the performance of a single-level erasure-coded (SLEC) storage system under DL workloads. Using GPEMU, we emulate training the ResNet-18 model on a V100 GPU with an 11GB subset of the OpenImages [33] dataset, where the dataset is stored in a remote erasure-coded storage system. The system is configured with ZFS and emulated HDDs using SLEC with varying stripe widths. Each disk is throttled to a read bandwidth of 30MB/s to simulate random read behavior typical of HDDs during DL training.

As shown by the green points in Figure 4.19, in the initial configuration (stripe width of 1, with no redundancy), the GPU utilization is as low as 7% due to the slow reads from a single emulated disk. As we increase the stripe width (e.g., 2+1 SLEC), the utilization nearly triples, since the system can now read data from three disks concurrently. As more disks participate in the distributed read process, GPU utilization continues to increase.

However, this assumes unconstrained inter-rack bandwidth. When we throttle the inter-rack

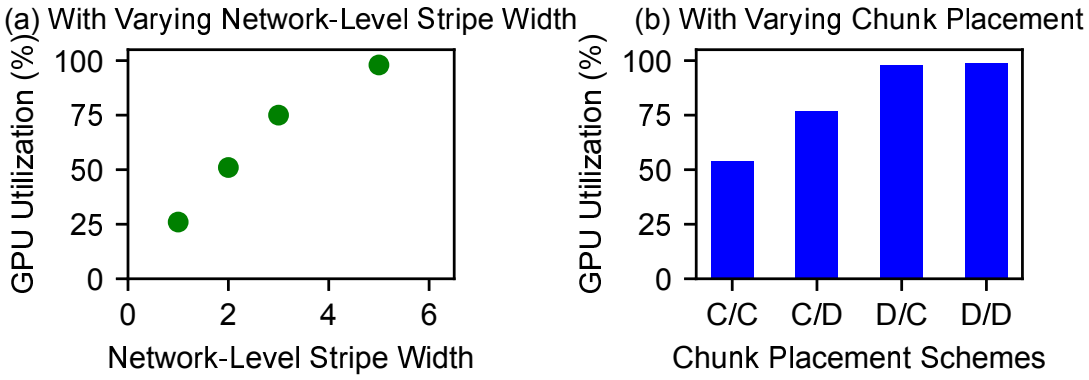


Figure 4.20: **GPU Utilization with MLEC Storage.** *The figures illustrate that (a) a wider network-level stripe can improve GPU utilization despite bandwidth limitations at the rack level, and (b) reading from an MLEC pool with different chunk placement schemes results in varying GPU utilization.*

bandwidth to 1Gbps, GPU utilization plateaus once the stripe width exceeds four, as the inter-rack bandwidth becomes the bottleneck (shown by the red points in Figure 4.19).

4.5.2 Evaluating GPU Utilization in MLEC Storage

Given the limitations seen with local SLEC storage, we evaluate MLEC storage, where data is inherently distributed across racks via network-level erasure coding, enabling us to take advantage of the aggregated bandwidth across multiple racks.

We first configure MLEC with (4+1) erasure coding at the local level and vary the stripe width at the network level. As Figure 4.20a shows, Though each rack's network bandwidth is constrained, as we increase the network-level stripe width, GPU utilization improves, reaching near 100% when the network stripe width is large enough.

We also evaluate the performance of a (2+1)/(2+1) MLEC pool with different chunk placement strategies in a cluster of 36 emulated disks across six racks, comparing $^{C}/_{C}$, $^{C}/_{D}$, $^{D}/_{C}$, and $^{D}/_{D}$ configurations. As shown in Figure 4.20b, GPU utilization varies depending on the chunk placement:

- With $^{C}/_{C}$, GPU utilization is 54%, as only 3 disks per rack take part in reading, underutilizing

each rack's network bandwidth. Moreover, only 3 racks are involved.

- With $^{C}/_{D}$, GPU utilization improves to around 77%, as declustered placement results in data being read from all 6 disks in each rack, fully utilizing the rack's network bandwidth. However, not all racks are involved.

- With $^{D}/_{C}$, GPU utilization reaches almost 100%, as all racks participate. Although a local clustered pool contains only 3 disks, declustered placement at the network-level allows both local clustered pools in each rack to take part in reading, fully utilizing each rack's network bandwidth.

- With $^{D}/_{D}$, GPU utilization is near 100%, as it fully utilizes each rack's network bandwidth and involves all racks.

In this setup, we assume each MLEC pool operates as a separate server because HDFS on ZFS does not support configuring multiple clustered pools within a single HDFS cluster. As a result, in MLECEmu, we configure each MLEC pool as an independent server. Consequently, it is more intuitive for the dataset to be stored in a specific MLEC pool, as users interact with individual HDFS servers.

A more efficient approach could involve a network-level storage manager that supports multiple clustered pools and dynamically distributes data across different MLEC pools. For instance, MarFS [90] implements this by distributing data across various MLEC pools through its internal hashing mechanism.

4.5.3 GPU Utilization Under Constrained Inter-Cluster Bandwidth

In the previous experiments, we demonstrated how MLEC storage can meet the bandwidth demands of DL workloads by distributing data across multiple racks. However, when inter-cluster network bandwidth is limited, the situation changes. To illustrate this, we restrict the network bandwidth between the DL training node and the MLEC storage cluster to 2 Gbps. We configure

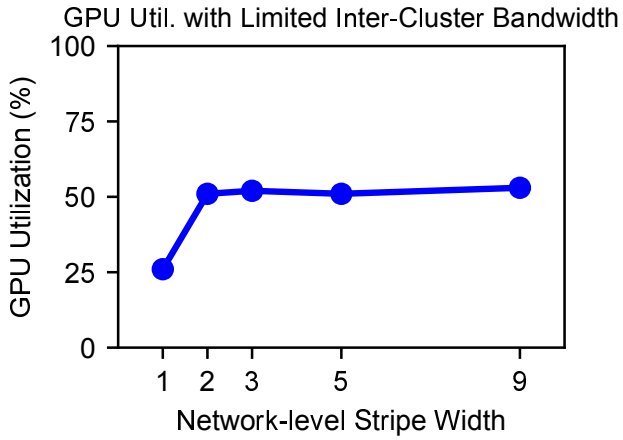


Figure 4.21: **GPU utilization with limited inter-cluster bandwidth.** *The GPU utilization stops to increase when the inter-cluster bandwidth is constrained.*

an $^{C}/_{c}$ MLEC storage with a (4+1) scheme at the local level and progressively increase the stripe width at the top level to include more racks.

As shown in Figure 4.21, the maximum GPU utilization is capped at 52%. When 3 or more racks are involved, GPU utilization ceases to increase because the inter-cluster bandwidth is no longer the bottleneck.

4.5.4 Evaluating Training Throughput During Catastrophic Failure Repairs

Finally, we assess the training throughput during catastrophic failure repairs using the $^{C}/_{c}$ (2+1)/(4+1) configuration with two different repair methods: R_{ALL} and R_{MIN}, with per-rack network bandwidth throttled to 500 Mbps. We simulate a catastrophic failure by emulating the simultaneous failure of two disks in a pool. We simulate the repair methods using projected repair durations and their impact on disk and rack bandwidth based on MLECSim. During the repair process, we limit repair bandwidth to 20% of the available capacity to ensure adequate bandwidth remains for DL workloads.

Figure 4.22 shows the training throughput over time during data repair. Before the failure, training throughput is approximately 900 samples/sec. During the repair, throughput drops to around

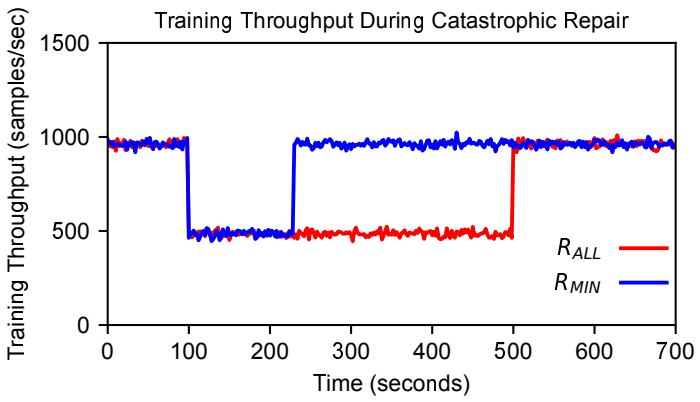


Figure 4.22: **Training Throughput During Catastrophic Local Repair.** *The figure illustrates the training throughput over time during catastrophic local failure repairs, comparing the impact of two different repair methods.*

480 samples/sec because only two of the three racks remain available to serve the DL workload, and network bandwidth is consumed by the repair process. Additionally, when reading lost data, HDFS must retrieve two chunks to reconstruct each lost chunk, doubling the read overhead.

With R_{ALL}, this performance degradation lasts about 400 seconds, as the entire local pool must be reconstructed via network repair. In contrast, with R_{MIN}, degradation time is shorter, lasting only about 130 seconds, since only one failed disk is repaired via network repair. After this, the training throughput returns to its original rate, even as the second disk continues local repair. This is because the rack's bandwidth limit, rather than disk bandwidth, is the bottleneck, allowing the remaining disk bandwidth to fully utilize the rack bandwidth even during local repairs.

4.6 Conclusion

In this chapter, we present an emulation-based approach to evaluate MLEC storage for deep learning (DL) workloads in a cost-effective manner, without the need for real GPUs or disk arrays. We first introduce GPEMU, a comprehensive GPU emulator specifically designed for DL workloads, equipped with extensive emulation features and support for a variety of DL configurations. Next, we introduce MLECEmu, an MLEC storage emulator capable of simulating the performance of

MLEC storage without using actual disk arrays. Finally, we demonstrate how GPEMU and MLE-CEmu can be used together to evaluate MLEC storage for DL workloads.

CHAPTER 5

SOPHON: A SELECTIVE PREPROCESSING OFFLOADING FRAMEWORK FOR REDUCING DATA TRAFFIC FROM REMOTE STORAGE IN DEEP LEARNING TRAINING

As we have shown in 4.5, while MLEC storage can provide substantial aggregated intra-cluster bandwidth for deep learning workloads with wide stripes and distributed reads from multiple disks and racks, the inter-cluster bandwidth can become a bottleneck for training throughput. In practice, inter-cluster bandwidth is often more limited than intra-cluster bandwidth [48, 80], making cross-cluster network bandwidth a critical constraint on DL workloads and GPU utilization [66, 103, 148, 167].

In this chapter, we introduce SOPHON (Selectively Offloading Preprocessing with Hybrid Operations Near-storage), a framework designed to selectively offload DL preprocessing tasks to remote storage servers to reduce data transfer traffic. SOPHON has two key components: (1) A two-stage profiler that collects essential metrics for offloading decisions. Offloading is activated only when the workload is identified as I/O-bound during profiling. (2) A decision engine that determines which samples to offload and identifies the specific operations to offload for each sample, balancing reduced traffic and CPU overhead. Together, these components enable SOPHON to provide tailored offloading strategies that meet the unique needs and constraints of each training scenario.

We evaluated SOPHON with deep learning training workloads that read data from remote MLEC storage, using GPEMU and MLECEmu introduced in Section 1.2. Our evaluation results show that SOPHON effectively enhances training efficiency, achieving a 1.2-2.2x reduction in training time compared to existing solutions.

As **Table 5.1** shows, SOPHON is the first work that implements data-selective offloading for DL preprocessing, where ”data-selective” refers to selecting specific samples for offloading based

	Operation Selective	Data Partial	Data Selective	To Near Storage
tf.data svc	—	—	—	—
GoldMiner	√	—	—	—
FastFlow	—	√	—	—
cedar	√	—	—	—
SOPHON	√	√	√	√

Table 5.1: **Existing Offloading [54, 145, 169, 170] vs. SOPHON.**

on each individual sample's characteristics.

In this chapter, we analyze a specific DL training workload to demonstrate the potential for traffic reduction through preprocessing offloading in Section 5.1. Section 5.2 introduces the design of SOPHON, followed by an evaluation of SOPHON for DL training workloads with remote MLEC storage using our emulators in Section 5.3. We discuss the use cases and limitations of SOPHON in Section 5.4 and examine potential challenges that MLEC storage may pose for preprocessing offloading in Section 5.5. Finally, we conclude in Section 5.6.

5.1 Preprocessing Analysis

A typical DL training iteration entails: (1) Fetching data from storage, (2) Preprocessing data on CPUs, (3) Transferring data to GPUs, (4) Forward propagation for predictions, (5) Backward propagation for parameter updates.

In this section, we analyze the preprocessing pipeline of a specific DL workload, revealing opportunities and challenges in minimizing data traffic through strategic offloading of preprocessing tasks. Our investigation is anchored in a case study on image classification workloads. We employ the official PyTorch example training script, sourced from its GitHub repository [21], conducting experiments on subsets of two key datasets in computer vision DL research: OpenImages [33] and ImageNet [127].

The workload's preprocessing pipeline consists of five key operations: (1) **Decode**: Converts raw binary (e.g., JPEG) to an image object for manipulation via Python libraries like PIL; (2)

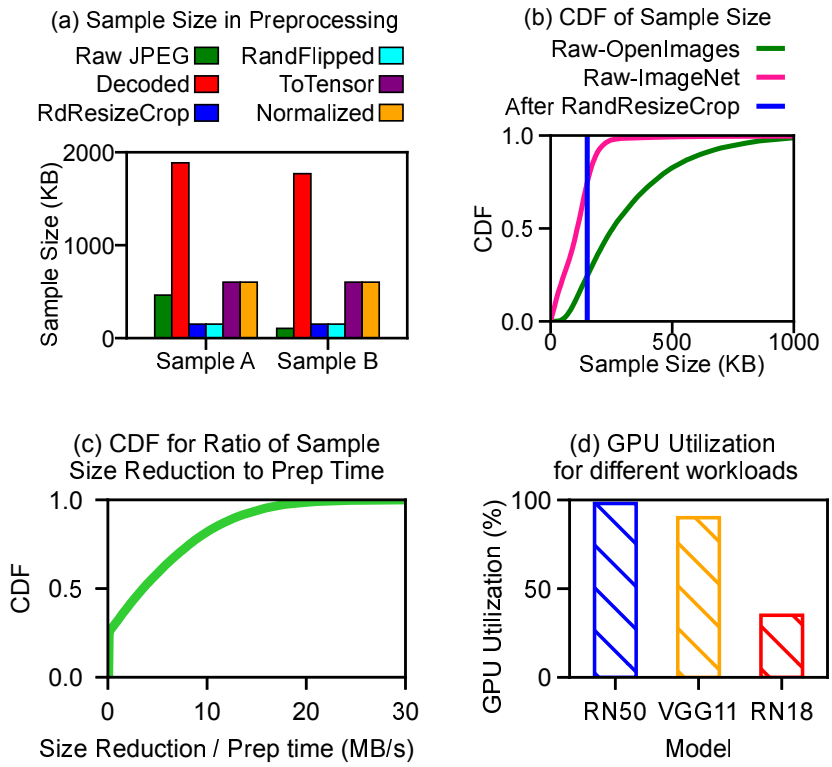


Figure 5.1: **Analysis of Preprocessing Pipeline.** *The figures are explained in Section 5.1.*

RandomResizedCrop: Crops a random image section and resizes it to a specified size; (3) **RandomHorizontalFlip**: Flips the image horizontally at random; (4) **ToTensor**: Transforms the PIL Image from a uint8 ([0,255]) list to a float Tensor, scaling to [0.0, 1.0]. (5) **Normalize**: Normalizes the tensor image with mean and standard deviation.

By measuring file sizes before and after each preprocessing step and assessing the preprocessing time cost, we arrived at several key findings:

Finding #1: Variations in file size across the preprocessing steps highlight opportunities for reducing data transfers through preprocessing offloading. For example, as illustrated in **Figure 5.1a**, Sample A's size drops from 462KB raw JPEG to 151KB post RandomResizedCrop—when the image is cropped and resized to 224x224 pixels, with each pixel's R/G/B value represented by 1 byte. This suggests that offloading Decode and RandomResizedCrop to the remote storage server before network transmission can notably reduce data traffic.

Finding #2: The observation that the minimum file size occurs before the final preprocessing step suggests a decomposed preprocessing offloading approach. Existing offloading frameworks often consider the entire preprocessing as a singular unit [54, 145], missing opportunities for data traffic reduction. As depicted in **Figure 5.1a**, the file size post-Normalize is 4x larger than after RandomResizedCrop and RandomHorizontalFlip, due to the ToTensor operation converting pixel values from 1 byte per R/G/B to 4 byte floats. Thus, to minimize network traffic, an offloading framework should permit the selective offloading of specific preprocessing operations.

Finding #3: The varied impact of preprocessing offloading across different images necessitates finer-grained decision making. For instance, Sample B's smallest file size occurs in its raw JPEG format, as shown in **Figure 5.1a**. Thus, unlike Sample A, Sample B would transfer more efficiently without any preprocessing offloading. As further illustrated by **Figure 5.1b**, while 76% of OpenImages exhibit size reductions post certain preprocessing operations, 24% are smallest in their raw JPEG format and should not undergo offloading. A similar pattern is observed with ImageNet, with 26% of images benefiting from offloading, while 74% do not, underscoring the need for a tailored offloading strategy.

Finding #4: Preprocessing traffic reductions have varying CPU costs. Reflecting on previous research, preprocessing tasks are notably CPU-intensive [54, 113, 145]. When such tasks are offloaded to remote storage nodes, which typically possess lesser CPU capabilities compared to compute nodes, considerable CPU overhead is incurred. This scenario highlights the need for weighing this overhead against efficiency gains. To better understand this tradeoff, we measured preprocessing time for each operation and image in the 12GB subset, with the raw data cached in memory and processed using 8 CPU cores running in parallel. For each image, we then calculated the preprocessing time cost needed to offload in order to reach the minimum data transfer.

As illustrated in **Figure 5.1c**, the ratio of file size reduction to preprocessing time across the OpenImages dataset serves as an indicator of the trade-off between network traffic savings and the CPU time cost. Notably, 24% of images attain their minimum size in raw JPEG and thus exhibit

a ratio of 0 and do not need offloading. For the remaining 76% of images, this ratio varies, necessitating a refined offloading strategy that prioritizes images yielding the highest network traffic savings per unit of CPU time, particularly when CPU resources at the storage node are limited.

Finding #5: DL workloads exhibit varying demands for preprocessing offloading, which calls for customized offloading decisions. As depicted in **Figure 5.1d**, the GPU utilization across three distinct models—trained using the same configuration of a V100 GPU, ample CPUs, and constrained bandwidth to remote storage—is different. Specifically, ResNet50, with high GPU compute intensity, achieves near-maximal GPU utilization, rendering it less susceptible to gains from preprocessing offloading. Conversely, ResNet18, with lighter GPU compute requirements, spends about 65% of its time in a data-fetching idle state, suggesting considerable offloading benefits. Hence, the decision regarding preprocessing offloading should not only factor in image size and preprocessing time but also account for the resource demands and characteristics of each workload.

5.2 Design

Based on Finding #1, there exists a substantial opportunity to improve data fetching efficiency in DL training through the strategic offloading of preprocessing tasks. However, Findings #2-5 also highlight several challenges that need to be addressed in order to fully capitalize on these opportunities. An efficient offloading framework must: (1) assess the need for preprocessing offloading to mitigate network traffic for specific workloads, (2) choose appropriate data samples for preprocessing offloading, and (3) select the precise preprocessing operations to offload for each sample.

To tackle these challenges, we introduce SOPHON (Selecti- vely Offloading Preprocessing with Hybrid Operations Near-storage), a solution engineered to selectively offload DL preprocessing to remote storage servers, aiming at minimizing data transfer traffic. SOPHON is designed to systematically navigate each of these decision points, utilizing online data analysis and adaptive

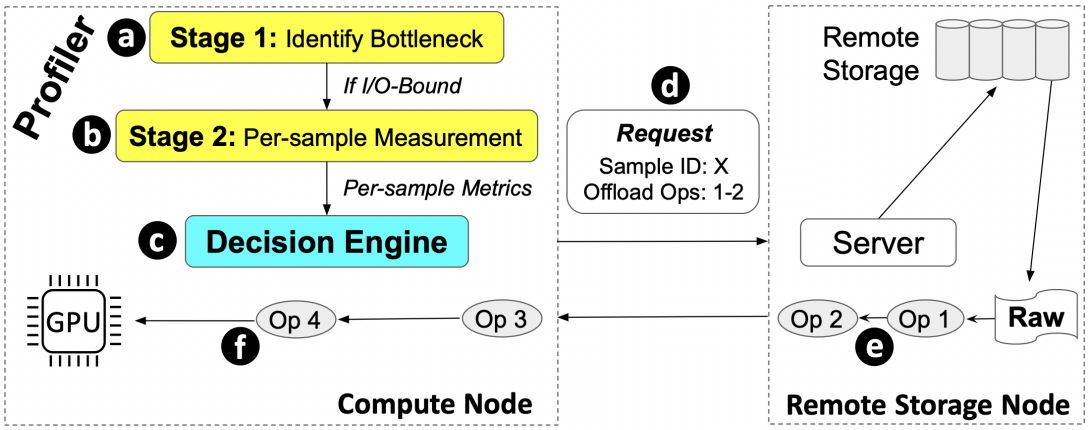


Figure 5.2: **SOPHON Design Overview.**

algorithms to tailor offloading decisions to the unique demands of each training scenario.

SOPHON has two key components: a two-stage profiler that collects essential metrics for informed offloading decisions, and a decision engine that determines the optimal offloading strategy using these metrics.

Figure 5.2 illustrates how SOPHON works. ⓐ The profiler first assesses GPU, CPU, and I/O throughput to determine if the given DL workload needs offloading to reduce I/O constraints. ⓑ If the workload is I/O-bound, SOPHON moves to the second profiling stage, noting the time and size changes for each preprocessing step across all samples. ⓒ With detailed metrics and knowledge of compute and storage node resources, SOPHON formulates the best offloading plan per sample. ⓓ Offloading directives for each sample are then incorporated into data fetch requests to the storage server, detailing the specific operations for offloading. ⓔ The storage server processes these operations as instructed, sending back the partially processed data to the compute node. ⓕ Finally, the compute node finishes any remaining preprocessing and forwards the data to GPUs for training.

5.2.1 Two-Stage Profiler

To minimize the profiling overhead while collecting essential metrics for informed offloading decisions, our approach harnesses a two-stage profiling process.

Inspired by Finding #5 and borrowing the idea from [113], the first stage briefly assesses the

primary bottleneck within the workload by measuring GPU, I/O, and CPU throughput. This is achieved by executing 50 batches under three distinct settings: (1) model training on the GPU using synthetic data to eliminate CPU or I/O delays, (2) data retrieval from remote storage, devoid of CPU or GPU processing to isolate I/O throughput, and (3) CPU-intensive preprocessing on the data cached during the second setting to gauge CPU throughput. This approach provides insights into the workload's throughput demands with minimal overhead (a typical training job spans over 50 epochs, each with thousands of batches). If the workload is I/O-bound, SOPHON proceeds to the second profiling stage; otherwise, it defaults to the standard training without offloading. CPU-bound scenarios may benefit from other solutions to mitigate preprocessing delays [54, 145].

In the second stage, we collect details on CPU time for preprocessing and changes in each sample's size through all preprocessing operations. This stage, requiring detailed, sample-specific data, involves processing the entire dataset, potentially incurring significant overhead. To minimize this, we use an on-the-fly profiling method: we proceed with the first training epoch without offloading any preprocessing tasks and collect essential per-sample metrics. This approach effectively lowers profiling overhead, facilitating efficient acquisition of performance data.

We currently assume identical CPU types on compute and storage nodes, allowing preprocessing times profiled on the compute node to be used for the storage node. We plan to explore heterogeneous CPU scenarios in the future.

5.2.2 Offloading Policy

Leveraging the metrics obtained through comprehensive profiling, SOPHON determines the most advantageous offloading strategy based on Findings #2-4.

Our strategy evaluates the potential size reduction and required preprocessing time for each data sample to reach its minimum size. Samples showing size reduction compared to their raw forms are considered for offloading. SOPHON measures *offloading efficiency* by the ratio of size reduction to preprocessing time, where a higher ratio suggests better potential for data traffic

reduction per CPU time spent. Therefore, samples are selected for offloading in descending order of efficiency, prioritizing those with the most significant effect on reducing data traffic.

SOPHON makes the decision based on four key metrics: (1) $\mathbf{T_G}$: The GPU time for one training epoch; (2) $\mathbf{T_{CC}}$: The CPU time on the compute node for local preprocessing, calculated as the total local preprocessing time divided by the CPU core count; (3) $\mathbf{T_{CS}}$: The CPU time on the storage node for offloaded preprocessing tasks, determined by dividing the total offloaded preprocessing time by the storage node's available CPU core count; (4) $\mathbf{T_{Net}}$: The time for data transfer from remote storage to the compute node over one epoch, derived from the total data traffic and the network bandwidth.

The first step uses a baseline profile without any offloading, characterized by T_{Net} as the predominant metric due to the I/O-bound nature of the workload and T_{CS} being 0 (no offloading). From this point, SOPHON selects the sample with the highest offloading efficiency, effecting a reduction in both T_{CC} and T_{Net} while elevating T_{CS}. The goal is to aggressively minimize network traffic until it ceases to be the limiting factor. This iterative selection of high-efficiency samples continues until either of two conditions is met: (1) T_{Net} ceases to be the predominant metric, or (2) no further samples with positive offloading efficiency remain. Through this algorithm, SOPHON minimizes network traffic without imposing excessive preprocessing load on the storage server.

5.2.3 Why Not Preprocess Just Once

One could contemplate a strategy where samples are selectively preprocessed just once to minimum size and then stored for reuse across all epochs. While this simplifies the process, it risks diminishing training accuracy. Random augmentations, typically applied during online preprocessing, are crucial for DL training accuracy and should be performed in each epoch. In contrast, our solution retains the original training's preprocessing logic and preserves accuracy.

5.3 Evaluation

We implement SOPHON in Python on top of PyTorch 1.8.0 in around 990 LOC. We utilize the gRPC framework to facilitate communication for data fetch requests and responses between the compute node and the remote storage server. Both the profiler and the offloading decision maker are encapsulated within a custom PyTorch data loader.

We evaluate SOPHON using small-scale emulation experiments with GPEMU and MLECEmu to quickly assess its performance and demonstrate its benefits. We plan to use more comprehensive and realistic settings in the future.

WORKLOAD BENCHMARKS: We benchmark the effectiveness of SOPHON on image classification tasks in deep learning training. To do this, we utilize the official PyTorch example training script, taken directly from its GitHub repository [21]. We use GPEMU to emulate the training of the AlexNet model, known for being compute-light and often bottlenecked by data fetching. Our experiments are conducted on subsets of two widely-used datasets in computer vision deep learning research: a 12GB subset of OpenImages [33] and an 11GB subset of ImageNet [127].

EXPERIMENT SETUP: We use a two-node setup: one serving as the compute node and the other as the storage server. Both nodes are equipped with two Intel Xeon Gold 6126 @ 2.60GHz processors, and, thanks to GPEMU, physical GPUs are not required. On the compute node, we allocate 48 logical cores to eliminate preprocessing bottlenecks present in the original workload, making I/O the primary bottleneck. GPEMU is used to emulate the training on an RTX-6000 GPU. The CPU allocation on the storage node is varied to evaluate the impact of CPU overhead introduced by offloading preprocessing.

To simulate a bandwidth-constrained environment, network throughput is capped at 500 Mbps. On the storage node, the data loader is connected to an emulated (4+1)/(4+1) MLEC storage system with 25 emulated disks, each throttled to a 200 Mbps read bandwidth. This configuration provides a total of 5 Gbps intra-cluster bandwidth. This setup reflects the common assumption that intra-

cluster bandwidth significantly exceeds inter-cluster bandwidth [48, 80, 88, 149].

SIMULATION OF REAL-WORLD CONFIGURATION: While we use small subsets that technically could fit into local storage, our experiments consistently fetch data from the remote storage node. This mimics real-world scenarios where datasets exceed local storage capacities. For example, the full OpenImages dataset totals 18TB [32, 103]. Each subset used in our experiments comprises over 40,000 images, randomly selected from the original dataset to represent the variety in sizes and preprocessing costs found in the full dataset.

We limit the network bandwidth to 500 Mbps to introduce an easy remote I/O bottleneck. In practical scenarios, training ResNet50 on ImageNet with 8 V100 GPUs requires nearly 16 Gbps of I/O bandwidth to fully utilize GPUs [103, 167], and even a 10 Gbps network could cause a significant remote I/O bottleneck.

BASELINES: We establish several baselines for comparison: **No-Off**, the original training pipeline without preprocessing offloading; **All-Off**, with all preprocessing operations of all samples offloaded to the storage node; **FastFlow** [145], a preprocessing offloading framework designed to alleviate CPU bottlenecks, which treats all preprocessing operations as a single unit and does not differentiate between data samples; and **Resize-Off**, which offloads only the Decode and RandomResizedCrop operations to the storage node, based on the observation that resizing reduces many images' sizes.

Our evaluation of SOPHON spans two distinct scenarios: one with ample CPU cores at the remote storage, and another where CPU resources are limited.

5.3.1 Ample CPU Cores on Storage Node

We start our evaluation using a storage node with ample (48) CPU cores to maximize the benefits of preprocessing offloading. **Figure 5.3** displays both training times and data traffic per epoch for all offloading policies.

All-Off has the longest training time across all policies, increasing data traffic by 1.9x for

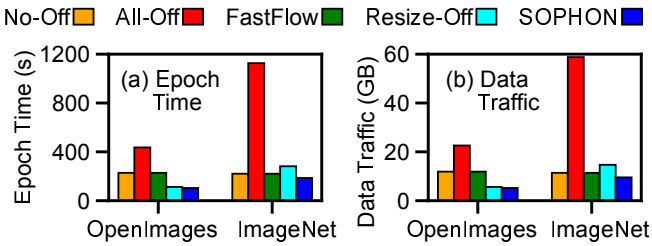


Figure 5.3: **SOPHON's effect with ample CPU cores.** SOPHON *cuts traffic/epoch time when storage node has ample CPUs.*

OpenImages and 5.1x for ImageNet when compared to No-Off. This is due to the data being converted into float tensors, which results in substantially larger sizes for most samples.

FastFlow chooses to not offload preprocessing in the evaluated setups. This decision is informed by its coarse-grained profiling analysis, which indicates that offloading all operations would lead to increased training time.

Resize-Off reduces data traffic by 2x for the OpenImages dataset compared to No-Off due to the large raw sizes of most images (76% of samples become smaller after Decode and RandomResizedCrop.) *However, Resize-Off increases traffic by 1.3x compared to No-Off for the ImageNet dataset* due to its smaller average image size (only 26% of samples become smaller after Decode and RandomResizedCrop.)

SOPHON improves performance for both datasets, thanks to its fine-grained offloading. For OpenImages, SOPHON achieves a 2.2x reduction in data traffic compared to No-Off, outperforming Resize-Off by not offloading preprocessing for samples that do not benefit. Unlike Resize-Off, SOPHON manages to reduce data traffic by 1.2x for ImageNet, thereby reducing training time. Selectively offloading preprocessing steps based on which images become smaller during preprocessing allows SOPHON to treat each sample optimally.

5.3.2 Limited CPU Cores on Storage Node

Next, we evaluate SOPHON's efficacy by varying the number of CPU cores allocated for preprocessing on the storage node. We focus on the OpenImages dataset, which has demonstrated greater benefits from preprocessing offloading. **Figure 5.4** shows the results for OpenImages.

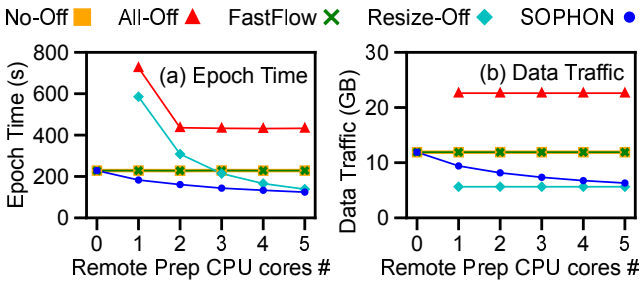


Figure 5.4: **SOPHON's effect with limited CPU cores.** SOPHON *finds the best solution when storage node has limited CPUs.*

All-Off results in the longest training time due to increased data traffic. Additionally, training time is further increased when only 1 CPU core is allocated for remote preprocessing, as the remote CPU overhead creates a bottleneck.

FastFlow consistently decides against preprocessing offloading as it anticipates that offloading *all* operations would increase the training time.

Resize-Off achieves the lowest data traffic among all configurations. However, its training time is not optimal as it offloads an excessive amount of preprocessing to the remote server, causing CPU overhead to create a new bottleneck. When the storage node has ≤ 2 CPU cores available for preprocessing, Resize-Off performs even worse than No-Off.

Finally, SOPHON exhibits the shortest training time among all policies, effectively balancing the trade-off between data traffic reduction and offloaded CPU overhead. Notably, there are diminishing returns for training time when allocating additional CPU cores. For instance, the transition from 0 cores to 1 leads to a 22-second reduction in epoch time, whereas moving from 4 to 5 CPU cores results in only a 9-second reduction. This demonstrates SOPHON's proficiency in selecting samples for offloading with the highest efficiencies, optimizing outcomes even under CPU constraints.

5.4 Use Cases and Limitations

While our work manages to reduce remote data traffic for specific DL training workloads, we understand that it might not help in some scenarios. Below we discuss the use cases of SOPHON

and scenarios where it might not be beneficial.

REMOTE I/O BOTTLENECK IN DL TRAINING: DL training often involves large datasets to enhance model accuracy. For instance, the Google OpenImages dataset totals 18TB [32, 103]. Such large datasets often exceed local storage capacities, leading to the use of remote cloud storage services to fetch data during training. In some GPU clusters, 97.3% of DL jobs store their data in cloud storage [167], resulting in a separation between GPU clusters and remote storage.

The rapid advancement in GPU computation speeds necessitates high-speed data transfers to avoid data stalls [66, 103, 105, 113, 116, 162, 167]. Furthermore, GPU clusters often run hundreds or thousands of DL training jobs simultaneously, putting substantial strain on the network between GPU clusters and remote storage. For example, a 400 V100 GPU cluster requires an aggregate I/O bandwidth of 200Gbps [167], while Azure's maximum egress bandwidth is only 120Gbps [41]. This remote I/O bottleneck is likely to worsen in the future due to the fast evolvement of GPUs [167].

SOPHON is effective for such scenarios where remote I/O bottlenecks may occur, as it reduces remote data traffic.

DL TRAINING WITH ESSENTIAL NEED FOR ONLINE PREPROCESSING: Many DL training jobs, especially computer vision models, require online preprocessing to enhance training accuracy. These workloads provide opportunities for SOPHON to reduce data traffic via selective preprocessing offloading.

NEAR STORAGE PROCESSING SUPPORT: Modern cloud storage services increasingly support near-storage data processing, facilitating the offloading strategies of SOPHON. For instance, Ceph enables near-storage data processing through dynamic object interfaces [154]. Similarly, Amazon S3 Object Lambda allows users to submit custom data processing code that is executed automatically before data is returned [5].

SCENARIOS WHERE SOPHON MIGHT NOT WORK: SOPHON may not help for Large Language Models (LLMs), where input data preprocessing is less critical for accuracy, limiting op-

portunities for preprocessing offloading. Though LLMs are becoming more popular, a substantial number of DL training jobs still exist in current clusters. This is because LLMs are often very expensive to run, while many DL models can achieve satisfactory accuracy at a much lower cost.

SOPHON assumes CPU-based preprocessing and currently doesn't support GPU-based strategies like NVIDIA DALI [30]. However, we believe our findings also points to new opportunities in GPU-based preprocessing scenarios. For example, one can selectively split preprocessing tasks between GPUs and CPUs to reduce CPU-GPU data transfers.

SOPHON doesn't help when the entire dataset can fit into local storage and thus remote I/O is not needed. For such training scenarios, many prior works have focused on alleviating the potential local I/O bottleneck through effcient caching and prefetching strategies[105, 113, 116].

5.5 SOPHON and Multi-Level Erasure Coding

We have introduced SOPHON, which effectively reduces network traffic between the compute cluster and storage cluster through selective preprocessing offloading. While the idea of SOPHON can be applied to any remote storage system, the nature of MLEC introduces specific challenges that may arise, which we hope future work will address. Here, we discuss some of these potential challenges.

Our current implementation assumes that preprocessing is offloaded to a centralized server within the remote storage cluster, which first gathers data from the MLEC storage cluster and then performs the preprocessing tasks. However, this approach could create significant overhead on the centralized node. A more efficient strategy would be to distribute the preprocessing work across each rack's server, which would not only decentralize the workload but also reduce inter-rack traffic within the cluster.

However, the nature of MLEC poses a challenge: the data is split and erasure-coded across multiple racks, with each rack storing only one chunk of the data stripe. Since erasure coding is applied at the block level rather than the file level, the encoder does not fully understand the file

structure. As a result, a single data sample (e.g., an image) could be spread across multiple racks, meaning that a single rack's server may not have access to the entire sample locally. This creates a challenge in efficiently distributing preprocessing tasks while maintaining data integrity.

Several possible solutions could address this challenge:

- **Cross-Rack Coordination for Complete Sample Reconstruction**: Implementing a cross-rack coordination mechanism where rack servers collaborate to reconstruct full samples before preprocessing could decentralize the workload, though it would introduce inter-rack communication overhead.
- **Selective Rack-Level Preprocessing Based on Data Locality**: Rack servers could preprocess only the chunks stored locally, with the compute node handling final aggregation. This would reduce inter-rack communication but require more sophisticated preprocessing logic.
- **Padding with Chunk Size Optimization**: Another approach is to add padding when storing data, ensuring that a complete file (e.g., an image) is stored within one rack. Proper selection of chunk size based on file size distributions can minimize padding overhead while maintaining the benefits of rack-local preprocessing.

Further complexity arises if we aim to offload preprocessing tasks directly to the storage devices, leveraging the compute power of technologies like computational storage [94]. This adds challenges because, in MLEC, each network-level chunk is divided into smaller local chunks for local erasure coding, with each disk storing only a small portion of the local chunk. In this case, additional solutions are required to address the complexities introduced by splitting the data across both network and local levels.

5.6 Conclusion

We reveal opportunities and challenges in reducing data traffic through strategic preprocessing offloading in DL training. We propose SOPHON, a framework that selectively offloads prepro-

cessing tasks to minimize data traffic, utilizing online profiling and adaptive algorithms to optimize for every sample. Our emulation results demonstrate that SOPHON can effectively reduce data traffic and training time.

CHAPTER 6

OTHER STORAGE WORK

In this chapter, we briefly mention our other work on availability and reliability of cloud storage systems.

6.1 Layered Contention Mitigation for Cloud Storage

In this paper [150], we introduce an ecosystem of contention mitigation supports within the operating system, runtime and library layers. This ecosystem provides an end-to-end request abstraction that enables a uniform type of contention mitigation capabilities, namely request cancellation and delay prediction, that can be stackable together across multiple resource layers. Our evaluation shows that in our ecosystem, multi-resource storage applications are faster by 5-70% starting at 90P (the 90thpercentile) compared to popular practices such as speculative execution and is only 3% slower on average compared to a best-case (no contention) scenario.

6.2 From Failure to Insight: Analyzing Disk Breakdowns in Large-Scale HPC Environments

Disk failure data provides valuable insights for preventing failures, enhancing storage robustness, guiding system design and deployment, and ensuring reliable operations at data centers. In our colloborated paper [71] with ORNL and LANL, we introduced two disk failure datasets collected from large-scale HPC production environments over the past five years, comprising over 5,000 failure records from more than 40,000 disks. We analyzed these datasets across multiple dimensions, including temporal, spatial, and relational trends, and performed a comprehensive reliability assessment. Our analysis yielded numerous observations and insights that influence various operational aspects of HPC storage systems. We believe this study offers a holistic understanding of disk failure trends likely to interest the HPC storage community.

CHAPTER 7

CONCLUSION AND FUTURE WORK

In this chapter, we conclude and discuss potential future directions for this work.

7.1 Conclusion

In this document, we have provided the most comprehensive design considerations and analysis of multi-level erasure-coded (MLEC) storage systems and explored its relationship to deep learning (DL) workloads.

We presented comprehensive design considerations and analysis of MLEC at scale, highlighting its multi-dimensional design space. We quantified MLEC's performance and durability using various evaluation strategies, identifying the MLEC schemes and repair methods that offer the best failure tolerance and significantly reduce repair network traffic. We also demonstrated that MLEC provides high durability, superior encoding throughput, and lower repair network traffic compared to other erasure coding (EC) schemes.

We introduced an emulation-based approach to evaluate MLEC storage under DL workloads. This includes GPEMU, a GPU emulator that facilitates fast and cost-effective evaluation and prototyping of deep learning system research without the need for real GPUs. We also introduced MLECEmu, a storage emulator that enables the evaluation of MLEC storage without requiring physical disk arrays. Using these emulators, we demonstrated how to evaluate MLEC storage under DL workloads and identify system bottlenecks.

Additionally, we have proposed SOPHON, a framework that selectively offloads preprocessing tasks at a fine granularity to minimize cross-cluster data traffic between DL training and remote MLEC storage. SOPHON uses online profiling and adaptive algorithms to optimize for each sample. We emulated SOPHON in DL workloads with remote MLEC storage, and our results show that SOPHON effectively reduces data traffic and training time.

7.2 Future Work

Exploring Additional Levels of Erasure Coding

We introduced multi-level erasure coding, focusing on two levels: network-level and local-level erasure coding. A potential future direction is to explore erasure coding with three or more levels. This approach could enhance durability for extremely large data storage systems, where data crosses multiple clusters or even zones, each with varying levels of network traffic, such as inter-zone, inter-cluster, inter-rack, and intra-rack traffic.

On one hand, redundancy across clusters would protect against cluster-level failures. On the other hand, bandwidth constraints increase at higher levels, for example, inter-cluster bandwidth is more limited than intra-cluster bandwidth. This calls for erasure coding across more levels, which introduces trade-offs in durability, storage overhead, computational complexity, and network traffic.

With more levels, analysis becomes more complex; simulations will involve more disks and might take longer to run. Exploring this direction could be valuable for scaling data storage systems.

Implementing Advanced MLEC Repair Methods

We demonstrated the configuration of MLEC storage using HDFS and ZFS, which formed the basis for MLECEmu. However, the current implementation only supports the R_{ALL} repair method, which repairs the entire local ZPool during catastrophic local failures. It does not support advanced repair methods (R_{FCO}, R_{HYB}, R_{MIN}) that can reduce repair traffic and improve durability.

Implementing these advanced methods is challenging due to the complexity of coordinating HDFS and ZFS during repair. The key challenges include: (1) ZPool suspends I/O during catastrophic failures, halting repairs; (2) ZFS must signal HDFS for network-level repairs of specific chunks, requiring robust checking mechanisms and well-designed APIs; and (3) HDFS must sup-

port repairs of smaller local chunks. Future work could focus on overcoming these challenges and implementing advanced repair methods in real systems.

Extending SODp to the Network Level and Applying it to MLEC

We have analyzed the effect of applying SODp to MLEC. The original SODp [96] was designed for local-level SLEC, as a variation of local-Dp, but with improvements to tolerate concurrent failures. The original SODp has not yet been applied to network-level erasure coding, as it was focused on local SLEC. A potential future direction is to extend SODp to support network-level erasure coding, allowing the design of MLEC with SODp at the top level.

This introduces challenges, particularly in adapting chunk placement for SODp so that each chunk is placed on a different rack, a problem not encountered in local SODp. Future work could explore these challenges and find solutions to extend SODp to network-level erasure coding.

Implementing SODp in Real Systems and Applying it to MLEC

Another future direction is to implement SODp in real systems. The original design of SODp in the paper [96] was theoretical and evaluated only through simulations. It has not yet been implemented in a real system. A promising direction would be to implement the chunk placement of SODp in real systems, such as ZFS, which already supports local-Dp.

This would allow SODp to be implemented within the existing framework of local-Dp. Once implemented, SODp could be applied to MLEC by configuring HDFS on top of SODp-ZFS to create C/SODp or D/SODp MLEC storage clusters.

Evaluating MLEC with Real Failure Traces

The current analysis results focus on distribution-based failures. A future direction could be utilizing the simulator to evaluate trace-based failures. This would require obtaining real failure traces

from large data centers. Fortunately, recent work [71] has released failure traces from national lab data centers, which could be used for this analysis.

Extending GPEMU with More and Finer-Grained Features

We believe we have built the major features of GPEMU to facilitate faster prototyping and evaluation of deep learning system research across various scenarios. However, it is not our intention to encourage the community to use GPEMU as a full replacement for GPUs. We acknowledge that GPEMU can be extended with more detailed features. Below, we discuss potential future extensions.

LAYER-LEVEL PROFILING: Currently, our profiling operates at the model level. An enhancement would involve collecting runtime statistics at the layer level (e.g., linear and convolutional layers), allowing model computation times to be derived from the combination of these layers.

INTER-GPU COMMUNICATION TIME: GPEMU currently does not emulate inter-GPU communication time. We believe this can be profiled and emulated similarly to our existing time emulation and supported by extending GPEMU.

COMPREHENSIVE SUPPORT FOR DISTRIBUTED TRAINING: GPEMU can be extended to emulate the network communication for gradient synchronization during Distributed Data-Parallel Training. We also aim to support other techniques like Fully Sharded Data Parallel (FSDP) [172] and Model Parallelism (MP) [132].

SUPPORT DL INFERENCE: While our work has focused on DL training workloads, GPEMU could be extended to emulate DL inference, enabling the evaluation and reproduction of system optimization papers on DL inference [81, 118, 165].

GPU SPATIAL SHARING: The current sharing support in GPEMU focuses on time sharing. An extension to support spatial sharing would allow multiple DL jobs to run concurrently on a GPU, accounting for interference between jobs [137, 147, 159].

SUPPORT LARGE LANGUAGE MODELS: GPEMU has mainly focused on traditional DL workloads. Given the rapid growth and popularity of large language models (LLMs), it would be valuable to extend GPEMU to emulate LLMs and foundation models.

Enhancing MLECEmu for Finer-Grained Performance and Latency Emulation

The current implementation of MLECEmu uses constant bandwidth throttling to control the throughput of emulated in-memory disks. However, in reality, disk throughput is not constant and varies based on factors such as file size and file location on the disk. A potential future extension would be to account for these factors to provide more realistic emulations.

Additionally, we currently only emulate throughput. Emulating disk latency would allow us to evaluate MLEC storage performance for latency-sensitive workloads, such as deep learning inference. This requires considering the characteristics of hard drives (such as seek time) and SSDs. This can be achieved by integrating existing disk simulators/emulators like DiskSim [44] and FEMU [107].

Evaluating SOPHON with More Complex Configurations and Workloads

The current analysis and evaluation of SOPHON use a small workload and small-scale emulations to quickly demonstrate its benefits. Future work could involve evaluating SOPHON with more realistic workloads and exploring its potential across other datasets and deep learning training categories, such as audio classification, video processing, and natural language tasks.

Extending SOPHON for Advanced Data Compression and Multi-Tenant Support

Data can be compressed to reduce size. Future work could involve designing a strategy to selectively compress preprocessed data, further reducing data traffic while considering potential increases in CPU overhead. This introduces a new trade-off between CPU overhead and traffic reduction, which could be an interesting area for exploration.

Additionally, future work could extend SOPHON to support environments with heterogeneous CPU types across compute and storage nodes.

Moreover, SOPHON could be explored in multi-tenant environments, where multiple jobs require preprocessing offloading. Future research could focus on developing a scheduler to efficiently allocate storage-side CPUs among multiple jobs, maximizing global training efficiency.

Extending SOPHON for Distributed Preprocessing in MLEC Setups

Another future direction for SOPHON is to expand its functionality to support distributed preprocessing in MLEC setups. The current implementation offloads preprocessing tasks to a centralized server within the remote storage cluster, which first gathers data from the MLEC storage cluster and then performs preprocessing. However, this centralized approach can create significant overhead on the server. A more efficient strategy could distribute preprocessing tasks across each rack's server, decentralizing the workload and reducing inter-rack traffic.

MLEC, however, presents a challenge: data is split and erasure-coded across multiple racks, with each rack storing only a single chunk of a data stripe. Since erasure coding operates at the block level rather than the file level, the system lacks full awareness of the file structure. As a result, a single data sample (e.g., an image) may be distributed across multiple racks, meaning that no single rack server has the complete sample locally, which complicates efficient distribution of preprocessing tasks.

Future work could explore solutions to address this challenge, such as implementing cross-rack coordination to reconstruct complete samples before preprocessing, or applying selective rack-level preprocessing based on data locality, where only locally stored chunks are processed, with final aggregation performed by the compute node. Another approach could involve adding padding to ensure that entire files (e.g., images) are stored within a single rack, with optimized chunk sizes based on file distribution to minimize padding overhead.

Another potential future direction is to explore offloading preprocessing directly to storage

devices, leveraging the computational power of emerging technologies like computational storage [94]. This approach introduces new challenges, as MLEC divides each network-level chunk into smaller local chunks, with each disk storing only a part of the local chunk. Future work could focus on managing these complexities, particularly in coordinating data across both network and local levels.

REFERENCES

[1] Hierarchical Erasure Coding: Making Erasure Coding Usable. https://www.snia.org/sites/default/files/SNIA_Hierarchical_Erasure_Coding_Final.pdf.

[2] 8.4. Configuration Tools Red Hat Enterprise Linux 7 — Red Hat Customer Portal. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/performance_tuning_guide.

[3] Allox GitHub. https://github.com/lenhattan86/allox.

[4] Amazon S3. https://aws.amazon.com/s3/.

[5] Amazon S3 Object Lambda. https://aws.amazon.com/s3/features/object-lambda/.

[6] Amazon SageMaker. https://aws.amazon.com/sagemaker/.

[7] Azure Blob Storage. https://azure.microsoft.com/en-us/products/storage/blobs.

[8] Azure Machine Learning - ML as a Service. https://azure.microsoft.com/en-us/products/machine-learning.

[9] Cgroups. https://en.wikipedia.org/wiki/Cgroups.

[10] Chameleon - A configurable experimental environment for large-scale cloud research. https://www.chameleoncloud.org.

[11] CORVAULT - Self-Healing, High Density Data Storage. https://www.seagate.com/products/storage/data-storage-systems/corvault/.

[12] Declustered raid. https://www.ibm.com/support/knowledgecenter/en/SSYSP8_5.1.0/com.ibm.spectrum.scale.raid.v4r23.adm.doc/bl1adv_introdeclustered.htm.

[13] Dell PowerEdge RAID Controller 12. https://infohub.delltechnologies.com/p/dell-poweredge-raid-controller-12/.

[14] Device Plugins — Kubernetes. https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/.

[15] Emulating multipath wireless links on CloudLab and FABRIC. https://witestlab.poly.edu/blog/emulating-multipath-wireless/.

[16] FastFlow GitHub. https://github.com/SamsungLabs/FastFlow.

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

[18] Google Cloud Deep Learning VM Images. https://cloud.google.com/deep-learning-vm.

[19] Gpemu github repository. https://github.com/mengwanguc/gpemu.

[20] How to Optimize Data Transfers in CUDA C/C++. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/.

[21] ImageNet Training in PyTorch. https://github.com/pytorch/examples/tree/main/imagenet.

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

[23] Linux I/O schedulers. https://wiki.ubuntu.com/Kernel/Reference/IOSchedulers.

[24] Loop device. https://en.wikipedia.org/wiki/Loop_device.

[25] MinIO GitHub. https://github.com/msr-fiddle/CoorDL.

[26] MLEC Artifact on Chameleon Trovi. https://tinyurl.com/mlec-artifact.

[27] MLEC Github repository. https://github.com/ucare-uchicago/mlec-sim.

[28] MLPerf Storage Benchmark Suite GitHub. https://github.com/mlcommons/storage.

[29] Muri GitHub. https://github.com/pkusys/Muri.

[30] NVIDIA DALI. https://developer.nvidia.com/dali.

[31] Object detection reference training scripts. https://github.com/pytorch/vision/tree/main/references/detection.

[32] Open Images Dataset Github. https://github.com/cvdfoundation/open-images-dataset.

[33] Open Images Dataset V7 and Extensions. https://storage.googleapis.com/openimages/web/index.html.

[34] ORNL's Alpine storage system. https://www.olcf.ornl.gov/olcf-resources/data-visualization-resources/alpine.

[35] Personal Communication with LANL, ORNL, and Seagate Engineers and Operators.

[36] PyTorch CUDA Asynchronous Execution. https://pytorch.org/docs/master/notes/cuda.html#asynchronous-execution.

[37] RabbitMQ: easy to use, flexible messaging and streaming — RabbitMQ. https://www.rabbitmq.com.

[38] Raid 5. https://searchstorage.techtarget.com/definition/RAID-5-redundant-array-of-independent-disks.

[39] Raid 6. https://searchstorage.techtarget.com/definition/RAID-6-redundant-array-of-independent-disks.

[40] RAMSSD GitHub. https://github.com/thustorage/ramssd.

[41] Scalability and performance targets for standard storage accounts. https://learn.microsoft.com/en-us/azure/storage/common/scalability-targets-standard-account.

[42] Scality ARTESCA: Object Storage for S3 Applications. https://www.scality.com/products/artesca/.

[43] Synergy GitHub. https://github.com/msr-fiddle/synergy.

[44] The DiskSim Simulation Environment (v4.0). http://www.pdl.cmu.edu/DiskSim/.

[45] Time-sharing GPUs on GKE — Google Kubernetes Engine (GKE) — Google Cloud. https://cloud.google.com/kubernetes-engine/docs/concepts/timesharing-gpus.

[46] Time-Slicing GPUs in Kubernetes — NVIDIA GPU Operator. https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/gpu-sharing.html.

[47] What is io-uring? https://unixism.net/loti/what_is_io_uring.html.

[48] Vitaly Abdrashitov, N. Prakash, and Muriel Médard. The storage vs repair bandwidth trade-off for multiple failures in clustered storage networks. In *IEEE Information Theory Workshop (ITW)*, 2017.

[49] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[50] Hiroaki Akutsu and Tomohiro Kawaguchi. Reliability analysis of distributed raid with priority rebuilding. In *Proc. USENIX Conf.*, 2013.

[51] Guillermo A. Alvarez, Walter A. Burkhard, and Flaviu Cristian. Tolerating Multiple Failures in RAID Architectures with Optimal Storage and Uniform Declustering. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, 1997.

[52] Guillermo A. Alvarez, Walter A. Burkhard, Larry J. Stockmeyer, and Flaviu Cristian. Declustered disk array architectures with optimal and near-optimal parallelism. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, 1998.

[53] GA Alverez, Walter A Burkhard, LL Stockmeyer, and Flaviu Cristian. Declustered disk array architectures with optimal and near-optimal parallelism. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*, pages 109–120. IEEE, 1998.

[54] Andrew Audibert, Yang Chen, Dan Graur, Ana Klimovic, Jiří Šimša, and Chandramohan A. Thekkath. tf.data service: A Case for Disaggregating ML Input Data Processing. In *Proceedings of the 14th ACM Symposium on Cloud Computing (SoCC)*, 2023.

[55] Sung Hoon Baek, Bong Wan Kim, Eui Joung Joung, and Chong Won Park. Reliability and performance of hierarchical RAID with multiple controllers. In *Proceedings of the 20st ACM Symposium on Principles of Distributed Computing (PODC)*, 2001.

[56] Ali Bakhoda, George L. Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[57] Yixin Bao, Yanghua Peng, Chuan Wu, and Zongpeng Li. Online Job Scheduling in Distributed Machine Learning Clusters. In *The 37th IEEE International Conference on Computer Communications (INFOCOM)*, 2016.

[58] Jeff Bonwick and Bill Moore. Zfs: The last word in file systems, 2007.

[59] Peter Braam. The lustre storage architecture. *arXiv preprint arXiv:1903.01955*, 2019.

[60] S. Chaudhary, R. Ramjee, M. Sivathanu, N. Kwatra, , and S. Viswanatha. Balancing efficiency and fairness in heterogeneous GPU clusters for deep learning. In *Proceedings of the 2020 EuroSys Conference (EuroSys)*, 2020.

[61] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. In *Computing Research Repository*, 2015.

[62] Zizhong Chen. Optimal real number codes for fault tolerant matrix operations. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2009.

[63] Liangfeng Cheng, Yuchong Hu, Zhaokang Ke, Jia Xu, Qiaori Yao, Dan Feng, Weichun Wang, and Wei Chen. LogECMem: Coupling Erasure-Coded In-Memory Key-Value Stores with Parity Logging. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[64] D. Colarelli and D. Grunwald. Massive Arrays of Idle Disks For Storage Archives. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC)*, 2002.

[65] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[66] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. Clairvoyant prefetching for distributed machine learning I/O. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[67] Zhuohui Duan, Haikun Liu, Xiaofei Liao, and Hai Jin. HME: A lightweight emulator for hybrid memory. In *Design Automation and Test in Europe (DATE)*, 2018.

[68] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[69] Daniel Ford, Franis Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlna. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[70] RM French. Catastrophic forgetting in connectionist networks. In *Trends Cogn Sci*, 1999.

[71] Anjus George, Meng Wang, Jesse Hanley, Garrett Wilson Ransom, John Bent, and Christopher Zimmer. From Failure to Insight: Analyzing Disk Breakdowns in Large-Scale HPC Environments. In *Fault Tolerance for HPC at eXtreme Scale (FTXS-24)*, 2024.

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

[73] Salvatore Di Girolamo, Daniele De Sensi, Konstantin Taranov, Milos Malesevic, Maciej Besta, Timo Schneider, Severin Kistler, and Torsten Hoefler. Building blocks for network-accelerated distributed file systems. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2022.

[74] Paul Glasserman, Philip Heidelberger, Perwez Shahabuddin, and Tim Zajic. Splitting for rare event simulation: analysis of simple cases. In *Proceedings of the 28th conference on Winter simulation*, pages 302–308, 1996.

[75] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. Deep Learning, 2016.

[76] Dan Graur, Damien Aymon, Dan Kluser, Tanguy Albrici, Chandramohan A. Thekkath, and Ana Klimovic. Cachew: Machine Learning Input Data Processing as a Service. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC)*, 2022.

[77] Alex Graves, Abdel rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.

[78] Kevin M. Greenan, James S. Plank, and Jay J. Wylie. Mean time to meaningless: MTTDL, Markov models, and storage system reliability. In *2nd USENIX Workshop on Hot Topics in Storage and File Systems (HOTSTORAGE)*, 2010.

[79] Juncheng Gu, Mosharaf Chowdhury, Kang G. Shin, Yibo Zhu, Myeongjae Jeon, Junjie Qian, Hongqiang Liu, and Chuanxiong Guo. Tiresias: A GPU Cluster Manager for Distributed Deep Learning. In *Proceedings of the 16th Symposium on Networked Systems Design and Implementation (NSDI)*, 2019.

[80] Shushi Gu, Fugang Wang, Qinyu Zhang, Tao Huang, and Wei Xiang. Global repair bandwidth cost optimization of generalized regenerating codes in clustered distributed storage systems. *IET Communications*, 15(19):2469–2481, 2021.

[81] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[82] James Lee Hafner and K Rao. Notes on Reliability Models for Non-MDS Erasure Codes. *IBM Res. rep. RJ–10391, 2006*, 2006.

[83] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[84] Frank Herold, Sven Breuner, and Jan Heichler. An introduction to beegfs. *ThinkParQ, Kaiserslautern, Germany, Tech. Rep*, 2014.

[85] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale Virtualization in the Emulab Network Testbed. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008.

[86] Mark Holland and Garth Gibson. Parity Declustering for Continuous Operation in Redundant Disk Arrays. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992.

[87] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. Performance Impact and Interplay of SSD Parallelism through Advanced Commands, Allocation Strategy and Data Granularity. In *Proceedings of the 25th International Conference on Supercomputing (ICS)*, 2011.

[88] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick P. C. Lee, Weichun Wang, and Wei Chen. Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage. In *Proceedings of the 19th USENIX Symposium on File and Storage Technologies (FAST)*, 2021.

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

[90] Jeffrey Thornton Inman, William Flynn Vining, Garrett Wilson Ransom, and Gary Alan Grider. Marfs, a near-posix interface to cloud objects. *; Login*, 42(LA-UR-16-28720; LA-UR-16-28952), 2017.

[91] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. Where Is My Training Bottleneck? Hidden Trade-Offs in Deep Learning Preprocessing Pipelines. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2022.

[92] Saurabh Kadekodi, Shashwat Silas, David Clausen, and Arif Merchant. Practical Design Considerations for Wide Locally Recoverable Codes (LRCs). In *Proceedings of the 21st USENIX Symposium on File and Storage Technologies (FAST)*, 2023.

[93] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. The Case for Unifying Data Loading in Machine Learning Clusters. In *The 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.

[94] Yangwook Kang, Yang suk Kee, Ethan L. Miller, and Chanik Park. Enabling Cost-effective Data Processing with Smart SSD. In *Proceedings of the 29th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2013.

[95] Zhuangwei Kang, Ziran Min, Shuang Zhou, Yogesh D. Barve, and Aniruddha Gokhale. Dataset Placement and Data Loading Optimizations for Cloud-Native Deep Learning Workloads. In *IEEE 26th International Symposium on Real-Time Distributed Computing (ISORC)*, 2023.

[96] Huan Ke, Haryadi S Gunawi, David Bonnie, Nathan DeBardeleben, Michael Grosskopf, Terry Grové, Dominic Manno, Elisabeth Moore, and Brad Settlemyer. Extreme protection against data loss with single-overlap declustered parity. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 343–354. IEEE, 2020.

[97] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020.

[98] Mahmoud Khairy, Zhesheng Shen, Tor M. Aamodt, and Timothy G. Rogers. Accel-sim: an extensible simulation framework for validated GPU modeling. In *Proceedings of the 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[99] Hyesoon Kim, Jaekyu Lee, Nagesh B Lakshminarayana, Jaewoong Sim, Jieun Lim, and Tri Pho. Macsim: A cpu-gpu heterogeneous simulation framework user guide. *Georgia Institute of Technology*, 2012.

[100] Jaeho Kim, Jongmin Lee, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Enhancing SSD reliability through efficient RAID support. In *Proceedings of the Asia-Pacific Workshop on Systems (APSys)*, 2012.

[101] Oleg Kolosov, Gala Yadgar, Matan Liram, Itzhak Tamo, and Alexander Barg. On Fault Tolerance, Locality, and Optimality in Locally Repairable Codes. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[102] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 26th Conference on Neural Information Processing Systems (NIPS)*, 2012.

[103] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An Informed Storage Cache for Deep Learning. In *Proceedings of the 18th USENIX Symposium on File and Storage Technologies (FAST)*, 2020.

[104] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. AlloX: compute allocation in hybrid clusters. In *Proceedings of the 2020 EuroSys Conference (EuroSys)*, 2020.

[105] Guillaume Leclerc, Andrew Ilyas, Logan Engstrom, Sung Min Park, Hadi Salman, and Aleksander Madry. FFCV: Accelerating Training by Removing Data Bottlenecks. In *2023 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023.

[106] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyung-Geun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. Refurbish Your Training Data: Reusing Partially Augmented Samples for Faster Deep Neural Network Training. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[107] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *Proceedings of the 16th USENIX Symposium on File and Storage Technologies (FAST)*, 2018.

[108] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, and Soumith Chintala. PyTorch Distributed: Experiences on Accelerating Data Parallel Training. In *Proceedings of the 46th International Conference on Very Large Databases (VLDB)*, 2020.

[109] Jie Liu, Bogdan Nicolae, and Dong Li. Lobster: Load Balance-Aware I/O for Distributed DNN Training. In *51st International Conference on Parallel Processing (ICPP)*, 2022.

[110] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. THEMIS: Fair and Efficient GPU Cluster

Scheduling. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[111] Krishna T. Malladi, Mu-Tien Chang, John Ping, and Hongzhong Zheng. FAME: A Fast and Accurate Memory Emulator for New Memory System Architecture Exploration. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2015.

[112] Jayashree Mohan, Amar Phanishayee, Janardhan Kulkarni, and Vijay Chidambaram. Looking Beyond GPUs for DNN Scheduling on Multi-Tenant Clusters. In *Proceedings of the 16th Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[113] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and Mitigating Data Stalls in DNN Training. In *Proceedings of the 47th International Conference on Very Large Databases (VLDB)*, 2021.

[114] Michael Moore, David Bonnie, Walt Ligon, Nicholas Mills, Becky Ligon, Mike Marshall, Elaine Quarles, and Sam Sampson. OrangeFS: Advancing PVFS. In *Proceedings of the 9th USENIX Symposium on File and Storage Technologies (FAST)*, 2011.

[115] Richard R. Muntz and John C. S. Lui. Performance analysis of disk arrays under failure. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB)*, 1990.

[116] Derek G. Murray, Jiří Šimša, Ana Klimovic, and Ihor Indyk. tf.data: A Machine Learning Data Processing Framework. In *Proceedings of the 47th International Conference on Very Large Databases (VLDB)*, 2021.

[117] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware Cluster Scheduling Policies for Deep Learning Workloads. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[118] Kelvin K. W. Ng, Henri Maxime Demoulin, and Vincent Liu. Paella: Low-latency Model Serving with Software-defined GPU Scheduling. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[119] Victor F Nicola, Perwez Shahabuddin, and Marvin K Nakayama. Techniques for fast simulation of models of highly dependable systems. *IEEE Transactions on Reliability*, 50(3):246–264, 2001.

[120] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD)*, 1988.

[121] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters. In *Proceedings of the 2018 EuroSys Conference (EuroSys)*, 2018.

[122] Jehan-François Pâris, S. J. Thomas J. E. Schwarz, Ahmed Amer, and Darrell D. E. Long. Highly reliable two-dimensional RAID arrays for archival storage. In *31th IEEE – International Performance Computing and Communications Conference (IPCCC)*, 2012.

[123] PA Rahman and G D'K Novikova Freyre Shavier. Analysis of Mean Time to Data Loss of Fault-Tolerant Disk Arrays RAID-6 based on Specialized Markov Chain. In *IOP Conference Series: Materials Science and Engineering*, volume 327, pages 022–086. IOP Publishing, 2018.

[124] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HOTSTORAGE)*, 2013.

[125] KV Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan Ramchandran. Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage and Network-bandwidth. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[126] Ohad Rodeh and Avi Teperman. zFS: A scalable distributed file system using object disks. In *Proceedings of the 12th NASA Goddard / 20th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, 2003.

[127] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. In *International Journal of Computer Vision (IJCV)*, 2015.

[128] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G. Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. In *Proceedings of the 46th International Conference on Very Large Databases (VLDB)*, 2013.

[129] Omar SEFRAOUI, Mohammed AISSAOUI, and Mohsine ELEULDJ. OpenStack: toward an open-source solution for cloud computing. In *International Journal of Computers and Applications (IJCA)*, 2012.

[130] Haiyang Shi and Xiaoyi Lu. TriEC: Tripartite Graph Based Erasure Coding NIC Offload. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2019.

[131] Haiyang Shi and Xiaoyi Lu. INEC: Fast and Coherent In-Network Erasure Coding. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2020.

[132] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

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

[134] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[135] Thomas J.E. Schwarz S.J., Jesse Steinberg, and Walter A. Burkhard. Permutation development data layout (PDDL). In *Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA-5)*, 1999.

[136] Dheeraj Sreedhar, Vaibhav Saxena, Yogish Sabharwal, Ashish Verma, and Sameer Kumar. Efficient Training of Convolutional Neural Nets on Large Distributed Systems. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018.

[137] Foteini Strati, Xianzhe Ma, and Ana Klimovic. Orion: Interference-aware, fine-grained gpu sharing for ml applications. In *Proceedings of the Nineteenth European Conference on Computer Systems (EuroSys)*, pages 1075–1092, 2024.

[138] Suhas Jayaram Subramanya, Daiyaan Arfeen, Shouxu Lin, Aurick Qiao, Zhihao Jia, and Gregory R. Ganger. Sia: Heterogeneity-aware, goodput-optimized ML-cluster scheduling. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP)*, 2023.

[139] Yifan Sun, Trinayan Baruah, Saiful A. Mojumder, Shi Dong, Xiang Gong, Shane Treadway, Yuhui Bao, Spencer Hance, Carter McCardwell, Vincent Zhao, Harrison Barclay, Amir Kavyan Ziabari, Zhongliang Chen, Rafael Ubal, José L. Abellán, John Kim, Ajay Joshi, and David Kaeli. MGPUSim: enabling multi-GPU performance modeling and optimization. In *Proceedings of the 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019.

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

[141] Alexander Thomasian. Multi-level RAID for very large disk arrays. In *ACM SIGMETRICS Performance Evaluation Review*, 2006.

[142] Alexander Thomasian and Yujie Tang. Performance, Reliability, and Performability Aspects of Hierarchical RAID. In *2011 IEEE Sixth International Conference on Networking, Architecture, and Storage (NAS)*, 2011.

[143] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications*. John Wiley and Sons Ltd., GBR, 2nd edition edition, 2001.

[144] Yuya Uezato. Accelerating XOR-based erasure coding using program optimization techniques. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2021.

[145] Taegeon Um, Byungsoo Oh, Byeongchan Seo, Minhyeok Kweun, Goeun Kim, and Woo-Yeon Lee. FastFlow: Accelerating Deep Learning Model Training with Smart Offloading of Input Data Pipeline. In *Proceedings of the 49th International Conference on Very Large Databases (VLDB)*, 2023.

[146] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th ACM Symposium on Cloud Computing (SoCC)*, 2013.

[147] Guanhua Wang, Kehan Wang, Kenan Jiang, Xiangjun Li, and Ion Stoica. Wavelet: Efficient DNN Training with Tick-Tock Scheduling. In *The Conference on Machine Learning and System*, 2021.

[148] Lipeng Wang, Songgao Ye, Baichen Yang, Youyou Lu, Hequan Zhang, Shengen Yan, and Qiong Luo. DIESEL: A Dataset-Based Distributed Storage and Caching System for Large-Scale Deep Learning Training. In *49st International Conference on Parallel Processing (ICPP)*, 2020.

[149] Meng Wang, Jiajun Mao, Rajdeep Rana, John Bent, Serkay Olmez, Anjus George, Garrett Wilson Ransom, Jun Li, , and Haryadi S. Gunawi. Design Considerations and Analysis of Multi-Level Erasure Coding in Large-Scale Data Centers. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2023.

[150] Meng Wang, Cesar A. Stuardo, Daniar H. Kurniawan, Ray A. O. Sinurat, and Haryadi S. Gunawi. Layered Contention Mitigation for Cloud Storage. In *IEEE 15th International Conference on Cloud Computing*, 2022.

[151] Meng Wang, Gus Waldspurger, and Swaminathan Sundararaman. A Selective Preprocessing Offloading Framework for Reducing Data Traffic in DL Training. In *16th USENIX Workshop on Hot Topics in Storage and File Systems (HOTSTORAGE)*, 2024.

[152] Neng Wang, Yinlong Xu, Yongkun Li, and Si Wu. OI-RAID: A Two-Layer RAID Architecture towards Fast Recovery and High Reliability. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2016.

[153] Zikun Wang and Xiangqun Yang. *Birth and death processes and Markov chains*. Springer-Verlag, 1992.

[154] Noah Watkins and Michael Sevilla. Using lua in the ceph distributed storage system. In *Proceedings of the Lua Workshop*, pages 16–17, 2017.

[155] Hakim Weatherspoon and John D Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems*, pages 328–337. Springer, 2002.

[156] Bingyang Wu, Zili Zhang, Zhihao Bai, Xuanzhe Liu, and Xin Jin. Transparent GPU Sharing in Container Clouds for Deep Learning Workloads. In *Proceedings of the 20th Symposium on Networked Systems Design and Implementation (NSDI)*, 2023.

[157] Huaxia Xia and Andrew A. Chien. RobuSTore: Robust Performance for Distributed Storage Systems. In *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2007.

[158] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. A Tale of Two Erasure Codes in HDFS. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[159] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, , and Lidong Zhou. Gandiva: Introspective Cluster Scheduling for Deep Learning. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[160] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[161] Lijie Xu, Shuang Qiu, Binhang Yuan, Jiawei Jiang, Cedric Renggli, Shaoduo Gan, Kaan Kara, Guoliang Li, Ji Liu, Wentao Wu, Jieping Ye, and Ce Zhan. In-Database Machine Learning with CorgiPile: Stochastic Gradient Descent without Full Data Shuffle. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2022.

[162] Chih-Chieh Yang and Guojing Cong. Accelerating Data Loading in Deep Neural Network Training. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019.

[163] Peifeng Yu and Mosharaf Chowdhury. Salus: Fine-Grained GPU Sharing Primitives for Deep Learning Applications. In *The Conference on Machine Learning and System*, 2020.

[164] Guangyan Zhang, Zican Huang, Xiaosong Ma, Songlin Yang, Zhufan Wang, and Weimin Zheng. RAID+: Deterministic and Balanced Data Distribution for Large Disk Enclosures. In *Proceedings of the 16th USENIX Symposium on File and Storage Technologies (FAST)*, 2018.

[165] Hong Zhang, Yupeng Tang, Anurag Khandelwal, and Ion Stoica. Shepherd: Serving dnns in the wild. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 787–808, 2023.

[166] Mi Zhang, Shujie Han, and Patrick PC Lee. A simulation analysis of reliability in erasure-coded data centers. In *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*, pages 144–153. IEEE, 2017.

[167] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Mingxia Li, Fan Yang, Qianxi Zhang, Binyang Li, Yuqing Yang, Lili Qiu, Lintao Zhang, and Lidong Zhou. SiloD: A Co-design of Caching and Scheduling for Deep Learning Clusters. In *Proceedings of the 2023 EuroSys Conference (EuroSys)*, 2023.

[168] Hanyu Zhao, Zhenhua Han, Zhi Yang, Quanlu Zhang, Fan Yang, Lidong Zhou, Mao Yang, Francis C.M. Lau, Yuqi Wang, Yifan Xiong, and Bin Wang. HiveD: Sharing a GPU Cluster for Deep Learning with Guarantees. In *Proceedings of the 14th Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[169] Hanyu Zhao, Zhi Yang, Yu Cheng, Chao Tian, Shiru Ren, Wencong Xiao, Man Yuan, Langshi Chen, Kaibo Liu, Yang Zhang, Yong Li, and Wei Lin. GoldMiner: Elastic Scaling of Training Data Pre-Processing Pipelines for Deep Learning. In *Proceedings of the 2023 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2023.

[170] Mark Zhao, Emanuel Adamiak, and Christos Kozyrakis. cedar: Composable and optimized machine learning input data pipelines. *arXiv preprint arXiv:2401.08895*, 2024.

[171] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Po. Understanding Data Storage and Ingestion for Large-Scale Deep Recommendation Model Training. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA)*, 2022.

[172] Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[173] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2022.

[174] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2018.