RESEARCH-ARTICLE

Flash-oriented Coded Storage: Research Status and Future Directions

ZHIYUE LI, Tsinghua University, Beijing, China

GUANGYAN ZHANG, Tsinghua University, Beijing, China

YANG WANG, The Ohio State University, Columbus, OH, United States

Published: 08 November 2024
Online AM: 19 December 2024
Accepted: 26 September 2024
Revised: 13 August 2024
Received: 15 November 2023

Open Access Support provided by:

Tsinghua University

The Ohio State University

Flash-oriented Coded Storage: Research Status and Future Directions

ZHIYUE LI, Department of Computer Science and Technology, Tsinghua University, Beijing, China
GUANGYAN ZHANG, Department of Computer Science and Technology, Tsinghua University, Beijing, China
YANG WANG, Department of Computer Science and Engineering, The Ohio State University, Columbus, United States

Flash-based solid-state drives (SSDs) have been widely adopted in various storage systems, manifesting better performance than their forerunner HDDs. However, the characteristics of flash media post some drawbacks when deploying SSD-based storage systems. First, flash media have limited program/erase cycles, making them vulnerable to media failures. Second, SSD foreground I/Os can suffer from inconsistent performance due to interference from background operations like garbage collection (GC). The major solution to the above problems is to introduce data redundancy. Redundant data can not only detect raw bit errors and recover lost data but also enable I/O scheduling to sidestep SSDs that are under performance degradation.

Compared with multi-replica, data coding is a more space-efficient way to provide redundancy. However, it is more challenging to simultaneously achieve low access latency, consistent performance, and fast recovery. This article examines the design of coded storage in existing storage systems, with a focus on flash storage systems, and how they address these challenges. The coded storage techniques are categorized into in-device coding, cross-device coding, and cross-machine coding. They are designed for different scenarios and purposes but share some design rationales in common. For each type of coded storage, we begin by presenting the theoretical bases, followed by an overview of how existing studies address the performance and endurance issues of coded storage from a systemic perspective. Finally, we review the history of coded storage, list several key insights from existing works, and speculate some promising directions for flash-oriented coded storage systems.

CCS Concepts: • **Information systems → Flash memory;**

Additional Key Words and Phrases: Flash storage, coded storage, ECC, RAID, erasure coding

ACM Reference Format:
Zhiyue Li, Guangyan Zhang, and Yang Wang. 2025. Flash-oriented Coded Storage: Research Status and Future Directions. *ACM Trans. Storage* 21, 1, Article 5 (January 2025), 37 pages. https://doi.org/10.1145/3708995

1 introduction

Flash-based **solid-state drives (SSDs)** are external storage devices based on flash media. Compared to their forerunner HDDs, SSDs have higher bandwidth and lower latency even under

This work was supported by the National Natural Science Foundation of China under Grant 62025203.
Authors' Contact Information: Zhiyue Li, Department of Computer Science and Technology, Tsinghua University, Beijing, Beijing, China; e-mail: sldjlzy@163.com; Guangyan Zhang (Corresponding author), Department of Computer Science and Technology, Tsinghua University, Beijing, Beijing, China; e-mail: gyzh@tsinghua.edu.cn; Yang Wang, Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, USA; e-mail: wang.7564@osu.edu.

ACM 1553-3077/2025/01-ART5
https://doi.org/10.1145/3708995

random workloads. So far, SSD-based storage has been widely adopted in various scenarios such as cloud service, high-performance computing, and big data analysis, demonstrating significant performance gains. For example, the Trinity supercomputer builds the SSD-based burst buffer to absorb burst I/Os [43]. Its burst buffer can provide 3.3 TB/s aggregated bandwidth, much higher than that (1.6 TB/s) of the HDD-based parallel file system. It is also demonstrated that parallel applications running on the SSD-based burst buffer can reduce the I/O time by 50% compared with directly using the underlying HDD-based storage [44].

However, the characteristics of flash media pose some drawbacks when deploying SSD-based storage systems. First, flash media are vulnerable to media failure, making the raw data stored in SSDs unreliable. Flash media have limited **program/erase cycles (P/E cycles)**, and the **raw bit error rate (RBER)** grows with flash media aging [35, 51, 142]. Second, SSD foreground I/Os can suffer from inconsistent performance due to interference from background operations like **garbage collection (GC)** [58, 75, 126]. Since SSDs are limited to out-of-place updates, they need to initiate GC when spare space becomes limited, potentially contending with active I/O operations for channel bandwidth.

The major solution to the above problems is to introduce data redundancy. Redundant data not only provide data protection but also enable I/O scheduling. First, redundant data can help to detect the raw bit errors in flash media or recover data that are lost or corrupted. Second, reading from a performance-degraded SSD can be avoided by reconstructing user data with redundant data. Both replication and data coding can provide data redundancy, but data coding is much more space-efficient than replication. For example, Meta previously utilized Haystack [7] to store warm BLOBs, employing a mix of triple replication and RAID-6 for safeguarding against disk, rack, and data center failures. However, due to the increasing quantity of warm BLOBs, Meta has transitioned to using f4 [92], which leverages Reed-Solomon codes and XOR operations to maintain equivalent fault tolerance. This shift has notably decreased the effective replication factor—from 3.6 to 2.11 [92]—thereby greatly enhancing storage efficiency.

Although coded storage is a space-efficient solution, it is challenging to simultaneously achieve low access latency, consistent performance, and fast recovery. For example, lower decoding latency and higher error-correction ability are contradictory goals for flash **error correction code (ECC)** [27, 51, 142]; partial stripe updates lead to I/O amplification and accelerate wearing in parity SSDs, presenting challenges for both **redundant array of inexpensive disks (RAID)** [86, 127] and distributed **erasure coding (EC)** [15, 49, 78] storage systems; accessing coded storage without proper management can intensify the performance variance caused by garbage collection; furthermore, data reconstruction for degraded reads can lead to I/O amplification, potentially overloading the entire system.

This article reviews how existing works tackle these challenges for coded storage, especially flash-oriented coded storage. We introduce a classification framework for SSD-based storage systems that takes into account their architectures. The architecture of an SSD-based storage system can range from a single-machine-single-device setup (such as a storage server with one SSD) to a single-machine-multiple-devices configuration (like a storage server equipped with an SSD array) or even a multiple-machines-multiple-devices arrangement (as seen in distributed SSD storage systems).

For each architectural category of storage systems, we provide a comprehensive summary of the coding techniques applicable to various storage types and examine the corresponding works. In a single-machine-single-device architecture, coding options include flash ECCs capable of detecting and correcting raw bit errors or EC implemented across multiple flash chips to withstand chip failures. In a single-machine-multiple-devices setup, RAID [96] can be employed to provide resilience against device failures within a single machine. In a multiple-machines-multiple-devices

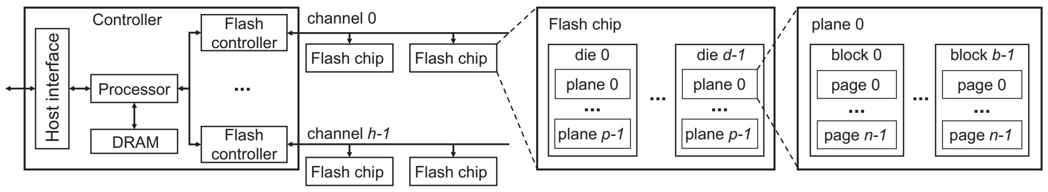


Fig. 1. Typical internal architecture of flash-based SSDs.

arrangement, distributed EC is capable of withstanding severe failures, including machine-level, rack-level, or even data-center-level failures. Although a storage system might employ multiple coding techniques at the same time, for the sake of clarity, we categorize them distinctly as in-device coding, cross-device coding, and cross-machine coding. For each coding, we outline the theoretical bases and present a comprehensive review of existing research based on optimization objectives such as minimizing decoding latency, mitigating tail latency, and enhancing the efficiency of partial stripe updates.

To inspire further research, we examine the evolution of flash storage and its associated coding techniques. Additionally, we extract key insights from existing designs of flash-based storage systems and offer conjectures on the prospective developments in coded SSD storage systems.

2 Background

In this section, we introduce the foundational background for flash-oriented coded storage. Section 2.1 outlines the typical architecture and mechanism of an SSD along with an examination of the reliability and performance issues in SSD storage. Section 2.2 provides an overview of data coding principles and the symbols used in this article. Finally, Section 2.3 reviews the architectures of existing flash storage solutions and discusses the associated data coding techniques.

2.1 Architecture and Mechanism of Flash SSDs

SSDs are solid-state storage devices that utilize integrated circuits to provide high-performance and persistent storage. Unlike HDDs, SSDs do not rely on moving parts to access the media, making them more friendly to random I/Os. This article focuses on the widely adopted flash-based SSDs that use flash [87, 88] as storage media, although in recent years some SSDs [53] are equipped with more advanced media (e.g., 3D Xpoint [123]) and present very different characteristics compared to flash-based SSDs [37, 125, 135]. In this article, we use SSD to refer to a flash-based SSD.

Figure 1 shows the typical architecture of a flash-based SSD. The flash controller exchanges data with the host by host interfaces. A DRAM is connected to the controller to buffer data and the mapping table of ***flash translation layer*** **(FTL)**. The controller connects to multiple channels that can be accessed in parallel. Each channel has multiple *flash chips*, and a flash chip has multiple logical units inside, namely, *die* (or *package*), *plane*, *block,* and *page*. *Cell* is the physical unit that stores data with voltage states. A cell with 2^m voltage states can store m bits. Flash that can save 1, 2, 4, or 8 bits per cell are called SLC, MLC, TLC, and QLC flash.

The characteristics of flash media decide the special access mechanism of SSDs. First, a physical page must be erased before being programmed again. This means that SSDs can only perform *off-place* update instead of *in-place* update. Second, erasing and programming have different granularities. Each page can be read or programmed independently, but all pages within a block must be erased together. When a page is updated, the old physical page is not erased immediately but is marked as *invalid* instead. SSDs employ background operations called *Garbage Collection* (GC) to reclaim these invalid pages. GC is triggered when the number of free pages is lower than a vendor-specific threshold. A block to erase may contain both valid and invalid pages. Before erasing, the valid pages must be copied to another block first, which will consume additional channel bandwidth. During off-place updates or GC, the physical position of a logical page may change. The FTL in an SSD maintains a mapping between the ***logical block address*** **(LBA)** and ***physical block address*** **(PBA)**. An SSD exposes a logical address space externally, and FTL translates the I/O logical addresses to relative physical addresses.

The above characteristics lead to two drawbacks for SSD storage. First, flash media are vulnerable to media failure, making the raw data stored in SSDs unreliable. Multiple factors can cause raw bit errors in flash media, such as program errors, cell-to-cell interference, data retention errors, and read disturb errors [10]. Besides, the reliability of flash media decreases with increasing P/E cycles [35, 51, 142]. These problems are more severe in high-density media that can store multiple bits per cell [132, 136, 138]. Apart from SLC, reading a flash cell needs to sense its voltage multiple times with different reference voltages, as each sense can only tell whether the cell voltage is higher or lower than the reference voltage instead of giving the precise voltage [142].

Second, SSD foreground I/Os can suffer from inconsistent performance due to interference from background operations like GC [58, 75, 126]. Performing GC on blocks that contain valid pages needs to copy out the valid pages first, which will incur extra loads on flash channels. These data copy I/Os can share the same queues with foreground I/Os at device level, channel level, or chip level [134], resulting in latency spikes of foreground I/Os. Besides, when SSDs are regarded as black-box devices, it is challenging for the host to know when GC is performing [58]. This will cause the host to suffer from unexpected performance degradation.

2.2 Data Coding Preliminary

Data coding is a space-efficient way to protect user data. The smallest unit that participates in data coding calculation is called a *symbol*. For binary user data, a symbol is a w-bit element from a *Galois* field (also called finite field) $GF(2^w)$ [102]. The addition and multiplication of the symbols are also defined in the Galois field. Addition in Galois field $GF(2^w)$ is equivalent to bit-wise ***exclusive OR*** **(XOR)**. In this article, we refer to this Galois field addition as XOR.

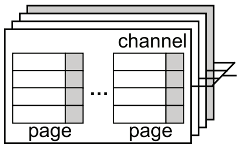
During data encoding, user data are divided into many data symbols. K data symbols are encoded into a *codeword* that consists of n encoded symbols. The number of redundant symbols is marked as m, which satisfies $m = n - k$. The *code rate* R is used to measure the extra storage overhead introduced by data coding:

$$R = \frac{k}{n}, 0 < k < n. \tag{1}$$

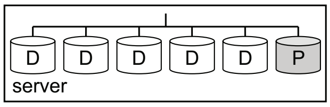
A higher code rate means the data coding is more space-efficient, but it may be weaker for data protection.

A coding either belongs to *systematic code* or *non-systematic code*. If the codeword contains the data symbols, then such a coding is called a systematic code. Otherwise, the coding is called a non-systematic code. The codeword of a systematic code can be regarded as a concatenation of k data symbols and $(n - k)$ parity symbols. A systematic code has the advantage that reading data symbols does not need to decode the whole codeword if no error happens.

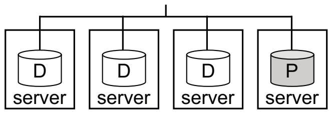
Data coding can be expressed in two forms, named *matrix form* and *polynomial form*. For the matrix form, data symbols and the codeword are represented as column vectors $\mathbf{x}$ and $\mathbf{y}$. Each coding has a *generation matrix* G that can encode x with $\mathbf{y} = G\mathbf{x}$. Every generation matrix has an equivalent *parity-check matrix* C that is used for error correction. For a received codeword $\mathbf{y}'$ to decode, the equation $C\mathbf{y}' = 0$ holds if there are no detectable errors in the codeword y'. This equation may also hold by chance if the number of errors exceeds the error correction ability of the coding.



(a) In-device coding.



(b) Cross-device coding



(c) Cross-machine coding

Fig. 2. Architectures of coded SSD storage. This figure only demonstrates the systematic codes, and the parity symbols are marked as gray. Besides, it omits data layout and assumes all parity symbols are in the same channel/device.

For the polynomial form, data symbols are represented as a polynomial. Assuming that the data symbols are $\{x_{k-1}, x_{k-2}, \ldots, x_1, x_0\}$, they are written as a $(k-1)$-order polynomial:

$$\mathbf{x}(D) = x_{k-1}D^{k-1} + x_{k-2}D^{k-2} + \cdots + x_1 D + x_0. \tag{2}$$

The generation polynomial $g(D)$ is an $(n-k)$-order polynomial whose coefficients are elements from the Galois field. The codeword polynomial $\mathbf{y}(D)$ can be generated by:

$$\mathbf{y}(D) = g(D)\mathbf{x}(D). \tag{3}$$

The redundant data may be used for different purposes. A coding can be an ***error detection code* (*EDC*)**, *ECC*, or *EC*. EDC, such as CRC and longitudinal/horizontal redundancy check, can detect errors in the received codeword with high code rates, but it cannot correct the errors. ECC can not only detect but also correct errors in a received codeword by adding more redundant data. Typical ECCs are Hamming code [38], Reed-Solomon code [99], and LDPC code [34]. EC can reconstruct lost or corrupted symbols, which can be detected by external methods such as a broken device, network failures, or failing ECC checks. With this additional information, EC can be constructed by EDC (like RAID-4 and RAID-5 [96] have the same coding with longitudinal/horizontal redundancy check) or ECC (RS code[99] used in distributed coded storage). In this article, we call data that are lost or corrupted as failed data.

2.3 Coded SSD Storage

The architectures of existing SSD storage can be categorized into three types. The single-machine-single-device architecture only uses one SSD as storage, like personal computers. The single-machine-multiple-devices architecture combines multiple SSDs into an array to build a high-performance and high-reliability storage server like the emerging large **all-flash arrays (AFAs)** [24, 31, 93]. The multiple-machines-multiple-devices architecture is a distributed storage system that aggregates multiple SSD servers as a whole. It can be used in data center storage like the public cloud storage [78].

All three kinds of SSD storage systems have reliability issues that the stored data may not be able to be read correctly. For single-machine-single-device storage, raw bit errors may occur in flash media (see Section 2.1). They are so ubiquitous in SSDs that the controller must use ECCs on raw pages to detect and correct potential errors (see Figure 2(a)). With the aging of an SSD, its ECC may not be able to correct all the raw bit errors. To prevent this issue, an SSD can perform another layer of coding across different channels. This additional redundancy can handle failures that are uncorrectable for ECC. We call the encoding techniques that are used within an SSD as *in-device coding*.

In single-machine-multiple-devices storage, data failures may happen due to various factors, such as broken devices or uncorrectable errors in the in-device coding. By building an EC across different devices, the redundant data can help to reconstruct the failed data (see Figure 2(b)). We denote these coding techniques as cross-device coding.

The multiple-machines-multiple-devices storage system also experiences data failure issues. Compared to the single-machine-multiple-devices storage, its failure root causes are more diverse, including broken machines, network failures, rack-level failures, or even the shutdown of an entire data center. Failed data of these kinds can be recovered with EC on SSDs of different machines (see Figure 2(c))). We call these coding techniques as cross-machine encoding.

For cross-device coding and cross-machine coding, a typical symbol size (like 8bit [141]) is so small that it is not friendly to SSD I/Os and the network transmission. Therefore, these data coding techniques typically encode data in a batch manner with a larger unit (like 4 KB or larger) called *chunk*. For each encoding operation, data chunks are encoded to generate one or more parity chunks. These chunks together form a *stripe*.

In this article, we refer to the SSD storage systems that use the above coding techniques as *coded SSD storage*. Note that a storage system may not use only one coding technique. For example, a single-machine-multiple-devices storage can use both in-device coding and cross-device coding, and a cross-machine coding can use both cross-device coding and cross-machine coding to form a hierarchical coding structure. For these cases, existing works typically consider each coding layer separately and treat the lower layers as black boxes, and we adopt this methodology as well.

3 In-device Coding

In this section, we first introduce the theoretical bases of in-device coding from a mathematical perspective. However, applying the coding directly to storage systems poses challenges from a systemic viewpoint. Consequently, we delve into existing works on coded storage systems designed to minimize write amplification, reduce soft-decision decoding latency, control tail latency, and mitigate raw bit errors.

3.1 Theoretical Bases

In-device coding consists of two main types. The first one is the ECC within pages, which detects and fixes raw bit errors. Commonly used codes are BCH code and LDPC code. The second one is cross-chip coding, which shields against uncorrectable errors in ECC or entire chip failures. This is done using the redundancy mechanism RAISE.

3.1.1 BCH Code. ***Bose-Chaudhuri-Hocquenghem code*** **(BCH code)** [45] is a class of cyclic ECCs that are built with polynomials over a Galois field. It is named after its inventors, R. C. Bose, D. K. Ray-Chaudhuri, and A. Hocquenghem. The key parameter in a BCH code is triplet (n, k, t), in which n is the number of symbols in a codeword and k is the number of data symbols. It can detect and correct no more than t bit flips in any position. BCH code is a generalization of some well-known codes, such as Hamming code [38] and Reed-Solomon code [99].

The construction of a BCH code can be summarized as follows: Here, we only talk about the primitive BCH code, in which

$$n = p^w - 1, w \in \mathbb{N}_+, p \text{ is a prime number.} \tag{4}$$

In binary data coding, p typically equals 2.

A prime polynomial of degree w is chosen to construct an extension Galois field $GF(p^w)$. This field has a primitive element a that can represent all elements in the field (except 0) as $\{a^0(1), a^1, a^2 \ldots, a^{n-2}\}$. Each of them has a minimal polynomial $f_i(D)$, where i is the relevant power. To construct a BCH code that can detect and correct t arbitrary errors, the first $2t$ minimal polynomials are chosen to generate a generation polynomial $g(D)$ with **least common multiple**

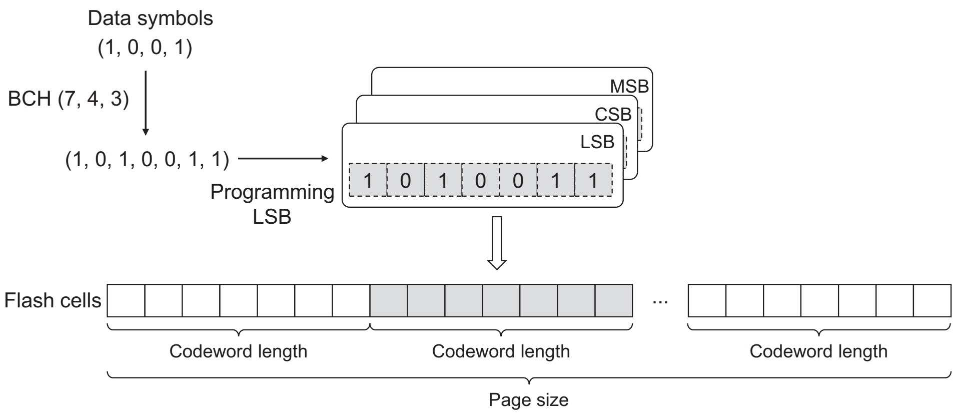


Fig. 3. An example applying $BCH\ (7, 4, 1)$ code in TLC flash. The codeword is programmed in the LSBs (least significant bits) of the flash cells.

(LCM):

$$g(D) = LCM(f_1(x), f_2(D), \ldots, f_{2t}(D)). \tag{5}$$

The data symbols can be encoded with the generation polynomial as in Equation (3).

Errors in a received codeword $y'(D)$ can be formalized as adding an error polynomial $e(D)$ to the codeword polynomial $y(D)$:

$$y'(D) = y(D) + e(D) = g(D)x(D) + e(D). \tag{6}$$

The $e(D)$ can be solved by substituting the roots of $g(D)$ into this equation. This makes $g(D)$ equal 0 and an equation set can be derived to solve the $e(D)$. The $e(D)$ is only solvable if the errors are no more than t. When $e(D)$ is solved, the original $x(D)$ can be recovered accordingly.

Here, we give a simple example to show how BCH code is constructed and implemented in flash storage. Assuming that we use 7-bit BCH codewords to tolerate 1-bit errors, a $GF(8)$ is constructed with the primitive polynomial $p(D) = D^3 + D + 1$. According to Equation (5), we need the minimal polynomials of a^1 and a^2 in the $GF(8)$, which are all $D^3 + D + 1$. Hence, we can get generation polynomial $g(D)$ by

$$g(D) = LCM(D^3 + D + 1, D^3 + D + 1)) = D^3 + D + 1. \tag{7}$$

The order of the polynomial is 3, so the length of data symbols k can be derived by

$$k = n - 3 = 4. \tag{8}$$

In Equation (7), we get the generation polynomial of a $BCH(7, 4, 1)$ code. This example is essentially the Hamming code, a particular case of the BCH code.

In practice, each page is split into multiple parts that are encoded, respectively. In the above simple example, assuming that the 4 bits to encode are $(1, 0, 0, 1)$, whose polynomial format is

$$x(D) = D^3 + 1. \tag{9}$$

The generated codeword can be derived as

$$y(D) = g(D)x(D) = (D^3 + D + 1)(D^3 + 1) = D^6 + D^4 + D + 1. \tag{10}$$

The 7 bits of the codeword are $(1, 0, 1, 0, 0, 1, 1)$. Figure 3 demonstrates the encoding process and how the codeword is programmed in the flash cell. The codeword is directly programmed on

$$C = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

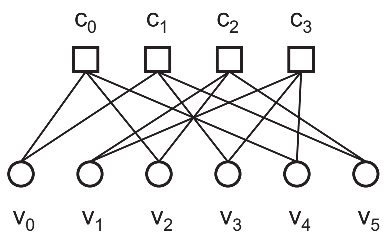


Fig. 4. An example of LDPC parity-check matrix and its relative Tanner graph.

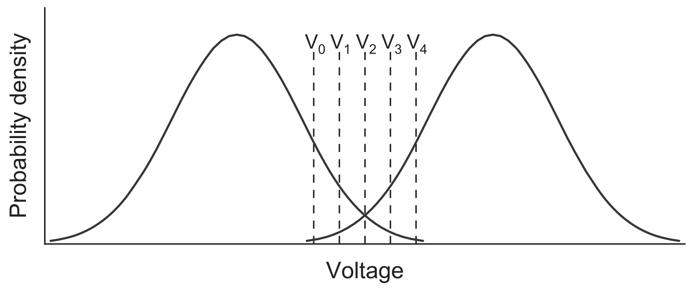


Fig. 5. Multi-level voltage sensing in LDPC soft-decision decoding.

relative cells in SLC flash that stores one bit per cell. In the multi-level flash, each cell can store multiple bits. Each codeword (as well as each page) only occupies one bit per cell.

3.1.2 LDPC Code. ***Low-density parity-check*** **(LDPC)** code is a linear code proposed by R. Gallager in 1962 [34]. Although LDPC was not paid much attention at the early stage due to the limited computing power, it has now been widely adopted in many fields, as it can approach the Shannon limit [85]. In recent years, many works have shown that LDPC has a better ability for error correction in flash memory [41, 142], making it very important for high-density SSDs that are less reliable [132, 136, 138]. The main characteristic of an LDPC code is that its parity-matrix C is sparse. A parity-check matrix is often randomly generated under some sparsity constraints. Figure 4 gives an example of an LDPC parity-check matrix and corresponding *Tanner graph*. The LDPC code can be applied in the flash cells in a similar way as shown in Section 3.1.1.

An LDPC codeword can be decoded with a *hard-decision* decoder or *soft-decision* decoder [59, 142]. Both of them initiate the values in variable nodes of the Tanner graph (V_0 to V_5 in Figure 4) and propagate the values with the check nodes (C_0 to C_3 in Figure 4) multiple times before getting a correct codeword. The difference between the two decoders is the values they propagate. The hard-decision decoder propagates discrete states (0 or 1) and the soft-decision decoder propagates the ***log likelihood ratio*** **(LLR)** of each transmitted bit. The soft-decision decoder has a better ability for error correction than the hard-decision decoder, but its implementation is more complex. A soft-decision decoder not only requires an estimation of channel distribution but also needs more accurate sensing of each bit in the received codeword.

A soft-decision decoder requires multi-level sensing to accurately estimate cell states. The voltage in a flash cell can change due to charge leaks, cell-to-cell interference, and read disturbances [10]. From a statistical viewpoint, a probability distribution function of cell voltage can be derived for soft-decision decoding. Figure 5 presents the cell voltage probability density of two adjacent states. The hard-decision decoding only needs one sensing level (i.e., V_2), but the soft-decision decoding needs multiple sensing levels (e.g., V_0 to V_4). Additional sensing provides more information for decoding but complicates the decoding process.

Table 1. Comparison of Different Methods that Reduce Write Amplification Inside SSDs

Method	Pros	Cons	Representative works
Time-varying ECC strength	Reduced space consumption by avoiding unnecessarily high ECC levels.	More complicated ECC encoder and decoder designs to support multiple ECC levels.	[46] (2015)
Hot-aware management	Reduced write amplification by handling hot data in a dedicated way.	More complicated flash FTL design to maintain data access frequencies.	[83] (2015)
Application-oriented ECC designs	Higher reliability and lower decoding latency.	Needs for additional application information.	[51] (2014)
Delayed parity updates in RAISE	Reduced write amplification from partial updates.	Needs for non-volatile memory.	[35] (2009)

3.1.3 RAISE. The ***redundant array of independent silicon elements*** **(RAISE)** is a cross-chip EC scheme that combines multiple independent flash chips to form a redundant group [35, 41, 95]. When data in one flash chip are lost due to the failed ECC decoding, it can be regarded as a data failure. The lost data can be reconstructed from the redundant data in other flash chips. RAISE generates parity chunks by XORing data chunks in different data chips. The parity chunks are stored in dedicated parity chips for fault tolerance.

3.2 Reducing Write Amplification

Write amplification refers to the problem that the amount of data written to the flash media is larger than the actual amount of user data. This issue is problematic, as it can occupy more device bandwidth and speed up media wearing. In-device encoding affects the extent of write amplification in two main ways. First, when employing ECCs of lower code rates, more storage space is required to accommodate parity data, leading to more frequent GC operations. This increased frequency of GC operations contributes to amplified writes, as it raises the likelihood that a block to be erased will contain valid data, necessitating additional data copying. Second, in RAISE stripes, each partial update should modify the parity data to ensure fault tolerance, further contributing to write amplification. Various existing approaches aim to address the write amplification issue. We summarize existing solutions on reducing write amplification inside SSDs in Table 1.

Time-varying ECC strength. ECCs with higher code rates have lower storage overhead, thereby alleviating the write amplification problem. Additionally, employing weaker ECCs can decrease the decoding latency. The necessary ECC strength varies over time. As the P/E cycle increases, the raw bit error rates of SSDs tend to rise [35, 51, 142]. Consequently, some works advocate for initially using weaker ECC for new SSDs and gradually strengthening it as the SSD ages [10, 41, 46, 136].

Hot-aware management. ECC requirements vary based on the hotness of data access. Many real-world applications exhibit skewed I/O patterns. Some data are hot and tend to be accessed frequently. Frequent reads on flash media increase the RBER due to the read disturb problem [10]. Frequent writes will generate many invalid pages and make GC more frequent. Consequently, some works propose hot-aware managements that group the hot pages in the same block and apply stronger ECC to these hot blocks [11, 36, 83]. For example, Luo et al. [83] find that the periodical data refresh in flash SSDs can timely correct the raw bit errors but also introduce a significant amount of additional writes. They propose to identify and physically group together write-hot data, which does not need refresh operations due to its frequent modifications.

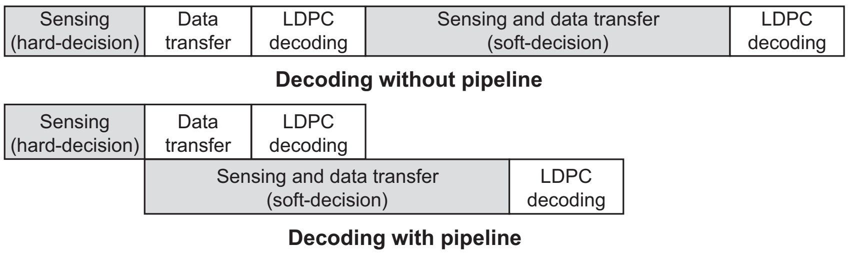


Fig. 6. LDPC soft-decision decoding procedure with and without pipeline.

Application-oriented ECC designs. The ECC designs can also be customized for special scenarios. For example, FlexECC [51] designs an adaptive in-device coding for the SSD-based caching systems, which is placed upon slower but cheaper storage (like HDD-based storage) for data caching. Data in an SSD-based cache can be either clean or dirty. For clean data that have a consistent copy in the underlying storage, FlexECC only adopts EDC for error detection, which lowers the decoding overhead. Data blocks that fail the EDC check are loaded from the lower storage. For dirty data that are newer than the version in underlying storage, FLexECC still adopts traditional ECC for protection. With FLexECC, the redundancy of clean data can be reduced to spare more storage space.

Delayed parity updates in RAISE. Partial updates on RAISE stripes can cause write amplification. One way to solve this problem is to delay the parity updates to the flash media and aggregate multiple partial updates that are on the same RAISE stripe. To solve this problem, Greenan et al. propose to organize data in a log-structured layout [35]. The data updates are performed in an out-of-place manner. New data are added to the log head, and old data are marked as invalid. In this way, it avoids expensive partial updates. Besides, it uses a host non-volatile cache to buffer the parity updates. When a RAISE stripe is full, the final parity block is written to the SSD. This provides the opportunity to merge the updates from different applications and reduce the write amplification.

3.3 Lowering Soft-decoding Latency

Previous works [94, 111, 142] have shown that LDPC soft-decision decoding outperforms hard-decision decoding concerning the error correction ability but at the cost of higher decoding latency. That is because soft-decision decoding needs multiple sequential voltage sensing to read cell data. Most flash controllers measure cell voltage with sense-amp comparators [124, 142]. For a particular reference voltage, the sense-amp comparator can tell whether the cell voltage is higher or lower than the given reference voltage, and sensing with different reference voltage must be serialized. Soft-decision decoding needs more sensing levels to get more accurate measurements of cell voltages, which is important for its error-correction ability [142]. There is a tradeoff between decoding speed and error-correction ability in LDPC soft-decision decoding, and previous works optimize the decoding latency in the following aspects.

Decoding pipeline. Zhao et al. [142] find that although the voltage sensing must be serialized on each cell, sensing can pipeline with the data transfer and controller-side decoding. They propose a decoding pipeline to cut off the decoding latency. As shown in Figure 6, decoding LDPC without pipeline goes through several stages. The flash controller first performs sensing and data transfer according to hard-decision decoding. If hard-decision decoding fails, then the controller will perform multiple cell sensing for soft-decision decoding. The proposed method is decoding with the pipeline, in which sensing for soft-decision decoding is immediately started regardless of whether hard-decision decoding succeeds. If hard-decision decoding succeeds, then the soft-decision decoding process will be terminated immediately. Otherwise, the soft-decision decoding proceeds with a shorter remaining time, therefore reducing the whole decoding latency.

Better reference voltage. In LDPC soft-decision decoding, the sensing level can be reduced by choosing better reference voltages, which can compensate for the diminished error-correction ability due to fewer sensing levels. Dong et al. [27] propose to use a non-uniform sensing strategy instead of the uniform sensing strategy [4]. With the non-uniform strategy, sensing voltages are placed within the "dominating overlap regions" of two adjacent cell states. However, within the dominating overlap region, the sensing voltages are still uniformly distributed, which may not be the best configuration. Some works [115, 116, 124] propose to directly solve the best sensing voltages by maximizing the **mutual information (MI)** between the originally programmed data and the sensed data in each cell. With cell voltage distribution models, an MI expression can be derived that contains the sensing voltages as parameters. Then, optimal sensing voltages are obtained by maximizing the value of MI expression with numerical methods.

Cell-wise decoding. Cell-wise decoding [74, 94] can reduce the number of sensing operations but at the expense of minor loss in error-correction ability. It is designed for flashes with multi-bit cells (e.g., MLC, TLC, and QLC). As opposed to page-wise decoding that reads each bit of a cell independently, cell-wise decoding considers all bits together for each read. For example, Lee et al. [74] propose a paired-page reading scheme for MLC flash, which replaces multiple soft-decision decoding sensing for the MSB page with one hard-decision decoding sensing for the LSB page. Another work [94] designs a new scheme that jointly encodes all bits within one cell instead of encoding each page independently. Their channel capacity analysis reveals that jointly encoding all bits and decoding them with hard-decision decoding only introduces minor channel capacity loss compared to page-wise soft-decision decoding while greatly reducing the sensing times and lowering decoding latency. These works present a reasonable tradeoff between decoding speed and error correction ability.

3.4 Taming Tail Latency

Copying valid pages during GC consumes chip bandwidth. This can interfere with foreground I/Os, resulting in latency spikes and long tail latency of device I/Os. Although major works aim to tame the tail latency in cross-device coding (see Section 4.4), solving this problem is also viable with in-device coding. Taming tail latency with in-device coding needs scheduling in finer granularity such as the flash chip and flash plane to better exploit the parallelism inside SSDs.

Yan et al. [134] tackle the GC-induced tail latency by designing fine-grained scheduling to exploit the parallelism of different flash channels and planes. As shown in Figure 7, they categorize GC blocking of existing SSDs into two types. The first type is controller blocking, in which GC tasks and device I/Os are serialized in the controller. Therefore, an ongoing GC task will influence I/Os on any plane. The second type is channel blocking, in which GC tasks and device I/O are serialized in channels. I/Os on planes that do not share the channel with the GC task will be uninfluenced.

Channel blocking weakens the GC-induced performance variation, but I/Os on planes that share the same channel with the GC task will still be blocked. To this end, Yan et al. design a new in-device coding technique called TTFLASH that can optimize the GC-induced blocking to the plane level. During the GC, TTFLASH utilizes an intra-plane copy mechanism to copy valid pages within one plane without occupying the channel bandwidth. A fine-grain pipeline is introduced to serve user I/Os on the same channel as GC during the GC intro-plane copy. This can cut off I/O waiting time and accelerate I/O speeds.

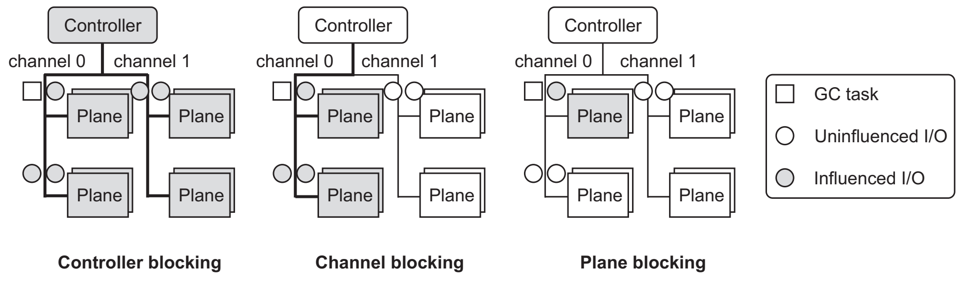
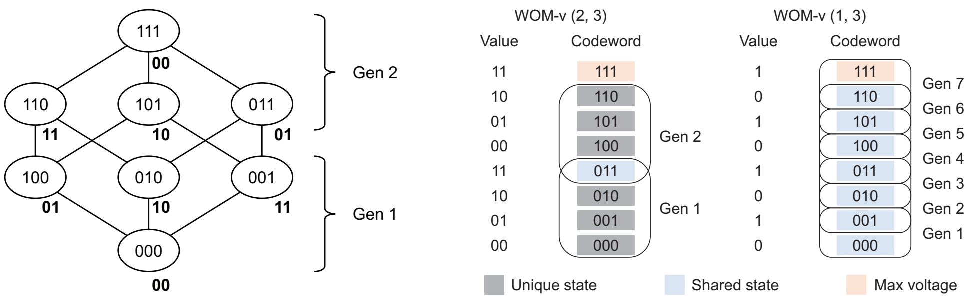


Fig. 7. Different GC blocking levels inside SSDs.



(a) Binary WOM code. (b) Non-binary WOM code.

Fig. 8. Examples of write-once memory (WOM) codes.

TTFLASH leverages data redundancy to provide GC-tolerant read. By scheduling GC in a round-robin manner, TTFLASH ensures that at most one plane in each plane group performs GC at a time. This guarantees that SSD reads can always avoid the plane undergoing GC either by directly reading from a GC-free plane or by reading from other planes and reconstructing the data when the target plane is performing GC.

3.5 Mitigating Raw Bit Errors

Error mitigation techniques are preventive measures that try to reduce the number of raw bit errors in flash media. Although orthogonal to the error correction techniques introduced before, some error mitigation techniques such as *write-once memory coding*, *bit labeling*, and *data scrambling* exploit data coding to lower the RBER of flash media, which can reduce the error correction pressures and extend SSD lifetime. For more flash error mitigation techniques, readers can refer to other works [10, 12].

Write-once memory coding. Write-once memory code [101] can extend the flash lifetime by reducing the number of erasing operations. As introduced in Section 2.1, a programmed flash cell cannot be modified before it is erased. Storage media with such a characteristic is called the **write-once memory (WOM)**. When raw flash bits are encoded with the WOM code, they can be programmed once and modified multiple times before being erased, which can effectively reduce the total P/E cycles and prolong the SSD lifetime.

A WOM code can be either binary or non-binary. First proposed in Reference [101], the binary WOM code is designed for write-once storage media whose basic storage unit has two states, such as the 0 and 1 in SLC flash. To support modifications, multiple storage units are grouped to form one codeword. Figure 8(a) presents a coding example that supports one program operation and one modification operation by encoding each 2-bit value into a 3-bit codeword. When an erased codeword is programmed, it changes to one of the states in the first generation according to the programmed value. A subsequent modification may keep the codeword intact if the value does not change, or alter the codeword to one of the states in the second generation.

Implementing the binary WOM code in multi-level flash media may restrict the number of usable codewords, leaving room for further optimizations [55]. The non-binary WOM code was first proposed many years ago [30] and has received intensive attention in recent years [33, 56, 133] due to the promotion of high-density storage such as MLC, TLC, and QLC flash. Figure 8(b) demonstrates an example of a recent non-binary WOM code, named WOM-v [55, 56]. An (m, n) WOM-v code encodes each m-bit value into an n-bit codeword. It enables modifications by dividing the flash cell state into multiple generations. Modifications on a storage unit can be achieved by increasing the generation. Besides, WOM-v further proposes shared states between adjacent generations to accommodate more generations, which effectively reduces the number of erasing.

Suitable bit labeling. SSDs save data by distinguishing different voltage states in flash cells. The mapping between a cell state and the bits it represents is called bit labeling. Designing a suitable bit labeling can help to reduce the RBER or extend the flash lifetime. *Gray code* is widely used in bit labeling in flash cells [16, 84]. The main characteristic of Gray code is that the adjacent codeword only differs by one bit. As a deviated cell state has a high probability of switching to its adjacent state, only one bit of the data will be wrong in this case, which helps to restrict the RBER of SSDs.

Data scrambling. Data scrambling in flash storage can help lower RBER by reorganizing the bits on flash media to avoid problematic patterns. For example, data randomization [13, 29] is used to change the patterns of the original data so the numbers of 0 and 1 bits are approximately the same [10]. Tanakamaru et al. [110] propose a stripe pattern elimination algorithm that can handle data retention errors and program disturb errors. Besides, some works propose to use the constrained codes [42, 63, 98] to limit the number of consecutive 0 or 1 but at the expense of increased storage space [94].

4 Cross-device Coding

In modern data centers, failures of whole SSDs are not rare [89, 103, 140]. With the growing storage density of flash media, merely relying on ECC may not be able to correct all raw bit errors [62, 103, 134]. An SSD can also fail when its controller is broken. Therefore, cross-device coding is necessary for data protection in single-machine-multiple-devices storage. This section begins with the theoretical foundations of cross-device coding from the mathematical perspective and the challenges when implementing them on SSD arrays. By summarizing four challenges of deploying SSD-based cross-device coding, we revisit what existing works have done to cope with these challenges, including modeling SSD RAID reliability, optimizing parity updates, taming tail latency, and redesigning on fast devices.

4.1 Theoretical Bases and Challenges of SSD Integration

The commonly used cross-device coding is the RAID that was proposed by D. A. Patterson in 1988 [96]. In the beginning, RAID refers to *redundant array of inexpensive disks*. The main idea of RAID is to combine a bundle of cheap but unreliable disks to provide high aggregated bandwidth while guaranteeing reliability. Nowadays, the performance of storage devices has developed a lot, but RAID, now referred to as *redundant array of independent disks*, is still widely adopted in storage systems for its ability to protect data reliability.

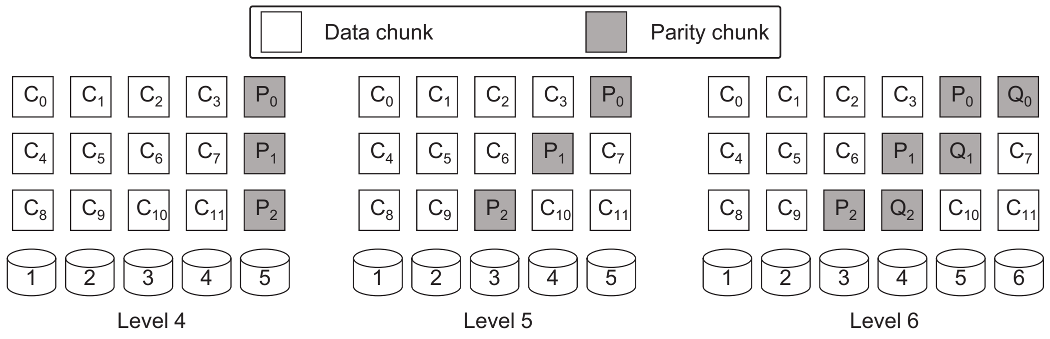


Fig. 9. Encoding process of RAID-4 to RAID-6.

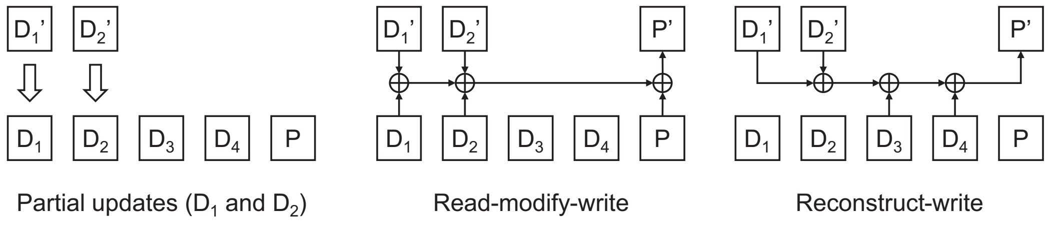


Fig. 10. Read-modify-write and reconstruct-write for partial updates in a RAID stripe. Assuming that chunk D_1 and D_2 are modified to D_1' and D_2', and the new parity chunk is P'.

4.1.1 Theoretical Bases. RAID has seven levels, from RAID-0 to RAID-6, among which five levels (RAID-1 to RAID-5) are first proposed [96] and two levels (RAID-0 and RAID-6) are supplemented later. We illustrate the encoding processes of RAID-4 to RAID-6 in Figure 9, as they are the frequently used data coding methods among the RAID levels.

RAID-4 can tolerate one device failure by adding a parity chunk in each RAID stripe. The parity chunk is derived by XORing the data chunks within one stripe. The main difference between RAID-4 and RAID-3 is that RAID-4 encodes data in larger units by arranging user data in a chunk-interleaving manner. Consecutive data of a chunk size fill one disk before proceeding to the next disk. RAID-4 can avoid multi-disk synchronization of each I/O and improve the parallelism of small I/Os [96].

RAID-5 optimizes the sequential read performance and balances the load for parity chunks. RAID-4 stores all parity chunks in one disk, which is problematic in two aspects. First, sequential read cannot exploit the aggregated bandwidth of all n disks, as the parity disk does not serve I/O requests when no disk failure happens. Second, the parity disk tends to have higher loads, as any partial update incurs loads on the parity disk.

Partial updates refer to the I/O requests that only update part of the data in a stripe. To keep the stripe fault-tolerant, the parity chunks should also be updated to be consistent with the new data chunks. Parity chunks can be updated with ***read-modify-write*** **(RMW)** or ***reconstruct-write*** **(RCW)**, as shown in Figure 10. With RMW, a delta of the parity chunk is calculated by getting the difference between the old and new versions of the modified data chunks. This requires reading the old versions of modified data chunks and the parity chunk. With RCW, the new parity chunk is calculated directly by reading the unmodified data chunks. The extra loads of RMW and RCW differ concerning the size of partial writes, and the optimal choice is discussed by previous works [107, 112]. However, both of them need to access the parity disk in RAID-4, resulting in unbalanced loads among different disks.

To solve the above problems, RAID-5 does not have a single parity disk but scatters the parity chunks to all disks in the array. As shown in Figure 9, the parity chunks in RAID-5 are placed on different disks in a round-robin manner. This can not only balance the loads for partial updates but also accelerate large sequential reads as the bandwidth of all disks is exploited.

RAID-6 refers to the cross-device coding methods that can tolerate failures of two disks. Each stripe contains two parity chunks as redundant data. The coding of RAID-6 is more complicated and has different realizations. Popular choices include the RS code [5], EVENODD [8], RDP [21], and X-code [131]. RAID-6 may encode data in a cross-stripe manner. The parity chunk in one strip may be the XOR of data that is not in the same stripe.

4.1.2 Challenges of SSD Integration. RAID was initially proposed in the HDD era for bandwidth aggregation and data protection. However, SSDs have very different characteristics compared to HDDs. Directly applying the RAID to the SSD array needs to consider several aspects [57]. We briefly point out four main issues to tackle when building an SSD-based RAID system and summarize existing works that target these issues in the following parts.

Reliability concerns. It may not always be beneficial to construct SDD-based RAID for reliability. Although the RAID enhances the reliability of SSD arrays by introducing redundancy, it causes write amplifications due to partial updates and more space consumption, resulting in faster wearing on SSDs. Hence, it is worth discussing the reliability models of SSD-based RAID systems.

Partial update handling. When designing an SSD-based RAID system, the developers should consider how to handle the partial updates, as it will cause I/O amplification, especially the write amplification, on SSD array. I/O amplification can degrade the performance of I/O requests, decrease SSD lifetime, and cause unstable performance.

Inconsistent performance. The performance of SSDs is less stable than that of HDDs due to the media characteristics. The unstable performance is further exacerbated on the SSD RAID array, as the raised I/O latency can propagate among different requests due to the I/O dependency during the parity updates.

Software bottlenecks. SSDs are becoming faster, and bottlenecks may shift from the hardware to the software. Outdated designs of software-based RAID solutions may become the bottleneck upon the ultra-fast NVMe SSDs.

4.2 Modeling SSD RAID Reliability

SSD-based RAID systems enhance storage reliability by introducing redundant data. However, this redundancy diminishes the lifetime of each single device due to two reasons. First, partial updates on RAID stripes modify both data chunks and parity chunks, resulting in write amplifications [6, 57, 60]. Second, redundant data occupy more device space, and GC is triggered more frequently due to less available space [91]. Both reasons cause increased P/E cycles and diminish the lifetime of flash media. To analyze the reliability of SSD-based RAID systems, existing works build different analytic models with different assumptions on SSD failures.

Kadav et al. [6, 60] analyze the reliability of SSD RAID by considering each SSD as a whole, eliminating the internal mechanisms. They assume that the **uncorrectable bit error rate (UBER)** of an SSD is only influenced by the actual P/E cycles, which can be derived by the number of page writes under the assumption of perfect wear leveling within each device. Based on this model, they find the problem of simultaneous broken devices in the common RAID schemes, such as the RAID-4 and RAID-5. For example, RAID-5 distributes the parity chunks evenly among different SSDs, and the probability of being written is very close among different devices. This makes all SSDs wear at approximately the same speed, raising the probability of data loss. To tackle this problem, they design the **differential RAID (Diff-RAID)**, which manually differentiates the wearing speeds of each SSD by assigning different portions of parity chunks, considering that parity chunks are updated more frequently than the data chunks. This design can help to avoid multiple SSDs breaking at the same time, decreasing the risks of data loss in the SSD RAID.

Diff-RAID controls the wearing speeds of different SSDs by assigning different proportions of parity chunks on different SSDs. However, it does not consider which combination of the parity chunk proportions provides the best SSD RAID reliability. Considering this, Li et al. [80] build a Markov chain model to analyze the reliability of SSD RAID under different configurations of parity chunk proportions. Like the Diff-RAID, this work assumes that each SSD can perform perfect wear leveling, and the error rate of each chunk is decided by the actual P/E cycles. In its Markov chain model, each state represents that the RAID system has a specific number of stripes that contains exactly one erroneous chunk. It also has a state in which at least one stripe has more than one erroneous chunk, which results in data loss under its RAID-5 scheme. The transition between different states is influenced by both the configuration of the parity chunk proportions and the recovery speed. In this way, it derives an expression that can compare the reliability of different RAID systems.

Moon et al. [91] also analyze the SSD reliability with the Markov chain model. The main difference from the above works is that they consider more types of failures, including whole-device and page-level failures. For page-level failures, it also regards the number of P/E cycles as the main source of raw bit errors, modeling their relationship based on the study in a previous work [108]. When a page is read for application I/O or GC, its raw bit errors can be corrected within the ability of its ECC. When the number of page raw bit errors exceeds the ability of the ECC, this page must be recovered with cross-device redundancy. The device-level failures also have relative failure and recovery rates. In this way, it can model the **mean time to data loss (MTTDL)** by substituting the above parameters to the Markov chain model.

Kishani et al. [71] further refine the SSD RAID reliability model by considering more types of SSD failures. By referring to a more recent study on SSD failures [103], they point out that apart from the raw bit error (bad symbols), other failures, such as bad blocks and bad chips, are not rare in industrial data centers, affecting the reliability of SSD arrays. With the fine-grained failure types, data loss is further divided into *array data loss*, *block data loss,* and *stripe data loss*. They modify the Markov chain model for each RAID scheme that takes all three failures into consideration. To derive the relationship between the rates of the above three kinds of failures and some observable parameters, such as the read, write, and erase counts, this article builds the linear regression models with field data [103]. Finally, a fault injector is built to simulate the failure rates of different RAID schemes according to I/O and erasure traces from real SSDs.

4.3 Optimizing Parity Updates

To achieve fault tolerance in a RAID stripe, the parity chunks need to be consistent with the data chunks. When part of the data within a stripe is updated (i.e., partial updates), the relevant parity chunks must be updated accordingly. Excessive partial updates have two adverse impacts [68, 90]. First, partial updates incur I/O amplification. Both RMW and RCW need to read more chunks before calculating the new parity chunks. This will increase the load on SSDs and interfere with the user I/Os. Second, frequent parity updates can impede the performance and endurance of SSDs. As SSDs do not support in-place updates, parity updates will accelerate space consumption and media wearing, triggering GC more frequently and decreasing SSD endurance. We summarize existing solutions in Table 2 and elaborate on them in the following part.

Table 2. Comparison of Different Methods that Optimize Parity Updates in SSD RAID Arrays

Method	Pros	Cons	Representative works
SSD-oriented parity logging	No need for reading the old data chunks.	More space occupation and needs for asynchronous parity merging.	[14] (2018)
Partial parity cache	Fewer SSD writes and wearings.	Needs for large non-volatile memory space.	[52] (2011), [19] (2014)
Log-structured RAID	No partial updates on full RAID stripes.	Poor concurrency and complicated metadata management.	[18] (2014), [20] (2015), [54] (2018)
Two-phase write	No need for reading old data chunks or updating parity chunks.	More space occupation and needs for asynchronous parity merging.	[86] (2012), [127] (2016), [58] (2021), [77] (2023)

Parity logging. Parity logging is first proposed to solve the parity updates problem in HDD RAID array [106]. It records the "parity update images," which is the XOR between old data chunks and new data chunks, in the parity disk in a log-structured manner. When merging the parity logs, the final parity chunk can be derived by XORing all parity update images of the stripe with the old parity chunk. For SSD RAID, Chan et al. further propose EPLOG [14], which exploits the out-of-place update characteristic of SSDs. When a data chunk in an SSD is updated, the new data are written in spare pages and the old pages are not erased. In this way, this new parity logging scheme only needs to record the XOR of the new data chunks in the parity disk, eliminating the need to read the old data chunks. This reduces the I/O load in the critical path. When merging the parity updates, both old and new data chunks are read to recalculate the parity update images and update the parity chunks. Afterward, the old data chunks can be reclaimed. The main challenge of EPLOG is that it needs more space to save the old data chunks. Besides, it needs to maintain additional metadata to locate both old and new data chunks.

Partial parity cache. The **PPC (partial parity cache)** is designed for RAID-5 to optimize the partial updates [19, 52]. It reduces I/O loads on the parity disk by caching the parity updates with **storage class memory (SCM)**. Besides, it leverages the out-of-place update characteristic of SSDs to further lower the I/O amplification.

PPC contains many cache entries, each of which has a cache for partial parity chunks and a bit vector to mark the modified data chunks. When a data chunk is updated, it updates the partial parity chunk and sets the relative bit of the bit vector. The partial parity chunk is derived by encoding the new version of data chunks that have been modified.

Figure 11 gives an example of PPC on a RAID-5 array. Initially, D_0 to D_3 form a full stripe with a parity chunk P_0. Assuming that data chunks D_1 and D_2 are modified to D_1' and D_2', the partial parity equals the XOR of D_1' and D_2'. This process does not incur any disk read I/O, as the out-of-place update characteristic of SSDs keeps D_1 and D_2 implicitly. To enable merging the parity chunk when a cache entry is flushed, PPC maintains a mapping table to record the current **physical page number (PPN)**, the old PPN (if any) and the **physical parity page number (PPPN)** of each **logical page number (LPN)**. PPC will incur one additional read I/O if a previously updated data chunk is written again (like D_1' to D_1'').

The challenge to deploying the PPC is that it needs to store the partial parity and the associated metadata for each cached stripe. This will quickly exhaust the cache space if the updated data spreads across many stripes and demonstrates little hotspot. Besides, before a partial parity is

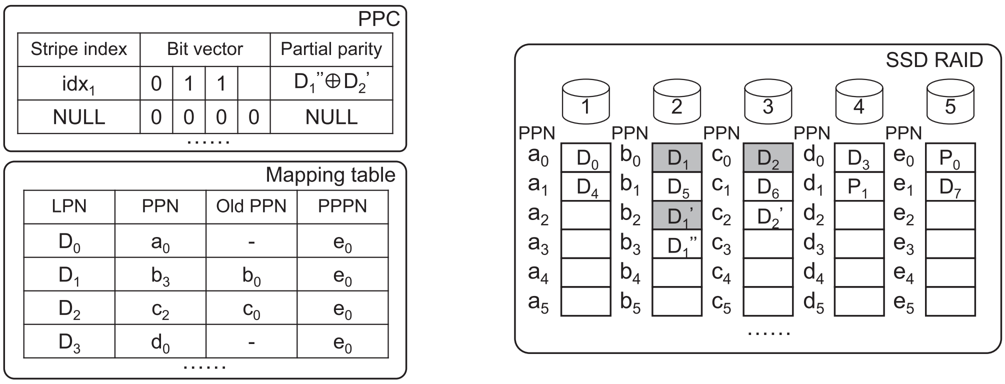


Fig. 11. PPC main data structures and an example of its deployment on a RAID-5 array.

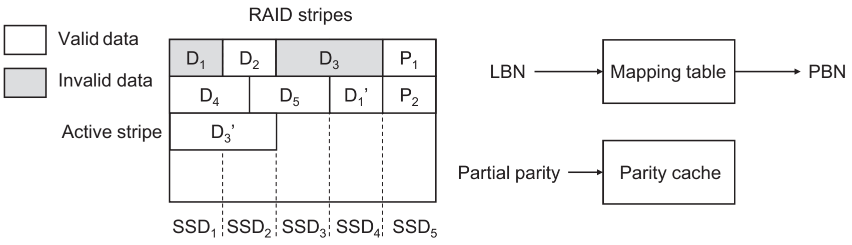


Fig. 12. Data layout of log-structured RAID with RAID-4 array.

committed to the SSD, the old data chunks must be kept to resist SSD failures, which will increase the temporary space overhead.

Log-structured RAID. Several works use the log-structured RAID to solve the partial update problems [18, 20, 54]. Figure 12 presents an example of log-structured 4+1 RAID-4.

A RAID array organizes its space in numerous RAID stripes. Unlike a traditional RAID array in which any RAID stripe can be written, only one stripe in log-structured RAID is writable. This stripe is also called *active stripe*. The active stripe can only be written in an append-only manner. When an active stripe is full, it becomes read-only. The parity chunk of the whole stripe is calculated and stored in the RAID array. A new blank stripe is chosen as the active stripe for subsequent writes.

Each write on the active stripe will update the parity chunk, which can still trigger the partial update problem. This can be solved by adding a SCM-based parity cache [20, 54] or mirroring a temporary parity chunk in separated disks [18]. Another way to optimize the parity update problem in log-structured RAID is *elastic striping* [64, 65]. In this method, the stripe length is variable. Each write can form a "partial stripe," making all writes to be full-stripe write. This can eliminate the need for parity updates but at the expense of more space consumption, as a RAID stripe can contain more than one parity chunk. It also complicates the RAID management for these additional parity chunks.

The log-structured RAID has the following advantages: First, all writes are aggregated into a limited number of active stripes, which can efficiently reduce the space needed to buffer partial parity chunks in SCM-based solutions. Second, the partial updates do not need to read the data chunks from the original RAID stripe, as log-structured RAID performs updates in an out-of-place manner. The active strip can be cached in the SCM buffer, reducing the read I/Os from the SSD RAID.

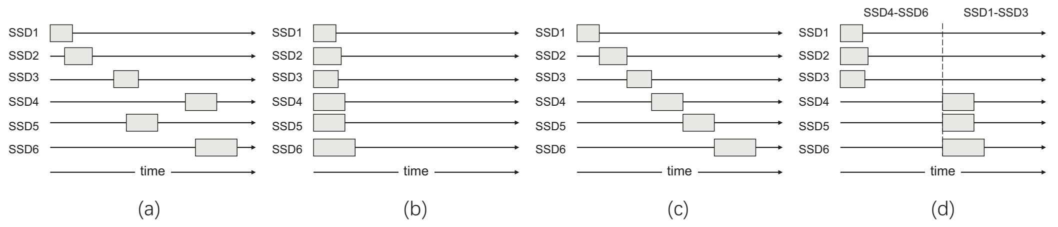


Fig. 13. Different coordination methods for GC within an SSD array: (a) local garbage collection (LGC), (b) global garbage collection (GGC), (c) request redirection (RR), (d) spatial separation (SS).

However, a log-structured RAID complicates host software designs. The host software needs to maintain a mapping between the LPN and the PPN to serve read requests. Besides, the host is also responsible for space management, such as space allocation and garbage collection. This is what FTL does inside one SSD, and the host-managed log-structured RAID extends this to the whole RAID array. Finally, aggregating multiple data flows into one stripe may incur concurrency problems, which can be alleviated by opening multiple active stripes for logging.

Two-phase write. Two-phase write optimizes partial updates on RAID stripes by dividing the partial updates into two phases. In the first phase, new data are saved as multiple replications on different disks. This can tolerate device-level failures while avoiding reading the old data and parity chunks for parity updates. When the system I/O load is not heavy, the two-phase write enters the second phase, in which the multiple replications are converted into RAID stripes to lower storage costs.

In the early years, some works [86, 127] designed hybrid disk arrays that compose both HDDs and SSDs. HDDs are used as parity disks due to their ability to perform in-place updates. Additionally, multiple replications can be stored across several HDDs, because they offer a lower cost per unit of storage compared to SSDs. However, recent works have started to store the multiple replications directly on SSDs [58, 77]. This mainly comes from two reasons. First, the capacities of SSDs have significantly increased in recent years, making it feasible to store multiple replications in SSDs [58]. Second, the performance gap between SSDs and HDDs has been growing, and this gap is expected to further expand with the increasingly fast NVMe SSDs. Hence, adding HDDs to an SSD RAID array can adversely impact the overall performance.

4.4 Taming Tail Latency

As shown in Section 2.1, background operations like GC may compete with foreground I/O for channel bandwidth, resulting in the unstable performance of user I/Os. This problem is further exacerbated in SSD storage with in-device coding, because both full-stripe writes and partial updates need to access multiple SSDs synchronously. Any I/O that encounters a performance-degraded SSD will slow down the whole request [22]. The unstable performance can result in long-tail latency of user I/Os, making the storage service providers violate their stringent ***service level objects* (SLOs)**, impairing user experience and negatively impacting revenues [23, 109, 113, 128]. We summarize existing works on taming tail latency in cross-device coding in Table 3 and elaborate on them in the following parts.

4.4.1 Internal GC Coordination. The unstable performance of SSDs can be tackled with *GC coordination*. It schedules SSD GCs at appropriate timing to lower background interference. We summarize existing GC coordination methods into four types and illustrate them in Figure 13.

Table 3. Comparison of Different Methods that Tame Tail Latency in SSD RAID Arrays

Method	Pros	Cons	Representative works
Global garbage collection	Reduced time window of inconsistent performance.	More severe performance variation during the scheduled GC windows and needs for in-device manipulation.	[70] (2011), [69] (2012)
Request redirection	Relatively stable performance at all times.	More I/O loads from degraded reads and needs for in-device manipulation.	[126] (2018), [75] (2021)
Spatial separation	GC-free in the front-end SSDs.	Lower overall SSD utilization and needs for in-device manipulation.	[105] (2014), [66] (2019)
Black-box solutions	No need for in-device manipulation and easy for practical deployments.	Hard to derive a theoretical performance guarantee.	[39] (2016), [40] (2020), [58] (2021)

Local garbage collection. When GC is not coordinated among different SSDs, each SSD decides its own GC timing, like when its spare space drops below a certain threshold. This is called ***local garbage collection*** **(LGC)**. In LGC, user I/Os can experience latency spikes frequently, as different SSDs may perform GCs at different time points. Besides, it can also cause unexpected performance degradation as it is not easy to know when the GC will start.

Global garbage collection. In light of the problems in LGC, ***global garbage collection*** **(GGC)** is proposed [69, 70] to coordinate GCs in different SSDs. The main idea of GGC is to make all SSDs perform GC at the same time, thus expanding the stable-performance intervals, in which no SSD is performing GC. Besides, GGC also gives information about when GC is performing, solving the problems of unexpected latency spikes. GGC can be implemented in either proactive or reactive modes. In proactive mode, the RAID controller proactively starts GC in all SSDs when certain conditions are met, like when the number of free blocks is below a threshold. In reactive mode, when one SSD starts GC spontaneously, the RAID controller will start GC in other SSDs at the same time. GGC can shorten the performance-degraded intervals. However, during the intervals of global GC, the I/O performance will drop substantially, as nearly all SSDs are performing GC.

Request redirection. To provide stable performance at all times, ***request redirection*** **(RR)** is proposed in subsequent works [75, 126]. The RR method adopts the opposite idea of GGC: Instead of starting GCs in all SSDs at the same time, it coordinates the GC time points so the number of SSDs that concurrently perform GC is no more than the number of redundant SSDs (i.e., $(n-k)$). In this way, RR enables I/O scheduling to avoid accessing the SSDs that are doing GC. If a write request targets an SSD that is doing GC, then the RAID controller can redirect it to another SSD that is not doing GC and record a mapping for this redirection. If a read request targets an SSD that is doing GC, then the RAID controller can reconstruct the required data with the data chunks and parity chunks in SSDs that are not doing GC (which is also known as *degraded read*). In this way, RR provides relatively stable performance at all times but with additional parity calculation and bandwidth consumption.

Spatial separation. Another method is to coordinate the GC in a ***spatial separation*** **(SS)** way [66, 105]. In the SS method, SSDs are divided into two interchangeable groups, named the front-end group and the back-end group. Only SSDs in the back-end group can perform GC to reclaim spare space. SSDs in the front-end group are forced to be GC-free. When the free space of the front-end group is not enough to support the GC-free goal, the front-end group and the back-end group exchange. It brings the benefit that I/O requests to the front-end group will not be interfered with by the SSD GC. This can either bring stable read performance [105] or write performance [66] of user I/Os. However, the overall utilization of this method is lower than the other methods, as only the front-end SSDs can serve the foreground write requests.

4.4.2 Black-box Solutions. Although GC coordination methods can alleviate the GC-induced performance variation in SSD storage, most of them cannot be directly applied to commodity SSDs due to their limited interfaces. GC is an internal task of SSDs and it is impossible to control it outside without special APIs from the SSDs. Besides, GC is not the mere factor of latency spikes in SSDs, like flushing the SSD write buffer can also cause performance variability [2, 58].

The above problems can be solved with methods that treat SSDs as black boxes and design external solutions to tame tail latency. This does not need any in-device management and can handle performance variations not restricted to GC.

The simplest way to do this is *request hedging* [22, 39, 40]. For I/O requests to data in a RAID stripe, if the data chunks do not return in t seconds, then the reader will send redundant requests to the parity chunks and seek to reconstruct the missing data chunks. The request hedging can be done proactively, sending redundant requests immediately (i.e., $t = 0$), such as the proactive mode in ToleRAID [39] or the way in Reference [22]. It can also be done reactively, sending redundant requests after a certain threshold t' (i.e., $t > t'$), like the reactive mode in ToleRAID [39].

The request hedging has the drawback that it has no information about which SSD is under performance degradation. Therefore, it either uses a proactive strategy that will produce times of extra loads on the storage or uses a reactive strategy that cannot handle the latency spike timely. To this end, some works propose to gather extra information to tame tail latency.

One way to achieve this is to identify the SSDs that are under performance degradation. For example, some works propose to monitor the history I/O response time from each SSD to decide whether an SSD is under performance degradation, such as FusionRAID [58] or the adaptive strategy in ToleRAID [39]. With this additional information, they can perform degraded read [39] or redirected write [58] to sidestep the performance-degraded SSDs in advance.

Another way is to perform latency prediction in I/O granularity. For example, LinnOS [40] exploits a lightweight neural network to identify I/O latency spikes. It models the problem as a classification problem, designing a **multilayer perceptron (MLP)** to predict whether an I/O operation will be slow. The input features are the history response time and the number of currently pending I/Os as input features, which do not involve any information within the device. Hence, LinnOS can identify slow devices I/Os are issued, enabling timely switching to the failover logic in advance.

4.5 Redesigning on Fast Devices

Conventional wisdom holds that CPUs are much faster than external storage and the bottlenecks of storage I/O stacks are mainly on accessing the external storage. This is true when the device is not so fast and the application I/O concurrency is not so high. However, as the storage devices become increasingly faster, traditional RAID I/O stack may have software bottlenecks under highly concurrent I/O requests.

Two recent works, ScalaRAID [137] and StRAID [118, 119], optimize the Linux software RAID stack to fit ultra-fast SSDs. They found that the Linux software RAID scales poorly when many user threads perform stripe writes concurrently. Deep analysis reveals that the intensive lock contention and the inefficient stripe-write workflow are the main reasons behind this. To this end, they design fine-grained lock mechanism [137] and stripe-thread architecture [118, 119], which can efficiently alleviate wasted CPU cycles on lock contention and improve the multi-thread scalability of SSD-based RAID array.

5 Cross-machine Coding

In multiple-machines-multiple-devices storage, cross-machine coding enables data protection under severe failures, including machine-level, rack-level, or even data-center-level failures. Cross-machine coding methods typically use distributed EC for data recovery, as data loss or corruption can be detected in other ways. In this section, we present the theoretical bases of cross-machine coding and the challenges when implementing them on distributed SSD storage. We summarize the challenge of network bottleneck in SSD-based cross-device coding and revisit how existing works can be leveraged to tackle this challenge.

5.1 Theoretical Bases and Challenges of SSD Integration

Cross-machine coding can be built with **Reed-Solomon code (RS code)**. RS code was proposed by I. S. Reed and G. Solomon in 1960 [99], and it is a **maximum distance separable (MDS)** code. When k data chunks are encoded into n chunks with RS code, it can recover any lost chunk when the number of simultaneous chunk failures is no more than m ($m = n - k$). Such a configuration is called an (n, k) RS code.

5.1.1 Theoretical Bases. We introduce the encoding and decoding process of the RS code in the matrix form (see Section 2.2). The parity check matrix C of an (n, k) RS code is a $m \times n$ matrix. Typical choices of parity check matrix are the *Vandermonde* matrix or *Cauchy* matrix.

Vandermonde-RS. The Vandermonde-RS is proposed in the initial work of RS code [99]. Assuming that $\{x_1, x_2, \ldots, x_n\}$ are n elements in $GF(2^w)$, an $m \times n$ Vandermonde matrix M_V is defined as:

$$M_V = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ x_1 & x_2 & x_3 & \cdots & x_n \\ {x_1}^2 & {x_2}^2 & {x_3}^2 & \cdots & {x_n}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ {x_1}^{m-1} & {x_2}^{m-1} & {x_3}^{m-1} & \cdots & {x_n}^{m-1} \end{pmatrix}. \tag{11}$$

The Vandermonde matrix is reversible if $\{x_1, x_2, \ldots, x_n\}$ are distinct elements.

Cauchy-RS. The Cauchy-RS is proposed in a subsequent work on RS code [9]. Assuming that $\{x_1, x_2, \ldots, x_m\}$ and $\{y_1, y_2, \ldots, y_n\}$ are elements in $GF(2^w)$, an $m \times n$ Cauchy matrix M_C is defined as:

$$M_C = \begin{pmatrix} \frac{1}{x_1+y_1} & \frac{1}{x_1+y_2} & \frac{1}{x_1+y_3} & \cdots & \frac{1}{x_1+y_n} \\ \frac{1}{x_2+y_1} & \frac{1}{x_2+y_n} & \frac{1}{x_2+y_3} & \cdots & \frac{1}{x_2+y_n} \\ \frac{1}{x_3+y_1} & \frac{1}{x_3+y_2} & \frac{1}{x_3+y_3} & \cdots & \frac{1}{x_3+y_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{x_m+y_1} & \frac{1}{x_m+y_2} & \frac{1}{x_m+y_3} & \cdots & \frac{1}{x_m+y_n} \end{pmatrix}. \tag{12}$$

A Cauchy matrix is reversible if the following conditions are met:

$$\forall i \in \{1, \ldots, m\}, \forall j \in \{1, \ldots, n\} : x_i \neq y_j \tag{13}$$

$$\forall i, j \in \{1, \ldots, m\} : x_i \neq x_j \tag{14}$$

$$\forall i, j \in \{1, \ldots, n\} : y_i \neq y_j. \tag{15}$$

Both Vandermonde and Cauchy matrices can be directly used as the parity check matrix when constructing an RS code. However, to ensure that the coding can recover any m failures, only the Cauchy matrix can be directly used to construct the generation matrix of an RS code without row or column transformation. This is because any sub-matrix of a Cauchy matrix, formed by the intersection of certain rows and columns, is also reversible if Equations (13) to (15) are met. This characteristic is obvious, as any sub-matrix of a Cauchy matrix is also a Cauchy matrix. The generation matrix G of a systematic RS-code can be derived with a k-order identity matrix I and an $m \times k$ Cauchy matrix M_C as:

$$G = \begin{pmatrix} I \\ M_C \end{pmatrix}. \tag{16}$$

This characteristic ensures that any k-order sub-matrix of G is reversible. Consequently, this coding can recover all chunks even if any m coded chunks are lost.

Lost chunks in a stripe can be reconstructed with the help of the generation matrix or the parity check matrix. Here, we use the generation matrix to illustrate the reconstruction process. Data chunks $\{x_0, x_1, \ldots, x_k\}$ are encoded into chunks $\{y_0, y_1, \ldots, y_n\}$ by:

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} G_{00} & G_{01} & \cdots & G_{0(k-1)} \\ G_{10} & G_{11} & \cdots & G_{1(k-1)} \\ \vdots & \vdots & \ddots & \vdots \\ G_{(n-1)0} & G_{(n-1)1} & \cdots & G_{(n-1)(k-1)} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{k-1} \end{pmatrix}. \tag{17}$$

Assuming that the data chunk x_i $(0 \leq i < n)$ is lost, a k-order square matrix G' can be derived from the generation matrix G by choosing any k rows (except for the ith row) from it. As matrix G' must be reversible in an RS code, any data chunk in $x_0, x_1, \ldots, x_k$ can be reconstructed with the surviving symbols y_j $(j \neq i)$. In this way, the lost chunk can be recovered. Multiple lost chunks can be reconstructed in a similar way.

The above reconstruction process involves matrix inversion. Normally, the complexity of reversing an n-order square matrix is $O(n^3)$. However, when the matrix is a Cauchy matrix, its reversing complexity can be reduced to $O(n^2)$ [9].

5.1.2 Challenges of SSD Integration. Unlike cross-device coding, cross-machine coding involves distributed I/O accesses, resulting in a potential bottleneck in the network. Although most existing works on cross-machine coding are not specially designed for SSD-based storage, the network bottleneck problem is more prominent in SSD-based storage than HDD-based storage [72]. The main reason is that the bandwidth of SSDs has been improving, while the bandwidth improvement of HDDs has been slow in recent years. The rising device bandwidth makes the network bandwidth a potential bottleneck in cross-machine encoding.

To better demonstrate the network bottleneck in SSD-based cross-machine coding, we list the performance of recent representative HDDs, SSDs, and network adaptors in Table 4. We can get two conclusions from the results.

First, both SATA HDDs and SATA SSDs have had little performance improvement in recent years, but the SATA SSDs have much higher bandwidth than SATA HDDs. This performance gap will further expand under random I/Os, which HDDs suffer to handle. The higher bandwidth of SSDs can potentially cause network bottlenecks when the system is equipped with Ethernet network adaptors varying from 100 Mbps to 1 Gbps. Some advanced network adaptors like 10 Gbps ones may also become bottlenecks during data recovery in cross-machine coding, as the storage can be formed with an array for higher aggregated bandwidth [58, 139], and a node may receive huge incast load from multiple servers for data recovery [97].

Second, the bandwidth of high-performance SSDs is comparable to that of high-performance network adaptors. As shown in the table, high-performance network adaptors like Mellanox Connect-X series can offer over 100 Gbps bandwidth, which can easily handle the performance requirement of a bundle of SATA SSDs. However, deploying high-performance NVMe SSDs is not rare in today's data centers [81, 118], and their peak bandwidth grows along with the bandwidth

Table 4. Performance Evolution of HDDs, SSDs, and High-performance Network Adaptors in Recent Years

Device Category	Device Model	Year	Peak Bandwidth
SATA HDD	Western Digital Ultrastar DC HC510	2015	249 MB/s
	Western Digital Ultrastar DC HC530	2020	255 MB/s
	Western Digital Ultrastar DC HC570	2022	291 MB/s
SATA SSD	Samsung 840 EVO	2014	540 MB/s
	Samsung 860 EVO	2018	550 MB/s
	Samsung 870 EVO	2020	560 MB/s
NVMe SSD	Samsung PM983 (PCIe Gen 3)	2019	3,200 MB/s
	Samsung PM9A3 (PCIe Gen 4)	2021	6,800 MB/s
	Samsung PM1743 (PCIe Gen 5)	2024	14,000 MB/s
NVMe Network Adaptor	Mellanox ConnectX-5 (PCIe Gen 3)	2020	100 Gbps
	Mellanox ConnectX-6 (PCIe Gen 4)	2021	200 Gbps
	Mellanox ConnectX-7 (PCIe Gen 5)	2022	400 Gbps

improvements of NVMe network adaptors. The network adaptors can also be easily saturated by those high-performance SSDs when building NVMe SSD arrays for high-performance storage.

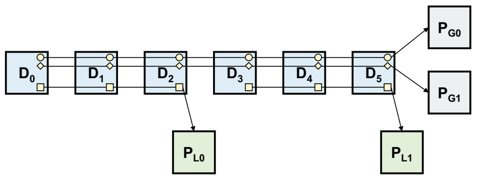
Considering the potential network bottleneck, optimizing network loads is important for SSD-based cross-machine coding. We summarize existing relevant works into three categories that can help to achieve this goal and elaborate on them in the following sections.

5.2 Designing Network-friendly Coding

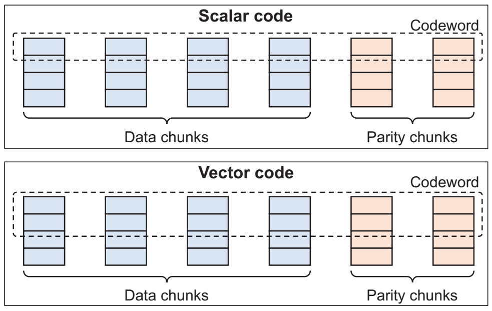
In the RS code, both data recovery and degraded read have larger network loads than transferring a single chunk, and the volume of transferred data increases as the number of data chunks in a stripe expands. Specifically, to reconstruct m' lost chunks ($m' \leq (n-k)$) in an (n,k) RS code, the recovery thread needs to read k surviving chunks. Such larger network loads can burden the cross-machine network. To reduce the amount of data transferred, network-friendly codings are proposed by existing works.

LRC. The **local reconstruction code (LRC)** [48] lowers the reconstruction load by organizing the data chunks into shorter stripes. When recovering a stripe, the data volume that goes over the network equals the number of data chunks k. Therefore, LRC reduces the reconstruction load by decreasing the k. The main challenge is that for the RS code, shrinking the stripe width will decrease the reliability under the same code rate. To this end, LRC designs local parity and global party chunks for fast reconstruction and high reliability. As shown in Figure 14(a), data chunks are divided into equal-sized groups, each of which has one local parity chunk (P_{L0} and P_{L1}). To keep high reliability, global parity chunks (P_{G0} and P_{G1}) are derived by encoding all data chunks of the stripe. In most cases where only one disk is slow or broken, the local parity chunks enable fast degraded read by only reading data in the local group. In rare cases, global parity chunks are involved when the local parity chunks cannot satisfy the degraded read, but its influence on the overall performance is low.

Vector codes. Compared to scalar codes like RS code, vector codes typically have lower recovery bandwidth. As shown in Figure 14(b), the basic coding unit of a scalar code is one symbol, while that of a vector code contains α symbols ($\alpha > 1$). Vector codes have lower recovery bandwidth, as the reconstruction can potentially be accomplished by accessing only a subset of the symbols in a



(a) Local reconstruction code.



(b) Scalar code and vector code. Each block represents an element in the finite field (i.e. a symbol).

Fig. 14. Demonstration of two network-friendly codings.

Table 5. Comparison of Different Methods that Reduce the I/O Amplification of Partial Updates in Cross-machine Coding

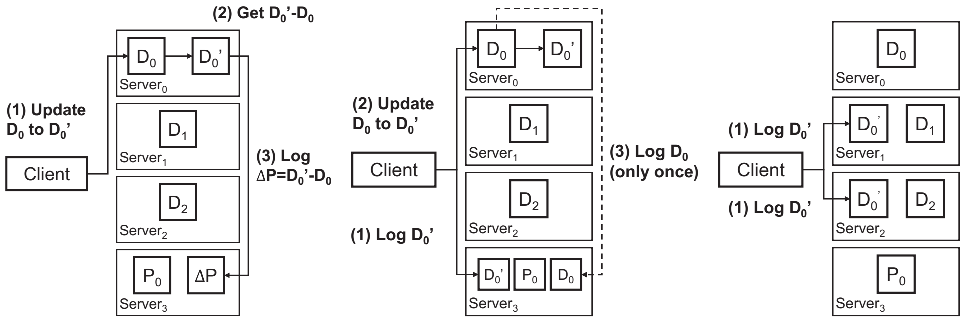
Method	Pros	Cons	Representative works
Parity logging	Reduced I/O randomness on the parity disks.	No network load reduction compared to the basic partial updates.	[15] (2014)
Speculative parity logging	Reduced network loads when data chunks are frequently updated.	Increased network loads when the data hotspot is rare.	[76] (2017), [120] (2022)
Delayed parity update	Further reduced network loads in the critical path.	More network loads when converting the replicas to stripes.	[32] (2021)

coding unit, therefore decreasing the data volumes needed for network transfer compared to the scalar codes [114]. Among the vector codes, the **minimum storage regenerating (MSR)** codes [26] have been paid great attention in recent years, as they are the MDS codes that have the smallest possible repair bandwidth. For an MSR code that encodes k data chunks into n chunks, its minimal recovery data volume is $\frac{n-1}{n-k}$ chunks, which is smaller than the recovery data volume of RS code (i.e., k) when both $(n-k)$ and k are larger than 1. Clay code [114] is a recently proposed MSR code and has been adopted in distributed storage systems like Ceph [122]. There are some challenges when deploying MSR-based storage. First, the total data volume in an MSR code stripe can be much larger than the RS code stripe of the same (n, k), which is unfriendly to small objects [104]. Second, when reconstructing lost chunks in the MSR code stripes, the mathematical structure of MSR codes makes their repair operations difficult to parallelize [79].

5.3 Reducing I/O Amplifications of Partial Updates

Like the cross-device coding, partial updates in the cross-machine coding will also incur the I/O amplification problem [130], resulting in higher loads on storage devices and the network [72]. We summarize existing works on taming tail latency in cross-device coding in Table 5 and elaborate on them in the following parts.

Parity logging. Partial updates on a distributed EC stripe will frequently modify the parity chunks, incurring additional reads on the old parity chunks. Jeremy et al. design parity logging with reserved space for cross-machine coding [15]. As shown in Figure 15(a), instead of performing in-place updates on the parity chunks, the deltas of a parity chunk are appended in a log-structured



(a) Parity logging. (b) Speculative parity logging. (c) Delayed parity update.

Fig. 15. Partial updates critical path operations of parity logging, speculative parity logging, and replayed parity update. In this example, we assume each stripe contains three data chunks and one parity chunk.

manner, eliminating reading the old parity chunks. Besides, each parity chunk reserves a free space behind it for parity logging, which can boost data locality when merging the parity chunks with their parity logging. In this example, the storage nodes have two chunks of inbound and one chunk of outbound loads in total.

Speculative partial write. Parity logging does not solve the problem that a partial update needs to read the old versions of data chunks when calculating the deltas of parity chunks. To this end, speculative partial write is proposed and implemented in subsequent works [76, 120, 121, 141]. As shown in Figure 15(b), instead of recording the deltas of data chunks in the parity disk, partial updates in the speculative partial write only send the new version of the data chunk to the parity disk, speculating that the old version of the relevant chunk has been stored in the parity disk. If the old version of the data chunk is not in the parity disk (i.e., this is the first update on this data chunk), then an additional read is issued for this data chunk. Speculative partial write can eliminate the read I/Os on data chunks that have been updated at least once. In this example, when the data chunk D_0 is updated for the first time, an additional chunk of transfer is needed between the data server and the parity server. Otherwise, the storage nodes only have two chunks of inbound loads in total.

Delayed parity update. Speculative partial write can reduce read I/Os only when data hotspots exist (i.e., a data chunk is partially updated multiple times). But it cannot benefit the first partial updates. Delayed parity update is proposed in Reference [32]. It shares the key idea with the two-phase write. As shown in Figure 15(c), when a stripe is partially updated, the new data chunk is sent to multiple machines for fault tolerance. This process does not involve reading the old data chunks even for the first partial update. The multiple replicas can be merged to the distributed EC stripe to lower storage costs asynchronously. In this example, the storage nodes have two chunks of inbound loads in total, but at the expense of more network loads when merging the multiple replicas to the EC stripe. This can be done when the foreground I/Os are not heavy to lower the interference.

Log-structured EC. The log-structured EC is another way to solve the partial update problem. In this scheme, each partial update simply marks the old data chunk as invalid and writes the new chunk in another place to form a new stripe. Hence, the parity chunks of the old stripe do not need to change. There are some papers designing efficient cross-machine coding for distributed storage systems with the append-only semantic [25, 100]. Some industrial distributed storage systems organize data in a log-structured layout and only support out-of-place updates such as **Windows Azure Storage (WAS)** [48], HDFS [129], and Alibaba Pangu [78]. The cross-machine coding in these systems can benefit from the append-only semantic concerning the partial update problem. The main drawback of the out-of-place storage scheme is that the read performance can be impaired. Each update on the same data chunk can span different ranges, so reading a chunk requires merging multiple updates that intersect with the target chunk.

5.4 Sidestepping Hotspot

In distributed storage, data may be accessed in a skewed way. The access frequency of some data is much higher than the others, resulting in data hotspots and load imbalance. The storage nodes that hold the hot data may respond to remote I/O requests slower, resulting in long tail latency [1, 3, 47, 50, 82].

The data hotspot can be alleviated by scheduling the application I/Os to the redundant data. Both multi-replica storage and coded storage have redundant data for scheduling. However, I/O scheduling on coded storage is more challenging, as reconstructing one data chunk needs to read k surviving chunks, incurring high I/O amplification. This will also increase the loads on the network, possibly adversely impacting the system performance. Besides, when the number of redundancy m is more than one, it will be more complicated to schedule the degraded read, as it has more choices of the set of source chunks.

Some works [1, 3, 47, 50, 82] design cross-machine coding systems that can adaptively choose the read strategy. They measure the I/O queue depth on each storage node and model the overall performance as a function with different read strategies, such as the normal reads or different choices of the degraded read. By treating this as an optimization problem, they can get the optimal probability that each client node should issue normal reads or degraded reads. These works can solve the I/O amplification problem, as they treat the system as a whole and the impacts of I/O amplifications are taken into consideration.

6 Key Insights and Future Directions

In this section, we present a timeline of existing works addressing data failures and performance instability in flash storage. We also summarize key lessons from these works and point out research trends to inspire future works.

6.1 Timeline of Flash-oriented Coded Storage

SSDs provide higher performance than HDDs, but the characteristics of flash media present drawbacks such as data vulnerability and unstable performance. We have reviewed existing flash-oriented coded storage systems that address these issues, focusing on their storage architecture (i.e., in-device coding, cross-device coding, and cross-machine coding), in the above sections. In this part, we present a brief timeline to clearly illustrate their developing history.

6.1.1 Enhancing Data Reliability. Ever since Fujio Masuoka proposed two types of flash memory named NOR flash [87] and NAND flash [88] in the 1980s, the reliability concerns have always been with flash storage. To address reliability issues, flash storage has integrated coding theory that was originally designed for communication systems to get reliable message transmission in an unreliable channel. Among various coding methods, some of them outperform the others considering the error correction ability, flexibility in coding configuration, and realization complexity of hardware circuits. Those coding methods include the Hamming code (1950) [38], BCH code (1959) [45], RS code (1960) [99], and LDPC code (1962) [34]. Recent coding methods address systemic problems during encoding and decoding while maintaining their error correction capabilities. For example, the Cauchy-matrix-based RS code (1995) [9] reduces the encoding and decoding complexity to quadratic complexity, and the **minimum storage regenerating (MSR)** code (2010) [26] reduces the data volumes needed for recovery.

Regarding flash-based cross-device and cross-machine coding, concerns arise about whether out-of-device coding can enhance storage reliability since the 2010s [60], as partial updates may cause write amplification in coded storage and accelerate flash wear. Subsequent works [71, 80, 91] build various analytic models to guide the systemic design of flash-oriented coded storage for high reliability. With theoretical support on the reliability of flash-based storage, numerous works focus on optimizing system performance in terms of decoding latency, I/O amplification, and load balancing.

6.1.2 Stabilizing Storage Performance. Addressing the unstable performance of flash-based storage, particularly the GC-induced performance variation, has been a long-standing topic in flash storage systems. In this article, we focus on leveraging data redundancy to schedule I/O requests, aiming to enhance performance stability. Some works achieve this by manipulating the GC process within SSDs to ensure that the number of SSDs that are performing GC concurrently is no more than the redundancy m, such as the TTFLASH (2017) [134] and the request redirection (2018) [126]. The advantage of these methods is that they can theoretically avoid slow devices, as both degraded reads and redirected writes can bypass performance-degraded devices.

The main challenge of this method is that it requires in-device manipulation to control the GC operations inside SSDs, which may not be feasible for commodity SSDs. Regarding these, some works design flash-based storage systems to detect performance-degraded SSDs by treating them as black-box devices, such as the ToleRAID (2016) [39], LinnOS (2020) [40], and FusionRAID (2021) [58]. These works efficiently address the performance variation problem of commodity flash storage systems in a best-effort manner.

Recently, with the promotion of NVMe SSDs, it has become possible to monitor and control the GCs inside SSDs, thanks to the more abundant device interfaces provided by the NVMe specification [28]. One recent work, IODA (2021) [75], has exploited this new interface to schedule the GC inside SSDs, achieving stable I/O performance with data redundancy while requiring only minimal firmware changes.

6.2 Lessons from Existing Works

This article mainly talks about how to build reliable and consistent storage, especially SSD-based storage, with data coding. Designing an efficient coded storage system is challenging, as the encoding and decoding processes can introduce non-negligible overhead. Besides, maintaining the consistency between user data and redundant data is also expensive. We derive some lessons from existing works for designing flash-oriented coded storage.

Designing systematic codes for cross-device coding and cross-machine coding. The ECC in in-device coding can be both systematic code and non-systematic code, but most EC in cross-device coding or cross-machine coding is systematic code. This is because ECCs are mainly used for correcting raw-bit errors in flash chips. Decoding is needed for each page read, as the SSD controller does not know whether there are bit flips in the page. By contrast, ECs are mainly used to recover lost data when erasure happens. Normal reads do not need to perform data decoding with a systematic EC. For example, in an (n, k) systematic cross-device or cross-machine coding, a normal read on a data chunk does not incur read amplification. If data is protected with non-systematic coding, then k chunks are needed to reconstruct the original data chunks, incurring high I/O amplification especially when I/Os are small.

Adopting dynamic code rate for lower storage cost. Using flash ECC with static code rates is not the most efficient way. The requirement for error correction ability is both time-varying and data-varying. For time-varying requirements, the reliability of flash media decreases with increasing P/E cycles, so it is better to strengthen the ECC as flash aging. For data-varying requirements, the data lost penalty differs with the scenarios, like data loss is acceptable when it is a cached copy of lower storage data. It is acceptable to use weaker ECC or even no ECC for these kinds of data, which can enhance the normal I/O performance. Existing work [138] has shown that ECC with dynamic code rates can reduce the average response time by 32.8% and decrease the write amplification by 37.8%.

Delaying parity updates for efficient partial writes. Tradeoffs are made to alleviate the overhead for maintaining the EC stripe consistency. For cross-device and cross-machine coding, the parity chunks are kept consistent with the data chunks to guarantee data recovery in case of failures. This strong consistency will adversely impact the performance of partial updates and incur I/O amplifications. Previous works solve this problem by trading off other aspects, such as temporary space overhead (parity logging and two-phase write) or altered data layout (append-only semantics). These tradeoffs are reasonable, considering that the SSDs are larger and more affordable in recent years, while the improvements in latency are much smaller [58].

Taming tail latency with I/O scheduling and GC coordination. When taming the I/O tail latency problem, write can be redirected to avoid the performance-degraded SSDs, but read needs degraded read to reconstruct target data with redundant data. For the degraded read, the user should consider when to perform degraded read (i.e., whether when a slow read is detected or just actively send redundant requests) and whether performing degraded read even spike latency occurs to avoid load flooding. Besides, GC coordination is an effective way to lower the performance influence of SSD GCs, but it requires in-device manipulations, which are challenging to implement for commodity SSDs.

6.3 Future Research Directions

For flash-oriented coded storage, we speculate the following research trends:

ECC with longer codeword. The flash media of SSDs is becoming more vulnerable as the storage density increases. Therefore, stronger ECCs are needed for future in-device coding. A stronger ECC can be achieved by lowering the code rates or increasing the codeword width. Lowering the code rates means introducing more redundant data for data protection. This will increase the space overhead and reduce the additional space for SSD GC, which will accelerate flash wearing and shorten SSD lifetime. However, an MDS ECC with a longer codeword has better error correction ability with the same code rate. Assuming that the coding configurations (n, k) of two MDS ECCs are (N, K) and $(2N, 2K)$, respectively, they have the same code rate $\frac{K}{N}$. However, when encoding user data of $2K$ bits, the latter one can correct arbitrary $2(N-K)$ bit flips, while the former one can only correct $(N-K)$ arbitrary bit flips at most. The main challenge is that longer ECC codewords also complicate the encoding and decoding processes, affecting the I/O latency and hardware complexity. A reasonable tradeoff between reliability and coding complexity should be made for ECC designs.

Application-aware coding. Achieving both strong coding and high encoding/decoding performance is challenging. A promising solution is to design application-aware coding techniques and selectively choose the code rate for user data. For example, important user data like file system metadata are protected with stronger codes. The codes can be weaker when SSDs are used as cache, as lost data can be reloaded from the secondary storage. The application-aware coding is easy to achieve for software encoding, but it is harder for in-device coding like page-level ECCs. To accomplish this goal, the SSDs need to expose more complex interfaces that allow the driver to choose the coding strategy for given data.

Host-informed and host-managed GC. In-device information and manipulation is helpful for taming the I/O tail latency derived from SSD GC. However, it is hard to achieve for commodity SSDs that only expose limited I/O interfaces. The **open-channel SSDs (OC-SSD)** give users entire control over the SSD and GC, but it complicates software design and may incur security concerns [17]. A promising direction is that the device only exposes more semantics than simple I/O interfaces. For example, the multi-streamed SSD [61, 73] presents an interface to store certain data within one block. This allows the user to separate hot data and cold data into different streams, thereby reducing the GC frequencies of the cold streams. In recent years, the **non-volatile memory express (NVMe)** specifications present many standards [28], such as the predictable latency [75] and **Zoned Namespaces SSDs (ZNS)** [67, 117], which all enrich the user's ability to know or control the GC inside the SSDs.

7 Conclusion

Flash-oriented coded storage has the potential to address the limited lifetime and fluctuated performance of flash memory at a small storage overhead, yet attaining low access latency, consistent performance, and fast recovery simultaneously remains a complex challenge. In this article, we categorize the coding techniques in current coded SSD storage into in-device coding, cross-device coding, and cross-machine coding. For each of them, we discuss the theoretical bases and elaborate on reviewing existing works designed for different goals. Additionally, we distill key insights and propose several research trajectories to inspire future investigations.

Acknowledgments

We thank all reviewers for their insightful comments and helpful suggestions.

References

[1] Vaneet Aggarwal, Jingxian Fan, and Tian Lan. 2017. Taming tail latency for erasure-coded, distributee storage systems. In *IEEE Conference on Computer Communications*. 1–9. DOI: https://doi.org/10.1109/INFOCOM.2017.8056997

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *USENIX Annual Technical Conference (USENIX ATC'08)*.

[3] Abubakr O. Al-Abbasi, Vaneet Aggarwal, and Tian Lan. 2019. TTLoC: Taming tail latency for erasure-coded cloud storage systems. *IEEE Trans. Netw. Serv. Manag.* 16, 4 (2019), 1609–1623. DOI: https://doi.org/10.1109/TNSM.2019.2916877

[4] Idan Alrod and Menahem Lasser. 2013. Fast, low-power reading of data in a flash memory. US Patent 8,433,980.

[5] H. Peter Anvin. 2007. The mathematics of RAID-6. http://ftp.dei.uc.pt/pub/linux/kernel/people/hpa/raid6.pdf

[6] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. 2010. Differential RAID: Rethinking RAID for SSD reliability. *ACM Trans. Stor.* 6, 2, Article 4 (July 2010), 22 pages. DOI: https://doi.org/10.1145/1807060.1807061

[7] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a needle in haystack: Facebook's photo storage. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association. Retrieved from https://www.usenix.org/conference/osdi10/finding-needle-haystack-facebooks-photo-storage

[8] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. 1995. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Trans. Comput.* 44, 2 (1995), 192–202.

[9] Johannes Blomer. 1995. *An XOR-based Erasure-resilient Coding Scheme*. Technical Report. ICSI.

[10] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. 2017. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proc. IEEE* 105, 9 (2017), 1666–1704. DOI: https://doi.org/10.1109/JPROC.2017.2713127

[11] Yu Cai, Yunxiang Wu, and Erich F. Haratsch. 2017. Hot-read data aggregation and code selection. US Patent 9,785,499.

[12] Yu Cai, Gulay Yalcin, Onur Mutlu, Erich F. Haratsch, Adrian Crista, Osman S. Unsal, and Ken Mai. 2013. Error analysis and retention-aware error management for NAND flash memory. *Intel Technol. J.* 17, 1 (2013).

[13] Jaewon Cha and Sungho Kang. 2013. Data randomization scheme for endurance enhancement and interference mitigation of multilevel flash memory devices. *ETRI J.* 35, 1 (2013), 166–169.

[14] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. Elastic parity logging for SSD RAID arrays: Design, analysis, and implementation. *IEEE Trans. Parallel Distrib. Syst.* 29, 10 (2018), 2241–2253. DOI: https://doi.org/10.1109/TPDS.2018.2818171

[15] Jeremy C. W. Chan, Qian Ding, Patrick P. C. Lee, and Helen H. W. Chan. 2014. Parity logging with reserved space: Towards efficient updates and recovery in erasure-coded clustered storage. In *12th USENIX Conference on File and Storage Technologies (FAST'14)*. USENIX Association, 163–176. Retrieved from https://www.usenix.org/conference/fast14/technical-sessions/presentation/chan

[16] Bainan Chen, Xinmiao Zhang, and Zhongfeng Wang. 2008. Error correction for multi-level NAND flash memory using Reed-Solomon codes. In *IEEE Workshop on Signal Processing Systems*. IEEE, 94–99.

[17] Junchao Chen, Guangyan Zhang, and Junyu Wei. 2023. A survey on design and application of open-channel solid-state drives. *Front. Inf. Technol. Electron. Eng.* 24, 5 (2023), 637–658.

[18] Tzi-cker Chiueh, Weafon Tsao, Hou-Chiang Sun, Ting-Fang Chien, An-Nan Chang, and Cheng-Ding Chen. 2014. Software orchestrated flash array. In *International Conference on Systems and Storage (SYSTOR'14)*. Association for Computing Machinery, New York, NY, USA, 1–11. DOI: https://doi.org/10.1145/2611354.2611360

[19] Ching-Che Chung and Hao-Hsiang Hsu. 2014. Partial parity cache and data cache management method to improve the performance of an SSD-based RAID. *IEEE Trans. Very Large Scale Integ. Syst.* 22, 7 (2014), 1470–1480. DOI: https://doi.org/10.1109/TVLSI.2013.2275737

[20] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. 2015. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. Association for Computing Machinery, New York, NY, USA, 1683–1694. DOI: https://doi.org/10.1145/2723372.2742798

[21] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. 2004. Row-diagonal parity for double disk failure correction. In *3rd USENIX Conference on File and Storage Technologies*. 1–14.

[22] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. DOI: https://doi.org/10.1145/2408776.2408794

[23] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating System Principles*. Retrieved from https://www.amazon.science/publications/dynamo-amazons-highly-available-key-value-store

[24] DELL EMC. 2023. VMAX all flash family. Retrieved from https://www.dellemc.com/en-us/collaterals/unauth/data-sheets/products/storage-2/h16051-vmax-all-flash-250f-950f-ss.pdf

[25] Haiwei Deng, Ranhao Jia, and Chentao Wu. 2021. A graph-assisted out-of-place update scheme for erasure coded storage systems. In *50th International Conference on Parallel Processing (ICPP'21)*. Association for Computing Machinery, New York, NY, USA, Article 20, 10 pages. DOI: https://doi.org/10.1145/3472456.3472502

[26] Alexandros G. Dimakis, P. Brighten Godfrey, Yunnan Wu, Martin J. Wainwright, and Kannan Ramchandran. 2010. Network coding for distributed storage systems. *IEEE Trans. Inf. Theor.* 56, 9 (2010), 4539–4551.

[27] Guiqiang Dong, Ningde Xie, and Tong Zhang. 2010. On the use of soft-decision error-correction codes in NAND flash memory. *IEEE Trans. Circ. Syst. I: Reg. Pap.* 58, 2 (2010), 429–439.

[28] NVM Express. 2022. NVM express base specification 2.0c. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0c-2022.10.04-Ratified-1.pdf

[29] Michele Favalli, Cristian Zambelli, Alessia Marelli, Rino Micheloni, and Piero Olivo. 2021. A scalable bidimensional randomization scheme for TLC 3D NAND flash memories. *Micromachines* 12, 7 (2021), 759.

[30] A. Fiat and A. Shamir. 1984. Generalized "write-once" memories. *IEEE Trans. Inf. Theor.* 30, 3 (1984), 470–480. DOI: https://doi.org/10.1109/TIT.1984.1056918

[31] FUJITSU. 2023. FUJITSU storage ETERNUS AF650 S3. Retrieved from https://sp.ts.fujitsu.com/dmsp/Publications/public/ds-eternus-af650-s3-ww-en.pdf

[32] Takayuki Fukatani, Hieu Hanh Le, and Haruo Yokota. 2021. Delayed parity update for bridging the gap between replication and erasure coding in server-based storage. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS@VLDB'21)*.

[33] Ryan Gabrys and Lara Dolecek. 2015. Constructions of nonbinary WOM codes for multilevel flash memories. *IEEE Trans. Inf. Theor.* 61, 4 (2015), 1905–1919.

[34] Robert Gallager. 1962. Low-density parity-check codes. *IRE Trans. Inf. Theor.* 8, 1 (1962), 21–28.

[35] Kevin M. Greenan, Darrell Long, Ethan L. Miller, Thomas J. E. Schwarz, and Avani Wildani. 2009. Building flexible, fault-tolerant flash-based storage systems. In *5th Workshop on Hot Topics in Dependability (HotDep'09)*.

[36] Keonsoo Ha, Jaeyong Jeong, and Jihong Kim. 2013. A read-disturb management technique for high-density NAND flash memory. In *4th Asia-Pacific Workshop on Systems*. 1–6.

[37] Frank T. Hady, Annie Foong, Bryan Veal, and Dan Williams. 2017. Platform storage performance with 3D XPoint technology. *Proc. IEEE* 105, 9 (2017), 1822–1833. DOI:https://doi.org/10.1109/JPROC.2017.2731776

[38] Richard W. Hamming. 1950. Error detecting and error correcting codes. *Bell Syst. Technic. J.* 29, 2 (1950), 147–160.

[39] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. 2016. The tail at store: A revelation from millions of hours of disk and SSD deployments. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association, 263–276. Retrieved from https://www.usenix.org/conference/fast16/technical-sessions/presentation/hao

[40] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. 2020. LinnOS: Predictability on unpredictable flash storage with a light neural network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, 173–190. Retrieved from https://www.usenix.org/conference/osdi20/presentation/hao

[41] Erich F. Haratsch. 2016. Media management for high density NAND flash memories. *Flash Mem. Summ.* 3, 3 (2016), 2–7.

[42] Ahmed Hareedy, Beyza Dabak, and Robert Calderbank. 2020. Q-ary asymmetric LOCO codes: Constrained codes supporting flash evolution. In *IEEE International Symposium on Information Theory (ISIT'20)*. IEEE, 688–693.

[43] Karl Scott Hemmert, Stan Gerald Moore, Michail A. Gallis, Mike E. Davis, John Levesque, Nathan Hjelm, James Lujan, David Morton, Hai Ah Nam, Alex Parga, et al. 2018. Trinity: Opportunities and challenges of a heterogeneous system. In *Cray User Group (CUG'18)*.

[44] Dave Henseler, Benjamin Landsteiner, Doug Petesch, Cornell Wright, and Nicholas J. Wright. 2016. Architecture and design of Cray DataWarp. In *Cray User Group (CUG'16)*.

[45] Alexis Hocquenghem. 1959. Codes correcteurs d'erreurs. *Chiffers* 2 (1959), 147–156.

[46] Jen-Wei Hsieh, Chung-Wei Chen, and Han-Yi Lin. 2015. Adaptive ECC scheme for hybrid SSD's. *IEEE Trans. Comput.* 64, 12 (2015), 3348–3361. DOI:https://doi.org/10.1109/TC.2015.2401028

[47] Yaochen Hu, Yushi Wang, Bang Liu, Di Niu, and Cheng Huang. 2017. Latency reduction and load balancing in coded storage systems. In *Symposium on Cloud Computing (SoCC'17)*. Association for Computing Machinery, New York, NY, USA, 365–377. DOI:https://doi.org/10.1145/3127479.3131623

[48] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in Windows Azure storage. In *USENIX Annual Technical Conference (USENIX ATC'12)*. USENIX Association, 15–26. Retrieved from https://www.usenix.org/conference/atc12/technical-sessions/presentation/huang

[49] Jianzhong Huang, Fenghao Zhang, Xiao Qin, and Changsheng Xie. 2013. Exploiting redundancies and deferred writes to conserve energy in erasure-coded storage clusters. *ACM Trans. Stor.* 9, 2, Article 4 (July 2013), 29 pages. DOI:https://doi.org/10.1145/2491472.2491473

[50] Longbo Huang, Sameer Pawar, Hao Zhang, and Kannan Ramchandran. 2012. Codes can reduce queueing delay in data centers. In *IEEE International Symposium on Information Theory Proceedings*. 2766–2770. DOI:https://doi.org/10.1109/ISIT.2012.6284026

[51] Ping Huang, Pradeep Subedi, Xubin He, Shuang He, and Ke Zhou. 2014. FlexECC: Partially relaxing ECC of MLC SSD for better cache performance. In *USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, 489–500. Retrieved from https://www.usenix.org/conference/atc14/technical-sessions/presentation/huang

[52] Soojun Im and Dongkun Shin. 2011. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. *IEEE Trans. Comput.* 60, 1 (2011), 80–92. DOI:https://doi.org/10.1109/TC.2010.197

[53] Intel. 2018. Intel Optane SSD 905P series specification. Retrieved from https://www.intel.com/content/www/us/en/products/sku/147529/intel-optane-ssd-905p-series-960gb-2-5in-pcie-x4-3d-xpoint/specifications.html

[54] Nikolas Ioannou, Kornilios Kourtis, and Ioannis Koltsidas. 2018. Elevating commodity storage with the SALSA host translation layer. In *IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'18)*. 277–292. DOI:https://doi.org/10.1109/MASCOTS.2018.00035

[55] Shehbaz Jaffer, Kaveh Mahdaviani, and Bianca Schroeder. 2020. Rethinking WOM codes to enhance the lifetime in New SSD generations. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'20)*.

[56] Shehbaz Jaffer, Kaveh Mahdaviani, and Bianca Schroeder. 2022. Improving the reliability of next generation SSDs using WOM-v codes. In *20th USENIX Conference on File and Storage Technologies (FAST'22)*. 117–132.

[57] Nikolaus Jeremic, Gero Mühl, Anselm Busse, and Jan Richling. 2011. The pitfalls of deploying solid-state drive RAIDs. In *4th Annual International Conference on Systems and Storage*. 1–13.

[58] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. 2021. FusionRAID: Achieving consistent low latency for commodity SSD arrays. In *19th USENIX Conference on File and Storage Technologies (FAST'21)*. USENIX Association, 355–370.

[59] Rinu Jose and Ameenudeen Pe. 2015. Analysis of hard decision and soft decision decoding algorithms of LDPC codes in AWGN. In *IEEE International Advance Computing Conference (IACC'15)*. 430–435. DOI: https://doi.org/10.1109/IADCC.2015.7154744

[60] Asim Kadav, Mahesh Balakrishnan, Vijayan Prabhakaran, and Dahlia Malkhi. 2010. Differential raid: Rethinking raid for SSD reliability. *ACM SIGOPS Operat. Syst. Rev.* 44, 1 (2010), 55–59.

[61] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'14)*.

[62] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. 2008. Characterization of storage workload traces from production Windows Servers. In *IEEE International Symposium on Workload Characterization*. 119–128. DOI: https://doi.org/10.1109/IISWC.2008.4636097

[63] Scott Kayser and Paul H. Siegel. 2014. Constructions for constant-weight ICI-free codes. In *IEEE International Symposium on Information Theory*. 1431–1435. DOI: https://doi.org/10.1109/ISIT.2014.6875069

[64] Jaeho Kim, Jongmin Lee, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2012. Enhancing SSD reliability through efficient RAID support. In *Asia-Pacific Workshop on Systems (APSYS'12)*. Association for Computing Machinery, New York, NY, USA, Article 4, 6 pages. DOI: https://doi.org/10.1145/2349896.2349900

[65] Jaeho Kim, Jongmin Lee, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2013. Improving SSD reliability with RAID via elastic striping and anywhere parity. In *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*. 1–12. DOI: https://doi.org/10.1109/DSN.2013.6575359

[66] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. 2019. Alleviating garbage collection interference through spatial separation in all flash arrays. In *USENIX Annual Technical Conference (USENIX ATC'19)*. USENIX Association, 799–812. Retrieved from https://www.usenix.org/conference/atc19/presentation/kim-jaeho

[67] Thomas Kim, Jekyeom Jeon, Nikhil Arora, Huaicheng Li, Michael Kaminsky, David G. Andersen, Gregory R. Ganger, George Amvrosiadis, and Matias Bjørling. 2023. RAIZN: Redundant array of independent zoned namespaces. In *28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 660–673.

[68] Youngjae Kim. 2015. An empirical study of redundant array of independent solid-state drives (RAIS). *Cluster Comput.* 18, 2 (June 2015), 963–977. DOI: https://doi.org/10.1007/s10586-015-0421-4

[69] Youngjae Kim, Junghee Lee, Sarp Oral, David A. Dillow, Feiyi Wang, and Galen M. Shipman. 2014. Coordinating garbage collection for arrays of solid-state drives. *IEEE Trans. Comput.* 63, 4 (2014), 888–901. DOI: https://doi.org/10.1109/TC.2012.256

[70] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. 2011. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST'11)*. 1–12. DOI: https://doi.org/10.1109/MSST.2011.5937224

[71] Mostafa Kishani, Saba Ahmadian, and Hossein Asadi. 2019. A modeling framework for reliability of erasure codes in SSD arrays. *IEEE Trans. Comput.* 69, 5 (2019), 649–665.

[72] Sungjoon Koh, Jie Zhang, Miryeong Kwon, Jungyeon Yoon, David Donofrio, Nam Sung Kim, and Myoungsoo Jung. 2019. Exploring fault-tolerant erasure codes for scalable all-flash array clusters. *IEEE Trans. Parallel Distrib. Syst.* 30, 6 (2019), 1312–1330. DOI: https://doi.org/10.1109/TPDS.2018.2884722

[73] Donghun Koo, Jaehwan Lee, Jialin Liu, Eun-Kyu Byun, Jae-Hyuck Kwak, Glenn K. Lockwood, Soonwook Hwang, Katie Antypas, Kesheng Wu, and Hyeonsang Eom. 2021. An empirical study of I/O separation for burst buffers in HPC systems. *J. Parallel Distrib. Comput.* 148 (2021), 96–108.

[74] Youngjoo Lee, Hoyoung Yoo, Injae Yoo, and In-Cheol Park. 2012. 6.4Gb/s multi-threaded BCH encoder and decoder for multi-channel SSD controllers. In *IEEE International Solid-state Circuits Conference*. 426–428. DOI: https://doi.org/10.1109/ISSCC.2012.6177075

[75] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. 2021. IODA: A host/device co-design for strong predictability contract on modern flash storage. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. Association for Computing Machinery, New York, NY, USA, 263–279. DOI: https://doi.org/10.1145/3477132.3483573

[76] Huiba Li, Yiming Zhang, Zhiming Zhang, Shengyun Liu, Dongsheng Li, Xiaohui Liu, and Yuxing Peng. 2017. PARIX: Speculative partial writes in erasure-coded systems. In *USENIX Annual Technical Conference (USENIX ATC'17)*. USENIX Association, 581–587. Retrieved from https://www.usenix.org/conference/atc17/technical-sessions/presentation/li-huiba

[77] Jun Li, Balazs Gerofi, Francois Trahay, Zhigang Cai, and Jianwei Liao. 2023. Rep-RAID: An integrated approach to optimizing data replication and garbage collection in RAID-enabled SSDs. In *24th ACM SIGPLAN/SIGBED*

International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'23). Association for Computing Machinery, New York, NY, USA, 99–110. DOI: https://doi.org/10.1145/3589610.3596274

[78] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu et al. 2023. More than capacity: Performance-oriented evolution of Pangu in Alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST'23)*. USENIX Association, 331–346. Retrieved from https://www.usenix.org/conference/fast23/presentation/li-qiang-deployed

[79] Xiaolu Li, Keyun Cheng, Kaicheng Tang, Patrick P. C. Lee, Yuchong Hu, Dan Feng, Jie Li, and Ting-Yi Wu. 2023. ParaRC: Embracing sub-packetization for repair parallelization in MSR-coded storage. In *21st USENIX Conference on File and Storage Technologies (FAST'23)*. 17–32.

[80] Yongkun Li, Patrick P. C. Lee, and John C. S. Lui. 2014. Analysis of reliability dynamics of SSD RAID. *IEEE Trans. Comput.* 65, 4 (2014), 1131–1144.

[81] Zhiyue Li and Guangyan Zhang. 2024. StreamCache: Revisiting page cache for file scanning on fast storage devices. In *USENIX Annual Technical Conference (USENIX ATC'24)*. 1119–1134.

[82] Guanfeng Liang and Ulaş C. Kozat. 2014. FAST CLOUD: Pushing the envelope on delay performance of cloud storage with coding. *IEEE/ACM Trans. Netw.* 22, 6 (2014), 2012–2025. DOI: https://doi.org/10.1109/TNET.2013.2289382

[83] Yixin Luo, Yu Cai, Saugata Ghose, Jongmoo Choi, and Onur Mutlu. 2015. WARM: Improving NAND flash memory lifetime with write-hotness aware retention management. In *31st Symposium on Mass Storage Systems and Technologies (MSST'15)*. IEEE, 1–14.

[84] Yina Lv, Liang Shi, Qiao Li, Congming Gao, Yunpeng Song, Longfei Luo, and Youtao Zhang. 2023. MGC: Multiple-gray-code for 3D NAND flash based high-density SSDs. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA'23)*. 122–136. DOI: https://doi.org/10.1109/HPCA56546.2023.10070946

[85] David J. C. MacKay and Radford M. Neal. 1997. Near Shannon limit performance of low density parity check codes. *Electron. Lett.* 33, 6 (1997), 457–458.

[86] Bo Mao, Hong Jiang, Suzhen Wu, Lei Tian, Dan Feng, Jianxi Chen, and Lingfang Zeng. 2012. HPDA: A hybrid parity-based disk array for enhanced performance and reliability. *ACM Trans. Stor.* 8, 1, Article 4 (Feb. 2012), 20 pages. DOI: https://doi.org/10.1145/2093139.2093143

[87] F. Masuoka, M. Asano, H. Iwahashi, T. Komuro, and S. Tanaka. 1984. A new flash E2PROM cell using triple polysilicon technology. In *International Electron Devices Meeting*. 464–467. DOI: https://doi.org/10.1109/IEDM.1984.190752

[88] F. Masuoka, M. Momodomi, Y. Iwata, and R. Shirota. 1987. New ultra high density EPROM and flash EEPROM with NAND structure cell. In *International Electron Devices Meeting*. 552–555. DOI: https://doi.org/10.1109/IEDM.1987.191485

[89] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. 2015. A large-scale study of flash memory failures in the field. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'15)*. Association for Computing Machinery, New York, NY, USA, 177–190. DOI: https://doi.org/10.1145/2745844.2745848

[90] Sangwhan Moon and A. L. Narasimha Reddy. 2013. Don't let RAID raid the lifetime of your SSD array. In *5th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'13)*. USENIX Association, 7.

[91] Sangwhan Moon and A. L. Narasimha Reddy. 2016. Does RAID improve lifetime of SSD arrays? *ACM Trans. Stor.* 12, 3 (2016), 1–29.

[92] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. 2014. f4: Facebook's warm BLOB storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, 383–398. Retrieved from https://www.usenix.org/conference/osdi14/technical-sessions/presentation/muralidhar

[93] NetApp. 2023. AFF A-Series all flash arrays. Retrieved from https://www.netapp.com/pdf.html?item=/media/7828-DS-3582-AFF-A-Series.pdf

[94] Daniel Nicolas Bailon, Johann-Philipp Thiers, and Jürgen Freudenberger. 2022. Error correction for TLC and QLC NAND flash memories using cell-wise encoding. *Electronics* 11, 10 (2022), 1585.

[95] Wen Pan and Tao Xie. 2018. A mirroring-assisted channel-RAID5 SSD for mobile applications. *ACM Trans. Embed. Comput. Syst.* 17, 4 (2018), 1–27.

[96] David A. Patterson, Garth Gibson, and Randy H. Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD International Conference on Management of Data (SIGMOD'88)*. Association for Computing Machinery, New York, NY, USA, 109–116. DOI: https://doi.org/10.1145/50202.50214

[97] Yi Qiao, Menghao Zhang, Yu Zhou, Xiao Kong, Han Zhang, Mingwei Xu, Jun Bi, and Jilong Wang. 2022. NetEC: Accelerating erasure coding reconstruction with in-network aggregation. *IEEE Trans. Parallel Distrib. Syst.* 33, 10 (2022), 2571–2583.

[98] Minghai Qin, Eitan Yaakobi, and Paul H. Siegel. 2014. Constrained codes that mitigate inter-cell interference in read/write cycles for flash memories. *IEEE J. Select. Areas Commun.* 32, 5 (2014), 836–846. DOI: https://doi.org/10.1109/JSAC.2014.140504

[99] Irving S. Reed and Gustave Solomon. 1960. Polynomial codes over certain finite fields. *J. Societ. Industr. Appl. Math.* 8, 2 (1960), 300–304.

[100] Yanjing Ren, Yuanming Ren, Xiaolu Li, Yuchong Hu, Jingwei Li, and Patrick P. C. Lee. 2024. ELECT: Enabling erasure coding tiering for LSM-tree-based storage. In *22nd USENIX Conference on File and Storage Technologies (FAST'24)*. USENIX Association, 293–310. Retrieved from https://www.usenix.org/conference/fast24/presentation/ren

[101] Ronald L. Rivest and Adi Shamir. 1982. How to reuse a "write-once" memory. *Inf. Contr.* 55, 1-3 (1982), 1–19.

[102] Luigi Rizzo. 1997. Effective erasure codes for reliable computer communication protocols. *SIGCOMM Comput. Commun. Rev.* 27, 2 (Apr. 1997), 24–36. DOI: https://doi.org/10.1145/263876.263881

[103] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash reliability in production: The expected and the unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association, 67–80. Retrieved from https://www.usenix.org/conference/fast16/technical-sessions/presentation/schroeder

[104] Yingdi Shan, Kang Chen, Tuoyu Gong, Lidong Zhou, Tai Zhou, and Yongwei Wu. 2021. Geometric partitioning: Explore the boundary of optimal erasure code repair. In *ACM SIGOPS 28th Symposium on Operating Systems Principles*. 457–471.

[105] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. 2014. Flash on rails: Consistent flash performance through redundancy. In *USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, 463–474. Retrieved from https://www.usenix.org/conference/atc14/technical-sessions/presentation/skourtis

[106] D. Stodolsky, G. Gibson, and M. Holland. 1993. Parity logging overcoming the small write problem in redundant disk arrays. In *20th Annual International Symposium on Computer Architecture*. 64–75. DOI: https://doi.org/10.1109/ISCA.1993.698546

[107] Dongdong Sun, Yinlong Xu, Yongkun Li, Si Wu, and Chengjin Tian. 2016. Efficient parity update for scaling RAID-like storage systems. In *IEEE International Conference on Networking, Architecture and Storage (NAS'16)*. 1–10. DOI: https://doi.org/10.1109/NAS.2016.7549400

[108] Hairong Sun, Pete Grayson, and Bob Wood. 2011. Quantifying reliability of solid-state storage from multiple aspects. *Proc. SNAPI* 11 (2011).

[109] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, 513–527.

[110] Shuhei Tanakamaru, Chinglin Hung, Atsushi Esumi, Mitsuyoshi Ito, Kai Li, and Ken Takeuchi. 2011. 95%-lower-BER 43%-lower-power intelligent solid-state drive (SSD) with asymmetric coding and stripe pattern elimination algorithm. In *IEEE International Solid-state Circuits Conference*. 204–206. DOI: https://doi.org/10.1109/ISSCC.2011.5746283

[111] Shuhei Tanakamaru, Yuki Yanagihara, and Ken Takeuchi. 2012. Over-10×-extended-lifetime 76%-reduced-error solid-state drives (SSDs) with error-prediction LDPC architecture and error-recovery scheme. In *IEEE International Solid-state Circuits Conference*. 424–426. DOI: https://doi.org/10.1109/ISSCC.2012.6177074

[112] Alexander Thomasian. 2005. Reconstruct versus read-modify writes in RAID. *Inf. Process. Lett.* 93, 4 (2005), 163–168.

[113] Beth Trushkowsky, Peter Bodik, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. 2011. The SCADS director: Scaling a distributed storage system under stringent performance requirements. In *9th USENIX Conference on File and Storage Technologies (FAST'11)*. USENIX Association. Retrieved from https://www.usenix.org/conference/fast11/scads-director-scaling-distributed-storage-system-under-stringent-performance

[114] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P. Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy et al. 2018. Clay codes: Moulding MDS codes to yield an MSR code. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*. 139–154.

[115] Jiadong Wang, Thomas Courtade, Hari Shankar, and Richard D. Wesel. 2011. Soft information for LDPC decoding in flash: Mutual-information optimized quantization. In *IEEE Global Telecommunications Conference (GLOBECOM'11)*. 1–6. DOI: https://doi.org/10.1109/GLOCOM.2011.6134417

[116] Jiadong Wang, Kasra Vakilinia, Tsung-Yi Chen, Thomas Courtade, Guiqiang Dong, Tong Zhang, Hari Shankar, and Richard Wesel. 2014. Enhanced precision through multiple reads for LDPC decoding in flash memories. *IEEE J. Select Areas Commun.* 32, 5 (2014), 880–891. DOI: https://doi.org/10.1109/JSAC.2014.140508

[117] Qiuping Wang and Patrick P. C. Lee. 2023. ZapRAID: Toward high-performance RAID for ZNS SSDs via zone append. In *14th ACM SIGOPS Asia-Pacific Workshop on Systems*. 24–29.

[118] Shucheng Wang, Qiang Cao, Hong Jiang, Ziyi Lu, Jie Yao, Yuxing Chen, and Anqun Pan. 2024. Explorations and exploitation for parity-based RAIDs with ultra-fast SSDs. *ACM Trans. Stor.* 20, 1, Article 6 (Jan. 2024), 32 pages. DOI: https://doi.org/10.1145/3627992

[119] Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, Jie Yao, and Yuanyuan Dong. 2022. StRAID: Stripe-threaded architecture for parity-based RAIDs with ultra-fast SSDs. In *USENIX Annual Technical Conference (USENIX ATC'22)*. 915–932.

[120] Bing Wei, Jigang Wu, Xiaosong Su, Qiang Huang, and Yujun Liu. 2022. Adaptive updates for erasure-coded storage systems based on data delta and logging. In *Parallel and Distributed Computing, Applications and Technologies*, Hong

Shen, Yingpeng Sang, Yong Zhang, Nong Xiao, Hamid R. Arabnia, Geoffrey Fox, Ajay Gupta, and Manu Malek (Eds.). Springer International Publishing, Cham, 187–197.

[121] Bing Wei, Jigang Wu, Xiaosong Su, Qiang Huang, Yujun Liu, and Fuhao Zhang. 2022. Efficient erasure-coded data updates based on file class predictions and hybrid writes. *Comput. Electric. Eng.* 104 (2022), 108441. DOI: https://doi.org/10.1016/j.compeleceng.2022.108441

[122] Sage Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *7th Conference on Operating Systems Design and Implementation (OSDI'06).* 307–320.

[123] Wikipedia. 2024. 3D XPoint. Retrieved from https://en.wikipedia.org/wiki/3D_XPoint

[124] Nathan Wong, Ethan Liang, Haobo Wang, Sudarsan V. S. Ranganathan, and Richard D. Wesel. 2019. Decoding flash memory with progressive reads and independent vs. joint encoding of bits in a cell. In *IEEE Global Communications Conference (GLOBECOM'19).* IEEE, 1–6.

[125] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2019. Towards an unwritten contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'19).*

[126] Suzhen Wu, Haijun Li, Bo Mao, Xiaoxi Chen, and Kuan-Ching Li. 2018. Overcome the GC-induced performance variability in SSD-based RAIDs with request redirection. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 38, 5 (2018), 822–833.

[127] Suzhen Wu, Bo Mao, Xiaolan Chen, and Hong Jiang. 2016. LDM: Log disk mirroring with improved performance and reliability for SSD-based disk arrays. *ACM Trans. Stor.* 12, 4, Article 22 (May 2016), 21 pages. DOI: https://doi.org/10.1145/2892639

[128] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. 2015. CosTLO: Cost-effective redundancy for lower latency variance on cloud storage services. In *12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15).* USENIX Association, 543–557.

[129] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. 2015. A tale of two erasure codes in HDFS. In *13th USENIX Conference on File and Storage Technologies (FAST'15).* USENIX Association, 213–226.

[130] Yifei Xiao, Shijie Zhou, and Linpeng Zhong. 2020. Erasure coding-oriented data update for cloud storage: A survey. *IEEE Access* 8 (2020), 227982–227998. DOI: https://doi.org/10.1109/ACCESS.2020.3033024

[131] Lihao Xu and Jehoshua Bruck. 1999. X-code: MDS array codes with optimal encoding. *IEEE Trans. Inf. Theor.* 45, 1 (1999), 272–276.

[132] Eitan Yaakobi, Laura Grupp, Paul H. Siegel, Steven Swanson, and Jack K. Wolf. 2012. Characterization and error-correcting codes for TLC flash memories. In *International Conference on Computing, Networking and Communications (ICNC'12).* IEEE, 486–491.

[133] Gala Yadgar, Eitan Yaakobi, Fabio Margaglia, Yue Li, Alexander Yucovich, Nachum Bundak, Lior Gilon, Nir Yakovi, Assaf Schuster, and André Brinkmann. 2018. An analysis of flash page reuse with WOM codes. *ACM Trans. Stor.* 14, 1 (2018), 1–39.

[134] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-Tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST'17).* USENIX Association, 15–28. Retrieved from https://www.usenix.org/conference/fast17/technical-sessions/presentation/yan

[135] Jinfeng Yang, Bingzhe Li, and David J. Lilja. 2020. Exploring performance characteristics of the optane 3D Xpoint storage technology. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 5, 1 (2020), 1–28.

[136] Engling Yeo. 2012. An LDPC-enabled flash controller in 40 nm CMOS. In *Flash Memory Summit.*

[137] Shushu Yi, Yanning Yang, Yunxiao Tang, Zixuan Zhou, Junzhe Li, Chen Yue, Myoungsoo Jung, and Jie Zhang. 2022. ScalaRAID: Optimizing Linux software RAID system for next-generation storage. In *14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22).* Association for Computing Machinery, New York, NY, USA, 119–125. DOI: https://doi.org/10.1145/3538643.3539740

[138] Tianqi Zhan, Xianpeng Wang, Dan Feng, and Wei Tong. 2020. AetEC: Adaptive error-tolerant erasure coding scheme within SSDs. In *IEEE 38th International Conference on Computer Design (ICCD'20).* 167–174. DOI: https://doi.org/10.1109/ICCD50377.2020.00041

[139] Guangyan Zhang, Zican Huang, Xiaosong Ma, Songlin Yang, Zhufan Wang, and Weimin Zheng. 2018. RAID+: Deterministic and balanced data distribution for large disk enclosures. In *16th USENIX Conference on File and Storage Technologies (FAST'18).* 279–294.

[140] Yuqi Zhang, Wenwen Hao, Ben Niu, Kangkang Liu, Shuyang Wang, Na Liu, Xing He, Yongwong Gwon, and Chankyu Koh. 2023. Multi-view feature-based SSD failure prediction: What, when, and why. In *21st USENIX Conference on File and Storage Technologies (FAST'23).* USENIX Association, 409–424. Retrieved from https://www.usenix.org/conference/fast23/presentation/zhang

[141] Yiming Zhang, Huiba Li, Shengyun Liu, Jiawei Xu, and Guangtao Xue. 2020. PBS: An efficient erasure-coded block storage system based on speculative partial writes. *ACM Trans. Stor.* 16, 1, Article 6 (Feb. 2020), 25 pages. DOI : https://doi.org/10.1145/3365839

[142] Kai Zhao, Wenzhe Zhao, Hongbin Sun, Xiaodong Zhang, Nanning Zheng, and Tong Zhang. 2013. LDPC-in-SSD: Making advanced error correction codes work effectively in solid state drives. In *11th USENIX Conference on File and Storage Technologies (FAST'13)*. USENIX Association, 243–256. Retrieved from https://www.usenix.org/conference/fast13/technical-sessions/presentation/zhao

Received 15 November 2023; revised 13 August 2024; accepted 26 September 2024