Shuffler: A Novel Read Reclaim Scheme for Superblock Management in NAND Flash

Jianwei Liao, Zhibing Sha, Yuanbo Li, Jun Li, Zhigang Cai, François Trahay

▶ **To cite this version:**

Jianwei Liao, Zhibing Sha, Yuanbo Li, Jun Li, Zhigang Cai, et al.. Shuffler: A Novel Read Reclaim Scheme for Superblock Management in NAND Flash. Transactions on Storage, In press, ⟨10.1145/3799982⟩. ⟨hal-05553835⟩

HAL Id: hal-05553835

https://hal.science/hal-05553835v1

Submitted on 16 Mar 2026

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons CC BY 4.0 - Attribution - International License

Shuffler: A Novel Read Reclaim Scheme for Superblock Management in NAND Flash

JIANWEI LIAO, ZHIBING SHA, and YUANBO LI, Southwest University of China, China
JUN LI, Nanjing University of Posts and Telecommunications, China
ZHIGANG CAI, Southwest University of China, China
FRANÇOIS TRAHAY, Samovar, Télécom SudParis, Institut Polytechnique de Paris, France

Superblock management is widely used in commercial products of flash memory, in which multiple blocks from different parallel units are organized as a superblock, to provide high-level access parallelism. However, excessive parallelism inadvertently amplifies the negative effects of read reclaim (RR) due to the larger scale of data migrations and space reclamation. This is because a superblock-granularity RR operation must be triggered when any block inside a superblock reaches the RR threshold, even though other blocks are far from the threshold. To improve the RR efficiency for superblock management, this paper proposes a shuffling-based RR scheme for unifying read accesses across all component blocks inside the superblock without any additional space costs. Specifically, it shuffles the blocks of superblock with the granularity of **card** that consists of multiple data pages in the RR process and ensures no impact on access parallelism after shuffling. As a result, all component blocks may serve a similar number of read requests and approach the RR threshold synchronously. Trace-driven simulation experiments demonstrate that our proposed method can reduce the RR operations and read latency by 14.5% and 12.9%, compared with conventional superblock-granularity RR schemes in flash memory, while preserving access parallelism.

CCS Concepts: • **Information systems → Flash memory**; • **Hardware → External storage**.

Additional Key Words and Phrases: NAND-based Flash, Superblock Management, Read Reclaim, Shuffling, Write Amplification, I/O Performance.

1 Introduction

The escalating density of NAND flash memory, while effectively reducing the per-unit cost, exacerbates its susceptibility to intrinsic circuit-level noise phenomena, including retention disturb, program disturb, and read disturb [1, 2]. Among these, read disturb represents a particularly insidious anomaly wherein the act of reading data from a specific flash page inadvertently alters the threshold voltages of adjacent, unread pages within the same block. The cumulative effect of these perturbations can precipitate read-disturb errors, thereby compromising data integrity [3]. To mitigate the deleterious impact of read disturb and forestall data corruption, the read reclaim (RR) mechanism is employed as a prophylactic measure. This technique entails the systematic migration of valid data pages from the affected block to another new block, followed by the erasure of the disturbed block to rejuvenate its storage capacity and ensure continued reliability [4–6].

To optimize access parallelism in NAND flash memory, the concept of a superblock has been introduced, wherein multiple physical blocks are logically aggregated into a single, higher-level entity [7, 8]. Specifically, physical blocks sharing the **same ID** across distinct parallel units (e.g., planes or channels) are consolidated into a unified superblock. Within this structure, pages located at the **same offset** across the constituent blocks are collectively termed a superpage, which serves as the fundamental unit for programming operations [9, 10]. This superblock-based architectural design ensures that maximum parallelism is consistently harnessed during read/write operations. This is because aligned write points within the superblock can enable multi-plane (MP) commands

J. Li is the corresponding author of this paper, E-mail: lijun951111@gmail.com.

Authors' Contact Information: Jianwei Liao; Zhibing Sha; Yuanbo Li, Southwest University of China, Chongqing, China; Jun Li, Nanjing University of Posts and Telecommunications, Nanjing, China; Zhigang Cai, Southwest University of China, Chongqing, China; François Trahay, Samovar, Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, France.

to efficiently complete operations on the underlying flash array in parallel, indicating the number of falsh transactions can be greatly reduced. Consequently, it can deliver sustained high throughput and enhance overall performance [10, 11]. Owing to these advantages, the superblock management strategy has been widely adopted in modern commercial flash memory products, such as *Samsung PM9D3* [12] and *Micron 2400* [13], underscoring its significance in contemporary flash memory architectures.

The superblock serves as the fundamental unit for both garbage collection (GC) and read reclaim operations, enabling efficient space management and access parallelism [11, 14]. Specifically, when the number of read accesses to any individual block within a superblock exceeds a predefined read reclaim threshold (e.g., 10K reads [15]), a superblock-level read reclaim operation is initiated. During this process, all valid data pages within the affected superblock are relocated to a free superblock, irrespective of whether the majority of these pages remain unaffected by read disturb and are not yet at risk of data corruption. This conservative approach, while ensuring data reliability, results in the premature migration of undisturbed valid data pages and incurs unnecessary block erase operations. Such inefficiencies not only degrade I/O responsiveness, but also exacerbate the write amplification factor (WAF)[1], which is the ratio of data pages written to the flash memory including extra flash writes caused by RR and GC operations to data pages initially issued by the host [16]. This highlights a critical trade-off between data reliability and performance optimization in superblock-based flash memory management.

To enhance the efficiency of superblock-granularity read reclaim, we propose a novel shuffling-based read reclaim scheme, termed ***Shuffler***. This scheme aims to uniformly distribute read accesses across all component blocks within a superblock, thereby reducing the frequency of read reclaim (RR) operations without incurring additional space overhead. In summary, this paper makes the following three contributions:

- We propose shuffling across component blocks of superblock with the unit of **card** that contains multiple data pages, during the RR process. Specifically, it exchanges cards between different component blocks based solely on their block-level read counts. Consequently, the scheme ensures a more balanced distribution of read accesses across the superblock, thereby mitigating localized read disturb and reducing the frequency of RR operations.
- We build a mathematical model to direct full/partial shuffling in the RR process, without impacts on superblock access parallelism. Specifically, data pages in the component block are divided into multiple cards and then redistributed across blocks in the superblock according to the principle of *the Latin square* [19], to minimize access dissimilarity across blocks without compromising the inherent access parallelism.
- We perform evaluation tests by replaying commonly used I/O traces of real-world applications. The results demonstrate our method can noticeably reduce RR operations and write amplification factors by 14.5% and 5.8% respectively, in contrast to conventional superblock-granularity RR schemes.

The rest of the paper is organized as follows: the background and motivations are shown in Section 2. The specifications of shuffling-based read reclaim are presented in Section 3. Section 4 depicts evaluation experiments and relevant discussions. Finally, we conclude the paper in Section 5.

[1]Write amplification is the nature of flash memory, which consumes extra program/erase cycles and then speeds up device wear-out [17, 18].

2 Background and Motivation

2.1 Background Knowledge and Related Work

NAND flash-based SSDs have emerged as a high-performance alternative to HDDs, dominating the storage market due to their speed and reliability [6]. A key feature of SSDs is their multi-level parallelism, which is achieved through a hierarchical structure: multiple channels, each connecting to several chips, which are further divided into dies and planes, and each plane contains hundreds of blocks [20]. To maximize this parallelism, SSD manufacturers aggregate blocks with the same block ID across multiple planes into a superblock, to boost performance. A superblock consists of multiple physical blocks sharing the same block ID across planes, ensuring all constituent blocks operate in unison. This requires that all blocks within a superblock remain in the same state (e.g., free or allocated) at any given time, necessitating data management schemes at the superblock level. In summary, superpages can enhance I/O throughput for read/write operations, but they degrade garbage collection and wear-leveling efficiency due to coarse-grained erase management [14].

Emerging research continues to explore optimizations such as superblock-level garbage collection and read reclaim, further enhancing the efficiency and reliability of superblock-based systems. To reclaim space, flash drives perform garbage collection at the granularity of superblock. Assuming superblock *A* is selected as the victim for garbage collection, all valid data pages within superblock *A* will be migrated to superblock *B*. Following the migration, the entire superblock *A* is erased, freeing it for future use [10]. As a result, superblock management inevitably leads to significant write amplification due to GC operations, which degrades both I/O performance and flash memory endurance [21]. To address the aforementioned issues, many novel garbage collection designs [22] and wear-leveling techniques [8, 14] have been proposed for superblock management. Considering flash blocks have different performances and characteristics because of process variation, Tseng et al. [10] have proposed organizing the blocks with strongly similar characteristics that are introduced by process-variation as one superblock. Similarly, Wang et al. [14] proposed to dynamically organize superblocks according to real-time wear levels of blocks, for maximizing access parallelism.

The RR strategy is proposed to avoid permanent data corruption caused by read disturb, through migrating data pages from affected blocks to other free blocks [23]. In the context of superblock management, SSDs generally trigger superblock-level read reclaim while the read count of any component block of the superblock approaches the RR threshold, even though that of other component blocks is still far from the threshold. In the RR process, all valid data pages within the superblock are indiscriminately migrated to other new superblocks. This approach leads to the same RR cases in the future, which in turn exacerbate write amplification, negatively impacting the overall performance and endurance of the flash memory system [10, 24]. Tracking read counts at the page level can help unify the distribution of read accesses during the RR process, thus reducing the number of RR operations caused by uneven read distribution across component blocks inside superblock. However, page-level read counters would inevitably require a significant amount of memory space, which could compete with data cache resources and negatively impact system performance [25].

We argue that, however, no existing RR schemes could well address the issue of unifying read accesses over component blocks inside the superblock without additional space overhead. Furthermore, existing schemes are insufficient in mitigating write amplification induced by RR operations in flash memory systems that employ superblock-level granularity for space management.

2.2 Motivations

We conducted an experimental study to compare block-based management and superblock-based management. Specifically, we utilized an open-source SSD simulator to replay several block I/O

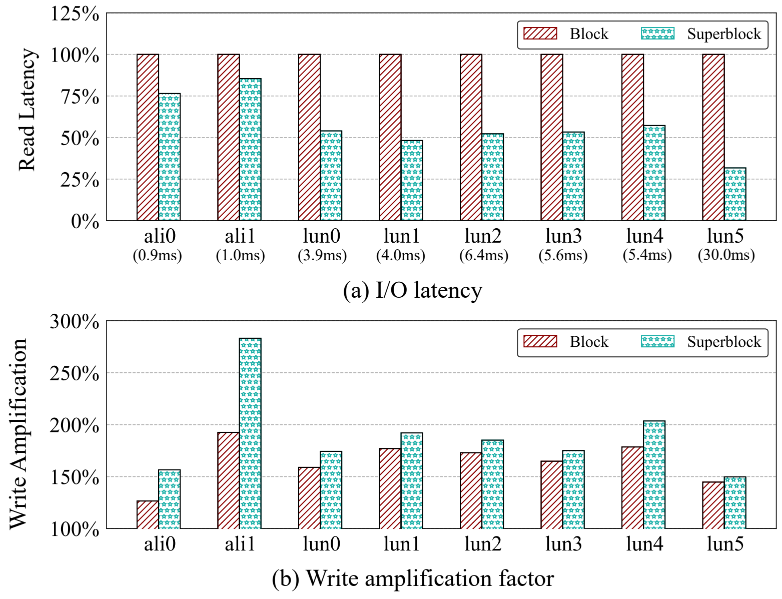

Fig. 1. Comparison of read latency and write amplification factor with block and superblock management for flash memory. All benchmarks are either read-intensive or small write footprint, and no GC operations were triggered, meaning only extra writes induced by RRs cause write amplification.

traces selected from two recent and widely adopted collections. The detailed experimental settings are described in Section 4.1. Figure 1(a) shows the comparison results of the I/O latency, and Figure 1(b) write amplification factor (WAF) while using two management approaches, where the value of 100% corresponds to the actual value of 1.0. As seen, superblock management significantly improves read responsiveness, outperforming block management by more than 14.5%. This enhancement is attributed to its superior utilization of access parallelism across multiple parallel units. However, superblock management also leads to a substantial increase in the write amplification factor, with WAF rising by up to 14.9% compared to block management. This confirms that superblock-granularity read reclaim operations contribute to higher write amplification, which in turn accelerates flash memory wear and reduces its overall endurance.

Furthermore, we conducted an in-depth analysis of read count variations across component blocks during superblock-granularity read reclaim (RR) operations. Figure 2 illustrates the concentration range and median values of block read counts within superblocks subjected to RR operations. The results reveal that the read counts for a significant portion of component blocks remain far below the RR threshold of 10K reads, as specified for triple-level cell (TLC) flash memory [15]. This phenomenon occurs because the superblock-granularity RR process is triggered as soon as any single component block approaches the RR threshold, regardless of whether the majority of blocks within the superblock have served only a minimal number of read accesses. Consequently, many blocks with low read counts are unnecessarily included in the RR process, leading to premature data migration and inefficient resource utilization.

These observations motivated us to design a novel read reclaim scheme tailored for superblock management in flash memory, aiming to uniformly distribute read accesses across all component blocks within a superblock, thereby reducing the frequency of RR operations. After achieving a

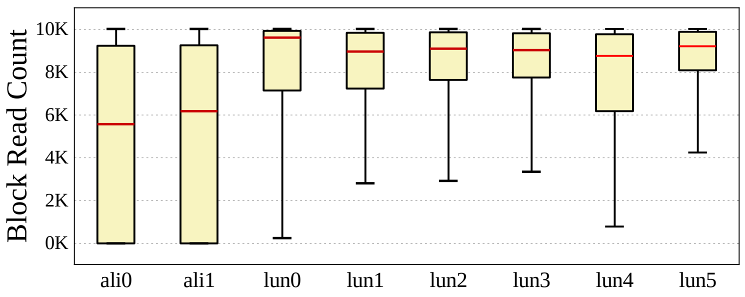

Fig. 2. Read count variations over component blocks when carrying out superblock-granularity RR operations, and the block read count to trigger the RR operation is set as 10, 000 [15]. In which, the yellow box represents the interquartile range between the 25th and 75th percentiles of read counts in RR blocks, with the red line marking the median read count per benchmark execution.

more balanced read distribution, the write amplification factor caused by RR operation can be minimized as well when running user applications.

3 Shuffling-based Read Reclaim

3.1 Architectural Overview

To address the issue of block read count dissimilarity within superblocks, we propose *Shuffler*, a novel scheme that performs data page shuffling during the read reclaim process. Figure 3 illustrates the main steps of *Shuffler*. As illustrated, the scheme initiates by evaluating read access imbalance across a superblock's component blocks, quantified as Δ. Based on the severity of this imbalance, *Shuffler* dynamically selects between full shuffling (the upper case in the figure) and partial shuffling (the lower case in the figure). Specifically, full shuffling is triggered when the read distribution across component blocks is extremely uneven, involving all component blocks in the exchange of data pages to achieve a balanced read distribution. On the other hand, partial shuffling is applied if the read access imbalance is moderate, targeting only a subset of component blocks in the superblock for data page exchanges.

We emphasize that *Shuffler* is meticulously designed to preserve access parallelism even after shuffling data pages. This is achieved by maintaining the superpage-level organization of data, ensuring that the superpage structure remains consistent throughout the read reclaim (RR) process. To accomplish this, we introduce a card-granularity *Latin* square mechanism to guide the shuffling of data pages within the superblock. In this approach, data pages within each block are divided into multiple cards, with the number of cards corresponding to the number of parallel units involved in the shuffling process. The *Latin* square framework can guarantee that each card will be redistributed only within the same column, thus preserving the alignment of superpages and maintaining access parallelism.

Through leveraging the *Latin* square, *Shuffler* can not only balance read accesses across component blocks to reduce the frequency of read reclaim operations, but also preserve the inherent parallelism of the superblock architecture. This dual focus ensures that the system maintains high data throughput and low latency while minimizing write amplification caused by RR operations. As a result, *Shuffler* emerges as a robust and efficient solution for optimizing read reclaim in flash memory systems that adopt superblock management, addressing both performance and endurance challenges.

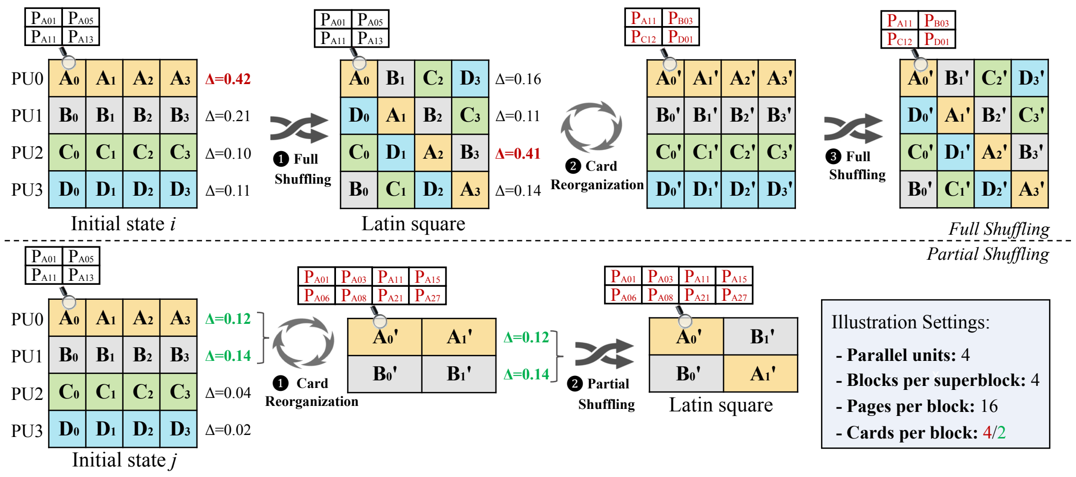

Fig. 3. Overview of *Shuffler* that supports card shuffling. The superblock has 4 blocks of A, B, C, D on their parallel units, and each treatment of card (e.g., A_0, which consists of 4 data pages of P_{A01}, P_{A05}, P_{A11}, and P_{A13}) occurs only once in each row and each column after shuffling according to the principle of the *Latin* square (see §3.2). The symbol of Δ means the read access imbalance level of the component block, which is the base of shuffling routine selection (see §3.3 and §3.4).

3.2 Latin Square Shuffling

Assuming a superblock consists of n blocks, and each block B_i has m pages. Let PR_{ij} denote the read count of page j in B_i, BR_i represent the total read count of block B_i, and σ_i^2 indicate the variance of read counts among its m pages. Without access to individual PR_{ij} values and given only the block-level BR_i for the n blocks in the superblock, we assume that the PR_{ij} values within each block are independent and identically distributed (i.i.d.). Thus, PR_{ij} can be defined as a random variable with:

$$E(PR_{ij}) = BR_i/m \tag{1}$$

$$D(PR_{ij}) = \sigma_i^2/m \tag{2}$$

Specifically, *Shuffler* aims to balance read counts across blocks by leveraging opportunities during migrating disturbed pages from block B_i to block B_j. To achieve this, during data page shuffling, the balancing scheme must satisfy the following constraints:

$$\begin{cases} \sum_i x_{ij} \cdot \dfrac{BR_i}{m} = \overline{BR} \quad j = 1, 2, \ldots, n & (3) \\ \sum_i x_{ij} \cdot \dfrac{\sigma_i^2}{m} = \overline{\sigma^2} \quad j = 1, 2, \ldots, n & (4) \\ \sum_j x_{ij} = m \quad i = 1, 2, \ldots, n & (5) \\ \sum_i x_{ij} = m \quad j = 1, 2, \ldots, n & (6) \end{cases}$$

where $\overline{BR} = (\sum_{i=1}^{n} BR_i)/n$, $\overline{\sigma^2} = (\sum_{i=1}^{n} \sigma_i^2)/n$, and x_{ij} is the number of pages migrated from B_i to B_j. Then, the expected total read count of these x_{ij} pages could be represented as $(BR_i/m) \cdot x_{ij}$, with variance $(\sigma_i^2/m) \cdot x_{ij}$.

If both BR_i and σ_i^2 are known, a specific adjustment strategy x_{ij} can be derived by solving the constraints. However, since each block only provides an aggregate read counter BR_i without page-level access counts, σ_i^2 remains unknown, rendering the constraints unsolvable. Fortunately, for any σ_i^2, $x_{ij} = m/n$ satisfies the system of equations. Thus, under the constraint of unavailable page-level read access information, a uniformly randomized adjustment scheme of *Latin* square is an equilibrium strategy in expectation. This serves as the theoretical foundation for the validity of *Latin* squares in such shuffling scenarios.

3.3 Read Imbalance Estimation and Card Shuffling

This section describes **how to** estimate read access imbalance levels across component blocks within a superblock and introduces a systematic approach to quantify read access disparities inside the superblock. It then explains **how to** organize and shuffle cards using a card-granularity framework, ensuring uniform read distribution while preserving access parallelism.

Assessing read imbalance. A superblock consists of *n* flash blocks distributed across parallel units. As defined, BR_i indicates the total read count of the i-th block. Equation 7 then defines an indicator to reflect the imbalance degree of read accesses over blocks inside the superblock.

$$\Delta = \frac{|BR_i - \bar{BR}|}{\bar{BR}} \tag{7}$$

where $\bar{BR}$ is the average block read count in the superblock.

Each component block within the superblock is assigned a unique value of Δ, which quantifies the imbalance level of read accesses of the superblock. Then, we employ this metric to determine the appropriate shuffling policy, full shuffling for highly uneven distributions, partial shuffling for moderate imbalances, or no shuffling when read accesses are balanced. This adaptive approach ensures efficient data redistribution in the RR process, for optimizing I/O performance and endurance of SSD devices adopting superblock management.

Shuffling principles and card organization. *Shuffler* performs data page shuffling with the unit of card, to unify read accesses across all component blocks during superblock-granularity read reclaim. Following the principle of *Latin* square, we first formulate card-shuffling rules to ensure the preservation of access parallelism. Let C_{ij} denote the j-th card from the i-th block at the i-th parallel unit, and the set of pages within C_{ij} is represented by:

$$C_{ij} = \{p_{ij_0}, p_{ij_1}, \ldots, p_{ij_{\frac{m}{n}-1}}\} \tag{8}$$

where m is total pages in a block, and n is the number of parallel units involved in shuffling.

Then, the updated card after shuffling can be denoted as $C_{i'j'}$, which should satisfy:

$$\begin{cases} i' = i - j \bmod n \\ j' = j \end{cases} \tag{9}$$

In summary, *Shuffler* triggers full shuffling if all component blocks in the superblock exhibit significant read workload imbalances, or partial shuffling if only some blocks are imbalanced. This decision is based on whether their Δ values, calculated using Equation 7, exceed predefined thresholds. If no imbalance is detected, the default read reclaim routine is applied, which does not reallocating data pages across blocks. Before initiating a new shuffling round, data pages within

Algorithm 1: Selection of shuffling routines

```
Input: $BR_i$ (i=0,1,...,n-1) of superblock $sb$, $\Delta_F$, $\Delta_P$.
Output: full/partial/no shuffling is done.
Function main()
    /*estimate block read imbalance*/
    $delta\_set$ = cal_blk_delta($BR_0$, ..., $BR_{n-1}$);
    $\Delta_{max}$ =get_max_delta($delta\_set$);
    /*carry out full shuffling on the superblock of sb*/
    if $\Delta_{max} \geq \Delta_F$ then
        /*form cards spanning on n parallel units*/
        form_card($sb$, $n$);
        full_shuffling($sb$);
        return;
    else
        $partial\_set$ ={};
        /*generate the partial shuffling set*/
        for $blk$ in $sb$ do
            if $delta\_set$[get_idx($blk$)] $\geq \Delta_P$ then
                add_to_set($partial\_set$, $blk$);
    /*partial shuffling or no shuffling*/
    if get_size($partial\_set$) >1 then
        form_card($partial\_set$, size($partial\_set$));
        partial_shuffling($partial\_set$);
    return
```

each block are reorganized into multiple cards, corresponding to the number of involved parallel units. Each card is constructed by randomly grouping data pages within the block, ensuring that all component blocks follow the same randomization rule. This preserves the superpage structure and maintains access parallelism during card reorganization.

3.4 Selection of Shuffling Routines

To direct the selection of shuffling routine, we measure the read imbalance status of the superblock by comparing all Δ values of component blocks with two predefined thresholds of Δ_F and Δ_P (with $\Delta_F > \Delta_P$). Then, the proper shuffling routine will be used for the current superblock-granularity RR operation, and Algorithm 1 illustrates the specifications. Note that the two pre-specified thresholds remain constant regardless of workload variations and are used to measure the imbalance level of read accesses among component blocks within the superblock, triggering data page shuffling when necessary. While accommodating workload-induced variations in shuffling frequency (the number of RR operations) and scope (the blocks invovled in the RR operation), the imbalance detection thresholds that trigger data shuffling remain fixed regardless of application characteristics.

As seen, it first calculates all Δ values for all component blocks, and obtains the maximum one labeled as Δ_{max}. Then, the full shuffling routine will be employed if Δ_{max} is greater than Δ_F, since it considers the read accesses across all component blocks are greatly uneven (Lines 2-10). If the condition is not satisfied, the component blocks will be included in the candidate set of

Table 1. Experimental settings of the *MQSim* simulator

Parameters	**Values**	**Parameters**	**Values**
Channel count	8	*Read latency*	0.038ms
Chip count	4	*Write latency*	0.35ms
Plane count	4	*Erase latency*	3.5ms
Block per plane	64	*GC threshold*	10%
Page per block	1024	*RR threshold*	10K
Page size	8KB	*FTL scheme*	superpage
DRAM data cache	64MB	*Superblock size*	4 blocks

partial shuffling, if their Δ value is larger than Δ_P (Lines 12-16). As a result, the partial shuffling routine will be used if the candidate set consists of more than one block, as shown in Lines 18-20. Otherwise, it will select the routine of no shuffling, indicating a conventional superblock-granularity RR operation will be performed.

4 Experiments and Discussions

4.1 Experimental Settings

We conducted trace-driven experiments using the *MQSim* [26] simulator to evaluate the proposed *Shuffler* scheme, utilizing its support for diverse configurations and its validation accuracy against real hardware platforms. We simulated a 64GB flash memory with superblock management at parallel units of planes, and configured its performance parameters by referring to [3, 15, 27, 28]. Table 1 shows the values of parameters in evaluation. The GC threshold defines the trigger condition for garbage collection operations, which are activated when the amount of available space falls below the predefined threshold [11]. We construct the superblock from 4 blocks distributed within one chip, each containing 4 planes. A larger superblock configuration with 32 blocks will be also evaluated in Section 4.4.1.

We selected **10** block I/O traces from recently published and widely-adopted collections, because their heterogeneous workloads robustly demonstrate our mechanism's adaptability across diverse I/O patterns. Among them, two traces are from the Alibaba center, including *alibaba_112*, and *alibaba_121* (labeled as *ali0* and *ali1*) [31]. Six I/O traces are from a real-world VDI application [32]. Specifically, these traces are **additional-01-***2016021616-LUN2*, *2016021619-LUN2*, *2016021619-LUN3*, *2016021620-LUN2*, *2016021620-LUN3*, and *2016021716-LUN0* (labeled as *lun0-lun5*). Additionally, we generated two synthetic workloads using *MQSim* (labeled as *gen0* and *gen1*), to purposely conduct hottest-read tests for triggering more RR operations and then verifying the effectiveness of *Shuffler*. The specifications on used traces in our evaluation are reported in Table 2. As read, the metric of **Rd R** means the read ratio, **Rd Size** indicates the average size of read requests, **Hot R** represents the ratio of data pages that have been read repeatedly to all data pages, and **FP** implies the total size of read and write footprint. In order to trigger more RR operations, we repeated the benchmarks 128 times.

Three comparison counterparts are used in our evaluation.

- ***Baseline***, which is the conventional read reclaim scheme for superblock management performs RR operations at the superblock granularity. It triggers a read reclaim process whenever any block within the superblock approaches the predefined RR threshold, regardless of the read workloads of other blocks.

Table 2. Specifications on the selected traces

Trace	Req #	Rd R	Rd Size	Hot R	FP
ali0	1,890,017	67.1%	19.8KB	59.9%	1.5GB
ali1	2,340,016	77.2%	20.4KB	70.5%	1.1GB
lun0	1,062,359	66.6%	20.3KB	27.7%	13.9GB
lun1	1,644,153	76.8%	16.1KB	38.2%	19.4GB
lun2	1,133,348	67.6%	25.7KB	28.1%	17.7GB
lun3	1,147,164	69.2%	23.5KB	29.3%	17.9GB
lun4	1,252,039	73.3%	23.2KB	33.6%	14.5GB
lun5	1,403,521	59.9%	31.6KB	27.0%	25.7GB
gen0	2,000,000	85.0%	16.2KB	64.9%	3.2GB
gen1	2,000,000	90.0%	16.0KB	49.0%	6.4GB

- ***Page***, which utilizes page-level read counts to track the read hotness of data pages in blocks [29]. It then groups data pages into component blocks based on their historical read hotness, aiming to balance read accesses across all blocks in future time windows. However, since each block contains hundreds of data pages, it incurs significant space overhead for maintaining page-level metadata. This overhead competes for memory space within data cache inside SSDs, reducing the available data cache capacity and then potentially degrading cache hits and I/O performance.
  Moreover, we aggregate the read counters of multiple consecutive logical pages into a single counter. These page group-based read counters are utilized to guide page migration during shuffling. Compared to the native *Page* approach, it can significantly reduce the space overhead required for read counters. We have implemented group sizes of 8 and 64 pages, labeled as *Page-8* and *Page-64*, respectively.
- ***Shuffler***, which is the card-granularity shuffling mechanism for read reclaim in superblock management. It redistributes data pages across component blocks using cards as the basic unit, based on their block-level read counts during the RR process. Thus, it can effectively unify read accesses across all blocks, reducing read imbalances while maintaining access parallelism and minimizing unnecessary data migrations.
  As discussed, *Shuffler* employs two thresholds, Δ_P and Δ_F, to determine whether partial shuffling or full shuffling should be triggered. Based on the results of our sensitivity studies (refer to Section 4.2), we set Δ_F to 0.30 and Δ_P to 0.10 by default.

4.2 Sensitivity of Parameters

This section carries out sensitivity tests to guide the configuration of two thresholds, Δ_P and Δ_F. Full shuffling is triggered when the read imbalance exceeds Δ_F while partial shuffling is triggered when it lies between Δ_P and Δ_F. If the thresholds are set too small, card shuffling may be triggered frequently, but random shuffling cannot always guarantee a further reduction in read imbalance. On the contrary, if the thresholds are set too large, fewer read reclaim (RR) operations may be triggered, which could delay the balancing of read workloads across the superblock.

Figure 4 shows the normalized results of RR counts after replaying the selected benchmarks with the proposed *Shuffler* scheme. On the one hand, smaller values for two thresholds tend to trigger data shuffling in RR processes but do not necessarily result in optimal I/O performance,

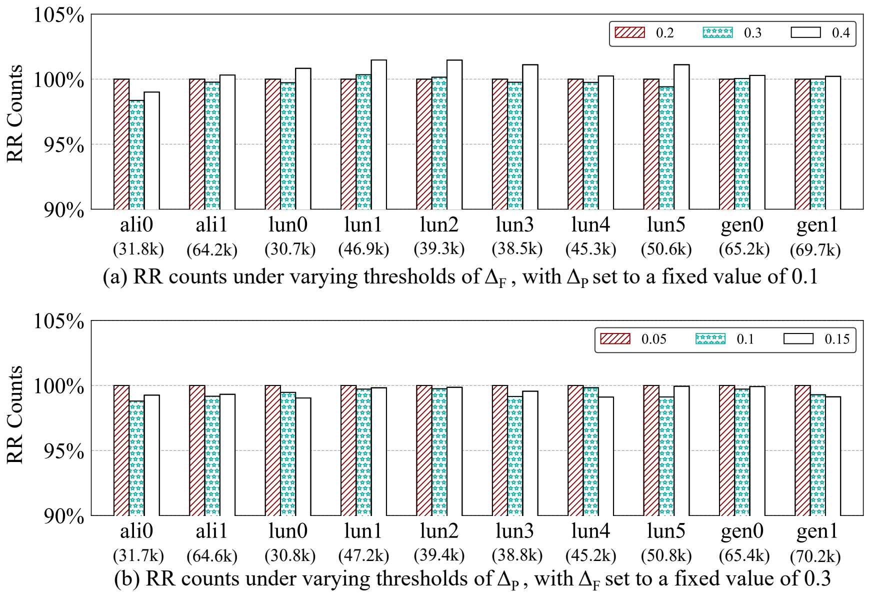

Fig. 4. The sensitivity analysis on parameters of Δ_F and Δ_P used for triggering full and partial shuffling in our proposal of *Shuffler*. In which, the legend items in subfigures (a) and (b) represent the different settings of Δ_F and Δ_P, respectively. Note that the results are normalized to the first configuration, and absolute values of the first configuration are shown below trace names.

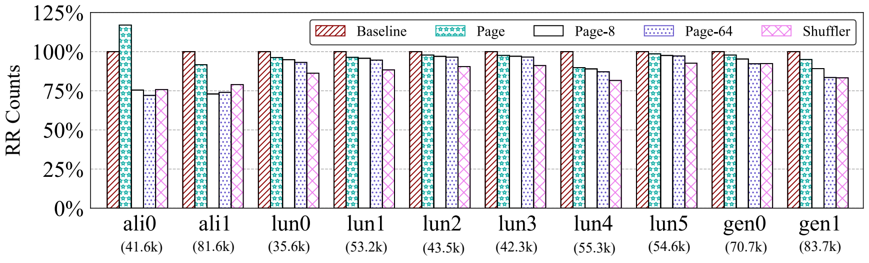

Fig. 5. Comparison of superblock-granularity RR operations after replaying the selected benchmarks. Note that the results are normalized to *Baseline*, and absolute *Baseline* values are shown below trace names.

as frequent shuffling cannot always guarantee an optimal balance in read distribution across the superblock. On the other hand, larger thresholds imply data shuffling will be triggered only if the read imbalance across component blocks become more obvious, which means it cannot promptly ensure the balance of read distribution across all blocks within the superblock. Based on sensitivity analysis, we recommend $\Delta_F = 0.3$ and $\Delta_P = 0.1$ as defaults since they can yield the least number of RR operations across experimental scenarios.

4.3 Performance Results and Discussions

To measure the validity of our proposal, we use the following metrics in our experiments: (a) *number of read reclaim*, (b) *read balance*, (c) *write amplification factor*, and (d) *I/O latency*.

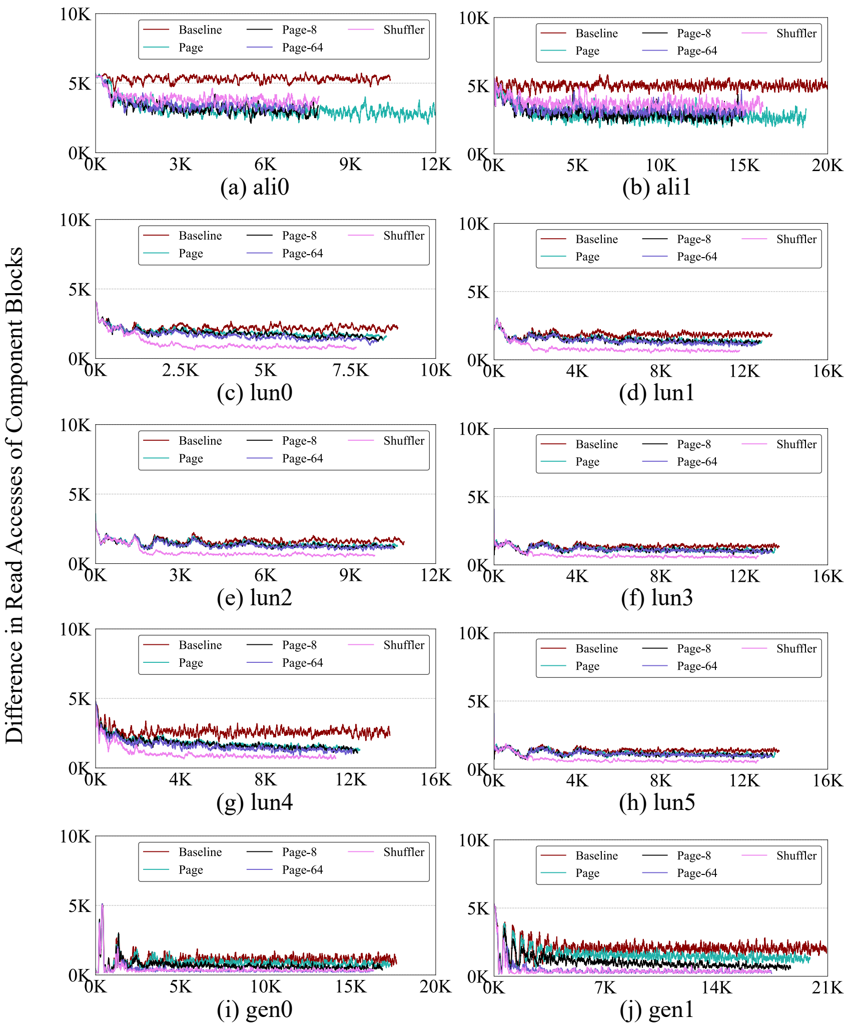

Fig. 6. The relationship between the convergence of read imbalance across component blocks and the number of RRs. In which, the Y-axis represents the difference in read access counts among component blocks within each superblock, and the X-axis shows the number of RR operations. Note that *Page*, *Page-8*, *Page-64* and *Shuffler* generally result in fewer RRs after replaying the selected benchmarks, in contrast to *Baseline*.

4.3.1 Read Reclaim Analysis. The primary objective of *Shuffler* is to minimize the number of RR operations. As illustrated in Figure 5, the results demonstrate the number of superblock-granularity RR operations after running the selected benchmarks. As observed, *Shuffler* and *Page* can decrease the number of RR operations by 14.5% and 7.8%, in contrast to *Baseline*. This fact verifies that exchanging data pages among component blocks of the superblock can effectively unify read distribution, thus reducing the number of RRs when running user applications.

It is worth mentioning that the *Page* scheme outperforms *Baseline* in most cases, but performs worse when running the *ali0* benchmark. This is because the *Page* method maintains page-level counters to track read hotness of data pages for directing page reallocation in the RR processes,

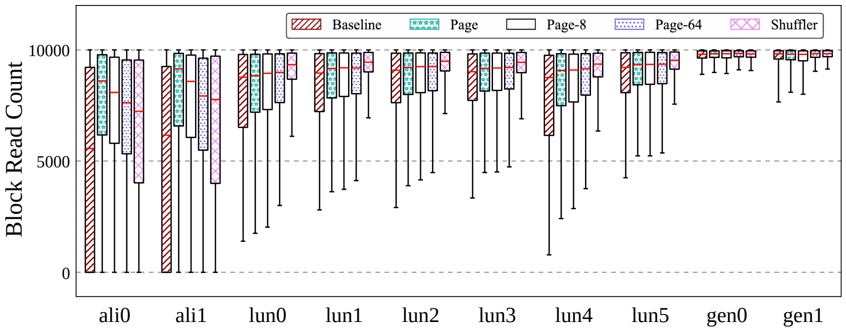

Fig. 7. Read count consistency in the read reclaimed superblocks after replaying the benchmarks with three read reclaim schemes.

which must reduce effective data cache capacity, thus degrading cache read hits[2] and more read accesses on the flash memory. Because *Page-8* and *Page-64* minimize the memory overhead for tracking read accesses, more cache capacity becomes available for absorbing read requests, thereby reducing the frequency of RR operations compared to the *Page* scheme. However, their group-level recording mechanism captures only coarse access patterns, which compromises tracking accuracy, as all pages within a group share a single access counter. In contrast, our proposed Shuffler completely eliminates the need for such read counters, thus introducing no cache space overhead. Although a card has a larger granularity, it is formed by randomly grouping pages within a block. This random composition allows our card-based shuffling to fundamentally perform page-level reorganization. Consequently, it offers finer-grained flexibility compared to the coarser *Page-8* and *Page-64* schemes, enabling unevenly accessed pages to be effectively distributed across the superblock over several shuffling rounds.

Another notable observation is that *Shuffler* performs slightly worse than *Page-8* and *Page-64* on the two *ali* traces. These workloads have small footprints of ~1GB and relative large hot ratios (see Table 2), where grouped read counters maintain sufficient cache efficiency and access tracking accuracy. But, we emphasize that *Shuffler* can perform the best across other workloads. This fact confirms our method efficiently unifies read distribution with zero memory overhead, delivering better results especially under changing access patterns in large footprint scenarios.

4.3.2 Read Balance in Superblocks. We depict the relationship between the convergence of read imbalance across all component blocks and the number of RR operations, to represent the level of read balance in read reclaimed superblocks. As the results shown in Figure 6, *Shuffler* can gradually converge the difference in the number of read requests on the component blocks of superblock, as the round of RR operations increases. Unlike *Baseline*, which does not exchange data pages during the RR process thus cannot change read differences across component blocks in the superblock, *Page* and two *Page*-group methods can decrease the read difference inside the superblock through page reallocation over component blocks to some extent.

In order to disclose the read access consistency in read reclaimed superblocks, we collect the range and the average of block read counts in the superblocks, and present the results in Figure 7. We can see that the read count of most component blocks is close to the RR threshold of `10K` while using *Shuffler*, implying better unified read workload distribution across blocks. In addition, it demonstrates that page-level or group-level read counters cannot always guarantee optimal

[2]See Section 4.3.4 for more details about the results of I/O latency.

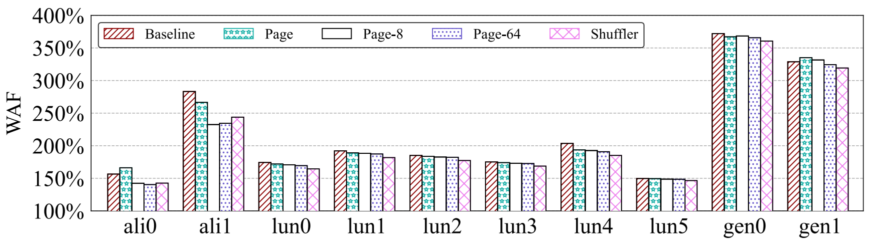

Fig. 8. Comparison of write amplification factors after replaying the selected benchmarks, with varied RR schemes.

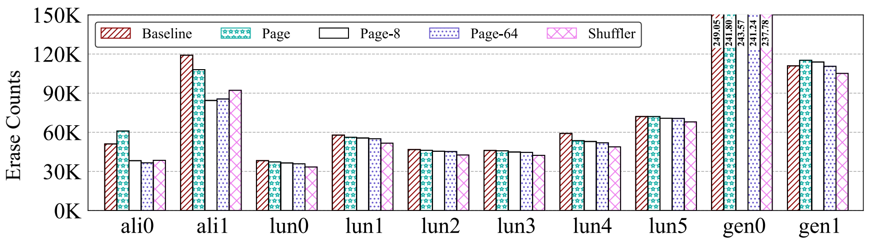

Fig. 9. Comparison of the lifetime of flash memory in terms of the number of block erases after replaying the selected benchmarks.

read balance after data reallocation in RR processes, because historical read patterns may not consistently reflect future workload distributions.

4.3.3 Write Amplification caused by RR. The lifetime of flash memory is limited by the number of program/erase cycles (P/Es), and a large write amplification factor means more page programs that will consume available P/Es and then degrade the endurance. Figure 8 presents the experimental results of write amplification induced by read reclaim operations after replaying the selected benchmarks. As shown, *Shuffler* demonstrates a significant reduction in the write amplification factor, achieving up to a 13.9% improvement compared to both *Baseline* and *Page*. This enhancement can be attributed to *Shuffler*'s ability to uniformly distribute read operations across component blocks. Consequently, it can effectively minimize the number of page migrations triggered by superblock-granularity RR operations, thus substantially mitigating write amplification.

In addition, we record the number of block erases that is the indicator of lifetime, and present relevant results in Figure 9. Clearly, it shows a similar tendency to the results of write amplification, since a large write amplification factor corresponds to more write data that will lead to more erase operations on the flash memory. This fact further proves that *Shuffler* can achieve the best lifetime of flash memory in all cases, because of the minimized write amplification.

4.3.4 I/O Latency. The I/O latency is the critical performance measurement in the context of flash memory. We recorded the average read/write latency after running the selected benchmarks, computed as the total read/write time divided by the total number of read/write requests. Figures 10 and 11 present read and write latencies separately, normalized to *Baseline*, to isolate operation-specific performance improvements. As shown, our *Shuffler* method achieves the best read responsiveness, reducing the read latency by 12.9%, 11.9%, 5.9%, and 4.5%, in contrast to *Baseline*, *Page*, *Page-8*, and *Page-64* respectively. This is because *Shuffler* can efficiently reduce the number of superblock-granularity RR operations and fewer RR-related I/O requests are generated, thus their interference

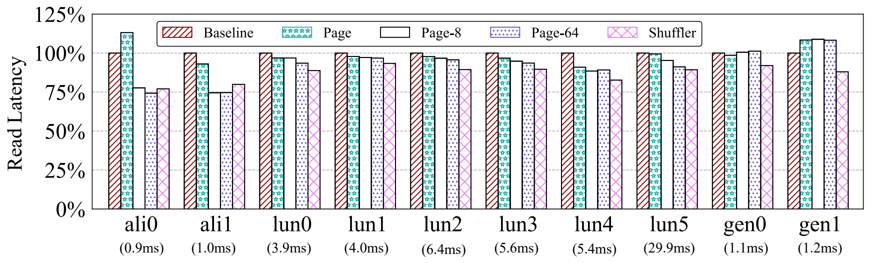

Fig. 10. Comparison of the average read latency after replaying selected benchmarks across three different RR schemes.

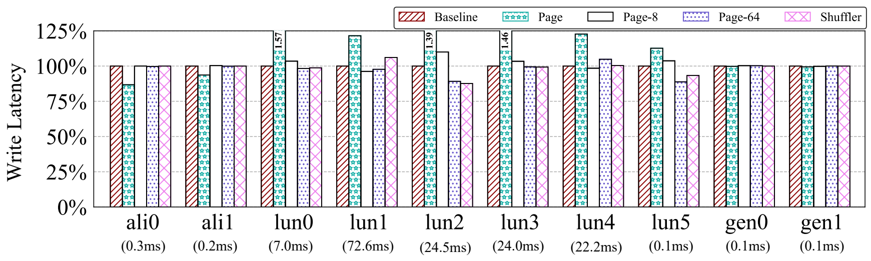

Fig. 11. Comparison of the average write latency after replaying selected benchmarks across three different RR schemes.

with user I/O requests can be minimized. Moreover, our proposal supports shuffling data pages in the card granularity in the RR process, which can help preserving access parallelism, as the superpage structure remains intact throughout the process.

On the other hand, all the methods generally reveal similar results of write latency. This is because all the optimization methods targets at RR operations, highly related to read latency. Another interesting clue is that the *Page* scheme performs the worst in most cases on the measure of the average write latency, though it can also yield a balanced read distribution across blocks in the superblock (see Section 4.3.2). We argue that *Page* requires maintaining page-level read counters to direct page reallocation in the RR processes, which must compete the cache space with user data, thus impacting cache use efficiency and I/O performance. Figure 12 presents the results of cache hit ratio for write and read requests after replaying the selected benchmarks. Clearly, *Baseline* and *Shuffler* achieve the same cache hit ratio, while *Page* and *Page-8* bring about less cache hits by 24.5% and 3.1%, when comparing to *Shuffler*, since they all require substantial cache space to hold page- or group-level counters. It is also worth noting that, the results of read cache hit ratios are relatively low in *lun* traces, since these traces unveil large access footprints and low hot ratios (see Table 2) Additionally, both synthetic workloads (i.e., *gen0* and *gen1*) follow random access patterns, so that the cache cannot effectively absorb incoming requests, resulting in consistently low hit ratios for both read and write requests in these two traces.

The long-tail latency is another indicator to reflect the quality of I/O service in the worst scenario. In general, a small value of long-tail latency can ensure better user experience of I/O service. Then, we collected the long-tail latencies of all methods. Figure 13 presents the results, where the x-axis represents I/O response time and the y-axis shows the Cumulative Distribution Function (CDF) for the slowest 1.0% of I/O requests. As seen, *Shuffler* outperforms other schemes in terms of

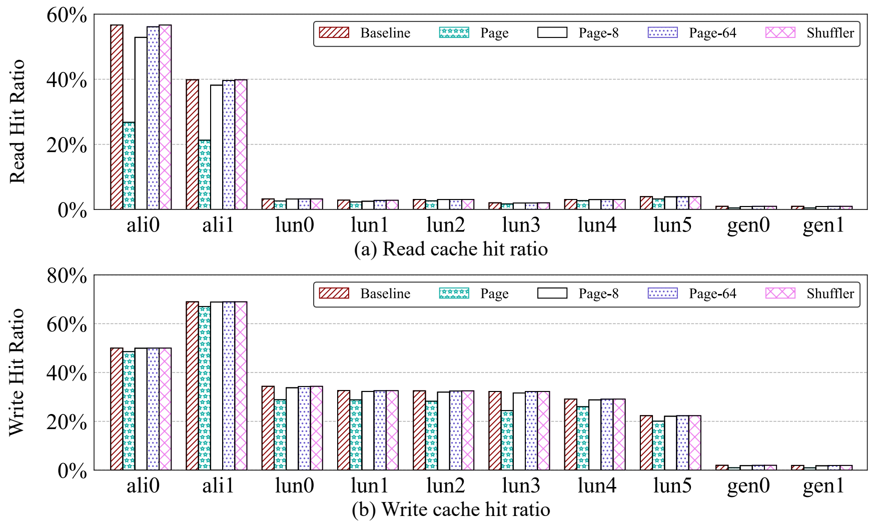

Fig. 12. Comparison of cache hit ratios. Note that *Baseline* and *Shuffler* yield the same ratio of cache hits, since they have the same size of data cache space and employ the LRU policy for cache management.

long-tail latency, and it can cut down the long-tail latency by 17.3%, 9.3%, 8.5% and 7.3% at the 99.99th percentile in contrast to *Baseline*, *Page*, *Page-8*, and *Page-64*, respectively. We attribute this improvement primarily to balanced read distribution achieved through data page shuffling during RR operations, which eliminates access contention, and secondarily to the removal of access counters that avoids the memory overhead inherent in *Page*, *Page-8*, and *Page-64* implementations.

4.3.5 Analysis on Access Parallelism. To evaluate access parallelism under different RR schemes, we introduce flash transaction count as a metric. Superblock architectures exploit SSD internal parallelism through multi-plane (MP) operations [10], where bundled reads/writes share a single transaction on the underlying flash array. In other words, a larger transaction count caused by a specific RR scheme directly reflects the less number of MP operations and the degraded access parallelism, when replaying the selected benchmark.

Figure 19 presents the normalized results of flash transactions after replaying the selected benchmarks, capturing only I/O request-induced transactions. As seen, *Page* performs the worst, and this is because the *Page* scheme distributes data pages across blocks within a superblock based solely on historical read patterns without considering the underlying MP organization. Consequently, this strategy severely restricts opportunities for parallel access through MP operations.

More importantly, our *Shuffler* mechanism achieves flash transaction parity with *Baseline*. Though absolute gains prove modest against *Page* and its variants, this parity confirms card-based shuffling can group data pages to match the underlying MP architecture. In other words, *Shuffler* can leverage MP operations at *Baseline*'s efficiency level, maintaining equivalent access parallelism despite the architectural redesign that reorganizes data pages across superblocks in RR processes.

4.4 Case Studies

4.4.1 Case Study on Scalability. To evaluate the impact of parallelism, we have conducted a case study under a large 32-superblock configuration, where all 4 planes in one chip across all 8 channels

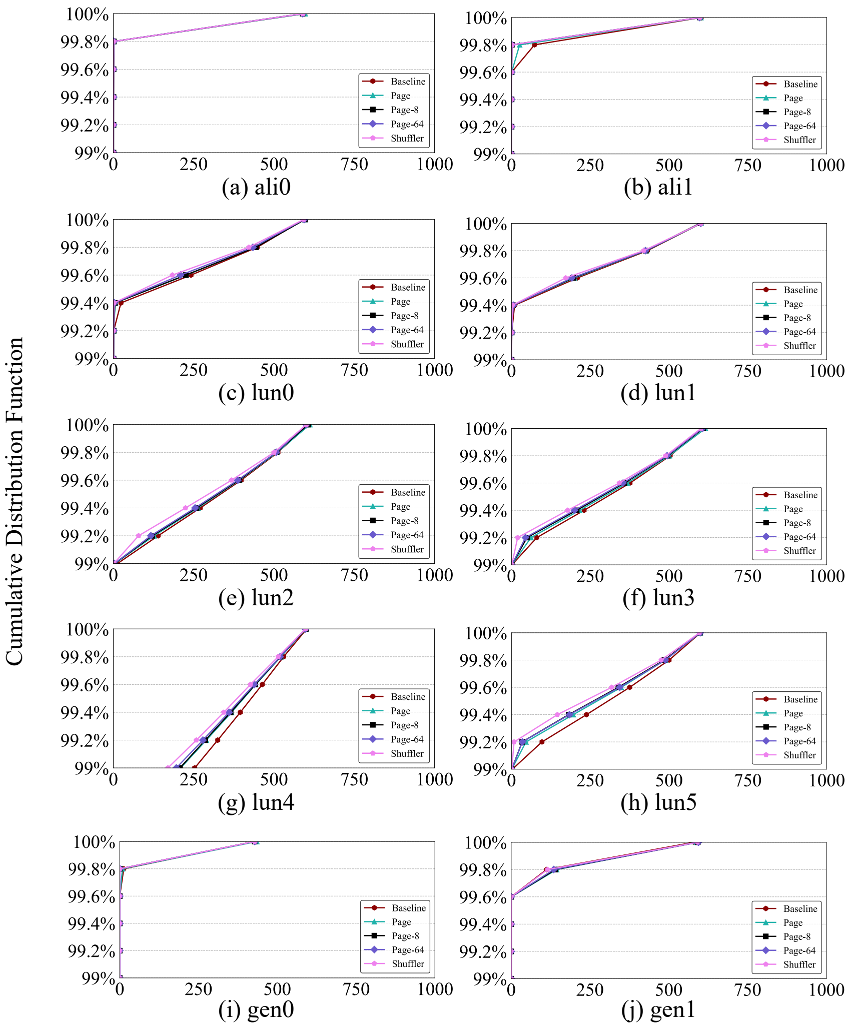

Fig. 13. Cumulative distribution function of I/O response time with different RR schemes. The unit on the X-axis of all sub-figures is millisecond.

were grouped into one superblock, while keeping other parameters consistent with the previous experiments. Figures 15(a) and 15(b) present the results of RR count and read latency, respecitvely.

As shown, *Shuffler* outperforms the compared methods in terms of RR count and read latency. Specifically, *Shuffler* can decrease the number of RR operations by 9.1%, 7.5%, 2.9% and 1.4%, so that it decreases the read latency by 10.4%, 8.4%, 2.3%, and 1.2%, in contrast to *Baseline*, *Page*, *Page-8* and *Page-64*. An interesting observation is that the improvement ratios of RR count and read latency remain similar between the large and small superblock settings. This can be attributed to two opposing factors. On the one hand, a greater number of block components within each superblock are involved in RR operations, making it more likely for read data to be distributed unevenly across different components, resulting in a greater improvement. On the other hand, the 32-block configuration of superblock supports greater parallelism, so that RR operations result in less impact on I/O latency, which moderates the overall improvement.

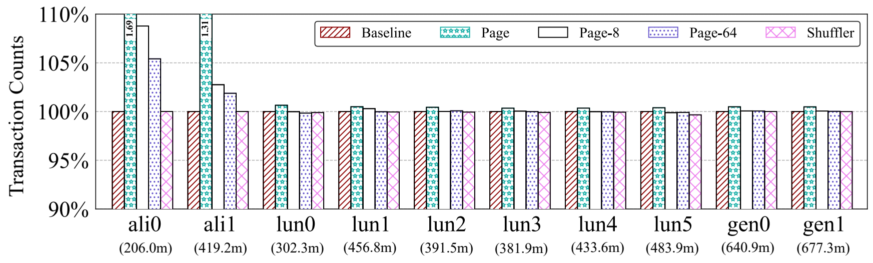

Fig. 14. The normalized number of flash transaction counts caused by I/O requests with different RR schemes.

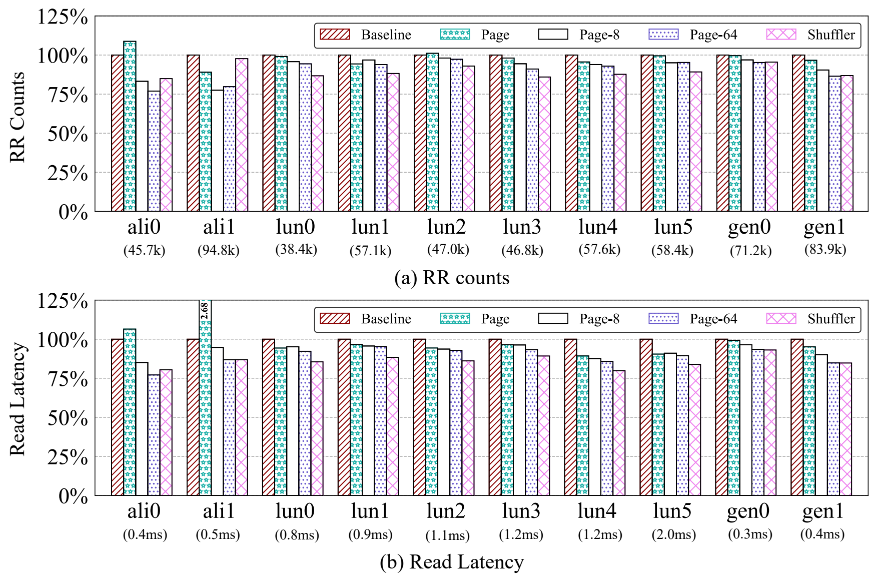

Fig. 15. Comparisons of different RR schemes with 32-block configuration of superblock.

4.4.2 Case Study on Cache Size. We have conducted a case study analyzing the impact of data cache size on system performance by replaying selected benchmarks. Performance metrics of cache hit ratio, read latency and RR count, were collected across cache configurations of 32MB and 128MB. Figures 16, 17, and 18 present the results of cache hit ratio, RR operation counts and read latency, respectively.

As shown, *Shuffler* consistently outperforms other RR approaches across varying data cache sizes. The key observation reveals an inverse relationship between the cache size and the number of RR operations, in which larger caches significantly reduce RR operations because the cached data can serve read requests directly, while the cache in traces of *lun* and *gen* cannot effectively absort incoming read requests that was discussed in Section 4.3.4. Then, we argue that *Shuffler* can effectively scale across different cache configurations while maintaining performance advantages.

4.5 Overhead Analysis

The *Shuffler* scheme analyzes block-level read counts to guide data shuffling, ensuring a balanced read distribution across component blocks within the superblock. Since block-level read counts

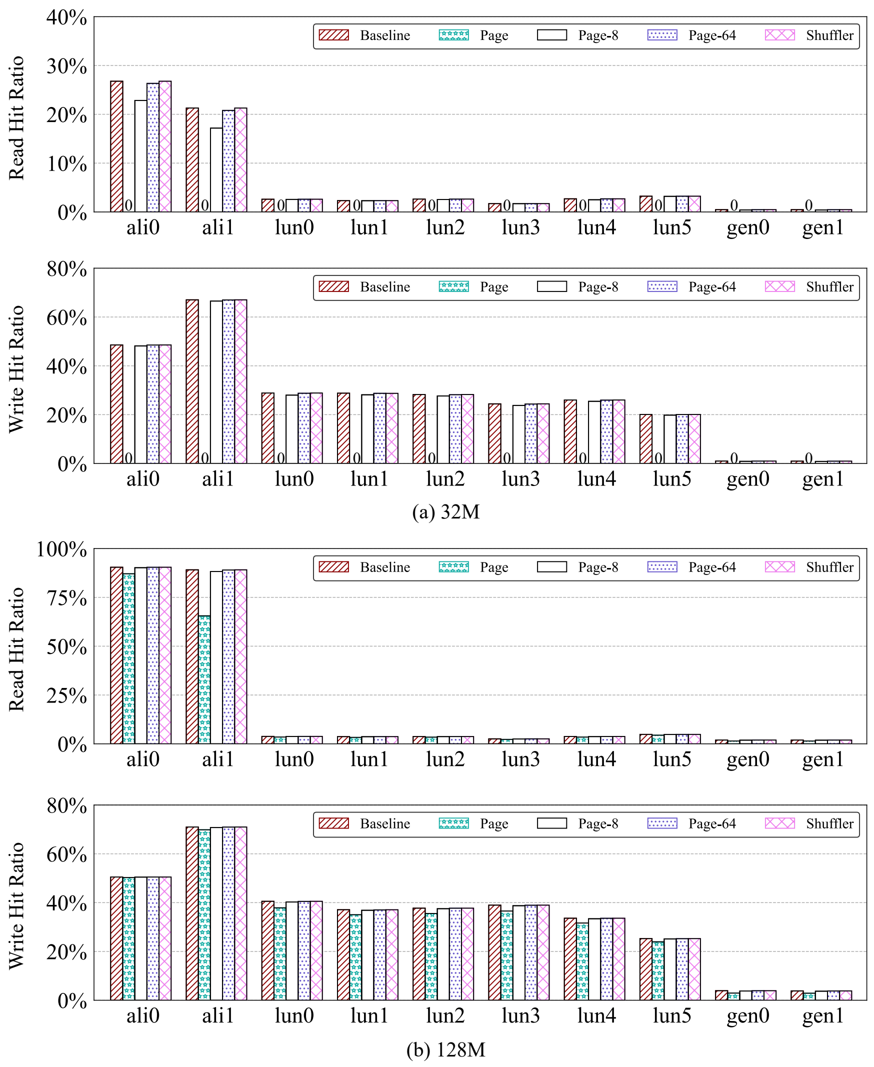

Fig. 16. Cache hit ratio comparison of different RR schemes with different size of data cache, after replaying the selected benchmarks. Note that, *Page* does not have the value of cache hit ratio in the 32M cache configuration, since the cache space is fully used for page-level counters.

are already tracked to determine whether a block approaches the RR threshold, *Shuffler* incurs no additional memory overhead.

The primary time overhead of *Shuffler* is caused by estimating the imbalance level of read workloads across component blocks and selecting the appropriate shuffling routine (full, partial, or none). Nevertheless, given the limited number of component blocks in a superblock, this time overhead remains negligible in practice. This ensures that the performance impact retains minimal while achieving the desired balance in read distribution.

To quantify the time overhead of *Shuffler*, especially during the large superblock configuration, we explicitly collect the runtime result of shuffling process. As shown in Figure 19, *Shuffler* only takes

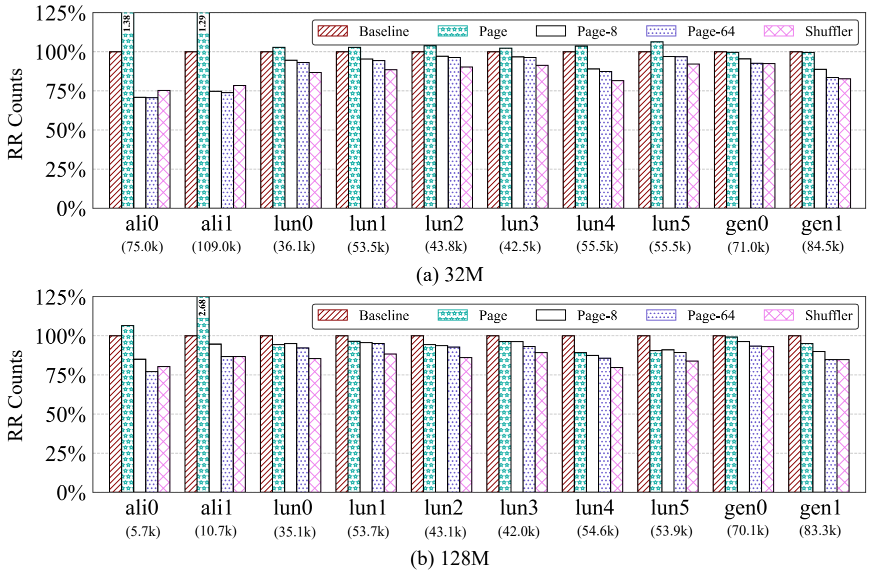

Fig. 17. RR comparison of different RR schemes under different size of data cache.

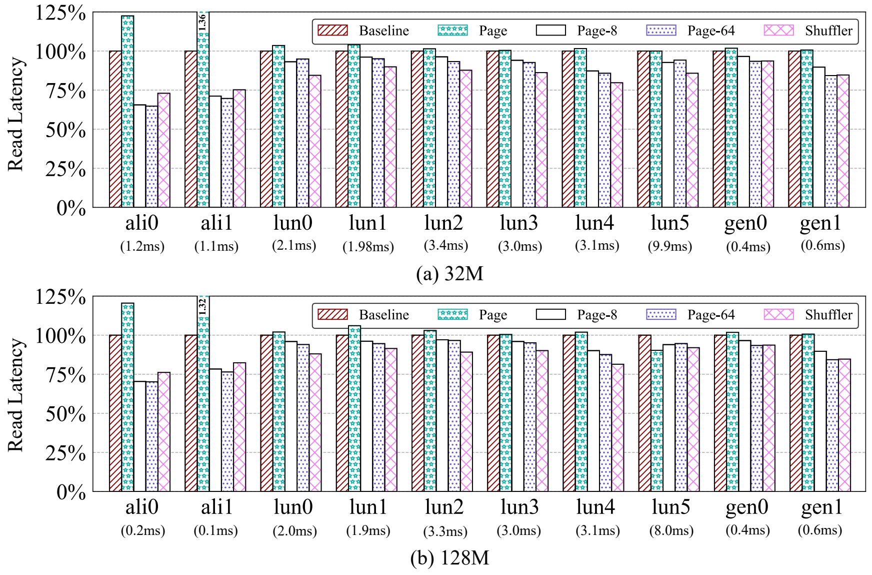

Fig. 18. Read latency comparison of different RR schemes with different size of data cache, after replaying the selected benchmarks.

45.7 seconds on average with 128-time repetitions of benchmarks in our experiments, accounting for less than 0.043% of the overall I/O response time, under 32-block configuration.

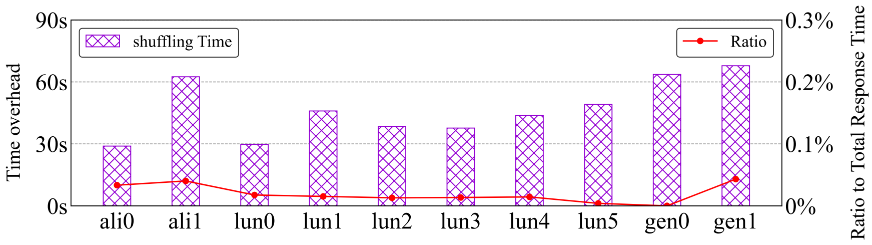

Fig. 19. Time overhead of *Shuffler* with 32-block configuration of superblock after repeating 128-time benchmarks.

4.6 Summary

With respect to comparing shuffling-based read reclaim and conventional read reclaim for superblock management in flash memory, we emphasize the following **two** key observations. *First,* page-level read counters may fail to ensure optimal read balance after RR operations, as historical read patterns do not precisely predict future read workloads. *Second,* our proposal effectively balances read accesses across component blocks within the superblock, significantly reducing write amplification factors. In addition, we argue that data page exchange between component blocks within a superblock does not impact SSD-level optimizations, such as data prefetching. This is guaranteed because the SSD controller maintains full addressability of relocated data pages at all times, and access parallelism is inherently preserved through superpage-aligned operations.

5 Conclusion

This paper has presented *Shuffler*, a novel read reclaim scheme designed for superblock management in flash memory. *Shuffler* significantly reduces the number of RR operations by unifying read accesses across the component blocks of a superblock, all without requiring additional space to track the read hotness of data pages. To this end, *Shuffler* introduces a card-granularity shuffling mechanism that redistributes data pages among the blocks of a superblock based on their read counts during the RR operation. As a result, it can ensure that read accesses are evenly distributed across component blocks while preserving access parallelism, as the superpage structure remains intact throughout the process.

Experimental results demonstrate that *Shuffler* outperforms existing superblock management schemes, reducing the number of RR operations and write amplification factors by up to 14.5% and 13.9%, respectively. These improvements highlight the effectiveness of *Shuffler* in optimizing flash memory performance and endurance, making it a promising solution for RR optimization.

Acknowledgment

The authors would like to thank anonymous reviewers for their valuable comments. The modified MQSim source code has been open-sourced at https://github.com/Yixi-sha/Yixi-sha-shuffler.

References

[1] Luo Y, Ghose S, and Cai Y et al. Improving 3D NAND flash memory lifetime by tolerating early retention loss and process variation. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (**POMACS**)*, 2018.
[2] Yu T, Wu C, and Liao Y. CRRC: Coordinating retention errors, read disturb errors and huffman coding on TLC NAND flash memory. *IEEE Transactions on Dependable and Secure Computing (**TDSC**)*, 2022.
[3] Liao J, Li J, and Zhao M et al. Read refresh scheduling and data reallocation against read disturb in ssds. *ACM Transactions on Embedded Computing Systems (**TECS**)*, 2022.
[4] Li J, Huang B, and Sha Z, et al. Mitigating negative impacts of read disturb in SSDs. *ACM Transactions on Design Automation of Electronic Systems (**TODAES**)*, 2020.

[5] Liu C, Lee Y, and Choi W et al. Prolonging 3D NAND SSD lifetime via read latency relaxation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**)*, 2021.

[6] Zambelli C, Crippa L, Micheloni R, and Olivo P. Investigating 3D NAND flash read disturb reliability with extreme value analysis. *IEEE Transactions on Device and Materials Reliability (**TDMR**)*, 2021.

[7] Micron. TN-29-28: Memory Management in NAND Flash Arrays Overview, 2016.

[8] Zhang W, Dong Z, and Zhu Y. Eddysuperblock: Improving nand flash efficiency and lifetime by endurance-driven dynamic superblock management. *IEEE Transactions on Very Large Scale Integration Systems (**VLSI**)*, 2022.

[9] Hwang J, Kim S, and Park D et al. ZMS: Zone Abstraction for Mobile Flash Storage. In *Proceedings of USENIX Annual Technical Conference (**ATC**)*, 2024.

[10] Tseng S, Chen T, and Yang M. Are Superpages Super-fast? Distilling Flash Blocks to Unify Flash Pages of a Superpage in an SSD. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (**HPCA**)*, 2024.

[11] Kim J, Jung M, and Kim J. Decoupled SSD: Rethinking SSD Architecture through Network-based Flash Controllers. *Annual International Symposium on Computer Architecture (**ISCA**)*, 2023.

[12] Introduction to Flexible Data Placement: A New Era of Optimized Data Management. https://download.semiconductor.samsung.com/resources/white-paper/FDP_Whitepaper_102423_Final.pdf, accessed in October 2024.

[13] Micron 2400 NVMe SSD. https://www.micron.com/products/storage/ssd/ client-ssd/2400-ssd. accessed in October 2024.

[14] Wang S, Wu F, and Yang C et al. WAS: Wear aware superblock management for prolonging SSD lifetime. In *Proceedings of Annual Design Automation Conference (**DAC**)*, 2019.

[15] Wu T, Ma Y, and Chang L, et al. Flash read disturb management using adaptive cell bit-density with in-place reprogramming. In *Proceedings of Design, Automation & Test in Europe Conference & Exhibition (**DATE**)*, 2018.

[16] Lee E, Kim J, and Bahn H et al. Reducing write amplification of flash storage through cooperative data management with NVM. *ACM Transactions on Storage (**TOS**)*, 2017.

[17] Lu Y, Shu J, and Zheng W. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of USENIX Conference on File and Storage Technologies (**FAST**)*, 2013.

[18] Sun P, You L, and Zheng S et al. Learning-based Data Separation for Write Amplification Reduction in Solid State Drives. In *Proceedings of ACM/IEEE Design Automation Conference (**DAC**)*, 2023.

[19] Keedwell A, and Dénes J. Latin Squares and Their Applications: Elsevier, 2015.

[20] Kim J, Kang M, and Han Y et al. Optimstore: In-storage optimization of large scale dnns with on-die processing. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (**HPCA**)*, 2023.

[21] Zhang X, Pei S, and Choi J et al. Excessive ssd-internal parallelism considered harmful. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems (**HotStorage**)*, 2023.

[22] Shahidi N, Kandemir M, and Arjomand M et al. Exploring the potentials of parallel garbage collection in SSDs for enterprise storage systems. In *Proceedings of the International Conference for High Performance Computing Networking Storage and Analysis (**SC**)*, 2016.

[23] Pang S, Deng Y, and Zhang G et al. Minato: A read-disturb-aware dynamic buffer management scheme for nand flash memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **(TCAD)***, 2024.

[24] Kim B, Choi J, and Min S. Design tradeoffs for SSD reliability. In *Proceedings of USENIX Conference on File and Storage Technologies (**FAST**)*, 2019.

[25] Cai L, Li J, and Sha Z, et al. PhasedRR: Read Reclaim Scheduling without Page-level Access Counting. In *Proceedings of The 38th International Conference on Massive Storage Systems and Technology (**MSST**)*, 2024.

[26] Tavakkol A, Gomez-Luna J, and Sadrosadati M, et al. MQSim: A framework for enabling realistic studies of modern Multi-QueueSSD devices. In *Proceedings of USENIX Conference on File and Storage Technologies **(FAST)***, 2018.

[27] Park S, Lyu J, and Kim M et al. 30.1 A 28Gb/mm 2 4XX-Layer 1Tb 3b/Cell WF-Bonding 3D-NAND Flash with 5.6 Gb/s/Pin IOs. In *Proceedings of IEEE International Solid-State Circuits Conference (**ISSCC**)*, 2025.

[28] Cho S, Kim B, and Cho H et al. AERO: Adaptive Erase Operation for Improving Lifetime and Performance of Modern NAND Flash-Based SSDs. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**)*, 2024.

[29] Cai Y, Ghose S, and Luo Y et al. Vulnerabilities in MLC NAND flash memory programming: Experimental analysis, exploits, and mitigation techniques. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture (**HPCA**)*, 2017.

[30] Narayanan D, Donnelly A, and Rowstron A. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (**TOS**)*, 2008.

[31] Alibaba block traces. [Online]. https://github.com/alibaba/block-traces.

[32] Lee C, Kumano T, and Matsuki T, et al. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of ACM International Systems and Storage Conference (**SYSTOR**)*, 2017.