ReadGuard: Integrated SSD Management for Priority-Aware Read Performance Differentiation

MYOUNGJUN CHUN, Seoul National University, Seoul, Korea (the Republic of)
MYUNGSUK KIM, Kyungpook National University, Daegu, Korea (the Republic of)
DUSOL LEE, Seoul National University, Seoul, Korea (the Republic of)
JISUNG PARK, POSTECH, Pohang, Korea (the Republic of)
JIHONG KIM, Seoul National University, Seoul, Korea (the Republic of)

When multiple apps with different I/O priorities share a high-performance SSD, it is important to differentiate the I/O QoS level based on the I/O priority of each app. In this paper, we study how a modern flash-based SSD should be designed to support priority-aware read performance differentiation. From an in-depth evaluation study using 3D TLC SSDs, we observed that existing FTLs have several weaknesses that need to be improved for better read performance differentiation. In order to overcome the existing FTL weaknesses, we propose `ReadGuard`, a novel priority-aware SSD management technique that enables an FTL to manage its blocks in a fully read-latency-aware fashion. `ReadGuard` leverages a new read-latency-centric block quality marker that can accurately distinguish the read latency of a block and ensures that higher-quality blocks are used for higher-priority apps. `ReadGuard` extends an existing suspend/resume technique to handle collisions among reads. Our experimental results show that a `ReadGuard`-enabled SSD is effective in supporting differentiated read performance in modern 3D flash SSDs.

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

Additional Key Words and Phrases: SSD, flash memory, read latency optimization, I/O priority

1 Introduction

Modern solid-state drives (SSDs) play a crucial role in serving apps that directly interact with users in large-scale data centers. Such latency-sensitive apps (e.g., web services [1], online transaction processing [2], and AI/ML inference apps [3, 4]) are commonly required to satisfy strict service-level agreements (SLAs). For instance, an online transaction processing app should process user requests and return responses with sub-second latency [5]. To meet SLA requirements, an ideal approach might be to develop a dedicated storage system for each app so that there is no interference among different apps. However, this approach is impractical for data centers due to its low cost-performance ratio, inefficient energy use, and extensive space needs [6, 7]. As a practical alternative, a data center employs shared storage systems that are shared among latency-sensitive apps as well as throughput-oriented apps (e.g., graph processing, data analysis, and backup tasks) that are less sensitive to I/O latency.

When a latency-sensitive app and a throughput-oriented app run concurrently in a storage system, we would desire the I/O latency of the latency-sensitive app to be shorter than that of the throughput-oriented app. To serve latency-sensitive apps with shorter I/O latencies in a shared storage system, several studies [8–14] have proposed

Authors' Contact Information: Myoungjun Chun, Seoul National University, Seoul, Korea (the Republic of); e-mail: mjchun@davinci.snu.ac.kr; Myungsuk Kim, Kyungpook National University, Daegu, Korea (the Republic of); e-mail: ms.kim@knu.ac.kr; Dusol Lee, Seoul National University, Seoul, Korea (the Republic of); e-mail: dslee@davinci.snu.ac.kr; Jisung Park, POSTECH, Pohang, Korea (the Republic of); e-mail: jisungpark@postech.ac.kr; Jihong Kim, Seoul National University, Seoul, Korea (the Republic of); e-mail: jihong@davinci.snu.ac.kr.

ACM 1553-3093/2024/7-ART
https://doi.org/10.1145/3676884

solutions that differentiate I/O latencies among multiple apps based on their priorities. FlashShare [8], for example, successfully reduced the average and p99-percentile read latency of a latency-sensitive app by enhancing the kernel-level I/O stack based on I/O priority. To address SSD-level read latency, which can constitute up to 91% of total I/O latency in a modern I/O stack with an NVMe interface [15], some works [11–14] have proposed SSD-level scheduling techniques that reorder read requests based on their I/O priority within device-level queues. In this paper, we argue that (1) existing priority-aware I/O management techniques at various I/O stack layers are not sufficient to differentiate SSD-level latencies among different apps in modern 3D flash SSDs, and (2) the I/O priority of an app should be carefully managed inside an SSD from the NAND block level to main FTL modules for priority-aware I/O performance differentiation. Since read latency is a crucial factor in determining the perceived I/O performance for the majority of applications, this paper focuses on differentiating read latency.

To understand how well a modern 3D flash SSD supports the read-latency differentiation requirement, we evaluated the read-latency (at the SSD level) distributions of apps using an NVMe SSD simulator with three priority I/O queues. Our simulation environment supports an SSD-level priority-aware scheduling mechanism proposed by previous studies [11–14] (see Section 2.4 for more details on the priority-aware FTL). Figure 1 shows the read-latency distributions of three apps, τ_{high}, τ_{mid}, and τ_{low}, where the I/O priority of τ_{high} is the highest while that of τ_{low} is the lowest.[1] Note that the read latency of Figure 1 represents the end-to-end read latency of an SSD from the time an app enqueues a read request to a submission queue to the time when the read response arrives at the host. As shown in Figure 1, the SSD did not adequately support read differentiation. In all three stages of the SSD lifetime, the average read latency of three apps was virtually indistinguishable regardless of the I/O priority of an app.

To identify the root causes of poor read differentiation over app priorities in our priority-aware SSD, we performed a comprehensive study from a NAND flash memory to an FTL and identified three main causes of poor read differentiation. First, the key modules of existing priority-aware FTLs (such as [11–14]) work in a read-latency-unaware fashion. For example, these FTLs assume that the read latency of flash blocks in an SSD is equal. Therefore, when the read latency of flash blocks is significantly different (as observed in modern 3D flash blocks), the existing priority-aware FTLs cannot properly support I/O requests with different priorities. For example, in our benchmark evaluations, we observed frequent block-quality inversions among apps, allocating blocks with shorter read latency to lower-priority apps. Second, conventional block quality measures (e.g., program/erase (P/E) cycles) are inadequate to differentiate the read latency of modern 3D flash blocks with high process variability. Since a large variation in the read latency of 3D flash blocks is directly related to the number

[1]See Section 3.2 for more details on the evaluated apps and SSD lifetime stages.

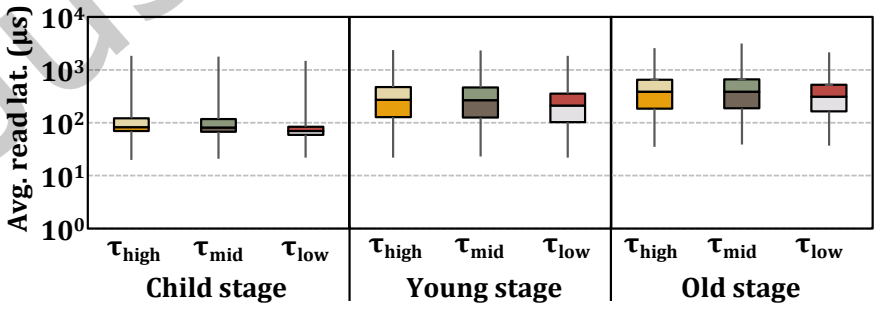

Fig. 1. Read-latency distributions among three apps.

of read-retry operations, a better block quality measure, which focuses on the read latency of flash blocks, is needed so that the number of read-retry operations of a block can be accurately predicted. Third, the existing priority-aware FTLs do not properly handle the conflict between two NAND read commands with different priorities. Existing command schedulers preempt only ongoing writes and erases over reads, without considering the case when a lower-priority read conflicts with a higher-priority read, which can cause a large delay for the higher-priority read when the lower-priority read requires a long latency to complete.

Motivated by our findings from the evaluation study, we propose a new integrated *priority-aware* flash management scheme, `ReadGuard`, which can better differentiate read latency between apps with different priority requirements. `ReadGuard` makes three key contributions. First, we propose a novel read-latency-centric block quality marker that can accurately represent the (worst-case) read latency *tREAD* of each block. By estimating the read latency of a block, not the reliability of data stored in the block (as in common block quality measures such as P/E cycles), the proposed block quality marker enables read-latency-aware block management in `ReadGuard`. Second, `ReadGuard` adopts a priority-aware block management scheme based on the proposed block quality marker. A `ReadGuard`-based FTL allocates blocks with short read latency to a higher-priority app and continuously monitors the quality level of the allocated blocks so that high-priority read requests can be serviced from the high-quality blocks. Third, we propose a priority-aware read-over-read command preemption mechanism. When blocks are managed based on their read latency in a priority-aware fashion, a higher-priority read should be able to preempt an ongoing lower-priority read command. Otherwise, a higher-priority read command will experience an excessive amount of delay because a lower-priority read tends to be serviced from a block with long read latency.

In order to evaluate the effectiveness of `ReadGuard`, we have implemented a `ReadGuard`-enabled FTL, `rgFTL`, using an open SSD simulation platform [11]. Our experimental results using various real-world workloads show that `rgFTL` can effectively differentiate the read performance of apps according to their I/O priorities. In `rgFTL`, the average read latency of the highest-priority app is up to 57.1% shorter than that of the lowest-priority app while the baseline FTL does not differentiate the read latency between these apps. Furthermore, `rgFTL` reduces the 99th-percentile read tail latency of high-priority apps by up to 55.5%. Although `rgFTL` needs additional block copy operations to avoid block-quality inversions among apps, their impact on SSD lifetime and performance is not significant. `RgFTL` increases the average write latency by by about 2.8% because additional page writes, which are needed to avoid block-quality inversions among apps, incur additional garbage collection.

2 Background

In order to support priority-aware read differentiation, our proposed technique requires understanding key parameters of NAND flash memory that affect the flash read latency. Therefore, we review the basics of the flash read latency and the impact of read errors on the flash read latency. We also briefly present an overview of an existing priority-aware SSD.

2.1 NAND Flash Memory Basics

A flash cell is the fundamental component of NAND flash memory. Figure 2 depicts its organization, with components such as the charge trap, blocking oxide, control gate, and tunnel oxide. Controlling the number of electrons in a flash cell's charge trap allows data to be stored. The threshold voltage (V_{th}) level of the flash cell distinguishes the binary data stored in it, which is either '0' or '1'. To change the V_{th} of the flash cell, electrons are either injected into or removed from the charge trap. This electron movement is facilitated by the tunnel oxide, a thin layer of insulating material between the substrate and the charge trap.

In a flash die, individual flash cells are organized into a hierarchical structure. Each flash die has several planes and hundreds to thousands of blocks within each plane. Each of these blocks is composed of multiple sub-blocks.

The sub-blocks are represented as matrices with rows and columns composed of flash cells. These horizontal rows, known as wordlines (WLs), connect the flash cells' control gates, whereas the vertical columns, known as bitlines (BLs), connect the cells' drain and source terminals. When a wordline (WL) is activated, the same voltage is applied to all cells of the target WL, allowing for simultaneous read and write operations across all cells on the WL. The type of NAND flash decides how many pages a single WL corresponds to. For example, in a triple-level cell (TLC) NAND flash memory, each WL is associated with three pages (MSB, CSB, LSB pages).

A read, a program, and an erase operation are the three fundamental operations of NAND flash memory. A *read* operation applies a specific voltage, read reference voltage V_{ref}, to distinguish between V_{th} levels of flash cells in target WL. The Flash chip determines the stored data by observing whether the current flows or not through the BLs. A *program* operation applies a high voltage (e.g., 20V) to the cell's control gate through its target WL. As a result of the voltage difference, electrons from the substrate tunnel through the gate oxide and are trapped in the charge trap, thus increasing the V_{th} level of the cell. An *erase* operation operates at the block granularity, whereas read and program operations operate at the page granularity. To erase the data within the target block, the flash chip applies a high voltage (e.g., 20V) to the source terminal. The voltage difference causes electrons to tunnel from the charge trap to the substrate via the tunnel oxide so that the V_{th} levels of all cells in the block are returned to the initial states.

2.2 Read Errors in NAND Flash Memory

Despite its nonvolatile nature, NAND flash memory is inherently prone to errors. Various error sources, such as retention loss [16, 17] and program disturb [18], can shift the V_{th} levels of flash cells beyond the V_{ref} value, leading to potential bit errors in NAND flash memory. Figure 3 shows the V_{th} distribution in a WL of MLC NAND flash memory, that employs four distinct V_{th} levels to store two bits per cell (E, P1, P2, and P3). Reference voltages (V_{ref0}, V_{ref1}, and V_{ref2}) are used to determine the V_{th} levels of flash cells. The V_{refi} reference voltage distinguishes P($i-1$) and P(i). In the initial state, as shown in Figure 3(a), all V_{th} levels can be reliably distinguished using reference voltages. Retention loss, however, causes unintended shifts in the V_{th} levels, making it more likely to overlap with V_{refi}. The stored bits in the overlapped region of flash cells flip, resulting in raw bit errors of read data.

The number of bit errors in read data is directly affected by the error characteristics of the target flash cells. The high voltage stress involved in repetitive program/erase operations (i.e., P/E cycles) accelerates the deterioration of a flash cell's tunnel oxide. This deterioration weakens its insulating capabilities, resulting in rapid charge leakage from the charge trap into the substrate. Furthermore, due to manufacturing process variations, particularly in 3D NAND flash memory, the initial thickness of the tunnel oxide may differ between flash cells [19–21]. Flash cells

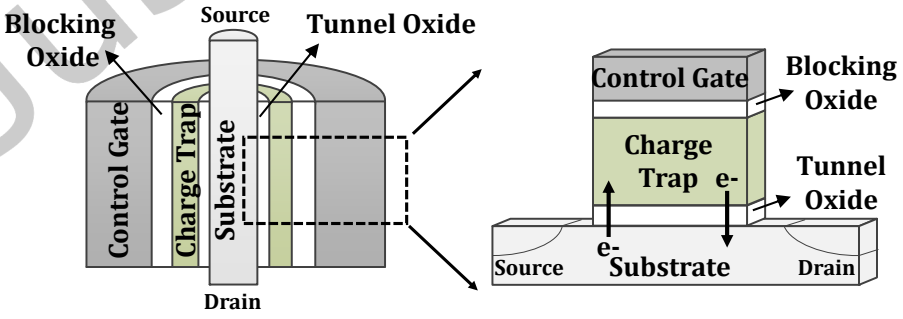

Fig. 2. An organization of a flash cell.

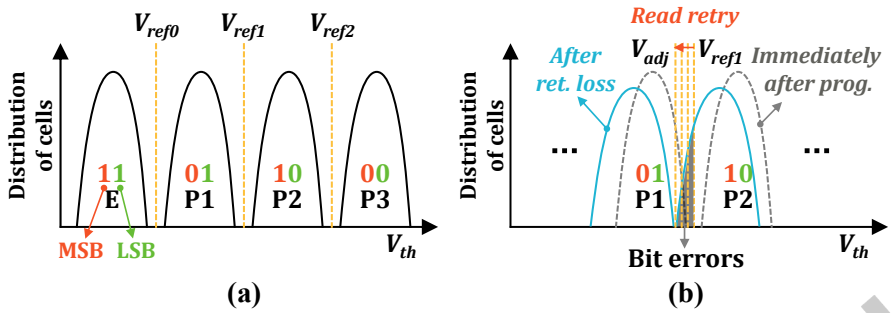

Fig. 3. Changes in V_{th} distribution of MLC flash cells and V_{ref} adjustment in a read-retry operation.

with a thinner oxide layer are inherently more susceptible to sustained voltage stress, causing them to wear out faster. As a result, even when two WLs are exposed to the same error sources, the number of bit errors in their stored data can vary significantly depending on their P/E cycles and inherent error characteristics.

2.3 Read-Retry in Modern SSDs and Its Impact on Read Latency

To ensure data reliability despite the heterogeneous error characteristics of modern NAND flash memory, modern SSDs use strong error-correcting codes (ECC) that can correct several tens of raw bit errors. Unfortunately, due to the high raw bit-error rate (RBER), even such a strong ECC often fails to correct all bit errors in modern NAND flash memory. To address this, when the RBER of a read page exceeds the ECC correction capability (i.e., the number of correctable bit errors), a modern SSD performs an *read-retry* operation. The read-retry operation repeats reading of the page with adjusted V_{ref} values until the page's RBER falls below the ECC capability or until a set number of retry attempts is reached [19, 20, 22–25]. Although read-retry is highly effective at ensuring data reliability, it significantly increases the effective read latency of NAND flash memory, almost linearly with the number of retry steps. Three read voltage adjustments are needed to retrieve the correct data (with V_{adj}) in Figure 3(b), for instance, resulting in four times the normal read latency.

In general, the device-level latency *tREAD* for reading a flash page can be expressed as $tREAD = (tR + tDMA + tECC) \times (N_{retry} + 1)$ where tR is the flash page access time, $tDMA$ is the data transfer time from a flash chip to a flash controller, and $tECC$ is the error correction time by the ECC engine. While tR, $tDMA$, and $tECC$ are fixed by flash manufacturers (i.e., they do not change during run time), N_{retry} significantly varies depending on the number of errors on the target page.

2.4 Overview of a Priority-Aware SSD

Although the flash read latency *tREAD* is a key parameter in deciding the read latency at the app level, the host-side read latency is significantly affected as well by SSD-internal states at the time of the read request issue. Figure 4 illustrates how a read request r is processed in a priority-aware SSD [11–14]. When the host issues r containing a target logical block address (LBA) range, an address of host-side read buffer, and its priority, it is first transferred to the host interface logic in the SSD (❶). The host interface logic splits the LBAs in the target request range into a series of m flash read requests (i.e., m transactions) (❷). An LBA address of each flash read request is converted to a physical page address (PPA) by the address translator. A transaction with a PPA (i.e., a read command to the PPA) is then enqueued into a per-chip queue that is responsible for serving the PPA (❸). The transaction scheduler decides which transaction is issued first by prioritizing the pending transactions in per-chip queues. When the status of the target flash chip is ready, the highest priority request is issued to the

flash chip (❹). An ECC engine corrects potential error bits of the requested page (❺) before the page is sent to the host-side completion queue (❻).

In existing priority-aware SSDs, regardless of a request priority, read commands are handled first over other flash commands (i.e., write and erase commands). When a read command is selected by the transaction scheduler, if a program command or an erase command is currently serviced at the same target flash chip, the transaction scheduler preempts the ongoing program/erase command by suspending its operation so that the read command can be serviced first [26, 27]. That is, read/write and read/erase interferences are minimized by command suspension techniques in the existing prior-aware SSDs. However, if the ongoing command is read, existing transaction schedulers [28] do not suspend the ongoing read command even if its priority is lower than that of the newly selected read command.

When a host read request r requires m flash reads to m LBAs, a_r^0, ..., a_r^{m-1}, the host-side read latency $L(r)$ is given by $max\{l(a_r^0), ..., la_r^{m-1}\}$ where $l(a)$ represents the read latency of a read request to the LBA a that measures from when a read request to the LBA a is fetched by an FTL to when the read request is completed. The read latency $l(a)$ of a read request to the LBA a consists of two terms, the flash-device latency for reading from the LBA a and the waiting time before a flash read command is issued for accessing the LBA a.

3 Root Cause Analysis

In this section, we explain key reasons for poor read differentiation in existing priority-aware SSDs. In a priority-aware SSD, if the I/O priority of r_i is higher than that of r_j, the transactions for r_i have a higher priority over those for r_j. Therefore, the transactions for a higher-priority request experience shorter waiting times over those for a lower-priority request, thus effectively supporting the waiting time differentiation over I/O priorities. For example, Figure 5 shows the average waiting time of the same three apps in Figure 1. Unlike the read-latency distributions (shown in Figure 1), the waiting times are quite nicely differentiated according to the I/O priority of the apps. Furthermore, since an incoming read command can preempt ongoing program/erase operations for serving the incoming read first, other flash operations do not interfere with read operations. Therefore, we start from the device level read latency *tREAD* to investigate the root causes of poor read differentiation.

3.1 Cause 1: Large Variations in Read Latency

Modern SSDs suffer from a large number of N_{retry} from the capacity-centric cell design (e.g., TLC and QLC) that makes SSDs to be more vulnerable to quick V_{th} shifts beyond V_{refi} after programmed [20, 21, 24]. Furthermore, the manufacturing process of modern flash chips introduces substantial process variability in terms of reliability

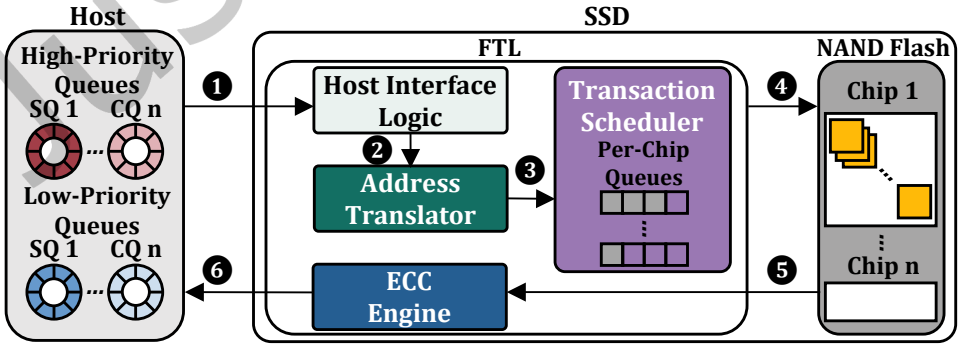

Fig. 4. Key steps of read request processing.

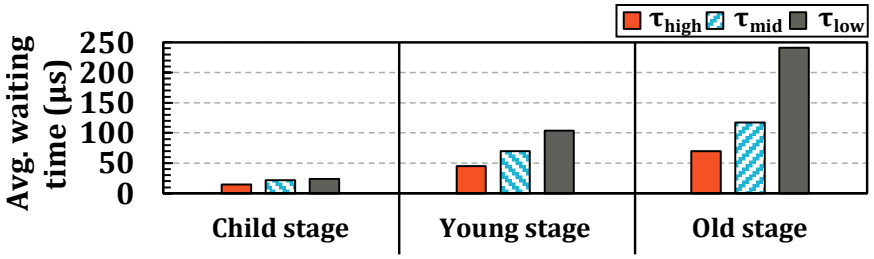

Fig. 5. Comparisons of average waiting times of three apps.

between flash blocks [21]. To understand how these trends affected the variation in read-latency distribution in an SSD, we performed comprehensive evaluations using 160 real 3D TLC flash chips. We measured the block-level read latency (i.e., N_{retry} of the worst page of a block) of more than 10,000 blocks under different P/E cycles and retention times. (When no confusion arises, N_{retry} of a block is used to mean N_{retry} of the worst page in a block.)

Figure 6 shows block-level N_{retry} distributions for the tested blocks under different operating conditions. We make two key observations from Figure 6. First, there are substantial block-level N_{retry} variations among flash blocks. For example, N_{retry} value of a block can vary significantly from 0 to 25. Such a large variation on N_{retry} between blocks can be attributed to inter-block process variability of a 3D flash manufacturing process as well as different operating conditions (e.g., P/E cycles or retention times). Therefore, when flash blocks experience different P/E cycles and retention times, they exhibit different levels of raw bit errors, thus resulting in significant variations of N_{retry}.

Furthermore, even under the same P/E cycle and retention time condition, N_{retry} of a block can differ by several times. For example, under the zero P/E cycle and 12-month retention time condition, the maximum N_{retry} value of tested blocks was 3× larger than the minimum N_{retry} value. The root cause of block-level N_{retry} variations is genetic process variability among blocks (e.g., different thickness of T_{OX} or flash cell size) resulting from the complex 3D NAND manufacturing procedure, as explained in Section 2.2.

Second, conventional P/E cycle-based block quality metrics are not adequate in predicting N_{retry} of the worst page in a block. That is, we cannot estimate N_{retry} of a block accurately using a block quality metric based on the P/E cycle and retention time. For example, N_{retry} values of blocks with the same P/E cycle (e.g., 3K P/E cycle) and the same retention time (e.g., 1-month retention time) are in the range between 8 and 16. If a N_{retry}

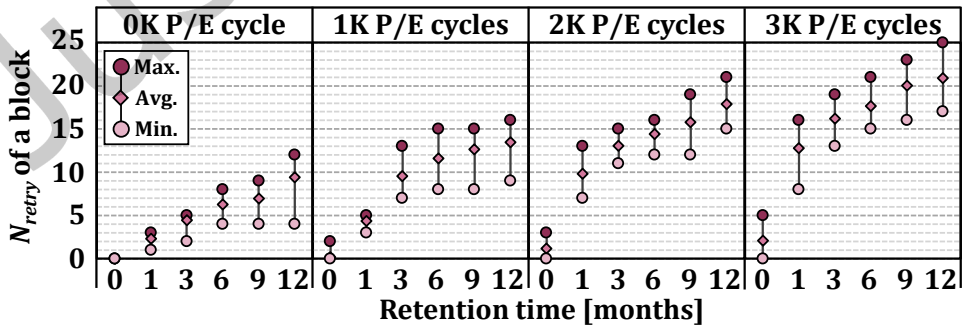

Fig. 6. Block-level N_{retry} distributions.

predictor were based on the P/E cycle and retention time only, it would be impossible to estimate N_{retry} of a block accurately. Our key observations from Figure 6 strongly suggest that we need a *read-latency-centric new measure* for predicting N_{retry} of a block so that we can manage SSD read requests in a priority-aware fashion.

3.2 Cause 2: Priority-Oblivious Block Management

It is commonly accepted that in existing priority-aware FTLs, it is not necessary to distinguish the quality of different flash blocks because a wear-leveling mechanism in an SSD tends to maintain the quality of all the blocks at a similar level. Therefore, although these FTLs honor the I/O priority until an I/O request is issued to flash chips at the frontend of FTLs, they do not employ priority-aware block management techniques in key FTL backend management modules.

To validate the claim that block quality management is not necessary in an FTL because of a wear leveler, we evaluated if similar quality blocks are allocated to apps regardless of their app priority. We extended our simulation environment [11] so that it can accurately reflect the real device characterization results from our characterization study (in Section 4.2).[2] We collected N_{retry} values of target blocks of read requests for three apps used in Figure 1 at three distinct SSD lifetime stages: a child stage (at 500 P/E cycles), a young stage (at 1K P/E cycles), and an old stage (at 3K P/E cycles). For this evaluation, We used three traces were collected from running Yahoo! Cloud Service Benchmark [29] using RocksDB [30] with three different access patterns: an update-heavy workload (KV_A), a read-intensive workload (KV_B), and a read-modify-write workload (KV_F). A high-priority app, τ_{high}, executes KV_B while two lower-priority apps, τ_{mid} and τ_{low}, run KV_F and KV_A, respectively.

Figure 7 shows block-level N_{retry} distributions of each app using box plots. In Figure 7, block-level N_{retry} values are normalized N_{retry} values between 0 and 1, where the lower value, the lower N_{retry} value. Unlike the common belief on the homogeneous quality because of an effective wear leveler of an FTL, the box plots of Figure 7 indicate that flash blocks with heterogeneous block quality were randomly allocated to three apps. For example, in all three SSDs, τ_{low} was allocated to better blocks than τ_{high}. Furthermore, at the old stage of the SSD lifetime, most poor blocks were allocated to τ_{high}. This observation strongly suggests that the block quality of flash blocks is not similarly maintained, thus requiring priority-aware block quality management for effective read-performance differentiation.

3.3 Cause 3: No Read-Over-Read Preemption

Existing command preemption techniques focus on suspending slow ongoing commands such as program and erase operations when a new read command is issued because their latency is 5.7× and 30.4× longer than that

[2]See Section 6.1 for a detailed description of our simulation environment.

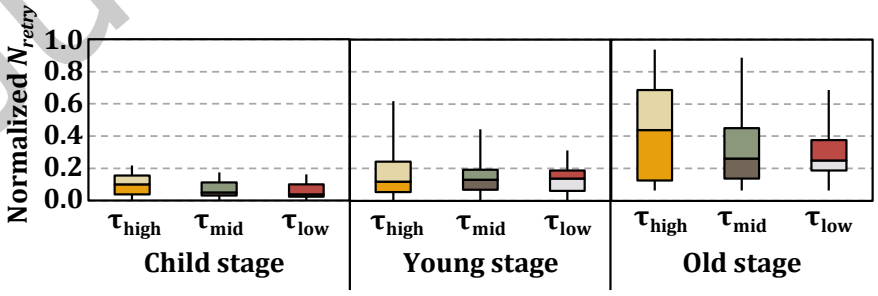

Fig. 7. Per-app block-level N_{retry} variations.

of a read, respectively [31, 32]. When a read command conflicts with another ongoing read command, the new read command must wait until the ongoing read command is completed although it is a higher-priority read command.

When there is little difference in read latency among different blocks (as incorrectly assumed in existing priority-aware FTLs), there may not be a strong need for supporting read-over-read preemption because an extra latency delay for a high-priority app may not be significant unless a large number of low-priority reads were intensively issued to the same target flash chip before the high-priority read. However, when an FTL supports read-latency-aware block management over priorities, a read-over-read preemption mechanism makes a big difference for high-priority reads, especially for their tail latency. In such an FTL, since the read latency of high-quality blocks would be shorter than that of low-quality blocks unless the read-over-read preemption is efficiently supported, the read latency of high-priority reads can be substantially degraded by low-priority reads.

To understand the impact of read-over-read preemption on the read tail latency when read-latency-aware block management is fully supported, we compared the read-latency distributions of three apps used in Figure 7, τ_{high}, τ_{mid}, and τ_{low}, under two FTL configurations, one with a read-over-read preemption mechanism, `rorFTL`, and the other without it, `nopFTL`. Both FTLs were configured to guarantee that a higher-priority app is allocated to blocks with shorter read latency. The 99-percentile tail latency of τ_{high} under `nopFTL` was up to 37.4% longer than that under `rorFTL` in the old SSD. Furthermore, the average read latency of τ_{high} was up to 25% longer than that under `rorFTL` as well.

4 Read-Latency-Centric Block Marker

To keep track of the heterogeneous block quality in terms of the read latency, we build a new block quality model that can accurately estimate *tREAD* of a block. Unlike many existing block quality markers [21, 33, 34] which focus on predicting the wear status of a block, `ReadGuard` needs a block quality marker that can be used to estimate *tREAD* of a block. Since N_{retry} is the only run-time variable in deciding *tREAD*, the proposed block quality model, the N_{retry} predictor $nr(B)$ of a block B, aims to predict the N_{retry} value of the slowest page in a block B. To the best of our knowledge, the proposed $nr(B)$ model is the first block quality metric that is specialized for estimating N_{retry}.

In developing an effective $nr(B)$ model, we explore a strong correlation between N_{retry} of a block B and the raw bit error rate (RBER) of the block B.[3] In order to understand the relationship between N_{retry} and RBER of a block, we conducted a characterization study using real 3D TLC NAND flash chips. (See Section 4.2 for a detailed description of our methodology.) Figure 8 illustrates how N_{retry} and RBER of a block are related to each other in different P/E cycles and retention months. As shown in Figure 8, N_{retry} of a block can be expressed as a step function of RBER of a block. Therefore, we take a two-step approach to building a $nr(B)$ model. As a first step, we predict the number of raw bit errors of the block B. Based on the estimated number of raw bit errors, we predict the N_{retry} of the block B.

As explained in Section 2.2, raw bit errors of a block consist of two major error sources: the bit errors caused by *program disturb* and *retention loss*, respectively. The bit errors by program disturb occur when pages in a block are programmed, while the bit errors by retention loss continually increase after the pages in the block are written. Therefore, it is logical to build a $nr(B)$ model using two submodels that correspond to two error sources of a block. Figure 9 conceptually illustrates how $nr(B)$ of a block B is decided under the proposed method. When $nr(B)$ of the block B is needed at time t_{cur}, two error attributes of the block B, $E_{init}(B)$ and $\Delta E(B)$, are computed. The $E_{init}(B)$ attribute (❶) represents the number of *initial* raw bit errors of the block B when its pages were most recently written at time t_s (where $t_s \leq t_{cur}$). The $E_{init}(B)$ attribute indicates how much the block B was affected by program disturbance. The $\Delta E(B)$ attribute (❷) is used to estimate the number of errors that have been

[3]Similar to the definition of N_{retry} of a block, we define the RBER value of a block as the RBER value of the worst page in a block.

accumulated to the block B since it was programmed at time t_s. The $\Delta E(B)$ attribute represents how much the block B was affected by retention loss during the time interval $(t_s, t_{cur}]$. By adding $E_{init}(B)$ and $\Delta E(B)$ values, the total number $E(B)$ of raw bit errors of the block B at time t_{cur} is computed (❸). Finally, $nr(B)$ is computed using a step function that relates $E(B)$ to $nr(B)$ (❹). Note that in Figure 9, the block B has an additional attribute $\texttt{age}(B)$. The $\texttt{age}(B)$ attribute, which represents the wear status of the block B, plays a key role in computing both $E_{init}(B)$ and $\Delta E(B)$ because both $E_{init}(B)$ and $\Delta E(B)$ are significantly affected by the wear status of the block B.

4.1 NAND Age Predictor: $\texttt{age}(B)$

The key prerequisite of estimating the number of raw bit errors of a block B is to know the accurate flash wear status of the block because the number of raw bit errors of the block varies significantly depending on the wear status of flash cells in the block [19, 21, 22]. For example, the number of retention error bits of a block under the same retention time condition (e.g., 12 months) can be several times different due to the varying wear status of the blocks. As discussed in Section 2.2, the flash wear status is closely related to the state of tunnel oxide in flash cells (i.e., a trap density). However, it is practically impossible to measure the trap density during run-time. As an alternative metric to differentiate the wear status of a flash block, several previous studies [21, 33, 34] have exploited the number $N(t)$ of retention bit errors *after* the t-month retention time at 30°C.[4] In our study, following the common industry practice (i.e., the JEDEC standard [38, 39]), we use $N(12)$ with the 12-month retention time.[5]

Although $N(12)$ is an accurate indicator of the flash wear status of a block, it is still not a practical metric to be used during run time because it measures *future bit errors* after 12 months since the block was programmed.

[4]Previous studies about NAND physics have shown that the number of retention bit errors has a near-linear relationship with the number of traps [35–37]. Furthermore, for recent multi-level flash memory, retention errors are responsible for the majority of the total number of raw bit errors, especially when the flash memory gets aged [19, 22, 24].

[5]Although different retention times (e.g., six months) may be effective as well, we use the 12-month retention time requirement in this paper because it is commonly used as the worst-case reliability condition in practice.

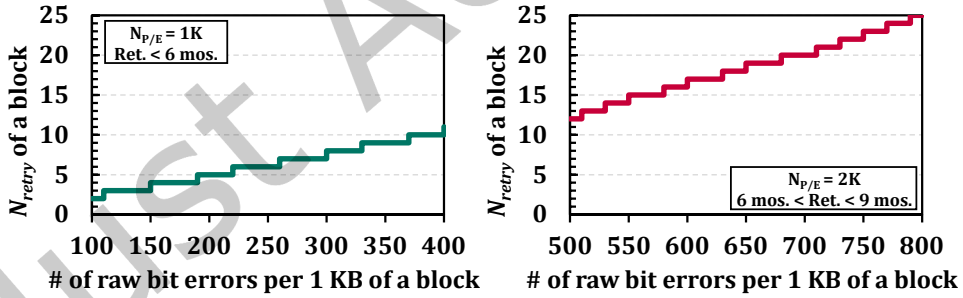

Fig. 8. A relationship between RBER and N_{retry} values of flash blocks under different operating conditions.

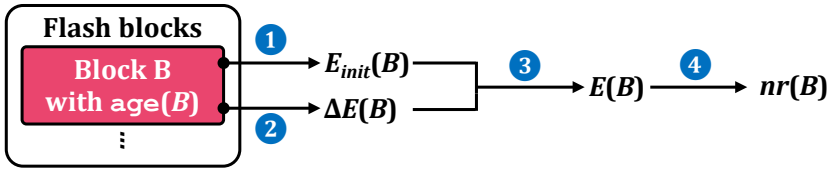

Fig. 9. An overview of predicting N_{retry} of a block B.

Table 1. A summary of variables for the age(B) attribute.

Variable	Description
$N_{P/E}$	P/E cycles
$tERASE$	Erase latency
E_0	No. of bit errors of an LSB page in the first WL right after program
L_{dwell}	Average length of time interval between successive erase (effectively at 30°C)

Because of this reason, $N(12)$ is mostly used *as an off-line metric* for characterizing the wear status of flash cells. Therefore, we need an accurate *online* predictor for $N(12)$. Note that the most common index for the flash wear status, the number $N_{P/E}$ of program and erase cycles, does not accurately predict $N(12)$ because it cannot account for key variations that affect the flash wear status such as process variability, I/O workload variations, and operating environment variations [21].

In order to design an accurate online predictor for $N(12)$, we use RealWear [21], which is one of the most accurate flash aging metrics. Our proposed $N(12)$ predictor of a block B, denoted as age(B), uses four run-time accessible parameters that are summarized in Table 1. In addition to $N_{P/E}$, three additional parameters are used: the erase latency $tERASE$, the number E_0 of bit errors of an LSB page in the first wordline, and the average length L_{dwell} of time intervals between successive erase operations at 30°C.[6] The additional three parameters are used to complement the weaknesses of $N_{P/E}$ as a wear status predictor. For convenience, age(B) is a normalized value by computing a ratio of $N(12)$ of the block B to the maximum number of bit errors that can be corrected by an ECC module. When the block B is *alive* (i.e., it can reliably store data under the 12-month retention requirement), $0 \leq \texttt{age}(B) \leq 1$. From the proposed predictor equation in [21], age(B) can be expressed as:

$$\texttt{age}(B) = c_0 + c_1 \cdot N_{P/E} + c_2 \cdot tERASE + c_3 \cdot E_0 + c_4 \cdot ln(L_{dwell}).$$

Five coefficients, c_0, c_1, c_2, c_3, and c_4, were estimated by the least-squares approximation method. The constant term c_0 reflects inborn defects from a manufacturing process. (See the reference [21] for a complete description of RealWear including its validation results as a NAND age predictor.)

4.2 N_{retry} Predictor Function: $nr(B)$

Model Construction. As described in Figure 9, the total number $E(B)$ of raw bit errors of a block B at time t_{cur} is calculated by a sum of $E_{init}(B)$ and $\Delta E(B)$ at time t_{cur}. The $E_{init}(B)$ value of the block B, which indicates how much flash cells were affected by program disturb after the block B was programmed, is known to have a strong and positive linear correlation with the wear status of flash cells [21]. Therefore, $E_{init}(B)$ can be expressed as:

$$E_{init}(B) = c_5 \cdot \texttt{age}(B) + c_6. \quad (1)$$

In Equation (1), the first term represents the number of raw bit errors of a block induced by the program disturbance effect and the second term represents the number of inborn bit errors of a block from a manufacturing process.[7] In order to find $E_{init}(B)$ at time t_{cur}, we use the $E_{init}(B)$ value at time t_s, the time when the block was most recently programmed. Since age(B) changes only when the block is erased, $E_{init}(B)$ at time t_s is still valid at time t_{cur} because the block B was erased most recently at time t_s.

[6]The impact of L_{dwell}, which models the effect of I/O workload variations and operating environment variations on the flash wear status, significantly varies depending on the operating temperature. For example, L_{dwell} of 1 hour at 50°C has the same impact as L_{dwell} of 13 hours at 30°C. We used the baseline temperature of 30°C in L_{dwell}. When L_{dwell} at T°C was measured, it is converted to L_{dwell} at 30°C using the Arrhenius's Law [21, 22].

[7]To decide c_5 and c_6, we used the non-linear least squares algorithm to fit $E_{init}(B)$ to the measurement data from a characterization study.

The $\Delta E(B)$ value of the block B at time t_{cur} indicates how many additional raw bit errors were accumulated to the block B since it was most recently programmed at time t_s. As retention loss and read disturbance account for most of the additional raw bit errors, the proposed function employs two variables to reflect these error sources: the data retention time T_{ret} and the number N_{read} of read operations to a block. As well known, the additional raw bit errors by retention loss have a logarithmic relationship with T_{ret} [19, 24, 25, 40], while the additional raw bit errors by read disturbance phenomenon are exponentially increased with N_{read} [41, 42].[8] Since the flash wear status (e.g., age(B)) of a block significantly affects the additional raw bit errors by each error source, our proposed $\Delta E(B)$ can be expressed as follows:

$$\Delta E(B) = c_7 \cdot (\texttt{age}(B) + c_8) \left\{ ln\,(1 + T_{ret}) + e^{(c_9 \cdot N_{read})} \right\}. \tag{2}$$

To derive five coefficients, c_5, c_6, c_7, c_8, and c_9, we used the non-linear least squares algorithm by fitting measurement data from our device characterization study to Equation (2).[9] The parameters c_0 to c_9 are fitting coefficients and constants to fine-tune the final polynomial equation to reflect the error characteristics of target chips, which can be determined via real-device characterization of the chips. In Equation (2), T_{ret} is the equivalent data retention time at 30°C. The specific thermal condition of 30°C in T_{ret} is needed because the impact of the retention time on the number of retention errors varies significantly depending on the data retention temperature. For convenience, we convert a data retention time $T_{ret}^{x^\circ C}$ at x°C to a data retention time T_{ret} at 30°C using the Arrhenius's Law [21, 22]. In order to find $\Delta E(B)$ at time t_{cur}, we use age(B) at time t_s and (t_{cur} - t_s) at 30°C. To find (t_{cur} - t_s) at 30°C, we read the current temperature from a thermal sensor in an SSD which is commonly adopted for internal management of an SSD such as performance throttling for thermal management [43].

Based on $E_{init}(B)$ and $\Delta E(B)$ at time t_{cur}, the total number $E(B)$ of error bits at time t_{cur} is computed by adding $E_{init}(B)$ and $\Delta E(B)$ at time t_{cur}. Based on $E(B)$, N_{retry} of the block B can be predicted by using a step function $S()$ as shown in Figure 8. The proposed N_{retry} predictor $nr(B)$, therefore, can be summarized as:

$$nr(B) = S(E_s(B) + \Delta E(B)). \tag{3}$$

Validation Methodology. To evaluate our proposed $nr(B)$, we performed comprehensive evaluations using 160 real 3D flash chips.[10] To avoid sample distortions, we divided our test samples into three groups: 60 chips for model construction, 60 chips for validating the adequacy of our model, and the other 40 chips for building a simulation environment (as will be explained in Section 6.1). In our evaluations, we carefully designed each measurement session following the test procedures of the JEDEC industry standards [38, 39, 46] for commercial SSDs. These standards specify the test methodology (e.g., a sample size or test conditions) and qualification criteria for evaluating NAND flash memory. One key recommendation for high-confidence characterization studies is to use more than 39 flash chips from 3 different wafers. Since we have used 60 flash chips from 5 wafers for designing the model, we believe that our sample size is sufficient to obtain statistically meaningful results. For model validations, we used another group of 60 chips. From each chip, to minimize potential distortion in our results, we evenly selected 16 blocks from different physical locations of the chip and tested all the pages in

[8]The retention errors shift program states to the left while the read disturbance errors shift erase state to the right. Therefore, these two error sources affect the number of bit errors independently.

[9]The next page describes the validation methodology in detail, including how we measured the RBER values of the tested blocks.

[10]In our characterization study, we used 48-layer 3D TLC flash chips from the same NAND flash manufacturer. Even though we were able to validate our new error model only for the specific type of chip (due to the limited access to real chips in academia), we strongly believe that our model (and the methodology to derive the model) can be used for a wide range of chips due to two reasons. First, our tested chips well represent modern 3D NAND flash memory because most commodity chips including SMArT/TCAT/BiCS [44, 45] have similar structures and cell types, e.g., vertical channel structures, gate-all-around transistors, and charge-trap type flash cells, thereby sharing key device characteristics. Second, we derive our model based on well-known error characteristics of NAND flash memory that are validated in a large body of prior work (using different types of chips) [19, 22], without relying on device-specific or technology-specific characteristics.

each block. To evaluate the impact of different variable combinations, we created test block samples for each combination by precisely controlling four variables of age(B) and T_{ret}. For example, to evaluate the impact of combinations of n L_{dwell} times and m T_{ret} times, we generated $n \times m$ samples, and each sample consists of 960 blocks (16 blocks × 60 chips).

Validation Results. In order to demonstrate that $nr(B)$ is an accurate predictor that predicts N_{retry} of a flash block, we present four key validation results. Figure 10(a) shows how well $nr(B)$ predicts N_{retry} of a flash block. The x-axis of the figure represents the predicted N_{retry} values $nr(B)$, while the y-axis shows a distribution of measured N_{retry} values from real flash blocks with the same $nr(B)$ value using a box plot. For a given v as a predicted N_{retry} value, a min-max plot shows a distribution of measured N_{retry} values within an interval $[min(v), max(v)]$ where $min(v)$ is the minimum measured N_{retry} value and $max(v)$ is the maximum measured N_{retry} value among the blocks with v as their N_{retry} predictor values. As shown in Figure 10(a), $nr(B)$ works properly as an on-line N_{retry} predictor of a block. The predicted N_{retry} values were very close to the measured N_{retry} values from real flash blocks. Especially, for all min-max plots shown in Figure 10(a), the difference between $min(k)$ and $max(k)$ at most 1 for all $k \geq 0$. Thus, if $nr(B_x) < nr(B_y)$, it is guaranteed that N_{retry} of $B_x \leq N_{retry}$ of B_y. That is, $nr(B)$ is sufficient to distinguish the difference in N_{retry} of flash blocks.

We also validated the $E(B)$ model from Equation (2) with the measured data under seven different conditions. Figures 10(b), 10(c), and 10(d), (e) compare the predicted $E(B)$ values with the measured ones when the data retention time was changed in young blocks (with age(B) = 0.2), moderately-worn blocks (with age(B) = 0.5), and old blocks (with age(B) = 0.8), respectively. Figures 10(f), 10(g), and 10(h) compare predicted $E(B)$ values with measured ones when the number of experienced read operations of a block was changed in young blocks (with age(B) = 0.2), moderately-worn blocks (with age(B) = 0.5), and old blocks (with age(B) = 0.8), respectively. Note that the data retention time in Figures 10(b), 10(c), 10(d), and 10(e) assumes the retention temperature of 30°C, and the read operations in Figures 10(f), 10(g), and 10(h) are fully sequential pattern. In all cases, the percentage root mean square error (%RMSE) is less than 10%, which means that $E(B)$ can accurately predict bit errors of the block B under various conditions (various $N_{P/E}$, T_{ret}, and N_{read}).

To demonstrate that our $E(B)$ model is a simple model to implement in practice with required high accuracy, we evaluated if the current $E(B)$ model includes redundant model variables. We evaluated the prediction accuracy of simpler $E(B)$ models with smaller model variables. Figure 10(d) compares %RMSE values of two simpler $E(B)$ models with the proposed $E(B)$ model based on the difference between predicted $E(B)$ values and the measured $E(B)$ values in each model. We considered two simpler models, C1 and C2, with three model variables only: (C1) the normalized $N_{P/E}$ value was used instead of age(B) value in Equation (1), and (C2) T_{ret} was not included in Equation (2). As shown in Figure 10(d), both simpler models exhibit much lower prediction accuracy over the proposed $E(B)$ model, thus demonstrating that the proposed $E(B)$ model has no redundant model variable.

5 Design of ReadGuard

The key contribution of the new block quality marker described in Section 4 is that it enables an FTL to manage its blocks based on the read latency level of each block. In this section, we describe ReadGuard, an integrated priority-aware flash management scheme based on our new block quality marker. A key design requirement of ReadGuard is to support the read performance differentiation in proportion to the app priority without affecting the performance/lifetime of a ReadGuard-enabled SSD. Figure 11 shows an overall organization of an FTL, called rgFTL, that employs the proposed ReadGuard scheme.

It is challenging for an FTL to determine whether an I/O request is latency-sensitive or not based on the limited information. As a more practical and straightforward solution, we assume that an app determines its own I/O priority based on a better understanding of I/O responsiveness. For passing I/O priority information from apps to the FTL via the kernel I/O stack, we modify the Linux kernel's process control block to keep the I/O priority

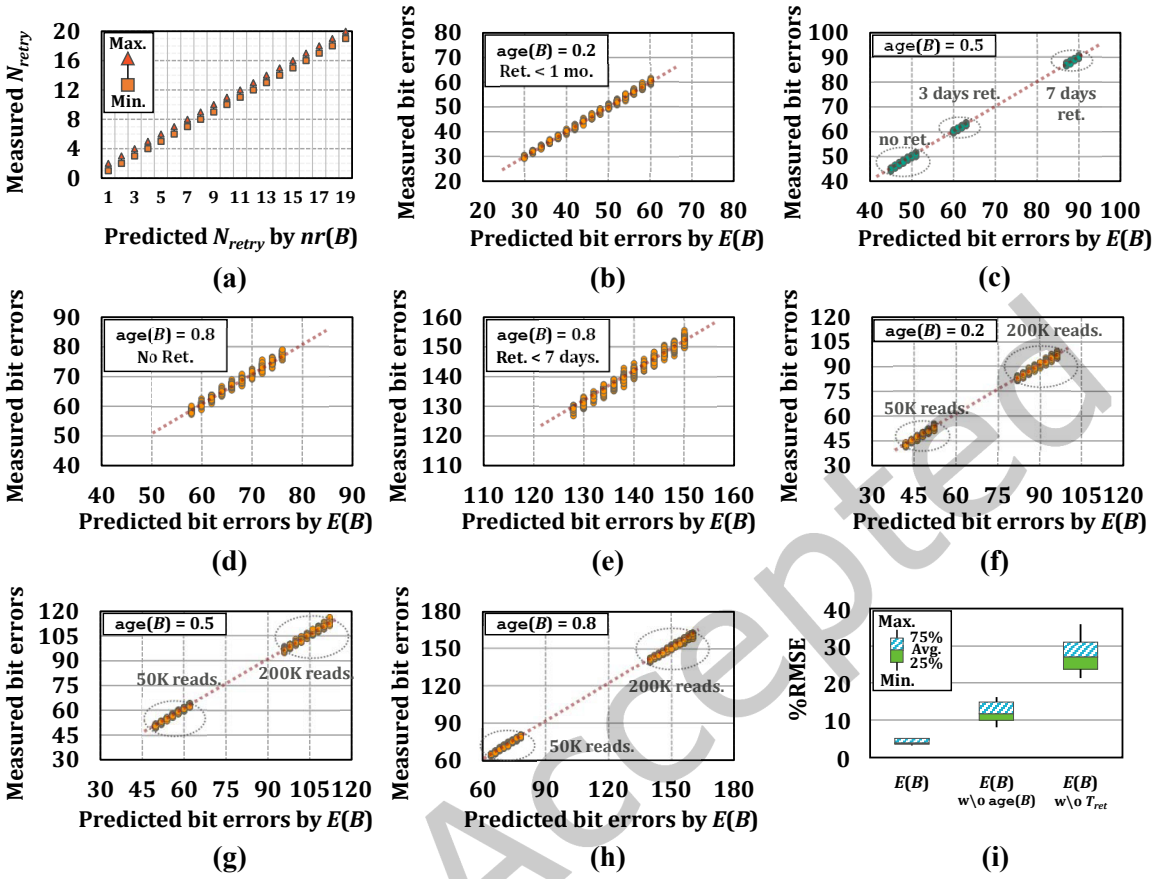

Fig. 10. Block quality function validation results.

value and employ NVMe's queueing-based I/O priority feature. During the initialization phase, an app sets its I/O priority value using a custom API. When an app issues an I/O request, the modified kernel I/O stack (block layer and NVMe driver) retrieves the app's I/O priority value from the process control block and inserts the I/O request into the appropriate NVMe queue based on the app's priority value. The host interface layer in the FTL then prioritizes the I/O request based on the priority level of the NVMe queue from which it comes.

As shown in Figure 11, the proposed `rgFTL`, which is based on an existing page-level FTL, has three key modules for supporting `ReadGuard`: the block grader (BGR), the priority-aware block manager (PBM), the WAF monitor (WM), and the extended suspend/resume arbiter (ESRA). The BGR module keeps track of the block quality $nr(B)$ of a block B. To this end, it manages an extended block status table (eBST) that stores parameters that are related to the block quality function. The PBM module is in charge of matching apps' priority with the quality of allocated blocks. To manage free blocks and used blocks based on their $nr(B)$ values, the PBM module uses `freePool` and `usedPool` (which work as a typical priority queue). The block allocator (BA) of the PBM module ensures that higher-priority apps use higher quality blocks over lower-priority apps at the initial block allocation time. The block quality monitor (BQM) of the PBM module monitors if there is block-quality inversion among allocated blocks for apps with different priorities. When there is block-quality inversion, the BQM module

resolves it by migrating data from quality-inverted blocks. The ESRA module preempts an ongoing operation when the operation conflicts with an incoming read request on the same flash chip.

In order for `rgFTL` to control the amount of additional writes by the BQM module (so that it cannot negatively affect the SSD lifetime), `rgFTL` dynamically changes the condition of block-quality inversion depending on the accumulated amount of writes by the BQM module. The WM module monitors the amount of extra writes from the BQM module and computes its proportion in the total amount of writes for SSD-internal management tasks (e.g., a garbage collector and a wear leveler). When the extra writes from the BQM module exceed the current upper bound, the WM module changes the condition of block-quality inversion to be more strict. If the extra writes from the BQM module are smaller than the current upper bound, the WM module modifies the condition of block-quality inversion to be easier to meet. (See Section 5.3.)

5.1 Block Grader

Algorithm 1 shows how the BGR module grades blocks by exploiting the proposed block-quality model in Section 4. To accurately estimate $nr(B)$ of a block B, the BGR module needs the following five values, the number $N_{P/E}$ of program/erase cycles, the data retention time T_{ret}, the number of reads N_{read}, the average length L_{dwell} of time intervals between successive erase operation, the erase latency $tERASE$, and the number E_0 of bit errors immediately after program[11], for each block. $N_{P/E}$ is simple to manage because it increments by one whenever the block B is erased. At time t, T_{ret} of the block B is estimated by $(t - t_0)$ where t_0 is the time when the block B was most recently programmed with its first page. The BGR module resets t_0 for each block B after the block B is erased. The BGR module modifies L_{dwell} whenever a block is erased using T_{ret} and the current temperature from a thermal sensor. Since the remaining two parameters, $tERASE$ and E_0, that are related to `age`(B) attribute, change slowly over the number of block erasures, the BGR module tracks these values at a coarse granularity, say every 100 P/E cycles. $tERASE$ is directly measured at the flash controller, and E_0 is measured by reading back the first page of a block immediately after the first page is programmed to the block. Once new $tERASE$ and E_0 are measured for a block, the BGR module updates the `age`(B) value for the block in the eBST. When $nr(B)$ value is

[11]Our metric uses E_0 of an LSB page in the first wordline among a block.

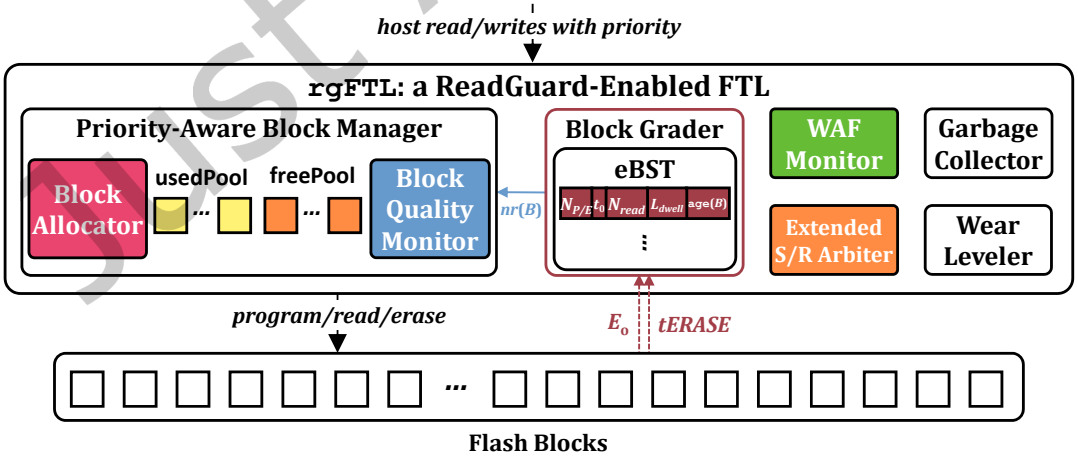

Fig. 11. An overview of the proposed `rgFTL`.

needed by `rgFTL` at time s for the block B, the BGR module returns $nr(B)$ using the current `age`(B) in the eBST, (s - t_0), and N_{read}.

In summary, the BGR module maintains five per-block parameters in the eBST, $N_{P/E}$, t_0, N_{read}, L_{dwell}, and `age`(B). However, its storage overhead is negligible because these parameters are managed at the block granularity. For example, four bytes would be sufficient to store each of the four parameters, so the extra memory of 5 MB would be sufficient for a 2 TB SSD (see Section 6.1). Although it also takes a few cycles to update parameters for every flash erase operation, the extra cycles would be negligible because the latency of erase operations is several orders of magnitude longer than the extra cycles.

Algorithm 1 Block grading

```
Initialize parameters for each block in eBST:
    N_{P/E} (Number of program/erase cycles)
    t_0 (Last programming time of block)
    N_{read} (Number of reads)
    L_{dwell} (Average time interval between erase operations)
    age(B)
function updateEraseRelatedParameters(block B, time t)
    Initialize two variables: tERASE and E_0
    Increment N_{P/E} by 1
    Update T_{ret} for B as (t − t_0)
    Reset t_0 for B after each erase operation
    Modify L_{dwell} based on T_{ret} and current temperature
    if N_{P/E} modulo 100 is 0 then
        Measure tERASE directly from the flash controller
        Measure E_0 by reading the first page of B after programming
        Update age(B) for B
    end if
    Free two variables: tERASE and E_0
end function
function updateReadRelatedParameter(block B)
    Increment N_{read} by 1
end function
function calculateBlockQuality(block B, time s)
    return Block quality nr(B) for B using age(B) in eBST, (s − t_0), N_{read}
end function
Main:
    When block B needs to be erased to serve a write request:
        Call updateEraseRelatedParameters(B, t)
    When a read request arrives to block B:
        Call updateReadRelatedParameter(B)
    When block quality nr(B) for block B is needed at time s:
        Call calculateBlockQuality(B, s)
```

Algorithm 2 Priority-Aware block management with three priorities

Initialize:
  `freePool`, `usedPool` (priority queues ordered by $nr(B)$)
  $\texttt{subP}_{high}$, $\texttt{subP}_{mid}$, $\texttt{subP}_{low}$ (subpools in `freePool` for each priority level)
  μ_{high}, μ_{mid} (per-priority inversion margins)
  GC_{thr} (GC threshold)
function ALLOCATEBLOCK(app τ_i, priority p_i)
  Allocate block B from $\texttt{subP}_i$
  Delete block B in $\texttt{subP}_i$
  Insert block B to `usedPool`
  if Number of free blocks in $\texttt{subP}_i \leq GC_{thr}$ **then**
    Invoke garbage collection for $\texttt{subP}_i$
  end if
  Update $\texttt{subP}_i$ pointers
end function
function DETECTINVERSION(priority p_i, priority p_j)
  Initilize allInversionIsResolved to *FALSE*
  // Detect all possible block-quality inversion
  while allInversionIsResolved is *FALSE* **do**
    Find the worst block B_w of τ_i and the best block B_b of τ_j
    if $nr(B_w)$ - $nr(B_b) > \mu_i$ **then** Call RESOLVEINVERSION(B_w, B_b)
    else
      Set allInversionIsResolved to *TRUE*
    end if
  end while
end function
function RESOLVEINVERSION(block B_{low}, block B_{high})
  // B_{low} is a lower-quality block and B_{high} is a higher-quality block
  if $\texttt{age}(B_{high}) < \texttt{age}(B_{low})$ **then**
    Migrate valid data of B_{high} to a new block in low-priority free pool $\texttt{subP}_{low}$
    Move B_{high} to high-priority free pool $\texttt{subP}_{high}$
  else
    Refresh valid data of B_{low} to a new block
    // T_{ret} and N_{read} are reset to 0
  end if
end function
Main:
  When a τ_i with p_i requires a free block:
    Call ALLOCATEBLOCK(τ_i, p_i)
  Every predefined interval:
    Call DETECTINVERSION(p_{high}, p_{mid})
    Call DETECTINVERSION(p_{high}, p_{low})
    Call DETECTINVERSION(p_{mid}, p_{low})

5.2 Priority-Aware Block Manager

The main role of the PBM module, which is the core component of `ReadGuard`, is to guarantee that blocks are managed so that the priority order of apps is ensured. Algorithm 2 shows how the PBM module manages blocks based on I/O priorities of apps. The PBM module maintains two ordered queues, `freePool` and `usedPool`, of free blocks and used blocks, respectively. Both `freePool` and `usedPool` are ordered by $nr(B)$ values. When the BA module of the PBM module allocates a free block to an app τ_i with priority p_i, it first searches for a proper free block (from `freePool`) that meets the priority order with the other apps. To make the search process more efficient, when N different priority levels are supported, blocks in the `freePool` are grouped into N subpools, $\texttt{subP}_0$, ..., $\texttt{subP}_{N-1}$, where blocks in $\texttt{subP}_i$ have higher quality over ones in $\texttt{subP}_j$ if $i < j$. When τ_i with p_i needs a free block, the BA module requests a free block from $\texttt{subP}_i$. Since each subpool covers a sequence of contiguous free blocks (that were sorted by their $nr(B)$ values), the BA module maintains one pointer for each subpool that points to the starting location of each subpool within `freePool`. Whenever a block is consumed from a subpool or a new block is added to `freePool`, subpool pointers are updated.

Although the BA module honors the priority order of apps when a block is initially assigned to a requesting app, it cannot completely prevent block-quality inversion because the quality of the allocated block dynamically changes. The BQM module is responsible for detecting block-quality inversion of blocks in `usedPool` during run time. In order to manage the number of extra writes for resolving block-quality inversions, we introduce two additional variables in deciding if block-quality inversion occurs between blocks. Per-priority inversion margins, μ_{high} and μ_{mid}, are used in deciding block-quality inversion in τ_{high} and τ_{mid}, respectively. In τ_{high}, when $nr(B_w)$ of the worst block B_w of τ_{high} is larger than $nr(B_b)$ of the best block B_b of τ_{mid} at least by μ_{high}, there exists block-quality inversion between τ_{high} and τ_{mid}. Similarly, in τ_{mid}, when $nr(B_w)$ of the worst block B_w of τ_{mid} is larger than $nr(B_b)$ of the best block B_b of τ_{low} at least by μ_{mid}, there exists block-quality inversion between τ_{mid} and τ_{low}. Note that the number of detected quality-inverted blocks is strongly dependent on two per-priority inversion margins. When the margins are set to a small value (e.g., 1), more block pairs may satisfy the condition of block-quality inversion. On the other hand, when the margins are set to a large value (e.g., 10), fewer block pairs can meet the condition. By dynamically adjusting μ_{high} and μ_{mid}, we control the number of quality-inverted blocks, thus managing the amount of extra writes needed for resolving quality-inverted blocks within an acceptable limit.

To find block-quality inversion, every predefined interval[12], the BQM module checks if the difference of $nr(B)$ values between the worst used block for τ_{high} and the best used block for τ_{mid} exceeds μ_{high}. Similarly, the same check is performed between τ_{mid} and τ_{low}. When the BQM module identifies two quality-inverted blocks, B_{high} and B_{low}, it resolves them by one of two methods, one based on $\texttt{age}(B)$ and the other based on T_{ret} and N_{read}. We assume that B_{high} is a higher quality block allocated for a lower-priority app τ_{low} and B_{low} is a lower quality block allocated for a higher-priority app τ_{high}.

As the first method, we check if there is inversion in $\texttt{age}(B)$ values of B_{high} and B_{low}. That is, we check if $\texttt{age}(B_{high}) < \texttt{age}(B_{low})$. This type of block-quality inversion is possible, for example, when a high-priority write-intensive app τ_{high} and a low-priority write-once app τ_{low} run together. If a block B_{high} were allocated to τ_{low} a long time ago, its $\texttt{age}(B_{high})$ value could be lower than that of a block that was recently allocated to τ_{high} because τ_{high} tends to quickly increase $\texttt{age}(B)$ values of its allocated blocks because of their frequent P/E cycles. We resolve the first type of block inversion by moving B_{high} to a free pool $\texttt{subP}_{high}$ of a high-priority app τ_{high} so that future block-quality inversion can be avoided by allocating B_{high} for a high-priority app τ_{high}. Furthermore, since data in B_{high} are moved to a free block B' in a free pool $\texttt{subP}_{low}$ of a low-priority app τ_{low}, $nr(B')$ should be larger than $nr(B_{high})$, thus mitigating future block inversion between τ_{high} and τ_{low}.

[12]Since N_{retry} of a block increases slowly by T_{ret}, we set default interval length by seven days at 30°C.

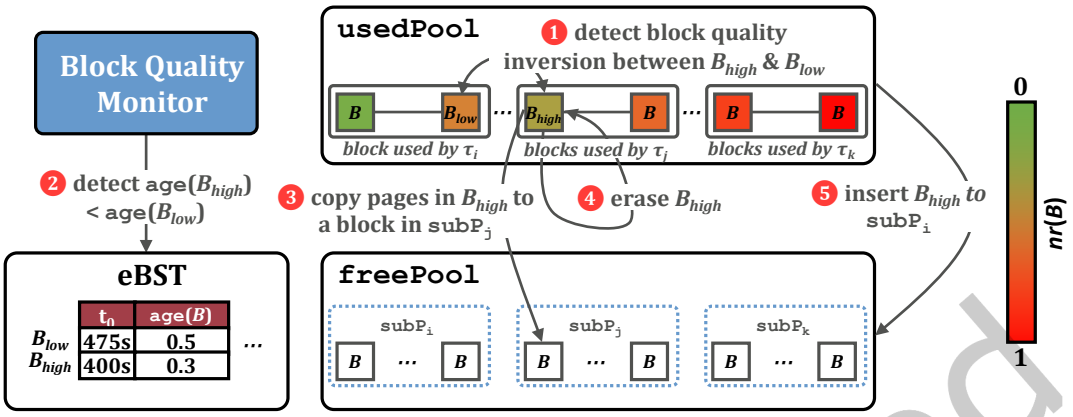

Fig. 12. An example of age(B) inversion handling.

Figure 12 illustrates how the BQM works when inversion in age(B) values occurs using a high-priority app τ_i and a low-priority app τ_j. The BQM module first detects block-quality inversion between B_{high} and B_{low} (❶) and finds age(B_{high}) < age(B_{low}) by referencing the eBST (❷). When the BQM module detects inversion in age(B) values, the BQM module moves data of the block B_{high} to a free block in subP$_j$ (❸ in Figure 12) and erases the block B_{high} (❹). After that, the BQM module forces the block B_{high} to be moved to subP$_i$ so that it can be used for a higher-priority app in a future block allocation time (❺).

If the first method cannot be applied, age(B_{high}) should be greater than or equal to age(B_{low}). Therefore, T_{ret} of B_{low} should be much larger than that of B_{high} if B_{high} and B_{low} could meet the condition of quality-inverting blocks. We resolve the second type of block inversion by refreshing (i.e., rewriting to the same block) data in B_{low}. Since T_{ret} of B_{low} resets to zero, the block quality of B_{low} should be better than that of B_{high}. Note that to avoid two types of block-quality inversion during run time, it is inevitable to move data between the affected blocks, thus possibly degrading the SSD lifetime. To avoid the SSD lifetime from being degraded by too many data movements from the BQM module, we need an intelligent mechanism to control the amount of data movements from the BQM module.

5.3 WAF Monitor

The main role of the WM module is to periodically monitor the write overhead from the BQM module and to properly adjust two run-time margin variables, μ_{high} and μ_{mid}, that are used in detecting quality-inverted blocks in τ_{high} and τ_{mid}, respectively. By modifying μ_{high} and μ_{mid} during run time, rgFTL can achieve high differentiation in read performance among apps with different priorities with little degradation on the SSD lifetime. Each margin variable is represented as a normalized number between 0 and 1 over the maximum observed N_{retry} value. In the current version of rgFTL, we allow the write overhead of the BQM module to be less than 10% of the total SSD-internal writes that are required for managing an SSD.[13]

In adjusting μ_{high} and μ_{mid}, the WM module considers the proportion p of additional writes by the BQM module over the total amount of writes for managing an SSD. The initial values of both margins are set to 0.1 so that the BQM module equally differentiates the read performance among three priorities. The WM module checks

[13] Although rather arbitrary, 10% was selected based on our observations from device characterization study. For all the tested flash blocks from 110 flash chips, they were still alive when their P/E cycles reached 110% of the maximum allowed P/E cycles.

the current p value at every monitoring interval.[14] When p is larger than 10%, the WM module increases the margin variables by 0.1 so that fewer blocks are detected as quality-inverting blocks, thus decreasing the write overhead from the BQM module. On the other hand, when p is smaller than 10%, the WM module decreases the margin variables by 0.1 so that more blocks can be detected as quality-inverting blocks, thus better differentiating read performance among apps with different priorities.

5.4 Extended Suspend/Resume Arbiter

The ERSA module extends an existing read-over-program/erase preemption mechanism [26] to support read-over-read preemption. Since the PBM module fully manages flash blocks in a priority-aware fashion in `rgFTL`, when a higher-priority read should wait for a lower-priority read to complete, the higher-priority read may suffer from an excessive delay because the lower-priority read is likely to be serviced by a block with the long read latency. A straightforward solution to this problem may be to immediately suspend the ongoing read command when a higher-priority read is issued. However, supporting an immediate read preemption mechanism is quite challenging because 1) a flash chip should be modified to support read suspend/resume commands, and 2) large hardware resources are required for saving the transient read-internal states when suspended and restoring the save states for resuming the suspended read.

In the ESRA module, we employ a lazy suspension mechanism when a high-priority read is issued while a low-priority read is ongoing. The low-priority read is allowed to complete the current read step. However, if the low-priority read requires a read retry step because of a failed read, the next read retry step is suspended, thus being preempted for the high-priority read.

The current design of the ESRA module minimizes the amount of saved read-internal states when a read command is preempted because only V_{ref} of the last failed read step needs to be saved. Furthermore, a maximum delay of one read-retry step is acceptable for higher-priority reads because it causes a marginal increase in *tREAD*. Figure 13 illustrates how `rgFTL` handles two reads with different priorities with and without the ESRA module.

[14]To observe a steady WAF value, we defined a monitoring interval as the time it takes to perform a sufficient number (e.g., 5% of total blocks) of garbage collections (GC).

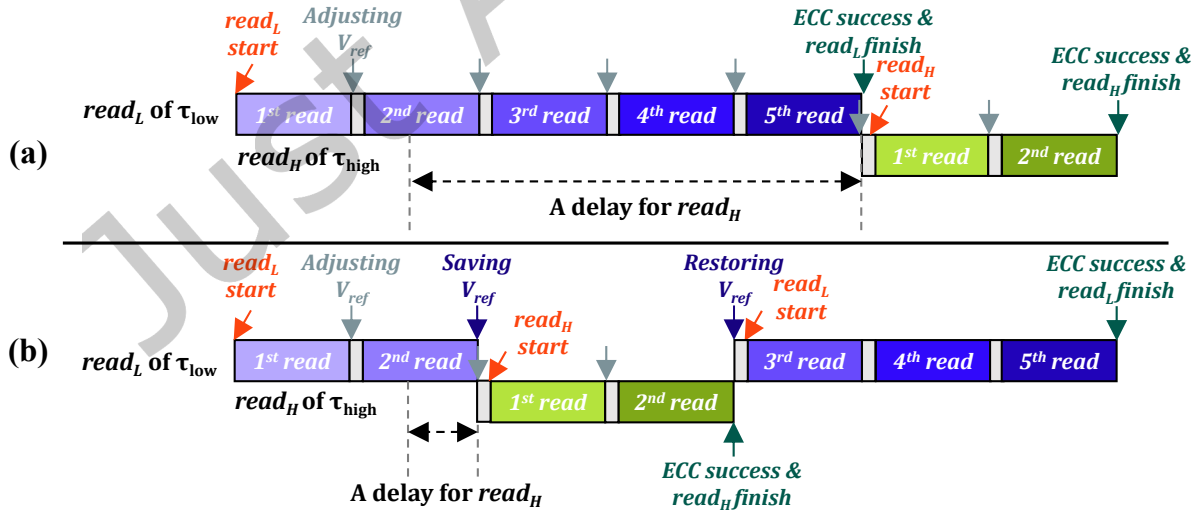

Fig. 13. An illustrative comparison of (a) `rgFTL` without the ESRA module and (b) `rgFTL` with the ESRA module.

As shown in Figure 13(a), without the ESRA module, a read command $read_H$ of τ_{high} must wait until the ongoing read command $read_L$ of τ_{low} that needs four steps V_{ref} adjustment (i.e., $N_{retry} = 4$) to complete. In contrast, as shown in Figure 13(b), the ESRA module waits until the ECC correction of the current $read_L$ is complete. If the ECC fails, suspends $read_L$ with its last used V_{ref} saved and issues $read_H$ first. After $read_H$ is completed with two read steps, the ESRA module restarts $read_L$ with the stored V_{ref} where its read step was suspended. Since the FTL decides if read retries are needed (by checking if the ECC succeeds or fails), the ESRA module requires no change to NAND flash chips. It requires only tiny DRAM space to store V_{ref} values (two V_{ref} values for τ_{high} and τ_{mid}, respectively, for supporting three priorities) of suspended read commands at the FTL level.

5.5 Other FTL Modifications

In `rgFTL`, a garbage collector and a wear leveler need to be modified because these modules indirectly affect how blocks are allocated to different apps. We design the garbage collector and the wear leveler in such a way that as a result of garbage collection and wear leveling, the number of block swap operations by the BQM module does not significantly increase.

Changes in Garbage Collector. The garbage collector of `rgFTL` is triggered based on the number of free blocks in each subpool instead of the number of total free blocks in a conventional FTL. Although there exists a large number of free blocks in `freePool`, when the number of free blocks in $\texttt{subP}_i$ is less than a threshold value[15], the garbage collector is invoked in the background and reclaims free blocks from `usedPool`. Each reclaimed free block B is inserted to a proper subpool (not necessarily to $\texttt{subP}_i$) according to its $\texttt{age}(B)$. When B is not inserted to $\texttt{subP}_i$, the BA module rearranges its subpools so that $\texttt{subP}_i$ gets a new free block. In order to avoid block-quality inversion during the garbage collection, when the garbage collector moves the valid pages of a selected victim block B_v to a target block B_t, B_t should be in the same priority group as B_v. That is, if B_v was allocated to τ_k, B_t must also be allocated to τ_k or must be from $\texttt{subP}_k$.

Changes in Wear Leveler. A common heuristic used in a wear leveler is to move write-hot data to a more reliable block [47–49]. However, if the wear leveler migrates data in a priority-oblivious fashion, data migrations by the wear leveler can cause block-quality inversion. For example, consider two apps τ_i and τ_j where the priority of τ_i is higher than that of τ_j. If τ_j is write-intensive, a wear leveler may be invoked so that write-hot data of τ_j can be moved to a more reliable block B_d (i.e., with a smaller $\texttt{age}(B)$). If the destination block B_d belonged to a higher priority group (e.g., in $\texttt{subP}_i$), the BQM module might trigger a block swap operation in the near future to prevent block-quality inversion from B_d. The block-quality inversion occurs because the block quality of B_d, which was allocated to a lower-priority app τ_j, can be better than that of the lowest quality block of a higher-priority app τ_i.

To avoid such successive block swap operations, the wear leveler of `rgFTL` employs an intra-priority mode as default. In the intra-priority mode, the wear leveler tries to minimize the difference in wear status of blocks that belong to the same priority group (i.e., used blocks for the same τ_i). Compared to a conventional wear leveler, the intra-priority mode alone may not be effective in leveling the wear status of the most-worn blocks when the data hotness is quite different depending on apps. For such rare cases, the wear leveler of `rgFTL` supports the inter-priority mode, which is invoked when the maximum difference in the wear status of all the blocks exceeds a threshold value. In the inter-priority mode, the wear leveler works in a conventional way, considering the wear difference of all the blocks, not the blocks only in the same priority group. Inevitably, the inter-priority mode will introduce additional block-quality inversion. However, if this inversion is quickly fixed by the BQM module, the wear leveler cannot reduce the maximum wear difference. Therefore, in the inter-priority mode, we tentatively

[15]We set the threshold value higher than the value that can maximize the internal parallelism of the SSD. That is, the modified garbage collector aims to maintain the number of free blocks in each `subP` as sufficient for composing a superblock.

disable the BQM module so that less reliable blocks can hold cold data longer, thus reducing the maximum wear difference among all the blocks.

6 Evaluation

6.1 Evaluation Methodology

Simulation Setup. To evaluate the effectiveness of `rgFTL`, we used an extended version of MQSim [11], a multi-queue SSD simulator with NVMe interface support. We extend MQSim in two directions to faithfully model a modern NAND flash-based SSD. First, we extended the NAND flash chip model of MQSim to simulate more realistic behavior of NAND flash blocks, based on real-device characterization of 40 3D TLC NAND flash chips. We modified the metadata structure of each simulated block to include multiple N_{retry} lookup tables that are indexed by a P/E-cycle interval and a data retention time. In the simulation setup stage, we randomly assigned a real characterized block to each simulated block and initialized the simulated block's N_{retry} lookup tables based on the characterization results of the real block. When simulating a read request to a page, the extended MQSim first queries the lookup tables with the current P/E cycle and retention time of the corresponding block and performs read retry operations N_{retry} times. Second, we modified the transaction scheduling unit of MQSim to support a read-over-read preemption mechanism as well as an existing read-over-program/erase preemption mechanism [26].

Table 2 summarizes the configurations of our evaluated SSD, which mimics a modern high-performance SSD. We configured the target SSD to have 2 TB capacity with eight channels, each of which has four 3D TLC flash chips. Each chip has four planes, and each plane consists of 1,888 blocks. Each block comprises 576 16 KB pages. We set flash operation timing parameters for *tREAD* (without read retry), *tPROG*, and *tERASE* to 45 μs, 400 μs, and 3.5 ms, respectively. We set the host interface to support a maximum bandwidth of 8.0 GB/s as specified by the PCI Express (PCIe) 4.0 standard [50]. A flash channel's I/O bandwidth can support 1.6 GB/s peak bandwidth, which is sufficient to support the host interface's peak bandwidth of eight channels.

To evaluate the effectiveness of `ReadGuard`, we built a `ReadGuard`-enabled FTL, `rgFTL`, and compared it to three different FTLs: `baseline`, `rgFTL`$^{--}$, and `rgFTL`$^{-}$. `Baseline` employs two features of modern priority-aware FTLs: priority-aware transaction scheduling [12] and read-over-program/erase preemption [26]. The transaction scheduler of `baseline` uses a strict priority-queuing mechanism. Chip-level queues are assigned a fixed priority order based on their priority level/request type and ready operations are dispatched to the target chip in the strict priority order of their corresponding queues. The other three FTLs, `rgFTL`$^{--}$, `rgFTL`$^{-}$, and `rgFTL`, are based on `baseline`. They employ priority-aware transaction scheduling with the read-over-program/erase preemption mechanism. `RgFTL`$^{-}$ works in the same way as `rgFTL` except that no read-over-read command preemption is supported in the ESRA module. Our objective in comparing `rgFTL` with `rgFTL`$^{-}$ is to evaluate the effectiveness of each approach: the write-side optimization by the PBM module and the read-side optimization by the ESRA module. `RgFTL`$^{--}$ works equally to `rgFTL` but does not support the block grader module. Instead of relying on the BGR module's predicted block quality, `rgFTL`$^{--}$ uses the most recent N_{retry} values for each

Table 2. Evaluated SSD configurations.

Configuration	2-TiB total capacity; 8 channels; 4 dies/channel; 4 planes/die; 1888 blocks/plane; 576 pages/block
Latencies (μs)	*tREAD* = 45; *tPROG* = 400; *tERASE* = 3500;
Bandwidth	8.0 GB/s external I/O bandwidth (PCIe 4.0, 4-lane); 1.6 GB/s channel I/O bandwidth

Table 3. Key I/O characteristics of six I/O traces.

Workload	Read ratio	Avg. read size (KB)	Avg. write size (KB)	Total read size (GB)	Total write size (GB)
Ali$_{2}$	0.29	20.89	13.56	111.90	276.39
Ali$_{46}$	0.38	22.28	6.45	140.96	228.48
Ali$_{81}$	0.43	28.42	10.72	50.79	66.66
Ali$_{121}$	0.92	21.55	6.51	3288.41	272.96
Ali$_{284}$	0.88	21.44	10.45	23.98	35.78
Ali$_{295}$	0.45	20.99	6.48	189.96	231.13

block to predict block quality. By comparing `rgFTL` with `rgFTL`$^{--}$, we can evaluate the effectiveness of our proposed block quality model in Section 4. Our study focuses on read performance, so all evaluated FTLs use the I-ES scheme, which can service read operations as soon as possible, among the three suspension schemes in [26]. When a read request collides with an ongoing erase operation, the I-ES scheme immediately terminates the current erase step to service the read request, and the suspended erase/program loop is resumed from the beginning.

Workloads. We conducted our experiments using six workloads obtained from large read-world I/O trace sets, AliCloud traces [51]. The AliCloud traces consist of 1,000 block I/O traces collected from a cloud block storage system over one month. From these trace sets, we carefully selected six representative traces with different read ratios and read sizes from the trace sets. Since the AliCloud traces were collected from HDD-based storage systems, we increased the host-side I/O intensity by shortening the time intervals between requests by an appropriate ratio (e.g., 1/10) to properly consider the high-performance SSD's processing speed. Using the selected six traces, we built five mixed workloads where each workload combines three traces: MixA = (Ali$_{121}$, Ali$_{2}$, Ali$_{284}$), MixB = (Ali$_{81}$, Ali$_{284}$, Ali$_{2}$), MixC = (Ali$_{295}$, Ali$_{46}$, Ali$_{121}$), MixD = (Ali$_{284}$, Ali$_{81}$, Ali$_{295}$), and MixE = (Ali$_{81}$, Ali$_{121}$, Ali$_{46}$). In each workload, the first trace mimics the highest-priority app, τ_{high}, the second trace mimics the mid-priority app, τ_{mid}, and the third trace mimics the lowest-priority app, τ_{low}.

6.2 Performance Evaluation

Read Performance Differentiation. We first evaluated the effectiveness of `rgFTL` in supporting differentiated read performance among apps based on their priorities. Figure 14 shows the average SSD-level read latency L_{avg} for each app in the five mixed workloads. We used three SSD configurations with different average block P/E cycles (as described in Section 3.2). All the measurements were normalized to the minimum host-side read latency (i.e., an ideal scenario where neither read retry nor waiting time exists).

We make three key observations from Figure 14. First, `rgFTL` clearly differentiates L_{avg} over each app's I/O priority in all the test scenarios, whereas `baseline` fails to do so in most scenarios. For the highest app τ_{high}, `rgFTL` provides 48.9% (25.1%), 62.2% (29.9%), and 57.1% (36.4%) shorter L_{avg} compared to τ_{low} (τ_{mid}) in the young, adult, and old SSDs, respectively. In contrast, `baseline` causes longer L_{avg} for τ_{high} compared to lower-priority apps in many cases. For example, in MixA, L_{avg} of τ_{high} 42.9%, and 30.7% higher than that of τ_{mid}, and τ_{low}, respectively. This is because the significant amount of read operations of τ_{high} in MixA, incurs a large amount of read-disturbance induced errors for read requests of τ_{high}, resulting in the large N_{retry}. `rgFTL` effectively differentiates L_{avg} with I/O priority, even with read-intensive τ_{high}.

Second, thanks to the capability of differentiating the read latency across apps, `rgFTL` significantly reduces the read latency of higher-priority apps by trading the read latency of lower-priority apps. Compared to `baseline`, `rgFTL` decreases L_{avg} for τ_{high} by 48.3% while increasing L_{avg} for τ_{mid} and τ_{low} by 22% and 45.3%, respectively, in

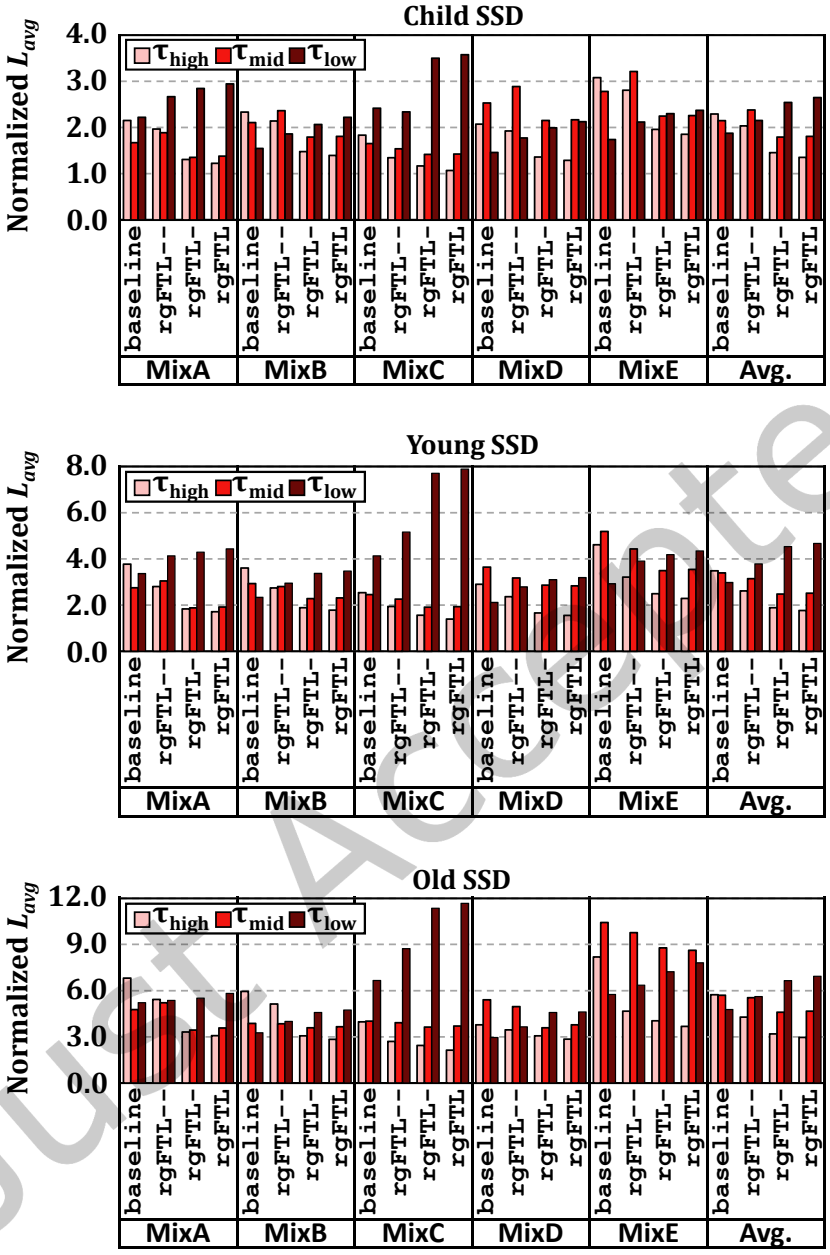

Fig. 14. Comparisons of normalized average host-side latency L_{avg} values for five workloads on three SSDs.

the old SSD. This clearly shows that `rgFTL` enables better utilization of large-capacity SSDs shared by multiple

applications, providing higher QoS performance for latency-sensitive apps by sacrificing the performance of lower-priority apps (likely less latency-sensitive).

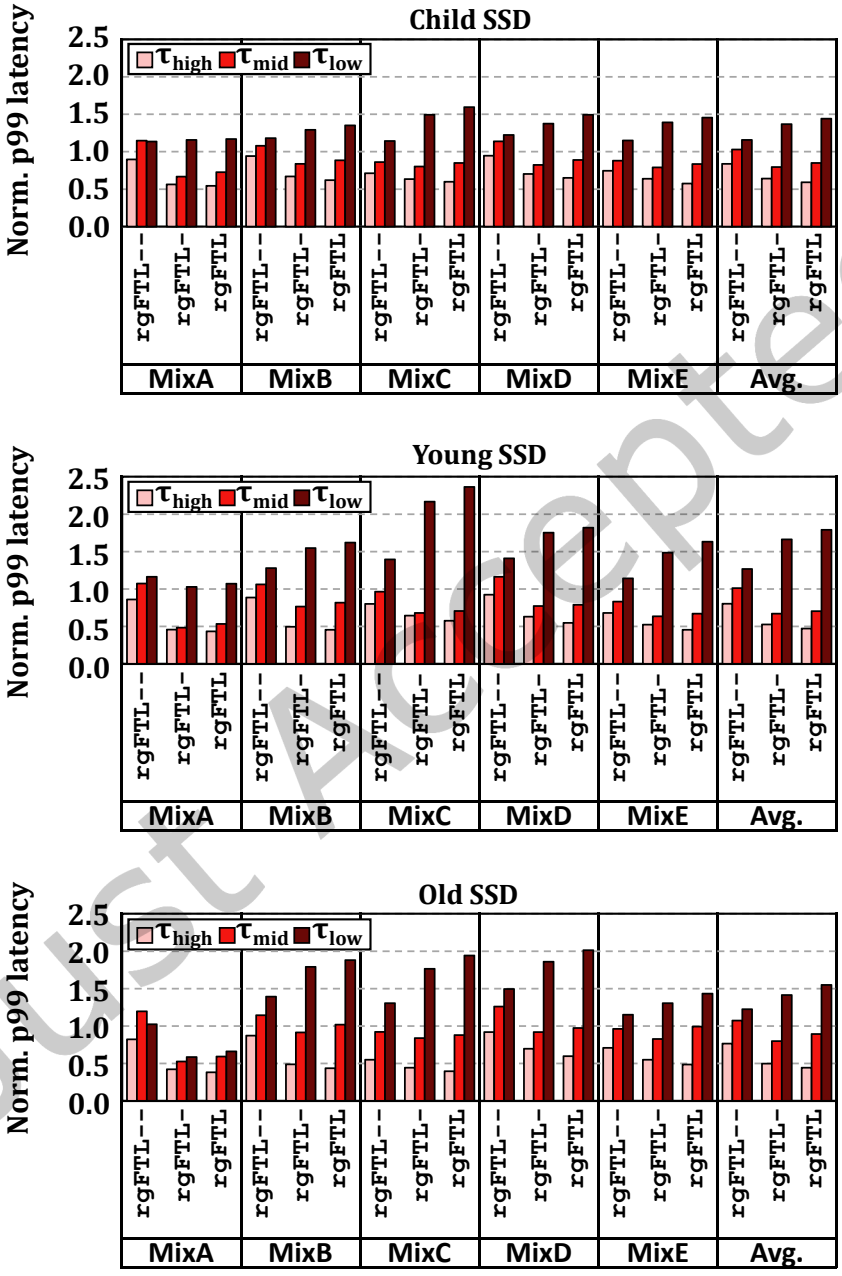

Fig. 15. Comparisons of normalized p99 host-side latency values for five workloads on three SSDs.

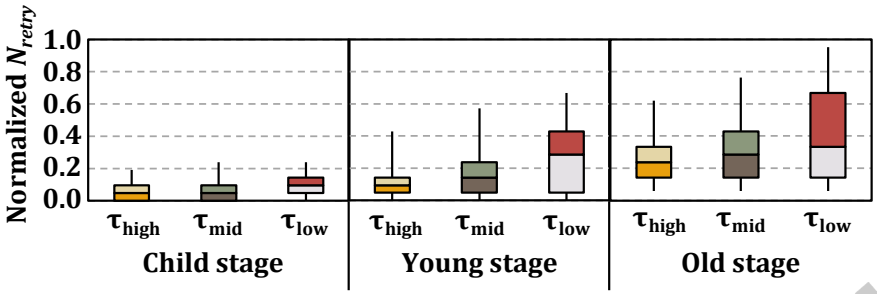

Fig. 16. Per-app block-level N_{retry} variations in rgFTL.

Third, read-over-read preemption provides considerable benefits for rgFTL to further improve the performance of higher-priority apps. While rgFTL⁻ (without read-over-read preemption) also significantly outperforms baseline in all test scenarios, rgFTL further reduces L_{avg} of τ_{high} compared to rgFTL⁻, especially when 1) the SSD gets aged, and 2) the workload is read dominant (e.g., 12.2% reduction for MixC, which τ_{low} is highly read-intensive, in the old SSD). This is because read retry can significantly increase the read latency as the block gets old, which, in turn, can cause a high-priority read to be blocked for a long time by a lower-priority read in rgFTL⁻.

Read Tail Latency. We evaluate the impact of ReadGuard on the tail latency of SSD read requests, which is a critical performance factor to many data-intensive applications [8, 52–56]. Figure 15 compares the 99th-percentile read latency (p99) of each app. All values in Figure 15 were normalized to the corresponding p99 value of each app in baseline.

We make two key observations from Figure 15. First, rgFTL significantly reduces the tail latency of τ_{high} compared to baseline, by 40.9%, 52.7%, and 55.5% on average in the young, adult, and old SSDs, respectively. Although suspended reads increase the tail latency of τ_{low}, latency-sensitive apps have stricter QoS requirements than throughput-oriented apps. Therefore, rgFTL can be a practical solution to meet the tail latency requirements of modern storage systems. Second, while effective in read-performance differentiation, rgFTL⁻⁻ is ineffective in reducing tail latency. The tail latency improvement of rgFTL⁻⁻ over baseline is only 16.3%, 19.7%, and 23.5% on average across all workloads in the young, adult, and old SSDs, respectively. This is because the recent N_{retry} value of a block is easily outdated (e.g., only one week retention time [24]) in modern NAND flash memory, thus rgFTL⁻⁻ cannot prevent large N_{retry} for cold data of τ_{high}. This shows that the N_{retry} prediction failure significantly affects SSD read tail latency, therefore, an accurate block-quality model is necessary to satisfy the QoS requirements of modern SSDs.

***N_{retry}* distributions.** To better understand how rgFTL outperforms baseline, we evaluate how rgFTL changes N_{retry} in apps with different I/O priorities. For this evaluation, we use the same workload in Figure 7. Figure 16 shows N_{retry} values per read for three apps in MixA. We normalize all values in Figure 16 to the maximum N_{retry} value. We observe that the higher the app's I/O priority, the lower the N_{retry} value. This clearly shows that the PBM module in rgFTL successfully manages block quality (i.e., N_{retry}) in a read-latency-centric fashion over the app priority, whereas a large block-quality inversion occurs in baseline (cf. Figure 7). Although the retention time of most blocks used for τ_{high} is quite long (with few update requests), the read latency of τ_{high}'s blocks was managed to be much lower than that of τ_{mid} and τ_{low}.

Based on our observations, we conclude that ReadGuard is an effective solution to better meet I/O performance requirements in modern computing systems where multiple apps with different I/O priorities share a large-capacity SSD.

6.3 Comparison to Prior Work

Prior works attempt to mitigate the overhead of frequent read-retry operations in modern NAND flash memory by 1) minimizing N_{retry} by deciding the near-optimal V_{ref} efficiently [20, 23, 25, 57–59] and 2) reducing the latency of the read-retry operation itself [24, 60]. Because existing read-retry mitigation techniques reduce read-latency variations caused by frequent read-retry operations, their application to an FTL may reduce the efficiency of our proposal for read performance differentiation.

To evaluate the effectiveness of our proposal when combined with existing read-retry optimization techniques, we built two FTLs, `baseline+` and `rgFTL+`. Based on `baseline`, `baseline+` employs two read-retry mitigation techniques [20, 24]. `RgFTL+` is an FTL that enables `ReadGuard` based on `baseline+`. Process similarity-aware optimization [20] reduces the number of retry steps by reusing V_{ref} values from previous read-retry operations on pages with similar error characteristics to the target page. Pipelined and adaptive read-retry [24] reduces the latency of a read-retry operation by pipelining consecutive retry steps using the existing cache read command [61] and dynamically reducing the chip-level read latency by exploiting the ECC margin of the final read-retry step. In this evaluation, we characterized N_{retry} values using the process similarity-aware V_{ref} adjusting technique and constructed each simulated block's N_{retry} lookup table in both FTL. Figure 17 compares the average SSD-level read latency L_{avg} of three apps in two FTLs for five workloads on three SSDs. All the measurements were normalized to the minimum host-side read latency.

We make two major observations. First, although the applied read-retry mitigation techniques successfully reduce L_{avg} of three apps, they fail to eliminate the overhead of read-retry, thus `baseline+` fails to differentiate read performance over each app's priority in most scenarios. For example, in MixA, L_{avg} of τ_{high} is 23.3%, 25.9%, and 29.3% higher than that of τ_{mid} in young, adult, and old SSDs, respectively. Second, `ReadGuard` is still effective in differentiating L_{avg} with I/O priority and reducing L_{avg} of τ_{high}, even when an FTL adopts existing read-retry mitigation techniques. `RgFTL+` reduces L_{avg} of τ_{high} by 22.5% (5.9%), 23.4% (20.4%), and 30.1% (23.4%) on average compared to τ_{low} (τ_{mid}) in three SSDs. Additionally, τ_{high}'s L_{avg} in `baseline+` is 26.6% shorter on the old SSD than that of `rgFTL+`.

Future generation NAND flash memory is expected to increase the frequency and number of read-retry operations, even with advanced mitigation techniques, due to its high error-prone characteristics resulting from increased density. Therefore, we believe that our proposal will be quite promising in satisfying the ever-increasing demand for I/O performance of latency-critical apps in future-generation SSDs.

6.4 Intra-Block Latency Variation

The current version of ReadGuard only considers variations in the N_{retry} values between blocks. However, in modern NAND flash memory, the N_{retry} values can vary across pages within a block due to various reasons, such as read-disturbance patterns, and process variations. [19, 20, 62]. However, we believe that inter-block variation in N_{retry} values considered in ReadGuard is much more significant than intra-block variation in most operating conditions.

To validate our claim, we evaluated the N_{retry} values for both the worst and best pages in the target block for every page read request in each workload. We then quantified inter-block variation as the difference between the worst pages across all blocks, and intra-block variation as the difference between the best and worst pages in the target block when the target page is read. Figure 18 compares the maximum intra-block and inter-block variation in MixA for four SSD lifetime configurations. From the results, we observed that the maximum inter-block variation is significantly larger than the maximum intra-block variation in all initial P/E cycle settings. Furthermore, as the SSD ages, the disparity between these two variations increases. For example, when the initial P/E values are set to 3500, the difference in N_{retry} values between the best and worst blocks may reach 16, while the difference between the best and worst pages within the target block may be only up to 3. This is because (i) all

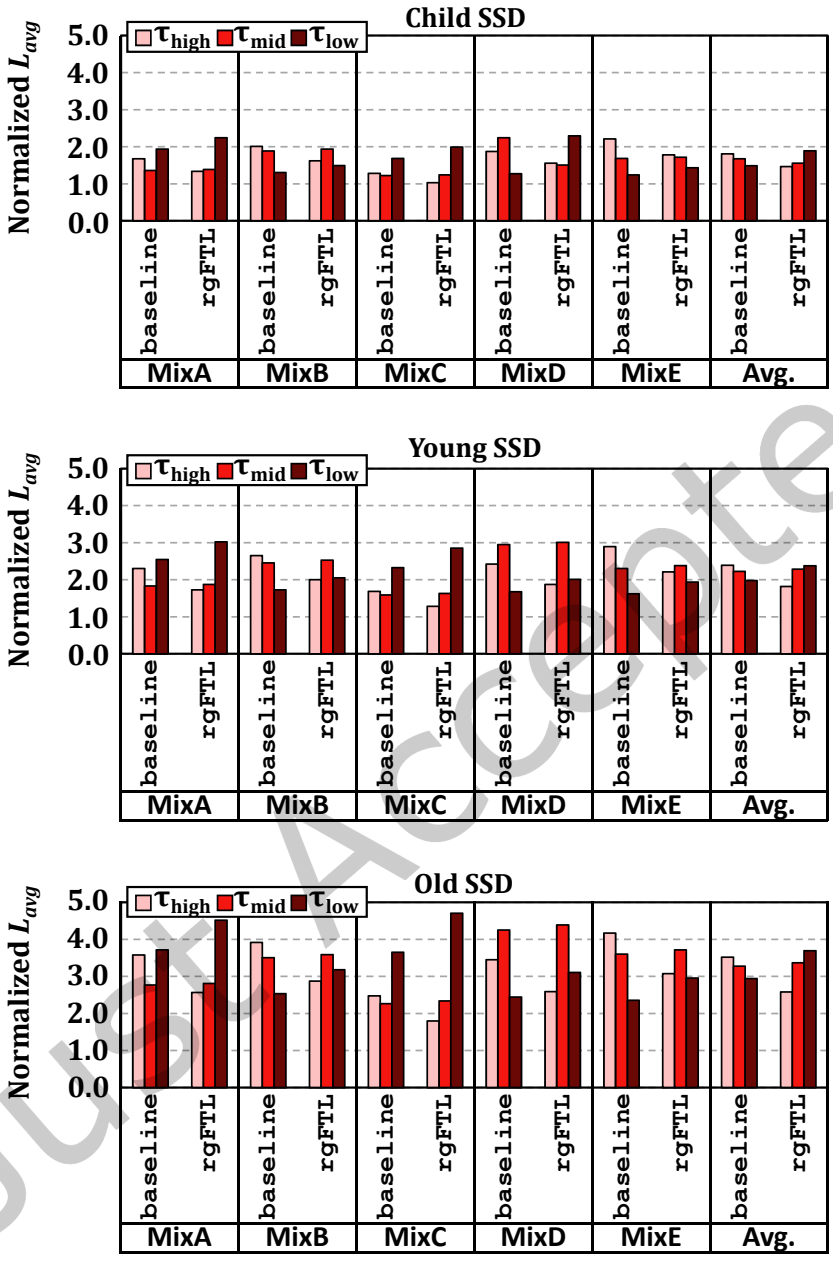

Fig. 17. The effectiveness of `ReadGuard` when combined with two existing read-retry mitigation schemes [20, 24].

pages in a block are written together in a short period and (ii) reading a page disturbs all other pages in the same block, which limits intra-block latency variation. In contrast, pages in different blocks can experience significantly

different retention times, read-disturbance effects, and block wear, thereby introducing high inter-block latency variation. As intra-block variation in the N_{retry} values is significantly smaller than inter-block variation, we concluded that ReadGuard, which focuses primarily on inter-block variation, is an efficient approach with lower overhead.

6.5 Overhead Evaluation

Impact of Block Migration Operations. To prevent block-quality inversion among apps, `rgFTL` requires to perform additional block migration operations. When the additional writes in `rgFTL` collide with user-level writes, the write latency of `rgFTL` may be degraded. Furthermore, these additional writes incur additional garbage collection invocations, thereby increasing the WAF value of SSD (i.e., more erase operations).

To evaluate the overhead of block migration operations in `rgFTL`, we measure the average SSD-level write latency of three app WAF values in each workload. Figure 19(a) compares the average write latency for five workloads in `rgFTL` and `baseline`. All values in Figure 19(a) were normalized to the average write latency in `baseline`. As shown in Figure 19(a), extra block migration overhead is marginal without affecting the SSD-level write latency. Even in the most write-intensive scenario (i.e., MixC), the average write latency of three apps was increased by 5.9% over `baseline`. Block migration operations from the BQM module do not have to be handled immediately; most block migration operations can be safely handled in the background with the lowest priority, without colliding with user write requests. A small increase in the write latency mostly comes from extra (foreground) garbage collection that is needed from additional block migration writes from the BQM module.

Figure 19(b) shows how data migration operations in `rgFTL` affect the WAF value of the old SSD. For this evaluation, the maximum proportion of additional writes in the WM module is set to 10% of the total internal writes of the SSD.[16] As shown in the figure, the increase in the WAF value in `rgFTL` is up to 7.9% compared to `baseline`. More erase operations are unavoidable in `rgFTL` because the proposed block-quality inversion management technique aims to reduce the read latency of τ_{high} by trading extra writes. However, by dynamically adjusting μ_{high} and μ_{mid} (used to detect quality-inverted blocks in τ_{high} and τ_{mid}) by the WM module, the increased WAF value can be suppressed to have a negligible impact on the lifetime of the SSD.

Impact of SSD Capacity. We analyze the impact of SSD capacity on our `ReadGuard`'s efficiency. First, we evaluate the efficiency of `ReadGuard` under varying SSD capacity (within a range of ⟨0.5 TB, 1 TB, 2 TB, 4 TB⟩). We observe only trivial variation in the reduction of L_{avg} of τ_{high} over `baseline`, e.g., less than 3% standard

[16] As explained in Section 5.3, 10% was selected based on our observations from the device characterization study.

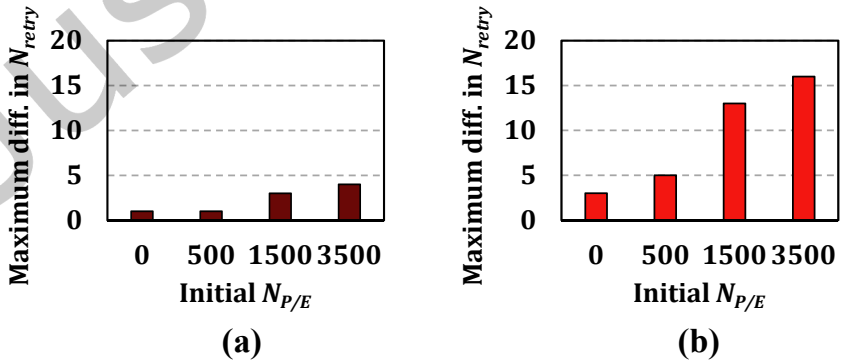

Fig. 18. Comparisons of maximum difference in (a) intra-block variation and (b) inter-block variation.

deviation across all workloads and SSD capacities. Second, we compare the space overhead of `ReadGuard` under various SSD capacity settings. The BGR module requires additional memory space to maintain five per-block parameters. Our simulated 2 TB SSD consists of 241,664 blocks (8 channels × 4 dies/channel × 4 planes/die × 1,888 blocks/plane), so the required memory space is around 5 MB. Therefore, assuming the block size is not changed, the space overhead of `ReadGuard` is proportional to the total number of blocks in SSDs, e.g., 2.5 MB for 1 TB SSD, and 10 MB for 4 TB SSD. Modern SSDs commonly have 0.1% DRAM module of total capacity (e.g., 2 GB DRAM for a 2 TB SSD), so the space overhead of `ReadGuard` is negligible. Furthermore, as the bit density of the flash chip increases, the block size is expected to increase. Table 4 shows how the size of a single block has been changed in modern 3D NAND flash memory. As shown in Table 4, the block size has increased by 2.6 times in 4 years. Therefore, we believe that the space overhead of our proposal will be negligible in future high-density SSDs composed of large blocks.

Based on the analysis, we conclude that `ReadGuard` efficiently supports read performance differentiation with low cost for wide range capacity.

Lifetime Impact. It is a reasonable concern that an SSD with `rgFTL` may have a shorter lifetime than the existing SSD because the wear leveler of `rgFTL` only focuses on flash blocks that belong to the same priority group in the intra-priority mode. To evaluate the performance of the wear leveler of `rgFTL`, we measure the $N_{P/E}$ values of flash blocks at different times while iterating MixD workload. For this evaluation, we limited the total capacity of the simulated SSD to 32 GB to enable faster experiments. The wear leveler of `rgFTL` is based on a widely used dual-pool algorithm [67]. The wear-leveling threshold value (i.e., allowed maximum difference in $N_{P/E}$ values of managed blocks) is set to 30, and the threshold value to invoke the inter-priority mode is 70.

Figure 20 visualizes the distributions of all $N_{P/E}$ values at four distinct times, t_0, t_1, t_2, and t_3. The $N_{P/E}$ values of flash blocks are sorted according to the priority of their stored data. Initially, the wear leveler of `rgFTL` operates

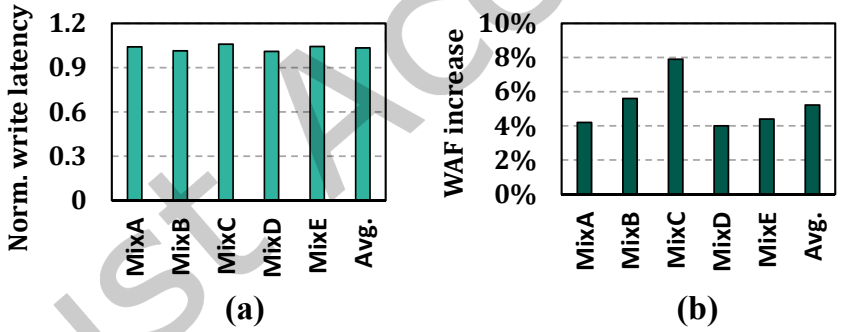

Fig. 19. Comparisons of (a) average write latency and (b) WAF values on the old SSD.

Table 4. Changes in the block size over time in modern 3D NAND flash memory.

Year	No. of pages per block	Block size (KB)
2018 [63]	1024	16384
2020 [64]	1472	23552
2021 [65]	2040	32640
2022 [66]	2816	45056

as the intra-priority mode, aimed at minimizing the difference in wear status of blocks belonging to the same priority group. At t_0, when 2.71 TB of writes have been completed, the maximum $N_{P/E}$ differences of the same priority blocks are properly managed, but the maximum difference in the $N_{P/E}$ values of all blocks exceeds the threshold value. Since the wear leveler operates as the inter-priority mode after t_0, the maximum difference of all blocks gradually decreases. The wear-leveler of `rgFTL` resumes its intra-priority mode operation once the maximum $N_{P/E}$ differences of all blocks are less than the threshold value (e.g., at t_3). Our evaluation shows that the wear leveler of `rgFTL`, which works on two levels, can adequately handle the lifetime side effect of `rgFTL`. When the maximum difference in $N_{P/E}$ values of all blocks increased due to the distinctive I/O pattern depending on apps, the inter-priority mode wear-leveling could effectively (and quickly) reduce the maximum difference in $N_{P/E}$ values of all blocks.

7 Related Work

Priority-Aware OS I/O Stack. Various techniques have been proposed at different OS I/O stack layers to differentiate I/O QoS levels [8–10, 68–71]. For example, the priority inversion problem was addressed at the file system level [68] and at the page cache level [9]. Researchers have proposed solutions at the lower layers of the I/O stack, such as the block queue layer [69–71], and at the device driver layer [8, 72]. However, these approaches have two limitations in differentiating the I/O performance of modern high-performance SSDs: (i) they do not consider the heterogeneity of flash device-level read latency, and (ii) modern high-performance host interface protocols, such as NVMe [72], bypass the I/O scheduler at the OS I/O stack to reduce latency.

I/O Scheduling at SSD Controller. Previous studies [12, 73, 74] have proposed SSD controller-level scheduling techniques for NVMe SSDs to address the lack of an I/O scheduling mechanism in the modern OS I/O stack layers. These sophisticated scheduling mechanisms were designed to ensure fairness by balancing interference

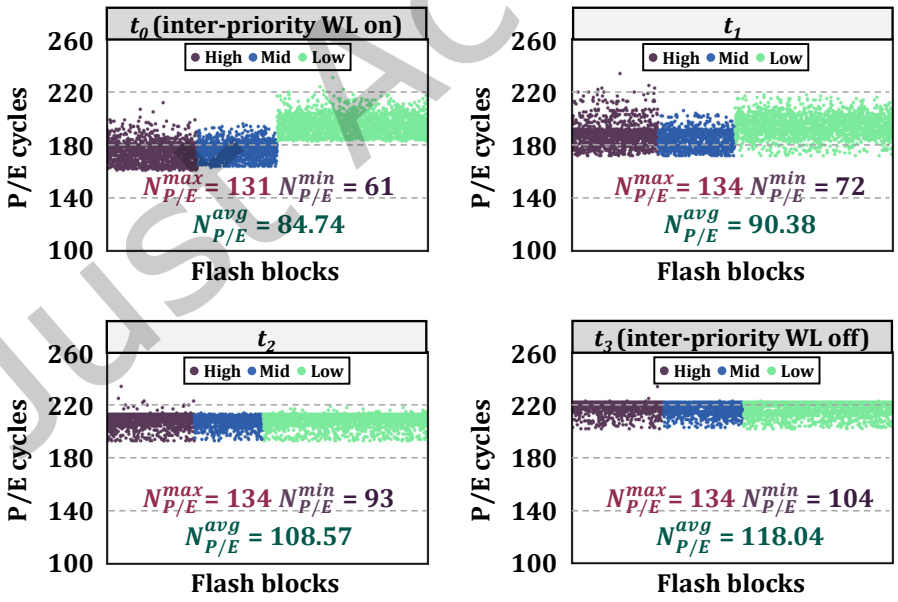

Fig. 20. $N_{P/E}$ distributions of flash blocks at four different times, t_0, t_1, t_2, and t_3.

effects between different apps in a shared SSD. Although the primary goal of these mechanisms differs from our proposal, latency-centric block management can be integrated with advanced scheduling techniques to improve their overall effectiveness. When apps with different I/O patterns share a single SSD, block-level latency variation may lead to unfairness among apps. For example, when two read-dominant apps with different I/O intensities share a single SSD, the high-intensity app unfairly slows down the low-intensity app. This is because a large number of reads from the high-intensity app cause frequent read disturbances, reducing the quality of flash blocks shared by both apps. ReadGuard can prevent unintended degradation of block-level read latency for one app due to another by (i) allocating separate free blocks to each app and (ii) managing app-level block quality to ensure fairness. For example, if an app has an average block-level read latency of 80 us when running alone, the PBM module in the shared SSD is responsible for maintaining that latency, which improves fairness.

Read Retry Mitigation. If read-retry operations are infrequent and the average N_{retry} value is less than one, our key assumption that block-level read latency varies is not valid. Therefore, existing read-retry mitigation techniques for modern NAND flash memory [20, 23–25, 57–60, 75, 76] can be considered as alternative solutions for read-performance differentiation. Unfortunately, most existing mitigation techniques are not generally applicable for this purpose. For example, Shim *et al.* [20] proposed an effective scheme that can reduce N_{retry} when horizontally adjacent wordlines within the same flash block are successively read. However, this scheme does not eliminate large variations in N_{retry} among different blocks. Li *et al.* [23, 25] proposed another mitigation technique that employs additional sentinel cells as a proxy of the error status of a flash page. However, this scheme is difficult to be used when the space constraint is tight (e.g., in cost-sensitive SSDs) because it requires about 10% more spare area per wordline to properly work. Park *et al.* [24] proposed a pipeline and adaptive read-retry scheme that reduces read-retry latency by utilizing the cache read command and trading ECC margin to decrease page-sensing latency. Although this scheme significantly reduces read-retry overhead, it focuses on reducing the latency of each read-retry step rather than the number of read-retries. As a result, our assumption that there is significant block-level read latency variation remains valid even with this scheme. Prior works [75, 76] optimized the decoding process of modern LDPC decoders to reduce soft decoding's long tail latency. However, this scheme does not eliminate large differences in N_{retry} between different blocks due to large latency variations in the number of read voltages of each soft level in the soft decoding process.

Error Prediction Models for Modern NAND Flash Memory. Several works have proposed error prediction models for modern 3D NAND flash memory [19, 21, 22]. Table 5 compares representative models to our proposal in terms of how comprehensively they considered various error sources of modern NAND flash memory. The new error model we develop in this work makes three key contributions over existing error models. First, it uses an accurate metric for flash wear rather than the traditional P/E-cycle count to reflect the effect of ambient temperature and inter-block variation on errors. Second, it comprehensively considers major error sources including retention loss and program/read disturbance. Third, it is rigorously validated using real 3D NAND flash chips.

Table 5. A comparison of the existing online RBER models.

Source	HeatWatch [22]	PV [19]	RealWear [21]	Our proposal
P/E cycle	O	O	O	O
Retention loss	O	O	X	O
Inter-block variation	X	X	O	O
Intra-block variation	X	O	X	X
Read disturbance	X	X	X	O
Ambient temperature	O	X	O	O

SSD-Level Read Performance Differentiation. To the best of our knowledge, this work is the first work to support block-level read performance differentiation. A recent work [77] proposes an adaptive read reclaim scheme based on the hint about expected read performance data from host apps. Even though the prior work aims to enhance SSD lifetime (but not to improve I/O performance) by preventing unnecessary read reclaims, its key idea (i.e., sending latency requirement hints) can be used for read performance differentiation. For example, one can differentiate the read performance of two applications app_a and app_b by setting their read-latency requirements differently, e.g., 100 us for app_a and 1 ms for app_b; the SSD controller then ensures lower read latency for app_a by more frequently performing read reclaim for app_a if a block's read latency becomes higher than 100 us. However, such an approach alone would be likely to introduce significant performance and lifetime overheads unless taking into account the block-quality variations as in `ReadGuard`. This is because, without considering the block quality, a latency-sensitive application's data can be stored in low-quality blocks, which inevitably causes early read reclaim to meet a high read-latency requirement. Note that such early read reclaims can occur (i) repeatedly by failing to use high-quality blocks and (ii) more frequently if the data of different applications is not stored separately as in `ReadGuard`.

8 Discussion

We believe that broad-area apps that require strict latency-based service-level agreements (SLAs) [78], ranging from traditional database apps to future machine learning apps, will benefit considerably from ReadGuard in shared storage systems. ReadGuard ensures that data from these apps is stored on higher-quality flash blocks than data from latency-insensitive apps (i.e., throughput-oriented apps). Furthermore, ReadGuard's capabilities can be extended to provide service differentiation [79] within a database application by selectively prioritizing more important users' data over others in order to protect the formal from performance degradation. Emerging ML apps are an additional promising use case for ReadGuard for two reasons. First, because the input data size of emerging ML apps, such as embedding tables in recommendation systems, is constantly increasing, storing all of this data in DRAM is not feasible [80]. As a result, moving input data from DRAM to large-scale SSDs is a promising solution for future ML apps. Second, some ML apps require strict SLAs to provide cloud-based ML capabilities to users [3]. ReadGuard can mitigate SLA violations resulting from the inherently slower performance of SSDs compared to DRAM by reducing both average and tail read latencies for latency-critical ML applications.

Although ReadGuard effectively differentiates read performance based on app I/O priority in modern flash-based storage systems, the current version of ReadGuard also has three limitations: (i) additional writes caused by block migration, and (ii) it does not account for intra-block read latency variation, and (iii) it may affect the utilization of plane-level parallelism. Compared to a baseline FTL, additional writes are performed in `rgFTL` for block migration operations to ensure priority order in block quality among different priorities. These additional writes may degrade (i) the lifetime of NAND flash memory and (ii) user-level write latency. The impact of additional writes on the lifetime of NAND flash memory can be minimized by carefully adjusting the WM module's predefined threshold value, which determines the block-quality inversion condition. For example, for error-prone NAND flash memory with a 1% P/E cycle margin, the predefined threshold value should be 0.01. With this threshold value, `rgFTL` may allow the most dynamic block-quality inversion after block allocation, but high-priority apps still benefit from priority-aware block allocation. In contrast, common NAND flash memory with a considerable P/E cycle margin, such as the NAND flash memory used in our study, can support stricter priority-based read performance differentiation by leveraging the margin with additional block migration operations. To minimize the impact of block migration operations on user-perceived write latency, `rgFTL` prioritizes block migration operations lower than user writes. As shown in Figure 19(a), this simple solution is quite efficient under most I/O patterns, including write-intensive workloads.

The current version of ReadGuard only considers inter-block read latency variations. However, in modern NAND flash memory, the read latency can vary across pages within a block due to various reasons, such as read-disturbance patterns, process variations, and target page types (i.e., LSB/CSB/MSB pages) [19, 20, 62]. Even though it is ideal to derive a more comprehensive model that considers such intra-block latency variation as well, we believe that our `ReadGuard` is also highly effective due to two reasons. First, inter-block latency variation considered in `ReadGuard` is much more significant than intra-block latency variation in many cases, as shown in Figure 18. This is because all pages in a block have similar retention times and read-disturbed counts, while pages in different blocks may experience significant differences in retention times and read-disturbance effects. Furthermore, it is common practice to minimize the sensing-time variation across page types (e.g., reading an LSB/CSB/MSB page requires 2/3/2 sensing operations), which limits intra-block latency variation. Second, it is challenging to consider intra-block latency variation in a cost-efficient manner due to the significant metadata overhead to keep track of each page's status. The current design of ReadGuard only requires small additional space (e.g., 5 MB for a 2 TB SSD) to keep per-block metadata, which makes it a highly cost-effective solution.

ReadGuard may affect the utilization of plane-level parallelism for read requests. To utilize plane-level parallelism, operations on multiple planes in a single chip must have the same page number. Pages with aligned page addresses that can be read concurrently from multiple planes may need to be read separately after block migration. This is because extra block migration operations performed by the PBM module may change the page address of copied data, as a block migration operation only copies valid data from the target block. However, as shown in our experimental results, the impact of additional block migration operations on read latency is negligible, since a baseline FTL already frequently invokes block migration operations for garbage collection and wear leveling. Furthermore, as flash manufacturers have recently proposed advanced flash architectures with independent row and block decoders for each plane [81, 82], we believe that the limitations of our proposal will be overcome in modern flash memory systems.

9 Conclusions

We have presented `ReadGuard`, an integrated priority-aware flash management technique that achieves read performance differentiation based on the I/O priority of apps in modern flash-based storage systems. `ReadGuard` fully manages flash blocks in a read-latency-centric fashion at the flash block level. To precisely distinguish flash blocks with different quality levels, `ReadGuard` proposed a novel read-latency estimator $nr(B)$ that accurately predicts N_{retry} of a flash block B. By leveraging $nr(B)$, we built `rgFTL`, a `ReadGuard`-enabled FTL, which ensures that higher-quality blocks are used for higher-priority apps. Our experimental results show that `rgFTL` effectively supports differentiated read performance among apps with different priorities without negatively affecting the SSD lifetime.

The current version of `rgFTL` can be further improved in several directions. For example, it manages flash blocks in a priority-aware fashion based on the longest *tREAD* of a flash block. Considering the well-known process variations among different WLs within a flash block, extending $nr(B)$ to a finer-grained level than a flash block level (e.g., sub-block level) may allow more accurate tracking of read latency variations. As a solution for *guaranteed* I/O QoS level support, it may be an interesting future direction to extend `rgFTL` to support strict read differentiation using a fine-grained read latency marker.

Appendix

This appendix contains our comprehensive device characterization results for 160 real 3D TLC NAND flash chips. We measured the number of retry steps (N_{retry}) of the worst page in a block for more than 2,560 blocks that are evenly selected from the 160 NAND flash chips under various operating conditions. We set six groups by varying operating conditions for this evaluation: (a) fresh, (b) after 1-week retention time, (c) after 1-month retention time,

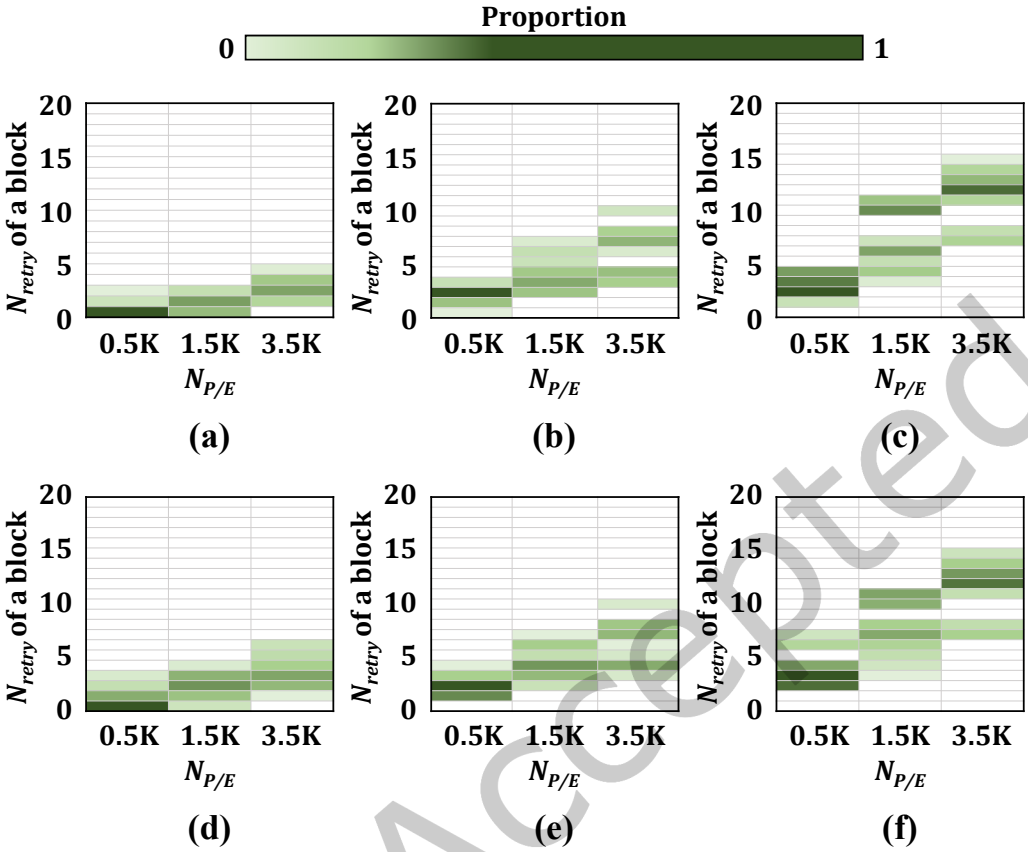

Fig. 21. Distributions of N_{retry} under various operating conditions: (a) fresh, (b) after 1-week retention time, (c) after 1-month retention time, (d) after 100 block read operations, (e) after 1-week retention time with 100 block read operations, and (f) after 1-month retention time with 100 block read operations.

(d) after 100 block read operations, (e) after 1-week retention time with 100 block read operations, and (f) after 1-month retention time with 100 block read operations. For the fresh group, we measured the N_{retry} value (i.e., block quality) of blocks immediately after programming, resulting in no retention time or read disturbance effect on all pages. To efficiently mimic the read disturbance effect on all pages in a block, we repeated a block read operation, which is a custom command for characterization that reads all pages sequentially. Figure 21 shows the probability of occurrence of different numbers of retry steps (in green scale) for six groups. A box at (x, y) represents the probability that a read requires a read-retry operation with y retry steps under x P/E cycles. From the results, we observed that (1) inter-block variation in N_{retry} values exists even under the same conditions, and (2) significant inter-block variation in N_{retry} values between fresh blocks and blocks after long retention time with read disturb effect, as shown in groups (a) and (f). We hope that this appendix will help readers understand the significant inter-block latency variation of modern NAND flash memory, which is the primary motivation for our study, as well as encourage future research.

References

[1] Mohd Tajammul and R Parveen. 2021. Cloud Storage in Context of Amazon Web Services. *International Journal of All Research Education and Scientific Methods* 10, 01 (2021), 442–446.

[2] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

[3] Xuan Sun, Hu Wan, Qiao Li, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. 2022. RM-SSD: In-storage computing for large-scale recommendation inference. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[4] Hu Wan, Xuan Sun, Yufei Cui, Chia-Lin Yang, Tei-Wei Kuo, and Chun Jason Xue. 2021. FlashEmbedding: Storing Embedding Tables in SSD for Large-Scale Recommender Systems. In *Proceedings of the ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*.

[5] Thomas Anderson, Adam Belay, Mosharaf Chowdhury, Asaf Cidon, and Irene Zhang. 2022. Treehouse: A Case for Carbon-Aware Datacenter Software. *arXiv* (2022).

[6] Yichao Jin, Yonggang Wen, and Qinghua Chen. 2012. Energy Efficiency and Server Virtualization in Data Centers: An Empirical Investigation. In *Proceedings of the IEEE INFOCOM Workshops (INFOCOM)*.

[7] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards Energy Proportionality for Large-Scale Latency-Critical workloads. *ACM SIGARCH Computer Architecture News* (2014).

[8] J. Zhang, M. Kwon, D. Gouk, C. Lee, M. Alian, M. Chun, M. Kademir, N. Kim, J. Kim, and M. Jung. 2018. FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[9] S. Hahn, S. Lee, I. Yee, D. Ryu, and J. Kim. 2018. FastTrack: Foreground App-Aware I/O Management for Improving User Experience of Android Smartphones. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[10] M. Liu, H. Liu, C. Ye, X. Liao, H. Jin, Y. Zhang, R. Zheng, and L. Hu. 2022. Towards low-latency I/O services for mixed workloads using ultra-low latency SSDs. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*.

[11] A. Tavakkol, J. Gomez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu. 2018. MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*.

[12] A. Tavakkol, M. Sadrosadati, S. Ghose, J. Kim, Y. Luo, Y. Wang, N. Ghiasi, L. Orosa, J. Gómez-Luna, and O. Mutlu. 2018. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*.

[13] J. Yoon, S. Devendrappa, and X. Ouyang. U.S. Patent 0075570A1, Mar. 2017. Reducing Read Command Latency in Storage devices. (U.S. Patent 0075570A1, Mar. 2017).

[14] T. Earhart and D. Pruett. U.S. Patent 10732895B2, August. 2020. Drive-level Internal Quality of Service. (U.S. Patent 10732895B2, August. 2020).

[15] M. Jung. 2020. OpenExpress: Fully Hardware Automated Open Research Framework for Future Fast NVMe Devices. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[16] J. Lee, J. Choi, D. Park, and K. Kim. 2003. Data Retention Characteristics of Sub-100 nm NAND Flash Memory Cells. *IEEE Electron Device Letters, vol. 24, no. 12, pp. 748-750* (2003).

[17] Y. Cai, Y. Luo, E. Haratsch, K. Mai, and O. Mutlu. 2015. Data Retention in MLC NAND Flash Memory: Characterization, Optimization, and Recovery. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[18] A. Torsi, Y. Zhao, H. Liu, T. Tanzawa, A. Goda, P. Kalavade, and K. Parat. 2010. A Program Disturb Model and Channel Leakage Current Study for Sub-20 nm NAND Flash Cells. *IEEE Transactions on Electron Devices, vol. 58, no. 1, pp. 11-16* (2010).

[19] Y. Luo, S. Ghose, Y. Cai, E. Haratsch, and O. Mutlu. 2018. Improving 3D NAND Flash Memory Lifetime by Tolerating Early Retention Loss and Process Variation. In *Proceedings of the ACM Measurement and Analysis of Computing Systems (POMACS)*.

[20] Y. Shim, M. Kim, M. Chun, J. Park, Y. Kim, and J. Kim. 2019. Exploiting Process Similarity of 3D Flash Memory for High Performance SSDs. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[21] M. Kim, M. Chun, D. Hong, Y. Kim, G. Cho, D. Lee, and J. Kim. 2021. RealWear: Improving Performance and Lifetime of SSDs Using a NAND Aging Marker. *Performance Evaluation* 145 (2021), 102153.

[22] Y. Luo, S. Ghose, Y. Cai, E. Haratsch, and O. Mutlu. 2018. HeatWatch: Improving 3D NAND Flash Memory Device Reliability by Exploiting Self-Recovery and Temperature Awareness. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[23] Q. Li, M. Ye, Y. Cui, L. Shi, X. Li, and C. Xue. 2019. Sentinel Cells Enabled Fast Read for NAND Flash. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.

[24] J. Park, M. Kim, M. Chun, L. Orosa, J. Kim, and O. Mutlu. 2021. Reducing Solid-State Drive Read Latency by Optimizing Read-Retry. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[25] Q. Li, M. Ye, Y. Cui, L. Shi, X. Li, T. Kuo, and C. Xue. 2020. Shaving Retries with Sentinels for Fast Read over High-Density 3D Flash. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*.
[26] S. Kim, J. Bae, H. Jang, W. Jin, J. Gong, S. Lee, T. Ham, and J. Lee. 2019. Practical Erase Suspension for Modern Low-latency SSDs. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.
[27] G. Wu and X. He. 2012. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension.. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*.
[28] Y. Kasorla, A. Schushan, A. Vega, E. Gurgi, and S. Ojalvo. U.S. Patent 9779038B2, Oct. 2017. Efficient Suspend-Resume Operation in Memory Devices. (U.S. Patent 9779038B2, Oct. 2017).
[29] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*.
[30] Facebook. 2013. RocksDB. http://rocksdb.org/. (2013).
[31] D. Kang, W. Jeong, C. Kim, D. Kim, Y. Cho, K. Kang, J. Ryu, K. Kang, S. Lee, W. Kim, H. Lee, J. Yu, N. Choi, D. Jang, C. Lee, Y. Min, M. Kim, A. Park, J. Son, I. Kim, P. Kwak, B. Jung, D. Lee, H. Kim, J. Ihm, D. Byeon, J. Lee, K. Park, and K. Kyung. 2016. 256Gb 3b/Cell V-NAND Flash Memory with 48 Stacked WL Layers. In *Proceedings of the International Solid-State Circuits Conference (ISSCC)*.
[32] 2013. Micron Announces 16 nm 128Gb MLC NAND, SSD in 2014. http://www.anandtech.com/show/7147/micron-announces-16nm-128gb-mlc-nand-ssds-in-2014. (2013).
[33] B. Peleato, H. Tabrizi, R. Agarwal, and J. Ferreira. 2015. BER-Based Wear Leveling and Bad Block Management for NAND flash. In *Proceedings of the IEEE International Conference on Communications (ICC)*.
[34] Y. Woo and J. Kim. 2013. Diversifying Wear Index for MLC NAND Flash Memory to Extend the Lifetime of SSDs. In *Proceedings of the Eleventh ACM International Conference on Embedded Software (EMSOFT)*.
[35] A. Chou, K. Lai, K. Kumar, P. Chowdhury, and J. Lee. 1997. Modeling of Stress-Induced Leakage Current in Ultrathin Oxides with the Trap-Assisted Tunneling Mechanism. *Applied physics letters* 70, 25 (1997), 3407–3409.
[36] S. Kamohara, D. Park, and C. Hu. 1998. Deep-trap SILC (Stress Induced Leakage Current) Model for Nominal and Weak Oxides. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS)*.
[37] S. Takagi, N. Yasuda, and A. Toriumi. 1999. A New IV Model for Stress-Induced Leakage Current Including Inelastic Tunneling. *IEEE Transactions on Electron Devices* 46, 2 (1999), 348–354.
[38] JEDEC. 2009. Electrically Erasable Programmable ROM (EEPROM) Program / Erase Endurance and Data Retention Stress Test (JEDEC22-A117). https://www.jedec.org. (2009).
[39] JEDEC. 2010. Stress-Test-Driven Qualification of Integrated Circuits (JEDEC JESD47). https://www.jedec.org. (2010).
[40] Y. Luo, S. Ghose, Y. Cai, E. Haratsch, and O. Mutlu. 2018. HeatWatch: Improving 3D NAND flash memory device reliability by exploiting self-recovery and temperature awareness. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*.
[41] Y. Cai, Y. Luo, S. Ghose, and O. Mutlu. 2015. Read Disturb Errors in MLC NAND Flash Memory: Characterization, Mitigation, and Recovery. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.
[42] Tianyu Ren, Qiao Li, Min Ye, and Chun Jason Xue. 2023. Read Disturb and Reliability: The Complete Story for 3D CT NAND Flash. In *Proceedings of the IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*.
[43] S. Lee and J. Kim. 2014. Effective Lifetime-Aware Dynamic Throttling for NAND Flash-Based SSDs. *IEEE Trans. Comput.* 65, 4 (2014), 1075–1089.
[44] R Micheloni, L. Crippa, and A. Marelli. 2010. *Inside NAND Flash Memories*.
[45] Seiichi Aritome. 2015. *NAND Flash Memory Technologies*.
[46] JEDEC. 2010. JEDEC Solid State Technology Assn., Solid-State Drive (SSD) Requirements and Endurance Test Method. https://www.jedec.org. (2010).
[47] Z. Jiao, J. Bhimani, and B. Kim. 2022. Wear Leveling in SSDs Considered Harmful. In *Proceedings of the ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*.
[48] F. Chen, M. Yang, Y. Chang, and T. Kuo. 2015. PWL: A Progressive Wear Leveling to Minimize Data Migration Overheads for NAND Flash Devices. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*.
[49] Z. Chen and Y. Zhao. 2020. DA-GC: A Dynamic Adjustment Garbage Collection Method Considering Wear-Leveling for SSD. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*.
[50] PCI-SIG. 2022. PCI Express M.2 Specification Revision 4.0, Version 1.1. (2022). https://pcisig.com/specifications.
[51] Jinhong Li, Qiuping Wang, Patrick P. C. Lee, and Chao Shi. 2020. An In-Depth Analysis of Cloud Block Storage Workloads in Large-Scale Production. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*.
[52] W. Kang and S. Yoo. 2020. Q -Value Prediction for Reinforcement Learning Assisted Garbage Collection to Reduce Long Tail Latency in SSD. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2240–2253.
[53] T. Zhu, M. Kozuch, and M. Harchol-Balter. 2017. Workloadcompactor: Reducing Datacenter Cost While Providing Tail Latency SLO Guarantees. In *Proceedings of the Symposium on Cloud Computing (SoCC)*.

[54] S. Yan, H. Li, M. Hao, M. Tong, S. Sundararaman, A. Chein, and H. Gunawi. 2017. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*.
[55] H. Litz, J. Gonzalez, A. Klimovic, and C. Kozyrakis. 2022. RAIL: Predictable, Low Tail Latency for NVMe Flash. *ACM Transactions on Storage (TOS)* 18, 1 (2022), 1–21.
[56] Z. Sha, J. Li, L. Song, J. Tang, M. Huang, Z. Cai, L. Qian, J. Liao, and Z. Liu. 2021. Low I/O Intensity-Aware Partial GC Scheduling to Reduce Long-Tail Latency in SSDs. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 4 (2021), 1–25.
[57] Nikolaos Papandreou, Nikolas Loannou, Thomas Parnell, Roman Pletka, Milos Stanisavljevic, Radu Stoica, Sasa Tomic, and Haralampos Pozidis. 2020. Reliability of 3D NAND Flash Memory with a Focus on Read Voltage Calibration from a System Aspect. In *Proceedings of the Non-Volatile Memory Technology Symposium (NVMTS)*.
[58] Yingge Li, Guojun Han, Sanwei Huang, Chang Liu, Meng Zhang, and Fei Wu. 2023. Exploiting Metadata to Estimate Read Reference Voltage for 3-D NAND Flash Memory. *IEEE Transactions on Consumer Electronics (TCE)* (2023).
[59] Meng Zhang, Fei Wu, Qin Yu, Weihua Liu, Yifan Wang, and Changsheng Xie. 2020. Exploiting Error Characteristic to Optimize Read Voltage for 3-D NAND Flash Memory. *IEEE Transactions on Electron Devices (TED)* (2020).
[60] Jinhua Cui, Zhimin Zeng, Jianhang Huang, Weiqi Yuan, and Laurence T Yang. 2022. Improving 3-D NAND SSD Read Performance by Parallelizing Read-Retry. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2022).
[61] Micron. 2004. Technical Note: NAND Flash Performance Increase Using the Micron PAGE READ CACHE MODE Command. https://www.micron.com/-/media/client/global/Documents/Products/Technical*****20Note/NAND*****20Flash/tn2901.pdf. (2004).
[62] C. Hung, M. Chang, Y. Yang, Y. Kuo, T. Lai, S. Shen, J. Hsu, S. Hung, H. Lue, Y. Shih, S. Huang, T. Chen, T. Chen, C. Chen, C. Hung, and C. Lu. 2015. Layer-aware Program-and-Read Schemes for 3D Stackable Vertical-Gate BE-SONOS NAND Flash Against Cross-Layer Process Variations. *IEEE Journal of Solid-State Circuits, vol. 50, no. 6, pp. 1491-1501* (2015).
[63] Seungjae Lee, Chulbum Kim, Minsu Kim, Sung-min Joe, Joonsuc Jang, Seungbum Kim, Kangbin Lee, Jisu Kim, Jiyoon Park, Han-Jun Lee, et al. 2018. A 1Tb 4b/cell 64-stacked-WL 3D NAND flash memory with 12MB/s program throughput. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*.
[64] Hwang Huh, Wanik Cho, Jinhaeng Lee, Yujong Noh, Yongsoon Park, Sunghwa Ok, Jongwoo Kim, Kayoung Cho, Hyunchul Lee, Geonu Kim, et al. 2020. 13.2 a 1tb 4b/cell 96-stacked-wl 3d nand flash memory with 30mb/s program throughput using peripheral circuit under memory cell array technique. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*.
[65] Tsutomu Higuchi, Takuyo Kodama, Koji Kato, Ryo Fukuda, Naoya Tokiwa, Mitsuhiro Abe, Teruo Takagiwa, Yuki Shimizu, Junji Musha, Katsuaki Sakurai, et al. 2021. 30.4 a 1Tb 3b/cell 3D-flash memory in a 170+ word-line-layer technology. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*.
[66] Ted Pekny, Luyen Vu, Jeff Tsai, Dheeraj Srinivasan, Erwin Yu, Jonathan Pabustan, Joe Xu, Srinivas Deshmukh, Kim-Fung Chan, Michael Piccardi, et al. 2022. A 1-Tb density 4b/cell 3D-NAND flash on 176-tier technology with 4-independent planes for read using CMOS-under-the-array. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*.
[67] Li-Pin Chang. 2007. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*.
[68] S. Kim, J. Kim, J. Lee, and J. Jeong. 2017. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*.
[69] Sandoval, O. 2017. blk-mq: Kyber multiqueue I/O scheduler. http://lwn.net/Articles/720071/. (2017).
[70] S. Yang, T. Harter, N. Agrawal, S. Kowsalya, A. Krishnamurthy, S. Al-Kiswany, R. Kaushik, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. 2015. Split-Level I/O Scheduling. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.
[71] Q. Zhang, D. Feng, F. Wnag, and Xie. Y. 2013. An Efficient, QoS-aware I/O Scheduler for Solid State Drive. In *Proceedings of the IEEE International Conference on High Performance Computing and Communications (HPCC)*.
[72] NVM Express. 2022. NVME Express Base Specification 2.0c. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0c-2022.10.04-Ratified.pdf/. (2022).
[73] Hao Fan, Yiliang Ye, Shadi Ibrahim, Zhuo Huang, Xingru Li, Weibin Xue, Song Wu, Chen Yu, Xuanhua Shi, and Hai Jin. 2024. QoS-pro: A QoS-Enhanced Transaction Processing Framework for Shared SSDs. *ACM Transactions on Architecture and Code Optimization* (2024).
[74] Byunghei Jun and Dongkun Shin. 2015. Workload-Aware Budget Compensation Scheduling for NVMe Solid State Drives. In *Proceedings of the IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*.
[75] Yajuan Du, Yuan Gao, Siyi Huang, and Qiao Li. 2023. LDPC Level Prediction Towards Read Performance of High-Density Flash Memories. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2023).
[76] Yajuan Du, Deqing Zou, Qiao Li, Liang Shi, Hai Jin, and Chun Jason Xue. 2017. Laldpc: Latency-aware ldpc for read performance improvement of solid state drives. In *Proceeding of the International Conference on Massive Storage Systems and Technology (MSST)*.
[77] Chun-Yi Liu, Yunju Lee, Myoungsoo Jung, Mahmut Taylan Kandemir, and Wonil Choi. 2021. Prolonging 3D NAND SSD lifetime via read latency relaxation. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[78] Che-Wei Chang, Geng-You Chen, Yi-Jung Chen, Chia-Wei Yeh, Pei Yin Eng, Ana Cheung, and Chia-Lin Yang. 2017. Exploiting Write Heterogeneity of Morphable MLC/SLC SSDs in Datacenters with Service-Level Objectives. *IEEE Trans. Comput.* (2017).

[79] Michael Mesnier, Feng Chen, Tian Luo, and Jason B Akers. 2011. Differentiated Storage Services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*.

[80] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 2021. RecSSD: Near Data Processing for Solid State Drive Based Recommendation Inference. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[81] Wanik Cho, Jongseok Jung, Jongwoo Kim, Junghoon Ham, Sangkyu Lee, Yujong Noh, Dauni Kim, Wanseob Lee, Kayoung Cho, Kwanho Kim, et al. 2022. A 1-Tb, 4b/Cell, 176-Stacked-WL 3D-NAND Flash Memory with Improved Read Latency and a 14.8 Gb/mm2 Density. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*.

[82] Ted Pekny, Luyen Vu, Jeff Tsai, Dheeraj Srinivasan, Erwin Yu, Jonathan Pabustan, Joe Xu, Srinivas Deshmukh, Kim-Fung Chan, Michael Piccardi, et al. 2022. A 1-Tb Density 4b/Cell 3D-NAND Flash on 176-Tier Technology with 4-Independent Planes for Read Using CMOS-under-the-Array. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*.

Received 19 September 2023; revised 9 May 2024; accepted 25 June 2024