GoFair: Global-Oriented Fairness Framework for Shared NVMe SSDs

Jiakun Li[1], Yulai Xie[1,2,*], Dan Feng[3,4], Heyu Zhang[1]
[1] School of Cyber Science and Engineering, Huazhong University of Science & Technology, Wuhan, China
[2] Shenzhen Huazhong University of Science & Technology Research institute, Shenzhen, China
[3] Wuhan National Laboratory for Optoelectronics, Huazhong University of Science & Technology, Wuhan, China
[4] Key Laboratory of Information Storage System, Ministry of Education

***Abstract*—Solid State Drives (SSDs) are widely used in cloud computing as high-performance storage solutions that offer fast data access speeds, low latency, and high reliability. The NVMe interface improves SSD performance by bypassing the software stack, but introduces fairness challenges in multi-tenant environments. Fairness is primarily affected by workload intensity and access patterns. Existing methods improve fairness primarily by reordering requests or transactions, or by increasing parallelism. However, existing methods cannot effectively improve overall fairness, as they overlook the processing capability of SSD. To the best of our knowledge, no prior work has provided a mechanism to align diverse workload intensities and access patterns with the processing capacity of SSDs. In this paper, we address this issue by proposing the Global-oriented Fairness Framework (GoFair). GoFair improves overall fairness by allocating chip resources across workloads according to the processing capability of individual SSD chips and the aggregate slowdown exhibited by each workload. We implement GoFair in the MQSim framework to compare its fairness improvements and throughput relative to established approaches across a broad spectrum of workload scenarios. Experimental results demonstrate that GoFair outperforms the state-of-the-art QoS-pro, improving fairness by 13.72%.**

***Index Terms*—Solid State Drives, Fairness, Multiple Applications, Transaction Scheduling**

I. Introduction

Solid State Drives (SSDs) have become the storage substrate in modern cloud infrastructures, due to their high I/O throughput, low latency, and improved reliability over hard disk drives (HDDs). This transition has been further accelerated by the advent and widespread adoption of the Non-Volatile Memory Express (NVMe) interface[1], which reduces software-stack overheads and exposes deep parallelism within SSDs. While these advances substantially elevate application performance, they also complicate resource sharing in multi-tenant environments, where co-located workloads with different characteristics compete for limited internal device resources[2]. In such environments, performance interference can manifest as pronounced unfairness[3], undermining service-level objectives (SLOs) and the predictability required by cloud customers[4].

As multiple tenants in cloud environments often share the same SSD, leading to contention for internal device resources[5, 6], numerous approaches have been proposed to improve fairness for shared SSDs. We can categorize existing research into three classes, namely host-level scheduling[7–9], hardware-level performance isolation in SSDs[3, 6, 10], and I/O scheduling within SSDs[4, 5, 11–13].

Existing solutions in the host-level and hardware-level categories exhibit distinct limitations. Host-level schemes, such as FlashFQ[8], regulate I/O dispatch at the driver or block layer to manage contention. However, they treat the SSD as a black box, lacking visibility into internal device states like garbage collection or geometric parallelism, which limits their ability to guarantee precise fairness. On the other hand, hardware-level isolation methods, including FlashBlox[3] and SSDKeeper[6], physically partition resources such as channels or dies among tenants. While this ensures strong isolation, it often suffers from resource fragmentation and underutilization, as rigid partitioning cannot adapt fluidly to highly bursty or skew-heavy workloads.

Within the SSD firmware, internal I/O scheduling attempts to bridge the gap between host demands and device constraints, yet current state-of-the-art methods struggle to achieve robust fairness due to three fundamental limitations. First, existing isolation mechanisms often operate at ineffective granularities. Approaches relying on physical chip-level partitioning, such as Clustering[11], are too coarse to handle diverse access patterns without causing load imbalance. Conversely, schedulers like FLIN[5] prioritize individual transaction latencies, operating at a level too fine to capture the aggregate, request-level slowdown effectively perceived by the tenant. Second, current metrics for contention are often one-dimensional. While schemes like Fuzzy[4] successfully adapt to varying I/O intensities via dynamic thresholds, they lack the semantic awareness to distinguish between heterogeneous access patterns (e.g., random vs. sequential), leaving pattern-driven contention unresolved. Finally, hybrid approaches such as QoS-pro[12], which combine parallelism exploitation with transaction scheduling, tend to inherit the limitations of their underlying scheduling logic (e.g., FLIN's transaction focus). Consequently, they fail to explicitly model the complex relationship between workload pressure and the bounded processing capabilities of SSD components.

Taken together, these limitations share a common root cause. Existing internal schedulers prioritize the optimization

* Corresponding Author

of local micro-architectural states, such as transaction ordering, instantaneous queue depths, or raw parallelism exposure. Consequently, these mechanisms fail to align broad workload-level demands with the heterogeneous and bounded processing capabilities of internal SSD components like chips, planes, and channels. Specifically, demand should be captured by metrics including request-level slowdown, intensity, and access patterns. Without an explicit and capacity-aware mapping from workload pressure to device-internal resources, fairness gains remain fragile, workload-specific, and sensitive to changes in mix and arrival dynamics.

In summary, our contributions are as follows:

- We identify that the root cause of unfairness in shared SSDs lies in the misalignment between diverse workload demands (e.g., intensity and access patterns) and the heterogeneous, bounded processing capabilities of SSD internal components. Existing methods fail to explicitly model this relationship, leading to fragile fairness guarantees.
- We propose GoFair, a global-oriented fairness framework that improves overall fairness by aligning chip resource allocation with the processing capability of individual SSD chips and the aggregate slowdown of each workload. GoFair introduces a comprehensive resource matching mechanism that includes load-aware mapping, resource-aware scheduling, and slowdown-aware suspending to balance performance and fairness.
- We implement GoFair in the MQSim framework and evaluate it against state-of-the-art approaches using real-world workload traces. Experimental results demonstrate that GoFair outperforms the best-performing baseline QoS-pro, improving fairness by 13.72% while maintaining high system throughput.

The rest of the paper is structured as follows. We begin with the background of NVMe SSD organizations and fairness metrics (Section II). Then, we delve into the motivation behind this paper (Section III). Next, we discuss the design of GoFair (Section IV). Subsequently, we detail the entire evaluation process for GoFair (Section V). We then review the related work (Section VI). Finally, we conclude the paper (Section VII).

II. Background

In this section, we first provide background on the organization of modern NVMe SSDs. We then describe the process of requests and transactions to analyze sources of inter-workload interference. Finally, we present an established methodology for quantifying SSD fairness.

A. NVMe SSD Organizations

As shown in figure 1, modern NVMe SSDs comprise a front-end controller and a back-end storage array, architected to expose substantial internal parallelism while minimizing host-side overheads. The front end is typically organized into three principal components: the Host Interface Logic (HIL), which terminates the NVMe protocol and fetches I/O

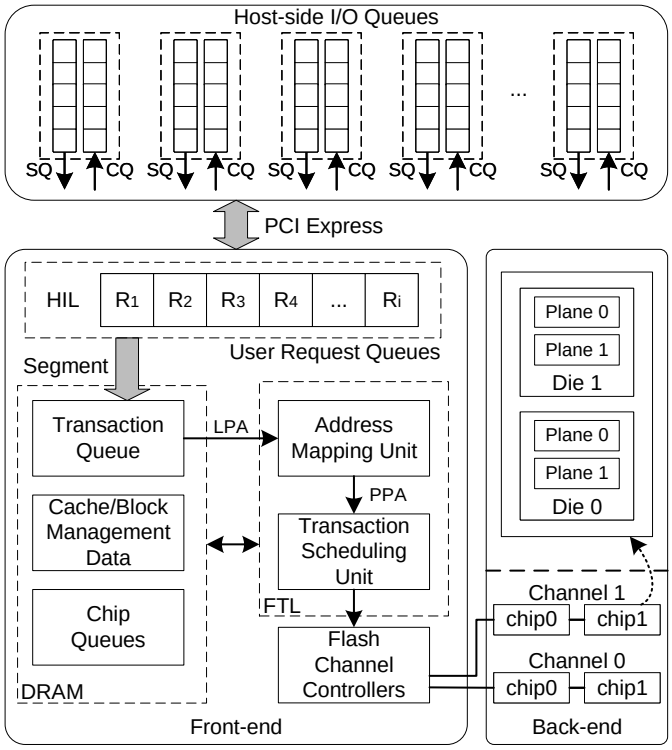

Fig. 1: NVMe SSD organization

requests from multiple host submission queues; the Controller (embedded processors and control logic), which parses host requests into page-granularity transactions and orchestrates management functions; and the Flash Channel Controllers (FCCs), which bridge front-end control to the storage back end and issue commands to the media along specific channels. The back end supplies four hierarchical layers: multiple channels, each attaching to one or more NAND chips; each chip containing one or more dies; each die comprising multiple planes; and each plane organized into blocks that consist of pages.

The request-processing pipeline proceeds as follows. Upon arrival, a host request is parsed into one or more transactions, each identified by an LPA. The flash translation layer (FTL) resolves LPAs to PPAs, typically employing out-of-place updates to respect erase-before-write constraints—allocating new physical pages for writes and retrieving the extant physical locations for reads, while preserving mapping consistency. Following translation, transactions are enqueued into per-chip software queues, commonly separated for reads, writes, and background maintenance activities (e.g., garbage collection). A transaction scheduling unit (TSU) then selects transactions from these chip-level queues and dispatches them to specific channels/chips/dies/planes, coordinating with device-level constraints to exploit internal parallelism and attenuate inter-workload interference.

At the media layer, reads and writes operate at page granularity, whereas erase operations occur at block granularity. The pronounced asymmetry between relatively fast

reads and slower program/erase operations, together with finite program–erase endurance, necessitates dedicated device-level management. The FTL addresses these concerns via out-of-place writes (invalidating prior versions), garbage collection to reclaim blocks with a high fraction of invalid pages (migrating any residual valid data prior to erase), and wear leveling to distribute erase cycles. Because these maintenance tasks contend for the same channels, chips, dies, and planes as foreground I/O, their interaction with TSU decisions materially influences latency, throughput, and fairness in multi-tenant settings.

B. Requests and Transactions Processing

In modern NVMe SSDs, host-issued I/O requests undergo a structured processing pipeline that bridges the logical representations used at the host with the physical organization of flash memory. Each request, typically expressed in terms of Logical Page Addresses (LPAs), is first retrieved from the host submission queues via the Host Interface Logic (HIL) and forwarded to the controller for interpretation. The controller decomposes complex requests into one or more atomic transactions, each aligning with the page-level granularity of the NAND media. This decomposition enables fine-grained control over command scheduling and parallel execution.

The Flash Translation Layer (FTL) plays a central role at this stage, performing LPA-to-Physical Page Address (PPA) mapping while adhering to NAND constraints such as erase-before-write. Write operations are executed out-of-place to avoid in-place modifications, creating invalid pages that must later be reclaimed through garbage collection. Read transactions retrieve data from their mapped physical pages. To improve throughput and reduce contention, transactions are classified by type—reads, writes, or background operations—and enqueued into separate per-chip queues. This separation allows the Transaction Scheduling Unit (TSU) to make informed scheduling decisions that balance latency sensitivity (favoring reads) with long-term maintenance needs (servicing garbage collection or wear leveling).

Dispatching transactions requires respecting multiple device-level constraints: channel bandwidth limitations, die-level concurrency, plane parallelism, and media readiness states. The TSU orchestrates these allocations to maximize utilization while mitigating cross-workload interference. In multi-tenant scenarios, scheduling policies must account for workload heterogeneity in access patterns, arrival rates, and request sizes, as these factors influence queue occupancy and service discipline. The interaction between foreground transactions and background activities is particularly critical, as internal maintenance operations can delay host-visible requests, exacerbate latency variability, and thereby affect workload fairness.

C. SSD Fairness

When multiple workloads execute concurrently on an SSD, a given workload can influence one or more co-running workloads. Therefore, it is necessary to evaluate SSD fairness using quantitative metrics. Fairness in shared SSDs concerns how evenly co-located workloads are slowed by mutual interference when they access a common device. We quantify the fairness of SSD based on existing researches[4, 5, 12, 13]. When different workloads of equal priority execute concurrently, fairness improves as the variance in their slowdown factors decreases. The SSD reaches the ideal of perfect fairness when all workloads exhibit identical slowdown. The slowdown of workload i can be calculated by Eq.(1), where RT_i^{shared} is the average response time of workload i when it runs concurrently with other workloads, and RT_i^{alone} stands for the average response time of workload i when it runs alone. The response time for each request is defined as the interval between the request's issuance at the host and the receipt of the SSD's response at the host.

$$slowdown_i = RT_i^{shared} / RT_i^{alone} \tag{1}$$

When computing the fairness of SSD, it is necessary to account for the slowdown experienced by different workloads to reflect relative fairness under co-run conditions. Fairness (F) is defined as Eq.(2), which is the ratio of the maximum and minimum slowdowns of all workloads in SSD. By definition, F lies in the interval $(0, 1]$. A smaller F signifies greater disparity between the minimally and maximally slowed flows, implying reduced fairness for the most-slowed flow. Therefore, higher F-values are desirable.

$$F = \min_i\{slowdown_i\} / \max_j\{slowdown_j\} \tag{2}$$

Fairness, however, must be considered alongside efficiency. Weighted speedup aggregates per-flow improvements (or degradations) relative to isolation and serves as a system-level indicator of overall performance[14]. We can calculate $weighted\ speedup$ (WS) by Eq.(3).

$$WS = \sum_i RT_i^{alone} / RT_i^{shared} \tag{3}$$

By comparing fairness (F) and weighted speedup (WS), we can evaluate both fairness and overall performance of shared SSDs.

III. Motivation

The widespread adoption of NVMe SSDs in multi-tenant computing environments has brought unprecedented opportunities for high performance and scalable storage design. However, efficient resource sharing among competing workloads remains an unresolved challenge. Although numerous fairness mechanisms have improved local scheduling, fundamental limitations persist in matching SSD internal processing capabilities to workload-specific resource demand. These gaps hinder robust performance isolation and contribute to unpredictable application slowdowns. The following three issues highlight critical deficiencies in existing approaches and motivate the need for more comprehensive solutions.

A. Incomplete Alignment Between Workload Demands and SSD Processing Capabilities

While prior schemes typically aim to balance request scheduling across SSD resources, they do so under the implicit assumption that all chips, channels, and dies possess identical performance capacity. In reality, SSD internals are highly heterogeneous—each chip or channel may exhibit variable processing throughput due to device wear, thermal conditions, or localized background operations such as garbage collection. Fairness algorithms generally ignore these per-chip or per-channel discrepancies, yielding suboptimal resource distributions.

This oversight becomes more pronounced in dynamic, multi-tenant contexts, where high-intensity workloads can saturate specific chips or channels, while lighter tenants are disproportionately affected by contention or delays on busy devices. Without capacity-aware scheduling, these workloads experience inconsistent response times and unpredictable throughput. Thus, the lack of mechanisms to explicitly quantify and address global load imbalances across all SSD resources represents a crucial limitation of current approaches.

B. Lack of Holistic Treatment of Workload Diversity

Workload heterogeneity presents additional obstacles for fairness. Tenants may issue requests with vastly different characteristics—such as sequential versus random access, read-dominated versus write-dominated traffic, or varying request sizes and arrival rates. Existing techniques typically draw simple boundaries based on transaction queue size or access pattern, yet fail to capture the multi-dimensional nature of workload diversity at the device level.

As a consequence, optimizations remain highly application-dependent and may deteriorate under mixed or evolving load conditions. For instance, fairness solutions effective for uniformly distributed transaction arrivals can break down in the face of bursty, skewed, or adversarial workloads that disproportionately stress specific SSD resources. True fairness mandates adaptive, per-workload analysis across both temporal and spatial dimensions, informed by continuous collection of workload features and slowdown metrics. The absence of such adaptation in current methods greatly constrains their effectiveness.

C. Absence of Global, Cross-Layer Resource Coordination

SSD fairness frameworks tend to operate narrowly within a single subsystem—be it front-end IO queuing, chip-level dispatching, or transaction-level reordering. Yet, fairness violations can originate anywhere within the hierarchy and propagate throughout the device. Local corrections made in isolation are insufficient to address broad load imbalances and system-wide contention, especially when complex background activities amplify resource competition and latency variability.

Moreover, the lack of global coordination prevents current frameworks from aligning total system capacity with workload-specific feedback. Without a responsive mechanism that measures and redistributes processing power according to observed workload slowdowns, the entire fairness pipeline remains susceptible to instability and oscillation. Achieving stable, robust performance isolation thus requires new methods capable of cross-layer monitoring, dynamic adjustment, and holistic resource management throughout the SSD.

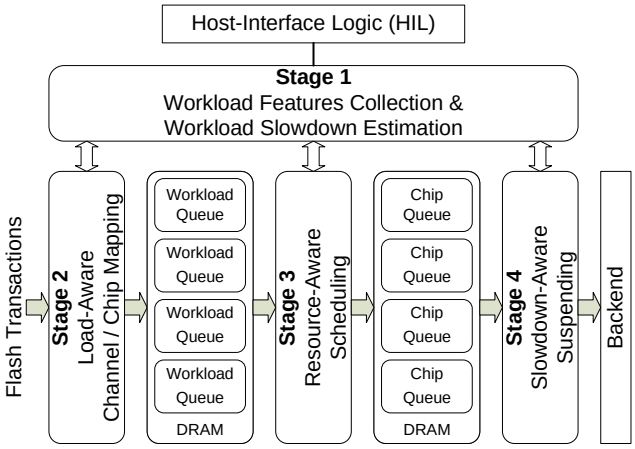

Fig. 2: Overview of GoFair.

In summary, the limitations of prevailing fairness solutions manifested as incomplete resource-to-demand alignment, insufficient workload diversity handling, and lack of global coordination—underscore the urgent need for more advanced frameworks. Addressing these gaps not only pushes the frontier of fairness research in SSDs, but also holds critical implications for multi-tenant cloud performance, predictable service delivery, and sustainable storage architecture design.

IV. Design of GoFair

In this section, we present the design of GoFair. GoFair is a fairness policy for shared SSDs that allocates internal chip resources in proportion to the resource demands of individual workloads. The core idea is to match per-request resource requirements with long-term slowdown estimates for each workload, thereby striking a balance between performance and fairness as closely as possible.

To achieve this goal, we address three key questions. First, how should the processing resources of individual SSD chips be defined, and how should the resource demands of different workloads be modeled, given that the service times for read, program, and erase operations differ substantially? Second, how can slowdown be estimated by jointly considering the resources the SSD can offer and the resources each workload requires, especially since slowdown is a temporal metric whereas SSD resources combine both time and space dimensions, necessitating cross-dimensional matching? Finally, given per-chip resource availability, workload-specific resource demands, and slowdown estimates, how should workloads be scheduled to improve inter-workload fairness?

A. Overview

In conventional SSDs, the controller optimizes for a single workload and strives to maximize internal parallelism.

However, traditional designs do not track which workload each request belongs to, which often leads to unfairness. GoFair, by contrast, must jointly consider overall parallelism and fairness across multiple co-running workloads. To this end, GoFair first models the SSD's resources so that they can be explicitly allocated. It then performs global resource matching by combining each request's resource requirement with a request-level slowdown estimate within its workload.

During the matching of per-workload requests to SSD resources, write requests can be placed on any channel, chip, die, or plane. GoFair exploits this placement flexibility, together with each workload's aggregate slowdown and resource demand, to reduce cross-workload interference. In contrast, read requests have fixed physical addresses and cannot be arbitrarily redirected. For reads, GoFair schedules with awareness of both per-workload resource needs and slowdown, lowering the slowdown of high-slowdown workloads while carefully managing the slowdown of low-slowdown workloads to improve overall fairness. In addition, GoFair leverages suspend operations, triggering them when necessary to balance aggregate throughput with fairness.

Figure 2 illustrates the overall workflow of our approach (GoFair). GoFair comprises four main stages: Workload Features Collection and Workload Slowdown Estimation, Load-Aware Channel and Chip Mapping, Resource-Aware Scheduling, and Slowdown-Aware Suspending.

In *Workload Features Collection* and *Workload Slowdown Estimation*, the module continuously ingests I/O requests from the HIL, extracts features for individual workloads, and estimates each workload's slowdown by combining workload characteristics, the SSD's processing capability, and the realized per-request execution outcomes. In *Load-Aware Channel and Chip Mapping*, the module maps write operations from different workloads to specific chips by accounting for each workload's slowdown and the instantaneous load across chips. In Resource-Aware Scheduling, the module allocates SSD resources over scheduling windows according to each workload's slowdown and the SSD's current processing capacity. In *Slowdown-Aware Suspending*, the module decides whether to issue a suspend operation based on the type of the ongoing operation, the workload it belongs to, the type of the imminent operation and its workload, as well as the corresponding slowdown information for those workloads.

B. Features Collection and Slowdown Estimation

Feature Collection. Because slowdown is computed at the request level, extracting features directly from the raw I/O stream yields more accurate, global estimates of per-workload slowdown. During the feature collection phase, we continuously gather per-stream features from the HIL (Host-Interface Logic) and use them both to compute slowdown and to drive the subsequent mapping, scheduling, and suspending stages. Concretely, we collect the following workload attributes:

- **Transaction Numbers of Each Request.** This records the number of flash-level transactions a request triggers (e.g., page reads or program operations), serving as a proxy for its resource footprint and potential contention.
- **Type of Each Request.** This identifies whether a request is a read or program (write), which is essential because their service times and interference characteristics differ substantially on SSDs.
- **Start LPA of Each Request.** This is the starting logical page address, enabling spatial locality analysis of a request.

Slowdown Estimation. During shared execution, the response time of each request is directly measured based on timestamps recorded within the Host-Interface Logic (HIL). Specifically, we record the enqueue time when the request enters the device queue and the completion time when its result is returned to the host. The difference between these two timestamps reflects the actual latency experienced under interference:

$$RT^{shared} = T^{RQ}_{comp} - T^{RQ}_{enq} \tag{4}$$

We define T^{RQ}_{enq} as the instant the host enqueues a request into the SSD command queue. Conversely, T^{RQ}_{comp} marks the completion event, specifically when the device notifies the host through the HIL.

To infer the performance of a workload running in isolation, we employ an analytical estimation model that reconstructs its expected execution timeline without interference. The controller creates a one-to-one mapping of *virtual chips* corresponding to the physical chips in the SSD. Each virtual chip maintains a record of its most recent transaction finish time, denoted as T^{chip}_{last}, representing the end of the previous operation.

For any new transaction, its estimated standalone latency is calculated as the sum of four components:

$$T^{TR} = T^{RQ}_{HIL} + T^{TR}_{xfer} + T^{TR}_{nand} + T^{TR}_{wait} \tag{5}$$

Here, T^{TR}_{nand} captures the intrinsic latency of NAND-level operations such as page read, program, or erase. T^{RQ}_{HIL} denotes the overhead at the host–interface boundary, and T^{TR}_{xfer} models the time required for internal data transfer between front end and back end. The queuing delay T^{TR}_{wait} accounts for serialization within the virtual chip and is estimated as:

$$T^{TR}_{wait} = max(T^{chip}_{last} - T^{TR}_{enq}, 0) \tag{6}$$

Once a transaction's alone latency is computed, T^{chip}_{last} is updated to reflect its predicted completion time, preserving sequential execution semantics within the virtual chip. A request typically comprises multiple transactions distributed across multiple dies or planes. To approximate the total isolated response time of the entire request, we take the maximum transaction latency within the same request:

$$RT^{alone} = max(T^{TR}) \tag{7}$$

This rule captures the critical-path behavior of internal parallelism—since a request completes only after its slowest

transaction finishes—while maintaining modeling simplicity. Combining the measured shared latency and the estimated isolated latency, the controller can continuously evaluate the real-time performance degradation of each workload.

C. Load-Aware Mapping

Unlike read operations, which must access specific physical addresses where data is stored, write operations (program requests) offer significant flexibility in data placement. GoFair exploits this flexibility to balance load across channels and chips while actively enforcing fairness. The primary objective of the Load-Aware Mapping module is to dispatch transactions from different tenants evenly across physical resources, ensuring that no single chip becomes a bottleneck while simultaneously correcting unfairness based on real-time slowdown estimates.

To achieve this, we first define the instantaneous load of a physical chip, denoted as L_{chip}. We characterize L_{chip} by the aggregate depth of the pending transaction queue on that specific chip. A higher L_{chip} implies a longer waiting time for any newly mapped transaction, thus increasing the latency of the request it belongs to.

The mapping strategy operates by comparing the current slowdown of the workload issuing the request, S_{wkld}, against a system-wide fairness threshold, S_{thresh}. The decision logic proceeds as follows:

- **Prioritizing High-Slowdown Workloads:** If a workload's slowdown exceeds the threshold ($S_{wkld} > S_{thresh}$), it indicates that the tenant is experiencing unfair performance degradation compared to others. To mitigate this, GoFair selects the chip with the minimum load ($min(L_{chip})$) among all available channels and chips. By mapping the transaction to the least congested resource, we minimize the queuing delay (T^{TR}_{wait}), thereby reducing the request's response time and helping the workload's slowdown converge toward the fair target.
- **Throttling Low-Slowdown Workloads:** Conversely, if a workload's slowdown is below the threshold ($S_{wkld} \leq S_{thresh}$), the tenant is running faster than its fair share, potentially at the expense of other workloads. To rectify this, GoFair maps the transaction to the chip with the maximum load ($max(L_{chip})$). Placing the transaction on a busy chip intentionally introduces queuing delay, increasing the workload's slowdown. Crucially, this strategy preserves the processing capacity of idler chips for workloads that are currently suffering from high slowdowns.

Algorithm 1 formalizes this decision process, illustrating how spatial placement decisions are leveraged to enforce temporal fairness constraints. The procedure begins by retrieving the real-time performance metrics of the active workload. Specifically, it calculates the current slowdown, denoted as $S_{current}$, and simultaneously scans the state of the underlying storage hardware to determine the instantaneous load L_{c_i} for every available chip in the set C (Lines 1–4). This initial state gathering is crucial, as it provides the necessary context to make an informed mapping decision that aligns with global system objectives.

Algorithm 1 Load-Aware Channel and Chip Mapping

Require: Write Transaction Tr, Workload W, Chip Set C, Threshold S_{thresh}

1: $S_{current} \leftarrow$ GetCurrentSlowdown(W)
2: **for** each chip $c_i \in C$ **do**
3: $L_{c_i} \leftarrow$ GetQueueDepth(c_i)
4: **end for**
5: **if** $S_{current} > S_{thresh}$ **then**
6: `/* Victim Strategy: Pick the least loaded chip to minimize latency */`
7: $c_{target} \leftarrow \arg\min_{c_i \in C}(L_{c_i})$
8: **else**
9: `/* Dominator Strategy: Pick the most loaded chip to yield resources */`
10: $c_{target} \leftarrow \arg\max_{c_i \in C}(L_{c_i})$
11: **end if**
12: Assign Tr to c_{target}
13: **return** c_{target}

The core logic of the algorithm branches based on the relationship between the workload's performance and the system-wide fairness target. If the workload is suffering from excessive delays (i.e., $S_{current} > S_{thresh}$), the algorithm identifies it as a "victim." In this scenario, the priority is latency reduction; therefore, the algorithm searches the chip set to find and select the target c_{target} with the minimum queue depth (Lines 5–7). This ensures the write transaction is processed as quickly as possible, aiding the workload's recovery. Conversely, if the workload is performing better than the fair share (i.e., $S_{current} \leq S_{thresh}$), it is treated as a "dominator." The algorithm then selects the most heavily loaded chip (Lines 8–10), intentionally subjecting the transaction to existing congestion to throttle the workload naturally.

Finally, the transaction is dispatched to the selected physical address (Line 12). By consistently applying this logic to every incoming write request, the system dynamically modulates the service time experienced by different tenants. This mechanism effectively converts the inherent spatial flexibility of flash memory programming into a fine-grained tool for temporal fairness control, ensuring that resource contention is managed proactively at the moment of data placement.

D. Resource-Aware Scheduling

While Load-Aware Mapping addresses the spatial distribution of I/O requests across different chips, it does not guarantee fairness once requests are queued on a specific chip. Without temporal control, a workload with a high request arrival rate could easily monopolize a chip's service time, starving other tenants even if they are mapped to the same resource. To address this, GoFair implements a *Resource-Aware Scheduling* policy that functions as a gatekeeper for per-chip execution. This module operates on a time-window basis, ensuring that

chip resources are shared proportionally according to tenant needs while maintaining high device utilization.

The core mechanism relies on tracking the cumulative service time consumed by each workload within a fixed scheduling window (T_{win}). We classify workloads into two categories based on their real-time performance relative to the fairness threshold: *favored* tenants (slowdown $< S_{thresh}$) and *suffering* tenants (slowdown $\geq S_{thresh}$).

Quota-Based Throttling for Fairness. The primary source of unfairness in shared SSDs is often a "noisy neighbor," which is defined as a workload that not only performs well but also aggressively consumes resources. GoFair imposes a strict resource quota (U_{limit}) on these favored tenants. When the scheduler prepares to dispatch the next operation for a chip, it inspects the accumulated usage of the candidate workload. If a favored tenant has already exceeded its quota U_{limit} within the current window, it is temporarily deemed "greedy." To protect the performance of other tenants, GoFair blocks this tenant from issuing further commands to the chip until the next scheduling window begins. This preemptive blocking effectively reserves the remaining time slots in the window for suffering tenants, allowing them to clear their backlogs and reduce their slowdowns.

Idle-Timeout Safeguard for Throughput. Strictly blocking active tenants can lead to a scenario where a chip sits idle because all eligible (non-blocked) tenants have empty queues, even though blocked tenants have requests waiting. This behavior, known as non-work-conserving scheduling, hurts the aggregate throughput of the SSD. To prevent resource wastage, GoFair introduces a fail-safe mechanism: the *Idle-Timeout Safeguard*.

The controller monitors the idle duration of each chip. If a chip remains inactive for a period exceeding a critical threshold (τ_{idle}) while there are pending requests in the blocked queues, the system infers that strict fairness is hindering system efficiency. In this case, the scheduler overrides the throttling rules. It scans all blocked workloads and forcefully schedules a request from the tenant with the highest current slowdown. This hybrid approach ensures a robust trade-off: in times of contention, fairness is strictly enforced via blocking; during periods of potential underutilization, the system reverts to work-conserving behavior to maximize performance.

Algorithm 2 delineates the operational flow of this policy, which is structured into two primary phases: filtering and dispatching. The procedure begins by measuring the chip's current inactivity duration, T_{idle}, and initializing the set of eligible candidates (Lines 1–2). The core logic (Lines 3–13) iterates through active workloads to evaluate their fairness status relative to their resource consumption. A critical check is performed here: if a tenant is identified as "favored" (i.e., low slowdown) yet has exceeded its usage quota U_{limit}, it is strictly excluded from the candidate pool. This preemptive exclusion serves as the primary mechanism for preventing resource monopolization by aggressive workloads.

Following the filtering phase, the scheduler determines the dispatch action based on candidate availability (Lines 14–23). In standard operation, requests are selected from the eligible set using a baseline policy like Round-Robin to ensure proportional sharing among well-behaved tenants. However, to prevent strict fairness from harming aggregate throughput, the algorithm includes a fail-safe: if the candidate set is empty but pending requests exist, the system checks if the chip's idle time exceeds τ_{idle}. In such instances, the throttling rules are overridden, and the scheduler prioritizes the workload with the maximum slowdown, dynamically reverting to work-conserving behavior to maintain high device utilization.

Algorithm 2 Resource-Aware Scheduling

Require: Workload Set W, Chip C, Window T_{win}, Limit U_{limit}

1: $T_{idle} \leftarrow$ Time since last operation on C
2: $Candidates \leftarrow \emptyset$
3: /* **Phase 1: Fairness-Based Filtering** */
4: **for** each workload $w_i \in W$ with pending requests **do**
5: $S_i \leftarrow$ Slowdown(w_i)
6: $Used \leftarrow$ ConsumedTimeSlice(w_i)
7: **if** $S_i < S_{thresh}$ **and** $Used > U_{limit}$ **then**
8: /* Tenant is favored and greedy: **Block** */
9: Continue
10: **else**
11: Add w_i to $Candidates$
12: **end if**
13: **end for**
14: /* **Phase 2: Dispatch or Override** */
15: **if** $Candidates \neq \emptyset$ **then**
16: Schedule request from $Candidates$ (e.g., Round-Robin)
17: **else if** $T_{idle} > \tau_{idle}$ **and** W has pending requests **then**
18: /* **Idle Timeout:** Override blocking to avoid waste */
19: $w_{suffering} \leftarrow$ Workload in W with max Slowdown
20: Schedule request from $w_{suffering}$
21: **else**
22: Wait (Chip Idle)
23: **end if**

E. Slowdown-Aware Suspending

While mapping and scheduling optimize resource distribution before execution begins, they cannot intervene once a flash command has been issued to the chip. NAND flash operations, particularly program and erase, have long latencies (hundreds of microseconds to milliseconds), which can block high-priority requests from suffering tenants. To address this, GoFair leverages the *Suspend/Resume* feature of modern NAND flash memory as a fine-grained, post-dispatch correction mechanism.

The decision to trigger a suspend operation is non-trivial because suspending incurs overhead (latency penalties for the suspended operation and command bus overhead). GoFair employs a strict policy that trades off these overheads only

when a significant fairness benefit is guaranteed. The policy evaluates two factors: **Slowdown Urgency** and **Hardware Feasibility**.

Slowdown Urgency. The controller continuously compares the slowdown of the workload currently owning the chip (W_{exec}) with the slowdown of the workload at the head of the wait queue (W_{head}). A suspend operation is considered justifiable only if the waiting tenant is in a worse state than the executing tenant. Specifically, if $Slowdown(W_{head}) > Slowdown(W_{exec})$, it implies that the executing operation is blocking a more "suffering" tenant. In this scenario, preempting the current operation helps the waiting tenant reduce its queuing delay, thereby balancing the global fairness.

Hardware Feasibility. Even if the urgency condition is met, suspension is physically constrained by the NAND flash operational hierarchy. Generally, an operation with lower latency can suspend one with significantly higher latency, but not vice versa. GoFair adheres to the standard priority rules:

- A **Read** request can suspend a **Program** or **Erase** operation.
- A **Program** request can suspend an **Erase** operation.

Algorithm 3 formalizes this decision process. The controller checks the state of each busy chip upon the arrival of a new request. If the incoming request belongs to a workload with a higher slowdown than the active workload, and the operation types permit preemption, a Suspend command is issued immediately.

Algorithm 3 Slowdown-Aware Suspending

Require: Active Chip C, Executing Transaction Tr_{exec}, Head of Queue Transaction Tr_{head}

$W_{exec} \leftarrow$ GetWorkload(Tr_{exec})
$W_{head} \leftarrow$ GetWorkload(Tr_{head})
$S_{exec} \leftarrow$ GetSlowdown(W_{exec})
$S_{head} \leftarrow$ GetSlowdown(W_{head})
`/* Condition 1: Fairness Benefit (Is the waiter more desperate?) */`
if $S_{head} \leq S_{exec}$ **then**
  `/* No fairness gain, do not suspend */`
  return false
end if
`/* Condition 2: Hardware Feasibility (Can we suspend?) */`
$Type_{exec} \leftarrow$ GetOpType(Tr_{exec})
$Type_{head} \leftarrow$ GetOpType(Tr_{head})
if ($Type_{head}$ is READ **and** $Type_{exec} \in \{PROG, ERASE\}$) **then**
  Issue **Suspend Command** to C
  return true
else if ($Type_{head}$ is PROG **and** $Type_{exec}$ is ERASE) **then**
  Issue **Suspend Command** to C
  return true
else
  `/* Operation combination not supported */`
  return false
end if

V. Evaluation

In this section, we present a comprehensive evaluation of our approach using simulation-based experiments. We conduct experiments using MQSim[15], a widely-used open-source SSD simulator that accurately models the major front-end and back-end components of modern SSDs.

A. Experimental Setup

1) Simulated SSD Configuration: Our experiments are conducted on a simulated NVMe SSD with configurations shown in Table I. The SSD features a PCIe 3.0 interface with NVMe 1.2 protocol, providing high-speed communication between the host and the device. The storage architecture consists of 8 channels, with 4 chips per channel, 2 dies per chip, and 2 planes per die, enabling four-level parallelism. Each plane contains 2048 blocks, and each block has 256 pages with 8 KB page capacity, resulting in a total user capacity of 480 GB. The device exhibits typical latency characteristics for read (75 μs), write (1,300 μs), and erase (3.8 ms) operations.

TABLE I: Configurations of the Simulated SSD

Parameter	Value
Host Interface	PCIe 3.0 / NVMe 1.2
User Capacity	480 GB
Channel/Chip/Die/Plane	8/4/2/2
Block/Page	2048/256
Page Capacity	8 KB
Latency of Read/Write/Erase	75 μs / 1,300 μs / 3.8 ms

The evaluated SSD configuration is representative of modern enterprise NVMe SSD architectures commonly adopted in prior MQSim-based studies. Although hardware parameters such as queue depth and chip count may affect the absolute performance and interference behavior, our work primarily focuses on the fairness mechanisms under realistic multi-tenant contention scenarios. A comprehensive sensitivity analysis across broader hardware configurations is left for future work.

2) Evaluated Workloads: We use real-world I/O traces to evaluate our system under realistic workload conditions. Table II presents the characteristics of 12 workloads collected from Microsoft, MSR Cambridge, and Florida International University (FIU) traces [16]. These workloads represent diverse application scenarios including OLTP applications (tpce, tpcc1, tpcc2), mail servers (exch), build services (bs), file servers (msnfs), remote access backends (lms-be), web servers (wsrch), financial services (fin), educational activities (cheetah), firewalls/web proxies (prxy), and source control systems (src). The workloads exhibit diverse characteristics in terms of read/write ratios (ranging from 0.10 to 0.99), average request sizes (2.25 KB to 126.09 KB), and arrival intervals (0.003 ms to 13.24 ms), providing comprehensive coverage

TABLE II: Characteristics of the Evaluated Workloads

Workload	Application	Description	Read Ratio	Avg. Read Size (KB)	Avg. Write Size (KB)
tpce	OLTP application	Transaction processing	0.92	8.00	12.58
tpcc1	OLTP application	Transaction processing	0.63	8.08	9.36
tpcc2	OLTP application	Transaction processing	0.61	8.14	9.55
exch	Mail server	Email storage	0.94	8.35	18.26
bs	OS build service	Build operations	0.53	7.78	7.72
msnfs	File server	MSN storage	0.80	14.88	10.54
lms-be	Remote access	Back-end service	0.78	110.66	126.09
wsrch	Web server	Apache service	0.99	15.15	8.61
fin	Financial service	Financial transactions	0.23	2.25	3.73
cheetah	Education	Educational activities	0.10	8.00	8.00
prxy	Firewall/proxy	Web proxy service	0.65	24.59	25.91
src	Source control	Version control	0.95	66.21	27.47

TABLE III: Mixes of Workloads

Mix ID	Workloads
Mix1	prxy, tpcc1, tpce, bs
Mix2	tpce, src, fin, tpcc1
Mix3	prxy, fin, tpcc1, lms-be
Mix4	tpce, src, tpcc1, exch
Mix5	exch, fin, cheetah, tpce
Mix6	fin, exch, src, lms-be
Mix7	exch, tpcc1, cheetah, bs
Mix8	tpce, tpcc2, lms-be, msnfs
Mix9	bs, tpcc2, exch, msnfs
Mix10	tpcc1, src, lms-be, prxy
Mix11	msnfs, tpce, cheetah, bs
Mix12	tpce, wsrch, fin, tpcc1
Mix13	lms-be, tpcc1, fin, bs
Mix14	tpcc2, cheetah, msnfs, exch
Mix15	src, tpcc1, tpce, exch
Mix16	bs, tpcc1, exch, prxy

of different I/O patterns. Although several traces originate from established benchmark repositories, they are still widely used in prior SSD scheduling and QoS studies, enabling reproducible evaluation across diverse I/O patterns.

3) Workload Mixes: To evaluate our system under multi-tenant scenarios, we create 16 different workload mixes by randomly combining four workloads from our evaluated set. Table III presents these combinations, which are designed to capture various interference patterns and resource contention scenarios that occur in shared SSD environments. Each mix represents a realistic multi-tenant scenario where different applications with varying I/O characteristics compete for SSD resources.

4) Baselines: To demonstrate the effectiveness of our framework, we compare it with three representative state-of-the-art schemes that target different design goals:

- **Clustering**[11]: A resource partitioning approach that dynamically groups flash chips into clusters and assigns specific workloads to dedicated clusters. By physically isolating the resources used by different tenants, it aims to eliminate inter-application interference and provide timing guarantees, serving as a representative isolation-based baseline.
- **FLIN** [5]: A fairness-aware I/O scheduler designed for modern NVMe SSDs. It employs a multi-stage scheduling mechanism within the SSD controller to identify and mitigate various sources of interference (e.g., I/O intensity, access patterns, and read/write disparity), aiming to ensure fair slowdowns among concurrent workloads.
- **QoS-pro**[12]: A recently proposed QoS-enhanced transaction processing framework. It redesigns the internal address translation and transaction scheduling layers (e.g., using global address mapping and capability-aware scheduling) to enhance fairness guarantees while preserving the internal parallelism of shared SSDs.

To ensure result stability, all experiments were repeated multiple times using identical configurations and workload traces. The observed variance across runs was negligible; therefore, error bars are omitted from the figures for clarity and readability.

B. Evaluation Metrics

We employ five key metrics to comprehensively evaluate the performance and fairness characteristics of our system:

Fairness. We measure fairness as the ratio of minimum to maximum slowdowns among concurrent workloads, where slowdown is defined as the ratio of average latency when running concurrently to average latency when running alone. A fairness value close to 1 indicates perfect fairness, while lower values indicate unfair resource allocation.

Normalized IOPS. We normalize the IOPS achieved by each workload mix to facilitate comparison across different configurations. Higher normalized IOPS indicates better utilization of SSD parallelism and overall throughput.

Maximum Slowdown (MS). This metric captures the worst-case performance degradation experienced by any workload in a concurrent environment. Lower MS values indicate better QoS guarantees for individual workloads.

Standard Deviation (STDEV). We compute the standard deviation of slowdowns across all concurrent workloads to measure the consistency of performance. Lower STDEV values indicate more uniform performance across workloads.

Weighted Speedup. This metric evaluates the overall system efficiency by computing the sum of inverse slowdowns across all workloads. Higher weighted speedup indicates better overall system performance while maintaining fairness.

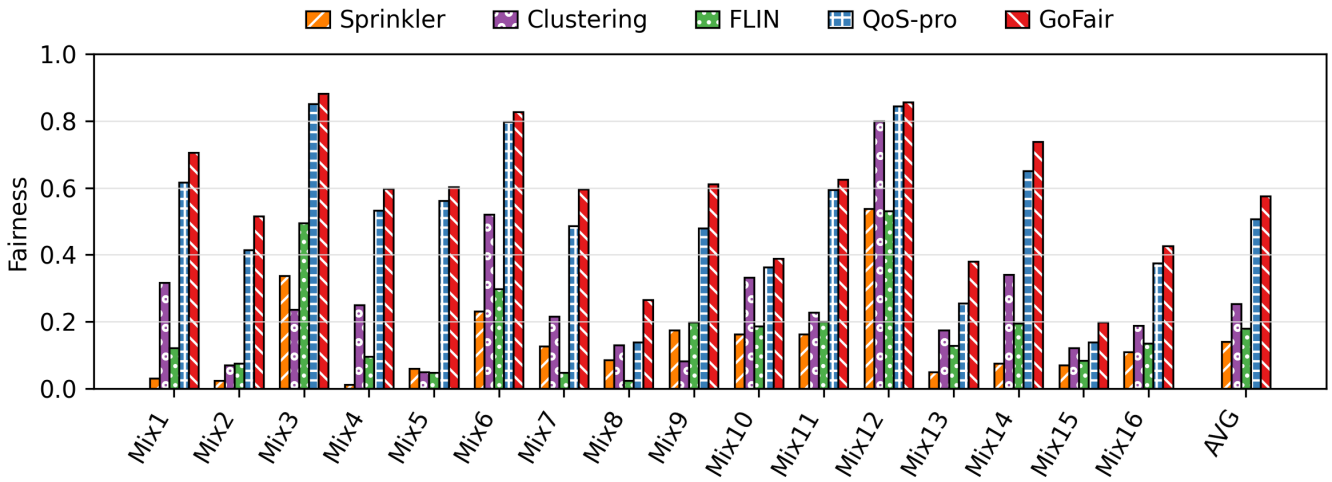

Fig. 3: Fairness comparison across 16 workload mixes. Higher is better.

C. Experimental Results

In this section, we present the experimental results of our proposed framework compared with the state-of-the-art scheduling schemes. We focus on five key metrics: Fairness, Normalized IOPS, Maximum Slowdown, Standard Deviation (STDEV) of slowdowns, and Weighted Speedup across the 16 workload mixes defined in Table III.

1) Fairness Analysis: We first evaluate the effectiveness of our framework in ensuring fairness among concurrent workloads. Figure 3 illustrates the fairness results for the 16 workload mixes. The fairness metric, defined as the ratio of the minimum slowdown to the maximum slowdown among concurrently running workloads, ranges from 0 (completely unfair) to 1 (perfectly fair).

As shown in Figure 3, GoFair consistently achieves high fairness scores across all mixes, with an average fairness of 0.576, significantly outperforming all baselines. In scenarios with high interference (e.g., Mix 2 and Mix 6), where read-intensive and write-intensive workloads co-exist, baseline methods often suffer from severe unfairness due to the dominance of aggressive workloads. In contrast, GoFair effectively mitigates this interference, maintaining a balanced slowdown distribution among all tenants.

Why GoFair achieves superior fairness. GoFair attains robust fairness because it enforces slowdown-aware control at multiple points along the SSD IO pipeline: (i) load-aware mapping exploits write-placement flexibility to steer high-slowdown workloads to lightly loaded chips while naturally throttling low-slowdown ones, (ii) resource-aware scheduling applies window-based quota throttling to prevent favored tenants from monopolizing per-chip service time, and (iii) slowdown-aware suspending leverages NAND suspend/resume to preempt long operations when a more suffering tenant would otherwise be blocked, jointly balancing fairness in both spatial placement and temporal execution.

Analysis of baseline limitations. In contrast, parallelism-oriented schedulers (e.g., Sprinkler) mainly maximize internal parallelism without explicit slowdown tracking, so aggressive tenants can dominate shared resources and amplify unfairness under read/write asymmetry, while FLIN and Clustering are constrained by limited granularity and/or rigid partitioning that adapts poorly to workload heterogeneity, and QoS-Pro, though more QoS-aware, can still struggle when interference is extreme and its scheduling decisions are too coarse to continuously equalize slowdowns.

2) Performance and System Efficiency: To assess the impact of our QoS mechanisms on the overall system throughput, we measure the normalized IOPS and weighted speedup.

Figure 4 presents the normalized IOPS for the evaluated mixes. While enforcing fairness often comes at the cost of aggregate throughput in traditional QoS schemes, our framework maintains high parallelism utilization. The results indicate that our method incurs negligible throughput overhead compared to parallelism-first schedulers, achieving comparable IOPS even in write-heavy scenarios.

Reasons for high throughput. GoFair achieves near-optimal throughput (averaging 98.2% of the baseline) primarily due to two design features that prevent resource wastage. First, the *Load-Aware Mapping* dynamically directs write traffic to the least congested chips to assist suffering workloads. This acts as an effective load balancer, ensuring that requests are evenly distributed across all available channels and chips, thereby maximizing global internal parallelism. Second, the *Resource-Aware Scheduling* incorporates a critical "idle-timeout safeguard." Unlike rigid fairness schemes that might leave a chip idle to penalize an aggressive tenant (non-work-conserving), GoFair monitors chip inactivity; if a chip remains idle beyond a threshold (T_{idle}), the scheduler overrides fairness constraints and dispatches requests from blocked tenants. This hybrid approach ensures that hardware resources are never wasted solely for the sake of fairness. In contrast, static partitioning methods like Clustering suffer from resource fragmentation, achieving only 79.2% of the baseline IOPS on average.

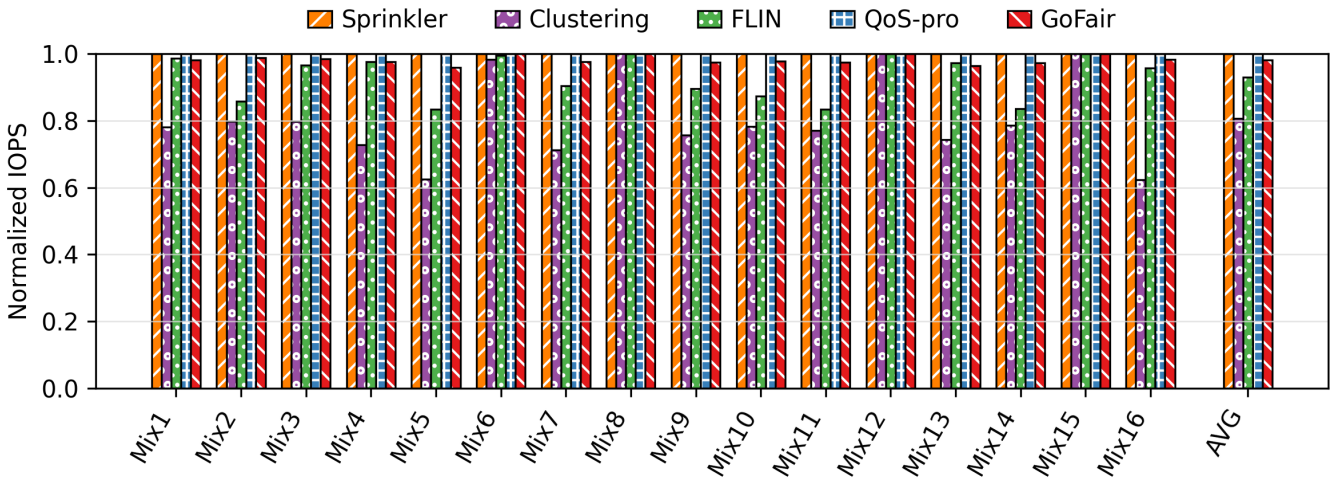

Fig. 4: Normalized IOPS across workload mixes.

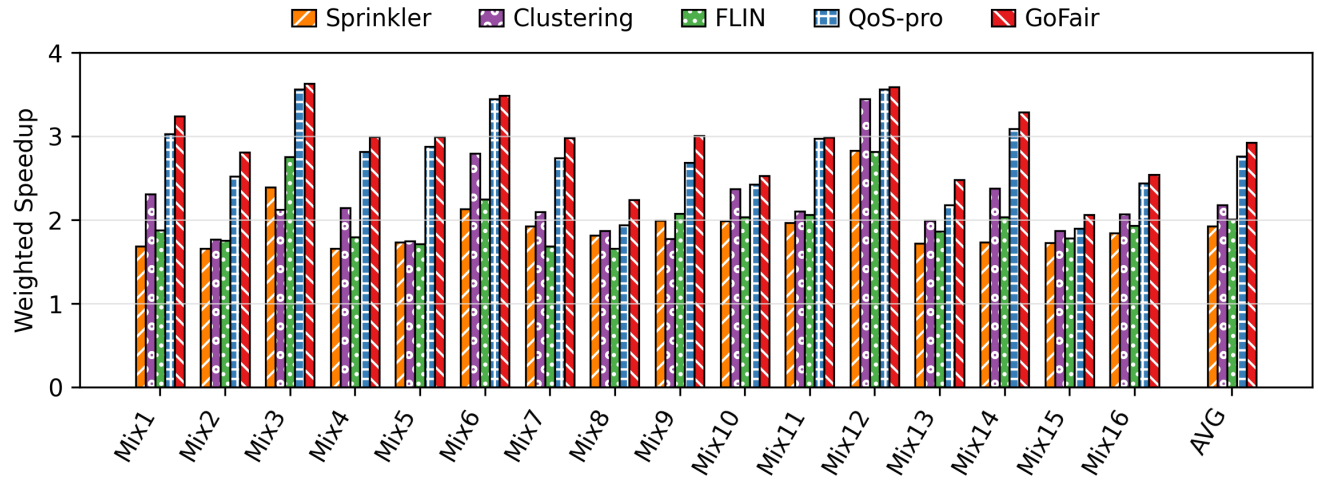

Fig. 5: Weighted speedup across workload mixes. Higher is better.

Discussion on Throughput Tradeoff. Although QoS-pro achieves slightly higher normalized IOPS in several workload mixes, this behavior is expected due to the different optimization emphases of the two frameworks. QoS-pro also considers fairness and QoS guarantees, but it primarily focuses on preserving SSD internal parallelism and maintaining high I/O utilization through work-conserving scheduling. In contrast, GoFair more aggressively prioritizes fairness through slowdown-aware throttling, load-aware request placement, and suspend-based interference mitigation. These fairness-oriented mechanisms inevitably introduce minor overheads, including additional queue waiting time and occasional suspend/resume penalties, which slightly reduce peak throughput. Nevertheless, the throughput loss remains limited, demonstrating that GoFair can substantially improve fairness and reduce slowdown variance without significantly compromising overall SSD efficiency.

Figure 5 shows the weighted speedup, which reflects the overall system efficiency. By preventing any single workload from monopolizing system resources, our framework improves the collective progress of all concurrent applications. We observe that our approach achieves the highest weighted speedup in the majority of the mixes, demonstrating its ability to balance individual QoS requirements with global system efficiency.

3) Latency Consistency and QoS Guarantees: Finally, we analyze the stability of performance guarantees using Maximum Slowdown (MS) and the Standard Deviation (STDEV) of slowdowns. These metrics are critical for evaluating the worst-case performance and the predictability of the system.

Figure 6 presents the results for Maximum Slowdown. A lower Maximum Slowdown indicates that the system successfully prevents any individual workload from suffering excessive delays. Our framework significantly reduces the Maximum Slowdown compared to the baselines, particularly in mixes containing latency-sensitive workloads.

Analysis of slowdown reduction. GoFair achieves the lowest average maximum slowdown, effectively capping the

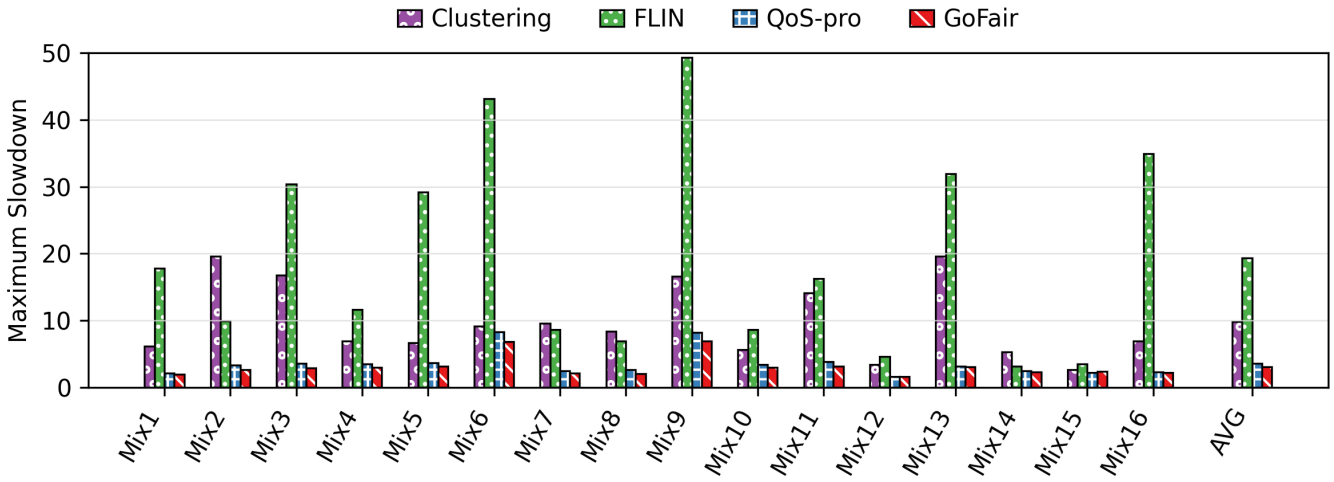

Fig. 6: Maximum Slowdown. Lower is better.

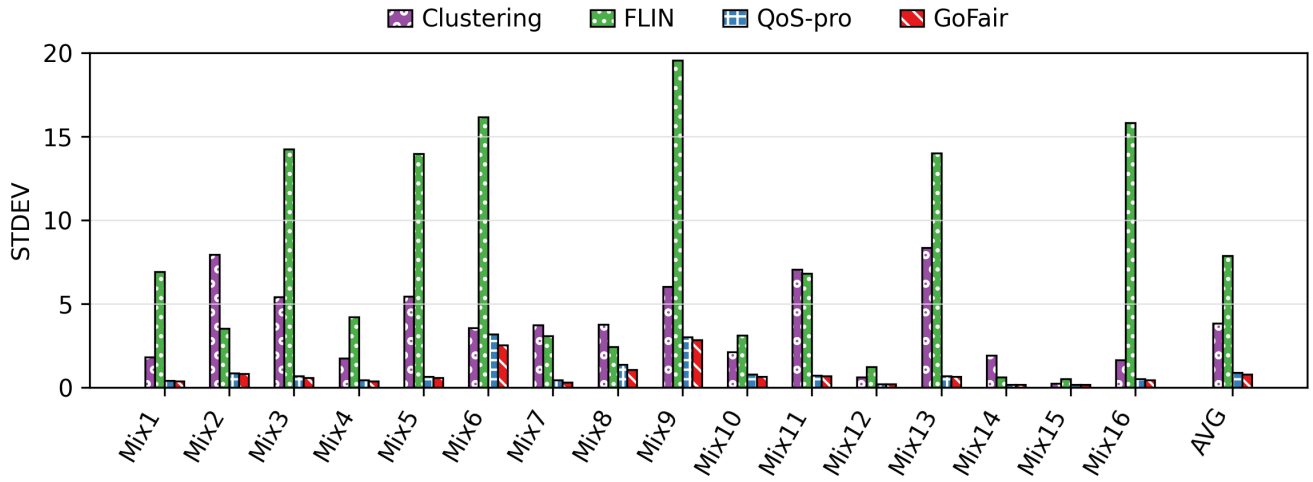

Fig. 7: STDEV of Slowdowns. Lower is better.

worst-case latency. This is because its *mapping* and *scheduling* policies explicitly prioritize "suffering" tenants—those with high slowdowns are routed to the least loaded chips and given scheduling priority, preventing any single workload from starving. In contrast, FLIN exhibits high volatility, as its static interference handling often fails to react fast enough to bursty traffic, leading to extreme delays in Mix 6 and 9. Clustering suffers when a workload's demand exceeds its assigned cluster capacity (e.g., Mix 2, 13), creating inescapable bottlenecks. While QoS-Pro performs well, GoFair further reduces maximum slowdown on average, proving that fine-grained, request-level intervention is necessary to protect vulnerable tenants in highly contended scenarios.

Furthermore, as shown in the STDEV results in Figure 7, our method achieves the lowest variation in slowdowns among concurrent workloads. This low standard deviation confirms that our scheduling policy treats all workloads equitably, providing predictable and consistent performance isolation regardless of the workload intensity or access patterns.

Comparison with QoS-Pro. The results show that both GoFair and QoS-Pro maintain a much lower STDEV than other baselines. This similarity exists because both methods use dynamic, feedback-driven scheduling to continuously adjust resource allocation based on real-time slowdowns, effectively preventing any single tenant from deviating too far from the group average. However, GoFair achieves a tighter distribution, particularly in mixes with heavy write interference (e.g., Mix 6). This is attributed to GoFair's *Slowdown-Aware Suspending* mechanism. While QoS-Pro can only reorder requests at the queue level, it forces read-intensive tenants to wait for ongoing write operations to complete (head-of-line blocking), which temporarily spikes the waiting tenant's slowdown and increases the variance across tenants. GoFair, by preempting these long-latency operations, immediately serves the victim tenant, keeping its slowdown closer to the others and thereby minimizing the standard deviation.

VI. Related Work

Existing research on multi-tenant isolation in SSDs primarily addresses three critical dimensions: resource scheduling, I/O isolation, and space isolation. In this section, we review these strategies, discussing the associated operational mechanisms and comparative trade-offs.

A. Resource Scheduling for Multiple Tenants

Early research largely concentrated on mitigating specific sources of interference to preserve Quality of Service (QoS). For instance, Kim et al. [17] presented an OPS isolation scheme designed to secure performance service level objectives (SLOs) in virtualized environments by explicitly mitigating the impact of Garbage Collection (GC) interference among virtual machines. Addressing contention at the I/O queue level, FlashFQ [18] operates as a fair queueing I/O scheduler. It balances fairness and responsiveness by leveraging throttled dispatch and I/O anticipation, thereby effectively managing resource contention between read and write requests.

More recent approaches have shifted focus toward internal cache management to further enhance isolation. Justitia [19] introduces a DRAM-based Over-Provisioning (OP) cache management mechanism to decouple tenant data. By employing distinct Static-OP and Dynamic-OP stages, it reduces data cache contention and minimizes the interference caused by background GC operations. Similarly, CoFS [20] coordinates SSD cache and Transaction Scheduling Unit (TSU) scheduling through a collaboration-aware scheme. It employs reinforcement learning to dynamically balance fairness at both the workload and flash levels. Despite these advancements, these approaches focus predominantly on cache-level or queue-level scheduling. They lack the capability for finer-grained plane-level isolation, which limits their efficiency in managing deep contention in high-density multi-tenant scenarios.

B. I/O Isolation for Multiple Tenants

A prevalent strategy for I/O isolation involves physically partitioning SSD resources to prevent interference entirely. FlashBlox [3] exploits the intrinsic parallelism of flash memory to dedicate specific channels and dies to individual tenants. To address the resulting wear leveling challenges, it dynamically manages wear imbalance, ensuring both rigorous isolation and a uniform SSD lifetime. Sharing this philosophy of hardware dedication, Clustering [11] dynamically groups flash chips into isolated clusters, assigning physical resources to specific workloads to eliminate inter-tenant interference. To improve resource utilization in such shared environments, Kim et al. [21] presented UPI, a scheme that quantifies and optimizes utility. By leveraging internal SSD parallelism more flexibly, UPI outperforms rigid static partitioning techniques in both latency and throughput.

Other works focus on optimizing the scheduling pipeline itself rather than enforcing strict physical partitioning. D2FQ [22] offloads the scheduling burden to the device by utilizing NVMe's weighted round-robin arbitration. This design reduces the traditional three-step host-level scheduling process to a single step, significantly improving block I/O performance while saving host CPU cycles. To address application-level unfairness inherent in modern multi-queue architectures, FLIN [5] implements a robust three-stage scheduling algorithm within the SSD firmware. Furthermore, QoS-pro [12] redesigns the address translation and transaction scheduling layers to prioritize low-intensity workloads, maintaining high internal parallelism. Despite their effectiveness, these strategies employ coarse-grained isolation mechanisms that lack the fine-grained control provided by plane-level isolation. Such precision is critical for balancing performance and space utilization while avoiding the pitfalls of over-segmentation.

C. Space Isolation for Multiple Tenants

Space isolation techniques typically target specific storage hierarchies or data structures to separate tenant data. SSDKeeper [23] features a self-adapting channel allocation mechanism for shared SSDs. It employs a machine learning-assisted algorithm to analyze multi-tenant access patterns and optimize channel allocation accordingly. Tailored specifically for Key-Value storage, Iso-KVSSD [24] maintains per-namespace LSM-trees to ensure strictly isolated access control. This approach achieves high read throughput with minimal write overhead compared to traditional shared LSM-tree baselines. In the context of redundancy and reliability, LiveSSD [25] offers a low-interference RAID scheme for hardware-virtualized SSDs. By utilizing high-speed NVRAM for parity storage and proactively updating parity during idle periods, it minimizes the I/O interference typically associated with RAID updates. However, these approaches concentrate primarily on channel-level, namespace-level, or RAID-level isolation. The absence of finer-grained plane-level partitioning restricts their ability to achieve flexible and efficient resource allocation, particularly in complex multi-tenant environments where workload characteristics vary rapidly.

VII. Conclusion

In this paper, we presented GoFair, a global-oriented fairness framework designed to address performance interference in multi-tenant NVMe SSDs. In contrast to existing approaches that prioritize local reordering or parallelism-centric optimizations, GoFair fundamentally aligns workload demands with the heterogeneous and bounded processing capabilities of internal SSD components. Through a comprehensive integration of load-aware mapping, resource-aware scheduling, and slowdown-aware suspending, GoFair effectively harmonizes resource allocation across diverse workloads. Our extensive evaluation using MQSim and real-world traces demonstrates that GoFair consistently outperforms state-of-the-art schemes, delivering a 13.72% improvement in fairness over QoS-pro while maintaining high system throughput.

Future work will focus on prototyping GoFair on physical SSD hardware and extending the framework to multi-SSD storage environments. We also plan to conduct broader sensitivity studies across different hardware configurations, such

as varying queue depths and chip organizations, to further evaluate the robustness of GoFair under diverse deployment scenarios. Additionally, we intend to integrate broader QoS guarantees, including strict tail-latency Service Level Objectives (SLOs), and refine our resource management strategies to better account for internal maintenance operations such as garbage collection.

ACKNOWLEDGMENTS

This work is funded by the National Key Research and Development Program (No.2022YFB4501300) and the Shenzhen Science and Technology Program, China (No.JCYJ20230807143706014).

During the preparation of this manuscript, the authors used AI-assisted tools for language translation, grammatical refinement, and LaTeX formatting assistance. All technical ideas, experiments, analyses, and conclusions were developed, verified, and approved by the authors.

REFERENCES

[1] NVM Express Workgroup, "Nvm express 2.3 specification." https://nvmexpress.org/specifications/, 2025.

[2] M. Hedayati, K. Shen, M. L. Scott, and M. Marty, "Multi-Queue fair queuing," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 301–314.

[3] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi, "FlashBlox: Achieving both performance isolation and uniform lifetime for virtualized SSDs," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, Feb. 2017, pp. 375–390.

[4] S. Tripathy, D. Sahoo, M. Satpathy, and M. Mutyam, "Fuzzy fairness controller for nvme ssds," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. New York, NY, USA: Association for Computing Machinery, 2020.

[5] A. Tavakkol, M. Sadrosadati, S. Ghose, J. Kim, Y. Luo, Y. Wang, N. Mansouri Ghiasi, L. Orosa, J. Gómez-Luna, and O. Mutlu, "Flin: Enabling fairness and enhancing performance in modern nvme solid state drives," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 397–410.

[6] R. Liu, X. Chen, Y. Tan, R. Zhang, L. Liang, and D. Liu, "Ssdkeeper: Self-adapting channel allocation to improve the performance of ssd devices," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 966–975.

[7] S. Park and K. Shen, "FIOS: A fair, efficient flash I/O scheduler," in *10th USENIX Conference on File and Storage Technologies (FAST 12)*. San Jose, CA: USENIX Association, Feb. 2012.

[8] K. Shen and S. Park, "FlashFQ: A fair queueing I/O scheduler for Flash-Based SSDs," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX Association, Jun. 2013, pp. 67–78.

[9] W. Choi, B. Urgaonkar, M. Kandemir, M. Jung, and D. Evans, "Fair write attribution and allocation for consolidated flash cache," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1063–1076.

[10] Y. Zhang, P. Huang, K. Zhou, H. Wang, J. Hu, Y. Ji, and B. Cheng, "OSCA: An Online-Model based cache allocation scheme in cloud block storage systems," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 785–798.

[11] G. Kim, S. Lee, and H. S. Chwa, "Dynamic chip clustering and task allocation for real-time flash," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 1153–1158.

[12] H. Fan, Y. Ye, S. Ibrahim, Z. Huang, X. Li, W. Xue, S. Wu, C. Yu, X. Shi, and H. Jin, "Qospro: A qos-enhanced transaction processing framework for shared ssds," *ACM Trans. Archit. Code Optim.*, vol. 21, no. 1, Jan. 2024. [Online]. Available: https://doi.org/10.1145/3632955

[13] Y. Zhou, F. Wang, Z. Shi, and D. Feng, "Parallelism or fairness? how to be friendly for ssds in cloud environments," in *2024 IEEE International Conference on Cluster Computing (CLUSTER)*, 2024, pp. 179–189.

[14] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IX. New York, NY, USA: Association for Computing Machinery, 2000, p. 234–244.

[15] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "MQSim: A framework for enabling realistic studies of modern Multi-Queue SSD devices," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, Feb. 2018, pp. 49–66. [Online]. Available: https://www.usenix.org/conference/fast18/presentation/tavakkol

[16] Storage Networking Industry Association, "SNIA IOTTA Trace Repository," http://iotta.snia.org/traces/block-io, 2008.

[17] J. Kim, D. Lee, and S. H. Noh, "Towards SLO complying SSDs through OPS isolation," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 183–189. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/kim_jaeho

[18] K. Shen and S. Park, "FlashFQ: A fair queueing I/O scheduler for Flash-Based SSDs," in *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX Association, Jun. 2013, pp. 67–78.

[Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/shen

[19] R. Liu, Z. Tan, L. Long, Y. Wu, Y. Tan, and D. Liu, "Improving fairness for ssd devices through dram over-provisioning cache management," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 10, pp. 2444–2454, 2022.

[20] Y. Zhou, F. Wang, Z. Shi, and D. Feng, "Cofs: A collaboration-aware fairness scheme for nvme ssd in cloud storage system," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 12, pp. 4490–4504, 2024.

[21] B. S. Kim, "Utilitarian performance isolation in shared SSDs," in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*. Boston, MA: USENIX Association, Jul. 2018. [Online]. Available: https://www.usenix.org/conference/hotstorage18/presentation/kim-bryan

[22] J. Woo, M. Ahn, G. Lee, and J. Jeong, "D2FQ: Device-Direct fair queueing for NVMe SSDs," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 403–415. [Online]. Available: https://www.usenix.org/conference/fast21/presentation/woo

[23] R. Liu, D. Liu, X. Chen, Y. Tan, R. Zhang, and L. Liang, "Self-adapting channel allocation for multiple tenants sharing ssd devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 2, pp. 294–305, 2022.

[24] D. Min and Y. Kim, "Isolating namespace and performance in key-value ssds for multi-tenant environments," ser. HotStorage '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 8–13. [Online]. Available: https://doi.org/10.1145/3465332.3470883

[25] Y. Zhou, F. Wu, W. Huang, and C. Xie, "Livessd: A low-interference raid scheme for hardware virtualized ssds," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 7, pp. 1354–1366, 2021.