USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Unleashing The Potential of Datacenter SSDs by Taming Performance Variability

Gohar Irfan Chaudhry, *MIT CSAIL;* Ankit Bhardwaj, *Tufts University;*
Zhenyuan (Zain) Ruan and Adam Belay, *MIT CSAIL*

https://www.usenix.org/conference/nsdi26/presentation/chaudhry

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

Unleashing The Potential of Datacenter SSDs by Taming Performance Variability

Gohar Irfan Chaudhry Ankit Bhardwaj[†] Zhenyuan Ruan Adam Belay
MIT CSAIL [†]*Tufts University*

Abstract

Storage disaggregation is widely adopted in today's datacenters to improve disk utilization. However, efficiently disaggregating SSDs remains a challenge because of their high performance variability. This is due to differences in flash characteristics, such as wear and model version; read/write interference; and SSD-internal operations like garbage collection. Existing systems can only manage these types of variability in isolation. We propose Sandook, a rack-scale block storage system that instead holistically manages them together to unlock higher performance. Sandook achieves this through a logically centralized architecture that can integrate multiple scheduling policies and respond effectively at both short and long timescales. It adaptively steers I/O to the best available SSDs, using techniques that enable greater routing flexibility for both reads and writes. Sandook does not require any special storage or network hardware. Our evaluation demonstrates that Sandook is capable of delivering the full performance potential of SSDs. It achieves a 30%–82% raw I/O throughput improvement over existing systems that tackle a single source of performance variability while maintaining sub-millisecond tail latency. For unmodified applications sharing a pool of SSDs, Sandook achieves a 12–94% improvement in end-to-end performance.

1 Introduction

Storage disaggregation—where a pool of disks are exposed over the network—is a key technique used in datacenters to drive up utilization [4, 37, 52, 56, 72, 74]. Solid-state drives (SSDs), however, are more difficult to pool than hard disk drives (HDDs) because of their high performance variability. As a result, the worst performing SSD can often limit overall performance. For example, in the common scenario where I/Os are statically striped across SSDs, Figure 1 reveals that *half the potential performance of the underlying pool is left untapped*. This variability is difficult to manage because it takes many forms including heterogeneity from model version, wear, and capacity [24]; sensitivity to load characteristics like the ratio between reads and writes [69]; and transient events like garbage collection (GC) [73].

Several existing systems have mitigated these sources of variability, but they each specialize in handling a specific type. For example, Gimbal [52], RackBlox [61] and LinnOS [23] can detect and route around transient events in individual SSDs, but lack awareness of their heterogeneity and load characteristics. On the other hand, Rails [69] and SWAN [34] segregate reads and writes to improve throughput, but cannot react to transient events. Unfortunately, because they each manage only one form of variability, they all leave significant performance on the table. However, combining these disparate approaches is difficult because they often place contradictory constraints on how to direct I/O to SSDs (§4). For example, normally isolating reads and writes to separate SSDs improves their throughput, but it can also harm it if one SSD is slower than normal due to wear. Moreover, if an SSD is designated to handle only writes, load must still be shifted away from it if it becomes congested due to GC.

To address these challenges, we present Sandook[1], a new approach to SSD disaggregation that provides distributed block storage while dynamically steering I/O across disks. To the best of our knowledge, it is the first system that can *holistically manage multiple forms of variability*. Sandook's main insight is that different forms of SSD variability occur at different timescales, and a different approach is needed to target each timescale. For longer timescales, a centralized design is best because it enables higher quality routing decisions. For shorter timescales, local decisions made closer to the SSDs are more effective because they enable faster reactions.

To reconcile this tension between decision quality and speed, Sandook includes a novel two-tier architecture that provides *logically centralized* scheduling. A global scheduler optimizes load distribution at a slower pace with a holistic view. Meanwhile, faster schedulers run on each client and execute the global plan while reacting to urgent events, such as shifting load away from congested SSDs. This design enables both high-quality and microsecond-scale I/O steering.

Another challenge Sandook had to overcome is maximizing flexibility in I/O steering—after all, the ability to steer I/O would have little benefit if most accesses must be directed to specific disks. Sandook uses different strategies for reads and writes to unlock greater flexibility in each. For reads, Sandook builds upon block replication, which it already uses for fault tolerance, to provide flexibility in routing read requests among replicas on different SSDs. For writes, Sandook adopts a log-structured design, allowing writes to be directed to any SSD regardless of current block locations. This high degree of freedom guarantees that scheduling policies can be acted

[1] An Urdu/Persian word (صندوق) meaning "box" or "chest" (for storage).

upon without restriction. For example, a congested SSD does not limit the options for reading/writing blocks, and an SSD in read-only mode need not experience interference from writes since they can be sent to other SSDs.

The combination of logically centralized scheduling and flexible I/O steering allows Sandook to mitigate multiple forms of SSD performance variability simultaneously. We demonstrate this by combining three existing scheduling policies that could previously target only one form of variability (§7). First, Sandook performs *profile-driven load steering* to adaptively align load with heterogeneity across SSDs. Second, Sandook *segregates reads and writes*, designating SSDs into distinct groups that predominantly serve either read or write requests. Finally, Sandook employs *storage congestion control* to reactively shift load away from congested SSDs and minimize their impact on overall performance.

To enable real-time performance monitoring, Sandook runs a storage agent with the SSDs on each machine. The agent automatically profiles its SSDs' performance to build a model that it periodically updates to account for dynamism in SSD behavior due to capacity and wear-level changes.

Sandook is able to manage performance variability and deliver significant performance improvements *without any specialized hardware* such as Open-Channel [5], SDN [38], NVM [53] etc. Additionally, Sandook simplifies the implementation of custom storage scheduling policies by delegating I/O scheduling to user-space. This allows Sandook to easily express a wide range of policies tailored to meet diverse rack-scale requirements (§7.4). It also provides a standard Linux block device interface so that existing applications can take advantage of its benefits transparently (§8), or applications can use a more efficient kernel-bypass interface when they demand higher performance. Finally, Sandook allows multiple tenants to safely share the same pool of SSDs by maintaining strict data isolation between them.

We evaluate Sandook with four common *unmodified* datacenter applications (an OLTP database, an ML training system, an image compression workload, and a block store service) on a testbed of ten NVMe SSDs (§9). The results show that Sandook improves application throughput by 12–94%, latency by 71–88%, and GPU utilization by 23%.

Sandook codebase is open-source [66].

2 Background and Motivation

2.1 SSD-Based Disaggregated Storage Systems

SSD-based disaggregated storage is becoming a key component in modern datacenters [4, 17, 36, 37, 39, 51, 52, 56, 72, 74, 78]. It offers better performance, fault tolerance, and scalability relative to locally attached SSDs. For users, this translates into significant benefits for data-intensive workloads such as data analytics, machine learning inference, and training. From the provider's perspective, pooling and sharing SSDs across applications and tenants improves SSD utilization, thereby reducing operating costs in the datacenter.

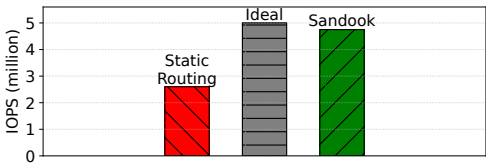


Figure 1: Disaggregated storage systems that perform static routing (*e.g.,* FDS [56]) only harness about 50% of the potential performance of the underlying SSDs, falling short of their intended goal to enhance storage efficiency. We introduce Sandook to bridge this performance gap.

To minimize request steering overhead and to scale with increasing numbers of SSDs, many existing systems employ *static steering*. This approach distributes storage blocks and requests to remote SSDs via sharding [50, 56, 72]. For its simplicity and effectiveness it is a de facto choice for many disaggregated storage systems such as flat datacenter storage (FDS), which utilizes hashing to map blocks across disks [56].

2.2 Problem: Substantial Untapped Performance

While these systems offer good scalability, their performance efficiency—how effectively they utilize the capabilities of the SSDs—is often overlooked. To investigate this, we conducted an experiment with ten datacenter grade SSDs (seven Samsung PM1725 and three Western Digital DC SN200).

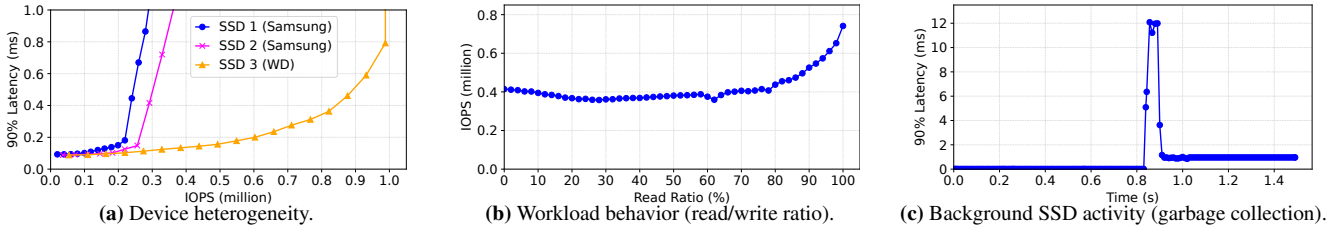
We implemented a prototype disaggregated system with static routing (replication factor of 2), similar to FDS [56]. We benchmarked it using a synthetic open-loop load generator with a 90% read ratio and Poisson arrival pattern, mirroring typical datacenter workloads [55, 77]. We measured its maximum 4K block IOPS achievable while maintaining a 1ms 90th-percentile latency target. We compare against the ideal performance potential of underlying SSDs, calculated by summing the maximum IOPS of each SSD under the same read ratio, without considering latency.

Figure 1 shows that static routing achieves only 2.5 million IOPS (MOPS), *a mere half of the all SSDs' combined potential*. This highlights that static request steering leaves a significant portion of SSD performance untapped. Such underutilization offsets the resource efficiency benefits of disaggregated storage, diminishing its practical value.

2.3 Root Cause: Performance Variability of SSDs

We investigated the root cause of the significant performance gap between static routing and ideal performance. Our analysis reveals that the fundamental issue lies in the *performance variability* of datacenter SSDs. This variability stems from several factors, including device heterogeneity (differences in models, wear levels, etc.), workload behavior (the fact that SSD performance changes based on read/write ratio), and background SSD activities (such as garbage collection).

Existing systems, with their static request steering, overlook



(a) Device heterogeneity. **(b)** Workload behavior (read/write ratio). **(c)** Background SSD activity (garbage collection).

Figure 2: The performance variability of SSDs stems from three key factors: a) Device heterogeneity: performance varies due to differences in models, wear levels, fragmentation levels, etc.; b) Workload behavior: performance heavily depends on the workload's read/write ratio; c) Background SSD activity: internal operations such as garbage collection cause temporary but severe performance disruptions.

this variability and treat all SSDs as uniformly performant. This results in hotspots and straggler SSDs that limit performance, preventing the system from realizing the full potential of SSDs and resulting in data stalls in expensive compute resources [9, 76]. Next, we will examine these sources of performance variability in datacenter SSDs in more detail.

2.3.1 Device Heterogeneity.

Datacenter SSDs exhibit significant performance variability due to device heterogeneity arising from two primary sources. First, SSDs from different vendors, or even different models from the same vendor, vary in their flash controllers, flash chips, and firmware, leading to performance discrepancies. For example, cloud providers such as Meta and Alibaba have reported many different SSDs active in their clusters [20, 24, 30, 45] – this is a common trend because of staged hardware rollout and correlated failure mitigation in datacenters. Second, even identical SSD models experience performance divergence over time due to differences in wear levels [10] and internal fragmentation [12]. These factors are heavily influenced by the specific workloads and usage patterns experienced by individual SSDs.

To illustrate this phenomenon, we measured the 4K block access performance of three SSDs in our testbed (two Samsung PM1725 and one Western Digital DC SN200) with a 90% read ratio. Figure 2a presents this significant variation. The Western Digital SSD outperforms the Samsung SSDs by 2.7–3.5×, achieving 0.99 MOPS under a one-millisecond 90th-percentile latency target, due to its faster flash chip. Importantly, even the Samsung SSDs exhibit performance differences. They achieve 0.28 and 0.36 MOPS respectively, representing a variability of 28%. This variation stems from their difference in wear levels and internal fragmentation.

This variability poses a significant challenge, requiring systems to be adaptive to the performance characteristics of each individual SSD to maximize overall performance.

2.3.2 Workload Behavior.

Flash SSDs exhibit significant performance variation depending on the workload's read/write ratio [43, 72, 75]. This asymmetry arises from NAND flash requiring block-level erases before writes, adding overhead not present in reads [33].

Figure 2b demonstrates this phenomenon. We measured SSD 1's maximum achievable IOPS under different read/write ratios and observed a drastic variation as the read/write ratio changes. With a read-only workload, the SSD delivers approximately 2× the IOPS compared to a write-only scenario. Notably, even a small percentage of writes, only 2%, degrades performance by 15% compared to a purely read workload. This sensitivity intensifies with increasing write percentages; a 10% write mix leading to a 30% IOPS decrease and a 20% write mix almost reducing the IOPS to 50%.

This sensitivity to the read/write mix exacerbates performance unpredictability, further complicating load steering decisions in disaggregated storage systems.

2.3.3 Background SSD Activities.

Activities such as garbage collection (GC) introduce performance fluctuations in SSDs [22, 29, 34, 35, 73]. NAND flash requires writing new data to fresh pages. GC reclaims space by identifying blocks containing outdated data, copying active data to new locations, and erasing the original blocks. These additional read/write/erase operations compete with regular user requests, temporarily degrading performance.

We demonstrate this with a write-only workload that triggers GC on SSD 1. As shown in Figure 2c, during active GC cycles (0.85–0.90 seconds), write latencies increase by more than 20×. This significantly degrades the SSD's ability to meet performance targets. The challenge lies in the fine-grained timescale at which these disruptions manifest. Even short GC bursts severely disrupt performance, making them extremely difficult to predict, detect, and react to.

3 Related Work

SSD disaggregation offers a potential solution to improving the low storage resource utilization in datacenters [36, 58, 60]. Notable examples include LeapIO [41], IODA [42], and systems utilizing NVMe-oF [18], which require hardware support, as well as FDS [56] and DLFS [78], which necessitate application code changes. Unfortunately, many of these systems overlook the performance variability inherent in SSDs [56], resulting in suboptimal performance that undermines the efficiency of disaggregation [9]. Below we elaborate on why existing systems fail to deliver close to ideal performance. Table 1 summarizes this comparison.

SSD Performance Variability. Several systems specialize

System	Congestion Control	R/W Segregation	Weighted Routing	Hardware Changes
FDS [56] (static), Decibel [54]	×	×	×	None
RackBlox [61], IODA [42], TTFlash [73]	✓	×	×	SDN, SDF, SSD firmware
LinnOS [23], MittOS [21]	✓	×	×	None
Gimbal [52]	✓	×	×	SmartNIC
Rails [69], SWAN [34]	×	✓	×	Extra memory (e.g., NVM)
Gecko [68]	×	✓	×	None
ReFlex [37], zQoS [48]	×	×	✓	None
Sandook	✓	✓	✓	None

Table 1: Sandook is the only storage system that transparently handles multiple sources of performance variability without changing hardware.

in managing a *single* form of SSD performance variability (§9.5). For example, Rails [69], SWAN [34] and Gecko [68] segregate reads and writes onto different SSDs, but they are not aware of when SSDs are congested. Others like RackBlox [61] rely on SDN [38] and SDF [58] to coordinate garbage collection, but cannot do this with commodity storage and network hardware. On the other hand, MittOS [21], LinnOS [23], and Gimbal [52] can detect congestion in SSDs, but they do not segregate reads and writes. All these systems also lack accurate performance profiles of the SSDs, so they waste resources by not avoiding congestion proactively. This lack of a holistic approach to performance variability ends up leaving substantial untapped performance.

In response, Sandook provides a novel two-tier architecture that enables it to achieve a) routing flexibility (through its log-structured design); b) global, data-driven decision-making (via its logically centralized controller); and c) rapid responsiveness to short-term variability (enabled by decentralized Sandook clients). This combination of capabilities enables the integration of multiple policies that can manage multiple forms of performance variability cohesively.

SSD Performance Modeling. One approach to mitigating the challenge of performance variability is through SSD performance modeling [2,23,28]. Approaches range from analytical models in systems like ReFlex [37] to machine learning techniques employed by systems such as LinnOS [23] and GrayBox [28]. Sandook employs offline profiling with continuous online monitoring to capture SSD performance and can be extended to incorporate other SSD metrics [44].

4 Challenges

Naively composing these techniques—profile-driven weighted routing [37], read/write (R/W) segregation [34,69], and storage congestion control [23, 52]—does not yield a robust rack-scale SSD disaggregation solution. In practice these mechanisms operate on different timescales, impose contradictory constraints, and often assume specialized hardware. Below we make these composition pitfalls explicit.

4.1 Mismatched Timescales

Profile-driven load steering must evolve slowly to reflect device heterogeneity and wear, whereas congestion control must react at sub-millisecond timescales to transient slowdowns (e.g., GC spikes). If both policies independently adjust routing without a contract, they fight each other: a global reweighting can immediately be undone by local backoff (and vice versa), causing oscillations and tail latency inflation. The challenge is to separate concerns by timescale while keeping the fast path safe and the slow path effective.

4.2 Conflicting Objectives

R/W segregation improves throughput by reducing interference, but it shrinks the candidate set for each operation (reads prefer R-mode SSDs; writes target W-mode SSDs). Weighted routing wants to *concentrate* load on faster devices, while congestion control wants to *diffuse* load away from SSDs that become temporarily slow. With static mappings (e.g., sharding/FDS-style placement), these constraints collide: (i) a write-only SSD in GC must still accept writes, (ii) a fast read SSD may be inaccessible if replicas are unlucky, and (iii) replication factor k couples read flexibility and write amplification. A correct composition needs more degrees of freedom than static placement provides, yet must preserve consistency, wear fairness, and capacity limits.

4.3 Metadata Management

Static mappings simplify systems but preclude adaptive routing. Log-structured placement restores routing flexibility (e.g., append-anywhere writes) but introduces new systems problems: scalable logical–physical mapping, timely trimming/-cleaning without synchronized stalls, crash-consistent metadata, and reconstruction after failures—all while preserving block device semantics and multi-tenant isolation.

4.4 Specialized Hardware

Prior systems often rely on specialized hardware or firmware support (e.g., SmartNIC JBOFs, NVM, SDN/SDF [38,58], or open-channel SSDs [5]) to obtain fine-grained signals or to coordinate GC [34,52,61]. Assuming unmodified commodity hardware means: (i) limited on-device telemetry, (ii) no control over the FTL, and (iii) microsecond-scale reactions must be achieved in software. The challenge is to detect disruptions using end-to-end signals and to actuate routing changes without bespoke dataplanes or firmware hooks.

4.5 Our Approach

Sandook overcomes these composition challenges through two key design decisions: **(1) Routing flexibility:** A log-structured placement layer for writes and replica choice for

reads decouples scheduling decisions from data placement, allowing R/W segregation, weighted routing, and congestion control to coexist without conflicts. **(2) Timescale separation:** A two-tier architecture cleanly divides responsibilities—the controller handles slow-changing decisions (modes, weights) while clients handle fast reactions (congestion backoff)—preventing oscillations between global and local adjustments. These principles enable Sandook to compose previously incompatible policies into a coherent system (§5).

5 Sandook Overview

Sandook addresses these composition challenges with two design principles. First, it exposes *routing flexibility* through a log-structured placement layer for writes and exploits replication for reads, so schedules are not blocked by immutable mappings. Second, it *separates control by timescale* via a *logically centralized* controller that computes a slow-moving per-SSD plan—a *mode* (read-mode or write-mode) and a *routing weight*—from global demand and learned performance profiles. *Client-side schedulers* execute that plan and may *temporarily down-weight* congested SSDs based on end-to-end signals to protect tail latency. This architecture composes R/W segregation, profile-driven weighted routing, and congestion control on *commodity* SSDs and network hardware.

Routing Flexibility. To guarantee freedom at dispatch time, Sandook uses a *log-structured placement layer* for writes (append-anywhere) and leverages *replication* to provide per-read *replica choice* (already a common practice in disaggregated storage systems for providing fault-tolerance). The system can steer writes to any suitable SSD and route reads to any of a block's replicas, independent of prior placements. This decoupling is what allows higher-level policies (R/W segregation, weighted routing, and congestion control) to act without tripping over one another.

Timescale Separation. Sandook employs *logically centralized scheduling* that produces a slow-moving plan and *client-side execution* that applies rapid corrections. The controller computes, per SSD, a *mode* (read-mode or write-mode) and a *routing weight* from global demand and SSD performance profiles. Clients then enforce this plan but may *temporarily down-weight* specific SSDs upon receiving end-to-end *congestion signals* from storage agents, protecting tail latency during short GC bursts or other transients. This division of labor avoids oscillations between global reweighting and local backoff while keeping the controller off the data path.

Architecture. Figure 3 illustrates the components that realize these principles. *SSD Agents* (§6.1) colocated with disks provide hardware-agnostic *observability* (periodic profiling and real-time signals) and serve I/O. *Sandook Controller* (§6.2) plans by aggregating signals and emitting per-SSD *modes* and *routing weights*. *Sandook Clients* (§6.3) execute the plan on the fast path, applying local, signal-driven *down-weighting* when needed, and issuing reads/writes against a log-structured, replicated block space.

Composable Policies. Within this substrate, Sandook integrates three policies that previously targeted disjoint axes of variability, now made *composable* by routing flexibility and timescale separation (§7): **(i) read/write segregation** (§7.1) assigns SSDs to *read-mode* or *write-mode* and steers requests accordingly, **(ii) profile-driven weighted routing** (§7.2) uses SSD performance profiles to assign per-SSD *routing weights* that align load with device heterogeneity, and **(iii) storage congestion control** (§7.3) performs fast, client-side *down-weighting* in response to agent *congestion signals*, shifting load away from transient stragglers without involving the controller. Table 2 summarizes how signals, decisions, and actions flow across agents, the controller, and clients.

Outcome. By pairing a flexible placement substrate with a two-tier control plane, Sandook composes existing ideas into a stable, hardware-agnostic solution for rack-scale SSD disaggregation: slow, globally consistent *modes+weights* unlock throughput, while fast, local *down-weighting* preserves tail latency—without modifying SSDs or the network stack.

6 Sandook Design

Next, we discuss the key design components in Sandook—SSD agent, Sandook controller, and Sandook client—and their interactions that enable adaptive I/O steering.

6.1 SSD Agent

Sandook employs an SSD agent on each storage server to manage attached SSDs. The SSD agent serves two core functions. First, it continuously monitors the load and performance metrics of attached SSDs, sharing this information with the Sandook controller to facilitate its decision making. Second, it handles block read and write requests issued by Sandook clients, forwarding them to the corresponding SSD. The memory and CPU overhead of monitoring the performance of (typically tens of) SSDs on a single storage server is negligible.

Profiling SSDs: The SSD agent registers new SSDs on the server with Sandook. It begins by generating a detailed performance profile, denoted as $\mathcal{P}(\text{load}, \text{rw\_ratio})$, by measuring the SSD's load-latency curves across various read/write ratios. To create these curves, the agent increases the load (in 1% increments) and measures the latency at each point. This profile captures performance information, including 50^{th}, 90^{th}, and 99^{th} percentile latencies, providing a granular view of the SSD's expected behavior under different workloads.

Since SSD performance can change over time due to factors like wear and fragmentation [8], the agent periodically reprofiles SSDs. This occurs during periods of low system load (e.g., at night) to minimize disruption. Any updates to the performance profile are shared with the Sandook controller to maintain its accurate system-wide view.

The agent shares the performance profile and capacity information with the Sandook controller. At this point, the new

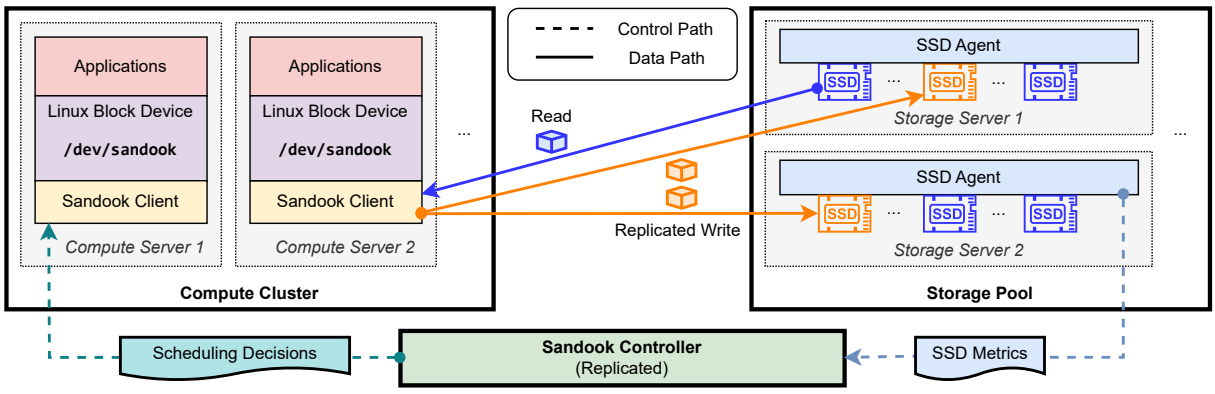


Figure 3: An overview of Sandook's architecture, highlighting the interaction between SSD agents (which monitor load and performance signals), the Sandook controller (which makes centralized decisions based on signals), and Sandook clients (which execute these decisions).

Policy	SSD Agents (Signal)	Sandook Controller (Decision)	Sandook Client (Action)
Read/Write Segregation (§7.1)	Gather read/write load information	Designate SSDs as read- (R) or write-mode (W)	Steer I/Os to dedicated R/W SSDs
Profile-Driven Load Steering (§7.2)	Profile and monitor SSD performance	Assign routing weights to SSDs	Route I/Os based on weights
Storage Congestion Control (§7.3)	Detect SSD performance disruptions	Bypassed for rapid response	Adaptively reduce traffic to congested SSDs

Table 2: An overview of Sandook's three key policies, which work in tandem to manage the performance variability of SSDs holistically.

SSD is made available for use to clients.

Detecting Congestion: The agent actively monitors SSDs for real-time load and short-term performance fluctuations. It tracks the load (number of read and write requests) and the latency of each request, aggregating these metrics over a 10ms window. Using Exponentially Weighted Moving Average (EWMA), it calculates average and tail latencies, capturing a reliable view of the SSD's current performance.

Detecting potential performance disruption (often caused by background activities such as garbage collection) is remarkably simple for the agent. It directly compares the real-time metrics against the established performance profile – deviations (exceeding a configurable threshold, 10% by default) signal likely congestion. In this case, the agent immediately informs Sandook clients, enabling them to take action.

Handling I/O: Another key responsibility of the SSD agent is to directly handle block read and write requests from Sandook clients. For read requests, each includes both the SSD's unique ID and the desired block offset within that SSD. If a valid block is being requested, the agent retrieves the data from the appropriate local SSD. It then returns the response to the client, optionally including a congestion signal if necessary.

Write requests, however, are handled differently due to Sandook's log-structured design. Each write request only includes the SSD ID. The agent dynamically allocates a new, unused block on the specified SSD to accommodate the write. Upon successful write completion, the agent returns the newly assigned block ID to the client so it can be used to refer to this block at the time of reading it.

We provide pseudo-code for I/O operations in §A.1.

6.2 Sandook Controller

We intentionally designed the Sandook controller to be lightweight and focused solely on the control plane. Its core responsibilities include gathering load and performance information from SSD agents, making centralized scheduling decisions, and sharing these decisions with Sandook clients. Additionally, the controller maintains a list of available SSD agents and SSDs, enabling simple discovery by Sandook clients. By keeping the controller thin and off the critical data path, we avoid potential performance bottlenecks and allow for easy replication, ensuring high availability and scalability.

Scheduling: The controller periodically queries all SSD agents to gather the latest load and performance information for each SSD in the system. We utilize a pull-based approach to ensure that the collected information reflects a synchronized snapshot in time. The controller aggregates the load data to calculate global read and write demand by summing the respective MOPS across all SSDs. Next, the controller leverages the aggregated load information and individual SSD performance profiles as input to the scheduling policies (§7). The output of the scheduling process is a list of per-SSD routing weights and their designated modes (read or write). This scheduling decision is then broadcast to all Sandook clients.

The controller iteratively performs this process every 200 milliseconds, carefully balancing the need for timely scheduling decisions with the importance of control system stability. Too frequently updating scheduling decisions can lead to system oscillation if there is not sufficient time for the system to

react and for the controller to observe its impact.

Trimming SSDs: The controller also coordinates the trimming of individual SSDs [32] ensuring minimal performance impact on the overall system. Due to Sandook's log-structured block layout, each write allocates a new block and invalidates the previous one. Trimming the invalidated blocks is crucial to maintain high write performance and prevent capacity exhaustion. To avoid disruption caused by simultaneous trims across SSDs, the controller employs a token-ring approach, rotating the token among SSDs to coordinate trim operations.

We provide pseudo-code for the trim operation in §A.2.

We intentionally keep the controller design simple and stateless. Scheduling policies are purely functional, and SSD load and performance information are just temporal states. The only persistent state maintained by the controller is the registry of SSDs and SSD agents within the system. While this registry could also be rebuilt by having SSD agents re-establish contact with the controller, we opt for a classic primary-backup replication approach to ensure high availability. This allows for rapid failover with minimal downtime in the event of primary controller failure.

6.3 Sandook Client

The Sandook client resides on user machines, bridging applications and the disaggregated SSDs within the Sandook system. It presents a familiar block device interface, transparently routing storage requests to the most suitable remote SSDs based on the Sandook controller's scheduling decisions.

Dispatching Requests: The Sandook client intercepts storage block requests made by applications (§8) and splits them into granular 4K blocks. In Sandook, we expose remote SSDs at this 4K block level, a choice that simplifies design by eliminating the internal fragmentation issues common in log-structured systems [62]. This approach forgoes the need for compaction and remains efficient due to the fast 4K random access speeds offered by modern SSDs, which are close to the sequential performance [65]. The client routes these 4K block requests independently and out-of-order to appropriate remote SSDs, exploiting the parallelism across remote SSDs.

Block Mapping: The Sandook client maintains a logical-to-physical mapping to translate logical block addresses visible to the user into their corresponding physical locations on remote SSDs. This mapping is implemented as a four-level nested table, mirroring the classical design of four-level page table [14]. This enables lazy metadata allocation for efficient memory usage, while allowing concurrent accesses with fine-grained locking at the leaf level. A key distinction arises from block replication: in Sandook, a single logical block maps to multiple physical blocks across different SSDs.

Crucially, this mapping mechanism enables strict data isolation. Each user has a distinct mapping, ensuring access only to their own allocated blocks. **Sandook does not support sharing a single logical volume across multiple clients**; each client maintains its own independent logical address space. This design avoids the complexity and performance overhead of cross-client synchronization for concurrent writes to shared blocks, while also eliminating potential security concerns and side-channels in multi-tenant settings. For enhanced security, we isolate each Sandook client from applications as a separate Linux process, inspired by microkernel approaches.

To process a 4K block request, the client first determines the potential remote SSD candidates. For read requests, the client consults the mapping table to identify the locations of all replicas. For write requests, any SSD can serve as a candidate due to the log-structured design.

Load Balancing: Next, the client applies the controller's scheduling decisions to these candidates, selecting the specific SSDs to be used (§7). Read requests target one location, while write requests require k locations, where k is the system's replication factor. For writes, the client updates the mapping with the new physical block locations returned by SSD agents. Old physical block locations are marked for recycling, and clients periodically push this information to SSD agents so that these blocks can be released.

Sandook can be configured to support different block sizes based on the workload requirements (*e.g.,* 512B etc.) For requests smaller than the block size, Sandook adds an amplification overhead similar to existing file systems.

6.4 Fault Tolerance and Crash Recovery

Sandook is carefully designed for fault tolerance and crash consistency, ensuring its reliability in disaggregating SSDs at rack-scale. We examine how each component handles failures and reconstructs lost state.

Controller Failure: The controller maintains two types of state: (1) *persistent state*, consisting solely of the SSD and agent registry, which is replicated across primary and backup controllers; and (2) *soft state*, comprising the current scheduling decisions (per-SSD modes and routing weights) and cached SSD performance profiles. Soft state is derived from SSD agent reports and can be fully reconstructed after a restart by re-querying agents. Upon primary controller failure, the backup seamlessly takes over; scheduling decisions are recomputed within one scheduling epoch (200 ms). A distributed coordination framework (e.g., Zookeeper [27]) prevents split-brain issues.

SSD Client Failure: The client maintains three types of in-memory state: (1) the controller's scheduling decisions, (2) the logical-to-physical block mapping, and (3) discarded blocks awaiting release. After a cold restart, recovery proceeds as follows: scheduling decisions are re-requested from the controller; the logical-to-physical mapping is recovered from local write-ahead logs (persisted on `fsync` or timeout); and lost discard information is reconstructed via the SSD agent reconstruction process described below. Sandook can optionally persist client-side mappings to shared storage for recovery from a different machine.

SSD Agent Failure: Existing systems rely on replication [1, 56, 71] and erasure coding [25, 67] for fault tolerance. Replication is widely used in performance-oriented storage systems because independent replicas provide *flexible read* routing, *simplify writes* by avoiding cross-device encoding, and preserve *low-latency* access by eliminating decoding on the read path. In contrast, erasure coding improves space efficiency but requires stripe-level coordination and incurs parity computation and decoding overhead. Sandook adopts replication rather than erasure coding. Our target environment—rack-scale SSD deployments—prioritizes performance, tail-latency control, and routing flexibility over raw capacity efficiency.

This choice also simplifies failure handling. Block replication ensures continued availability during agent failure; clients steer requests to other SSDs. Upon agent restart, the only state requiring reconstruction is the free block bitmap for attached SSDs. The agent broadcasts a reconstruction request to all Sandook clients, which respond with lists of physical blocks currently in use (obtained by scanning their mapping tables). The agent computes the complement to rebuild free block information. This process completes in seconds for typical deployments and does not block ongoing I/O to other SSDs.

Overall, Sandook provides persistence guarantees comparable to locally attached cloud storage, with higher performance.

7 Sandook Policies

Sandook's flexible architecture enables the simultaneous integration of various scheduling policies. In this section, we will discuss three specific policies that work in tandem to manage the performance variability of disaggregated SSDs.

7.1 Read/Write Segregation

Sandook employs a read/write segregation policy to mitigate read/write interference within individual SSDs, preventing the performance degradation it can cause (§2.3.2). This policy uses a dynamic sliding window mechanism to designate a subset of SSDs within the pool as write-mode, while the remaining SSDs become read-mode.

The Sandook controller dynamically calculates the size of the write-mode window based on the system's current workload. It begins by aggregating the read and write loads of each SSD, reported by SSD agents, to determine the system-wide read demand R and write demand W. Next, using performance profiles $\mathcal{P}$ of each SSD, the controller calculates the minimal number of SSDs required to meet the global write demand W. These SSDs form the write-mode window.

To ensure fairness and balance wear, the sliding window of write-mode SSDs periodically shifts across a logical ordering of the SSDs. This shift happens on every scheduling epoch of the controller, providing all SSDs with the opportunity to switch between read and write modes.

Sandook clients aim to strictly enforce the segregation policy by directing read requests exclusively to read-mode SSDs and write requests exclusively to write-mode SSDs. However, in rare cases where all replicas of a read request reside on write-mode SSDs, Sandook makes an exception. Rather than stalling the read request, which would disrupt application performance, clients are allowed to read the block from write-mode SSDs. This is acceptable since such situations are infrequent, as evidenced in our evaluation (§9.5.2).

7.2 Profile-Driven Load Steering

Sandook employs performance-profile-driven load steering to manage SSD performance variability caused by device heterogeneity (§2.3.1). This policy optimizes load distribution by incorporating performance capabilities of each SSD.

The Sandook controller determines the optimal load distribution for both read and write requests. It focuses on two primary objectives: meeting the system's global read demand (R) and write demand (W), and minimizing the overall latency.

The controller leverages linear programming (LP) to achieve this optimization. Focusing on read requests (with the write optimization following a dual structure), the controller first gathers performance profiles $\mathcal{P}_i$ of each SSD_i from SSD agents. It then derives a latency metric (by default 90^{th}-percentile latency), represented as $L_i(r_i)$, for each SSD_i based on its current read load r_i. The LP formulation aims to find the optimal read load distribution:

$$\textbf{Minimize: } \max_{i=1,\ldots,N} \{L_i(r_i)\}, \text{ where } N \text{ is the \# of read SSDs}$$

$$\textbf{Constraints: } \sum_{i=1}^{N} r_i = R \quad \text{(ensuring read demand is met)}$$

$$r_i \geq 0, \forall \text{ read SSD}_i \quad \text{(non-negative load)}$$

To obtain a good solution within a reasonable timeframe, Sandook employs a hill-climbing algorithm [64]. Initially, reads are evenly distributed across read-mode SSDs. The algorithm then iteratively identifies the SSD with the highest L and the one with the lowest L. It then transfers a small portion of load (1% by default) from the worst-performing SSD to the best-performing SSD. The process continues until the overall latency objective no longer improves.

7.3 Storage Congestion Control

Sandook employs a storage congestion control policy to rapidly mitigate the performance impact of transient disruptions on individual SSDs, usually caused by the SSD's internal background activities such as garbage collection (§2.3.3).

SSD agents are capable of identifying transient disruptions by comparing real-time performance metrics of each SSD against their performance profile $\mathcal{P}$ (§6.1). To enable fast reaction, this policy bypasses the central controller; the SSD agent immediately notifies impacted Sandook clients by piggybacking a congestion signal with the I/O response.

Reacting to this signal, Sandook clients employ a congestion control algorithm inspired by rate-based TCP congestion control algorithms [6, 7, 16]. For each storage round trip time

(RTT) where a congestion signal is received, the client reduces its routing weight to that specific SSD by half, easing the load on the congested device.

Once congestion signals cease, the client initiates a gradual recovery process. For each RTT, the routing weight is incrementally increased by a small percentage (1% by default) of the original weight. This probing approach allows the client to safely discover the available throughput of the SSD without overloading it. The process continues until the routing weight fully returns to its original value, ensuring optimal performance once the SSD has stabilized.

7.4 Extensible Policy Engine

Sandook performs I/O scheduling in user-space, similar to prior work for CPU scheduling [15, 26]. It is designed to be extensible in implementing, deploying and testing custom scheduling policies for achieving different efficiency objectives. Sandook focuses on improving utilization while minimizing tail latency, though other policies can be easily expressed. For instance, Sandook can factor in capacity management by accounting for the used capacity on each SSD or perform wear-management by considering erase cycles of each SSD when assigning routing weights. Sandook's policy engine can also prioritize storage requests based on user-defined application requirements, maintaining separate sets of routing weights for different users to ensure Quality-of-Service (QoS).

8 Implementation

Sandook is implemented in approximately 9,000 lines of C++-23 code. Its modular implementation comprises three user-space binaries (the SSD agent, Sandook controller, and Sandook client), ensuring ease of deployment. We built them upon Caladan's threading and networking runtime [15] for high TCP performance, and SPDK [70] for fast, low-overhead storage access. Our careful optimization of Sandook's core runtime allows it to easily saturate our 100 GbE NIC and achieve 2.6 million 4K IOPS for remote storage.

Sandook provides a standard Linux block device interface, leveraging the recent Linux `ublk` [31] feature for seamless integration with existing applications. It intercepts all Linux I/O system calls (such as `open`, `read`, and `write`) made to the exposed block device and forwards them to the user-space Sandook client using `io_uring` [46]. To improve scalability, we employ multiple `io_uring` queues.

For applications seeking optimal performance with minimal overhead, Sandook provides a kernel-bypass interface. This offers direct access to Sandook functionality, eliminating the need for Linux system calls. We provide C, C++, and Rust bindings that mirror a familiar block API, including functions such as `Read(block_id)`, with both synchronous and asynchronous options for flexible integration.

To ensure efficient interaction between Sandook's components, we developed a custom, high-performance RPC stack. It maintains per-core TCP flows for scalability, employs adaptive batching to amortize overhead [3], and performs locality-aware flow steering to optimize cache footprint. As a result, it can deliver more than 1 MOPS RPC throughput per CPU core—ample performance for rack-scale SSD disaggregation.

We carefully optimized Sandook's business logic to overcome potential scalability bottlenecks. Key optimizations include sharding data structures into per-core components (e.g., the SSD agent's performance metric counters, which are sharded and later aggregated); employing fine-grained locking for fast concurrent updates (e.g., the block mapping in Sandook clients, which requires update on every write request); and utilizing Read-Copy Update (RCU) for lockless access to read-mostly data structures (e.g., the Sandook controller's scheduling decisions stored in Sandook clients).

Routing Overhead: Compared to static routing, Sandook introduces minimal additional overhead. On the *client side*, static routing computes a hash to select an SSD, while Sandook performs a weighted random selection among cached replica locations—both are O(1) operations requiring no synchronization or network round-trips. The additional client memory overhead is limited to per-SSD routing weights and mode designations (a few bytes per SSD), as the logical-to-physical mapping table is comparable in size to metadata maintained by any block-level storage system. On the *controller side*, the scheduling computation (hill-climbing over SSD profiles) runs only once per scheduling epoch and scales linearly with the number of SSDs—negligible for rack-scale deployments. Congestion signals are piggybacked on existing I/O responses, adding no extra network messages compared to baseline storage RPCs.

9 Evaluation

9.1 Setup.

We used a pool of ten NVMe SSDs, seven Samsung (PM1725) and three Western Digital (DC SN200). They exhibited varying performance, with read and write throughput ranging from 0.38–1.05 MOPS and 0.32–0.49 MOPS, respectively, due to model differences and usage wear. Figure 2a illustrates the performance profiles of three representative SSDs.

The testbed consists of ten machines: four client machines running storage applications, five storage machines (two SSDs each), and one machine hosting Sandook's centralized controller. Each machine ran Ubuntu 23.04 with Linux kernel v6.5, equipped with an Intel Xeon E5-2680 v4 CPU, 64 GB DDR4 memory, interconnected via a 100 GbE network.

We trimmed all SSDs and flushed the Linux page cache on all machines before launching experiments. Unless otherwise specified, we configured Sandook with a replication factor of 2. Section 9.6 presents the results of other replication factors.

9.2 Comparison with Other Systems

To understand the importance of Sandook's holistic approach to manage SSD performance variability, we compared it to ex-

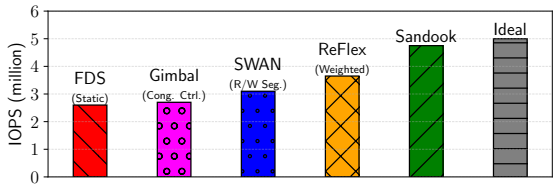


Figure 4: Existing systems address only one type of SSD performance variability leaving significant performance potential untapped: the baseline FDS achieves 52% of ideal IOPS, Gimbal's congestion control policy reaches 54%, SWAN's R/W segregation policy achieves 62%, and ReFlex's policy of static weighted routing reaches 73%. In contrast, Sandook enables 95% of the ideal IOPS.

isting systems that address only one source of variability. Our comparisons include: a baseline FDS [56] with hash-based I/O steering that overlooks SSD performance variability; Gimbal [52] that reacts to short-term variability by employing storage congestion control on the baseline; SWAN [34] that focuses on mitigating read/write interference by applying segregation on the baseline; and ReFlex [37] that responds to device heterogeneity by performing static weighted routing, with weights based on peak SSD IOPS, on the baseline. We faithfully emulate the core policies of these systems atop Sandook, *without using any specialized hardware*.

Figure 4 shows the results. In a 90% read workload, FDS achieved 2.6 MOPS, Gimbal [52] 2.7 MOPS, SWAN [34] 3.1 MOPS, and ReFlex [37] 3.6 MOPS, underutilizing SSDs as compared to the ideal potential of SSDs (5 MOPS).

In contrast, Sandook's holistic approach enables it to deliver 4.75 MOPS, closely approaching ideal performance. This translates to 76%, 53.2%, and 31.9% higher IOPS than storage congestion control, read/write segregation, and static weighted routing, respectively.

9.3 Performance of Colocated Storage Applications

We evaluated Sandook with four common unmodified datacenter storage applications.

LeanStore [40] is a high-performance storage engine for online transaction processing, optimized for multi-core CPUs and NVMe SSDs [19]; we run YCSB-B [13] (Zipfian distribution with 95% reads and 5% updates) for object lookups.

Machine learning training involves training a Unet3D CNN model [11] using PyTorch [59] on a 180GB dataset. We use MLPerf [49] to simulate an NVIDIA A100 GPU [57].

LZ4 image compression compresses ImageNet ILSVRC2015 dataset images [63] using LZ4 [47].

Storage server is a high-performance open-source block storage server [15], a latency-critical application. We use an open-loop load generator with Poisson arrivals and Zipfian block address distribution, in line with prior work [15].

9.3.1 Methodology.

We compared Sandook with two representative baseline configurations. The first is static routing (as used in FDS [56]),

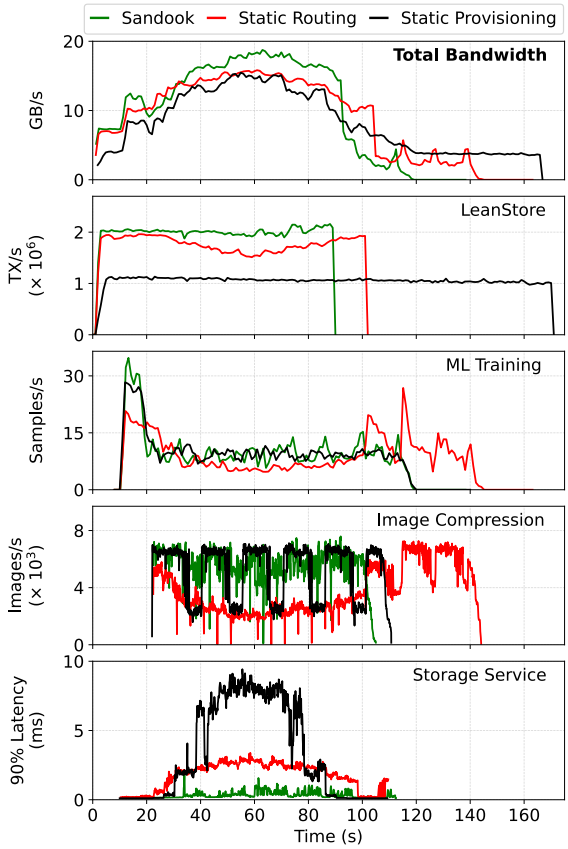


Figure 5: Sandook outperforms existing approaches in a realistic scenario of colocating four storage applications. Compared to static routing and static provisioning, Sandook unlocks a 27% increase in peak bandwidth (top figure), boosts application performance by 12–94% (figures 2–4), and ensures sub-millisecond tail latency even under peak load (bottom figure).

that statically steers storage requests via address hashing.

Our second baseline is static provisioning, where the same total number of SSDs are partitioned and assigned to individual applications. Applications exclusively used their allocated SSDs via static routing. This ensured strict SSD performance isolation among applications, representing today's practice for achieving the best storage performance [60]. We allocated the number of SSDs to applications in proportion to their peak storage throughput demand, resulting in four SSDs to storage server (the most storage-intensive application) and two SSDs each to the other three applications.

We examined Sandook's performance in a realistic rack-scale setup in order to compare to existing approaches. We colocated four typical storage applications on our pool of ten NVMe SSDs. Three throughput-focused applications—LeanStore, ML training, and image compression—are initiated sequentially at $t = 0, 10, 20$ s, respectively, running to completion. For these, higher throughput and a lower completion time is better. The latency-critical storage server started

at $t = 10$ s, with its load gradually increasing from 0.1 MOPS to 2.0 MOPS until $t = 62$ s before reducing at the same rate. For this, lower latency is better.

We measured the total storage bandwidth usage and the end-to-end performance of the applications with Sandook as the storage backend, comparing it to static routing and static provisioning. Ideal results would indicate both a higher total storage throughput and improved application performance.

Figure 5 depicts the results. As shown in the top figure, the total bandwidth increased until $t = 62$ s as more applications were started and the storage server's load ramped up. Sandook reached a peak bandwidth of 19 GB/s, 27% higher than the 15 GB/s achieved by both static routing and static provisioning. This resulted from Sandook's flexible request steering, unlocking SSDs' performance potential untapped by static routing and static provisioning. Bandwidth decreased post $t = 63$ s as we scaled down the storage server's load.

In LeanStore, static provisioning showed the lowest performance (1.1 million transactions/s) being restricted to two statically assigned SSDs. Both Sandook and static routing initially achieved 2 million transactions/s by taking advantage of more SSDs. As we increased the storage server's load, static routing started to falter due to straggler SSDs (that are slower than others) and heightened read/write interference (due to colocated applications). Sandook maintains stable, high performance throughout, showing a 12% and 94% end-to-end performance improvement over static routing and static provisioning, respectively. It is able to complete the required number of transactions sooner than both baselines.

In ML training, static routing lagged with an average of 8 samples/s and 80% GPU utilization. This slowdown and GPU underutilization primarily resulted from *data stalls* during the loading stage—periods where the GPU sits idle waiting for training data from storage. These stalls are a direct consequence of read/write interference and straggler SSDs under static routing. Static provisioning achieved better performance, an average of 10 samples/s and 98.5% GPU utilization, avoiding interference via strict SSD isolation. Sandook achieved similar high performance even with multiplexed SSDs, as it mitigated interference through read/write segregation and avoided stragglers via adaptive routing. This demonstrates that Sandook's storage optimizations translate directly into improved utilization of expensive compute resources (GPUs), reducing costly idle time. Sandook completes the training job at par with static provisioning and faster than static routing.

Static routing also underperformed in image compression (an average of 3751 images/s), again due to read/write interference, whereas static provisioning, with its isolated SSDs, achieved an average of 5135 images/s. Sandook, mitigating interference and leveraging additional SSDs as needed, outperformed both with an average of 5520 images/s. It is able to complete the compression job faster than both baselines.

For the latency-critical storage server, static routing initially exhibited the worst 90$^{\text{th}}$-percentile latency (2 ms), primarily due to transient storage congestion (caused by GC) and read-/write interference. As load intensified beyond $t = 30$ s, static provisioning struggled and fell behind; the overwhelming load (exceeding 1.5 MOPS and triggering 6 GB/s bandwidth demand) saturated its four SSDs, resulting in serious performance disruption. At peak load around $t = 60$ s, static routing and static provisioning exhibited significant tail latencies of 3.5 ms and 9 ms, respectively. In contrast, Sandook maintained sub-millisecond tail latency throughout by effectively managing storage congestion and read/write interference, achieving latency reductions of 71-88% compared to static routing and static provisioning, respectively.

Overall, these results demonstrate that in a realistic rack-scale environment, Sandook successfully unlocks an additional 27% of SSD performance. This translates to an improvement in application throughput by 12—94%, a reduction in application latency by 71–88%, and an increase in GPU utilization for ML training up to 23%, over the baselines.

9.4 Raw Storage Performance

We evaluated Sandook's ability to unleash the performance of heterogeneous SSDs. We focused on raw storage performance rather than end-to-end application performance.

We used a synthetic microbenchmark to measure 4K block RW performance under different RW ratios. We compared Sandook's results to both the ideal baseline and static routing. The ideal throughput represents what could be achieved with an optimal load distribution across SSDs, exhaustively calculated based on workload and each SSD's performance profile. We omit the static provisioning baseline as this experiment only involves one application. A successful outcome for Sandook would show performance significantly exceeding static routing and closely approaching the ideal.

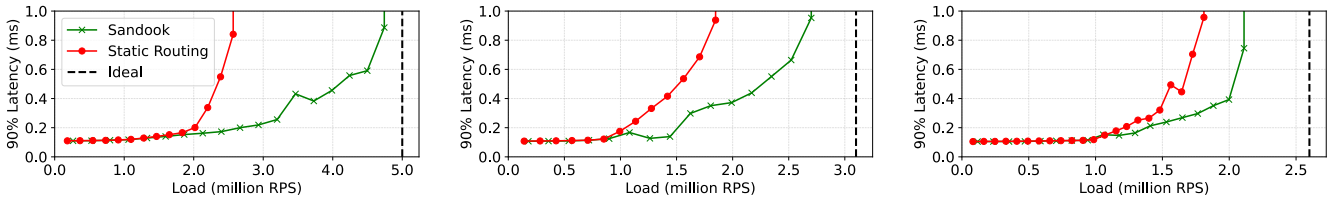
Figure 6 presents the results. In a read-heavy workload (10% writes, Figure 6a), the ideal throughput under a one-millisecond 90$^{\text{th}}$-percentile latency target was 5.0 MOPS. However, static routing achieved 2.6 MOPS, underutilizing performance by 48%. Sandook significantly narrowed this gap, reaching 4.75 MOPS, only 5% away from the ideal.

For a balanced workload (50% writes, Figure 6b), Sandook achieved 2.7 MOPS, only 13% below the ideal and 50% higher than static routing. Likewise, in a write-heavy scenario (70% writes, Figure 6c), Sandook delivered 2.1 MOPS, close to the ideal (within 19%), surpassing static routing by 22%.

These results demonstrate Sandook's ability to unlock the potential of SSDs, delivering performance significantly exceeding the baseline and close to theoretical optimal.

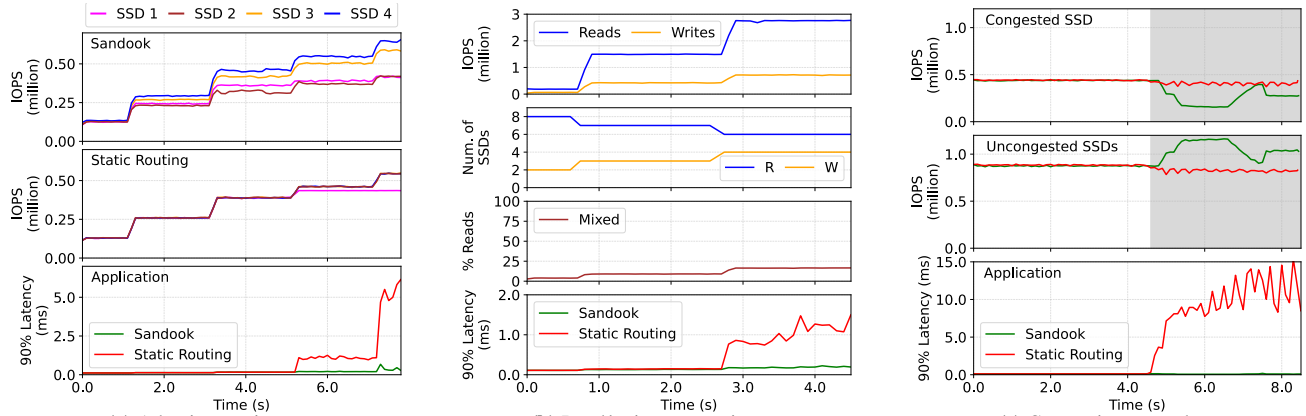
9.5 Drill-Down Experiments

We evaluated the performance impact of each design choice in Sandook using the same experimental setup described in §9.4.



(a) Read heavy (10% writes). **(b)** Read-Write balanced (50% writes). **(c)** Write heavy (70% writes).

Figure 6: Sandook demonstrates near-optimal SSD block access performance across various R/W ratios, outperforming static routing.



(a) Adaptive routing. **(b)** Read/write segregation. **(c)** Congestion control.

Figure 7: **(a)** Static routing leads to the saturation of an SSD but underutilization of others, thereby failing to maintain low tail latency. Conversely, Sandook adaptively steers load based on SSD performance profiles, achieving consistently sub-millisecond tail latency.
(b) Sandook balances write-mode SSD allocation in response to increasing write loads, ensuring a low mixed request ratio to mitigate R/W interference, thereby achieving sub-millisecond latency. Conversely, static routing lacking such a mechanism, faces latency spikes.
(c) When GC occurs (shown in gray), static routing experiences disruptions, due to its inability to detect and respond. In contrast, Sandook dynamically decreases the load on affected SSD and gradually increases it post-GC, maintaining sub-millisecond 90th-ile latency.

9.5.1 Profile-Driven Load Steering.

We investigated Sandook's ability to harness performance heterogeneity across different SSDs. We used four SSDs (for simplicity in illustration) with varying performance levels, ranked from the most worn (lowest performance) to the least worn (highest performance). To assess Sandook's performance, we conducted the synthetic microbenchmark comparing it against static routing. Ideally, Sandook should demonstrate the ability to adaptively distribute storage load across SSDs according to their performance profiles, maintaining stable tail latency.

Figure 7a shows the result. The first part of the figure illustrates the 90th-percentile latency for Sandook and static routing under increasing loads. Subsequent parts present the load distribution among the SSDs in both systems. Static routing results in an even distribution of load across all SSDs initially. At $t = 7.2$ s, when the total load reaches 1.85 MOPS, the first SSD became saturated, causing significant latency spikes in static routing (over 5 ms).

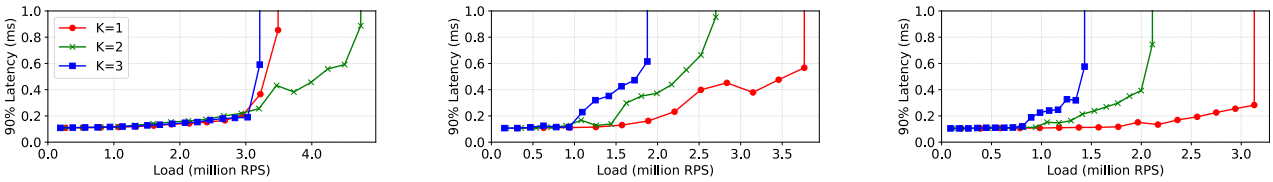
Conversely, Sandook's adaptive weighted routing successfully redistributed the load, effectively managing the performance differences and maintaining sub-millisecond latency.

This result demonstrates that Sandook can effectively manage performance heterogeneity of SSDs.

9.5.2 Read/Write Segregation.

We examined Sandook's ability to mitigate SSD's read/write interference by segregating requests. We ran the synthetic microbenchmark on ten SSDs, using a write ratio of 10%, with progressively increasing load. A desirable outcome for Sandook would demonstrate dynamic allocation of an appropriate number of write-mode SSDs to handle the load while effectively isolating read from write requests.

Figure 7b presents the results. Initially, with a low write load of 0.01 MOPS, Sandook designated only two SSDs for write-mode (aligning with the replication factor) and used the remaining SSDs for read-mode. As the load increased, Sandook adaptively allocated more SSDs for writes; specifically, three write SSDs at 0.50 MOPS (write), and four at 0.75 MOPS (write). By segregating requests, Sandook consistently maintained a low read/write mix ratio, defined as the proportion of requests landing in SSDs operating in the opposite mode. This ratio remained within 4%-16% during the experiment, enabling Sandook to effectively mitigate read/write interference and achieve sub-millisecond 90th-percentile latency. Conversely, static routing, which collocates read and write requests without managing interference, encountered higher latency, reaching 1 ms at $t = 3.5$ s, and 2 ms at $t = 4.5$s.



(a) Read-heavy. (10% writes) **(b)** Read-Write balanced. (50% writes) **(c)** Write-heavy. (70% writes)

Figure 8: Performance impact of varying replication factors in Sandook, measured under different read/write ratios.

This result shows that Sandook's read/write segregation technique successfully mitigates interference.

9.5.3 Storage Congestion Control.

We assessed Sandook's capability to manage transient storage congestion, such as that induced by SSD's garbage collection (GC). We simplified our illustration with a three-SSD setup, where one SSD was pre-filled to near capacity to ensure that subsequent writes would quickly trigger GC. We offered a constant load of 0.9 MOPS to benchmark Sandook's performance under congestion and compared it with static routing. A good result would show that Sandook is able to quickly redistribute the load from the congested SSD to others, minimizing performance impact.

Figure 7c shows the result. The pre-filled SSD initiated GC at $t = 4.5$ s. Unlike static routing, which failed to identify the congestion and suffered a noticeable increase in its 90$^{\text{th}}$-percentile latency (approaching 15 ms), Sandook promptly detected and responded to congestion. Specifically, between $t = 5$ s and $t = 5.3$ s, Sandook reduced the load on the congested SSD from 0.45 MOPS to 0.19 MOPS. It kept the new load distribution while the GC was ongoing. After the completion of GC at $t = 6.8$ s, Sandook gradually increased the load back on the previously affected SSD. A similar pattern was observed starting at $t = 7.5$ s with another GC event occurring. Remarkably, throughout the experiment, Sandook's proactive approach in congestion management maintained a low, sub-millisecond 90$^{\text{th}}$-percentile latency.

This result illustrates Sandook's capability to effectively mitigate transient storage congestion.

9.6 Impact of Replication Factor

We assessed Sandook's performance with various replication factors k—1 (no replication), 2 (the default in other experiments), and 3—using the synthetic microbenchmark.

Figure 8 summarizes the results. In a read-heavy workload (10% writes, Figure 8a), $k = 3$ showed the least performance (3.2 MOPS at a one-millisecond 90$^{\text{th}}$-percentile latency target) due to the overhead of tripling writes and assigning proportionally more write-mode SSDs, thus reducing the performance of reads. $k = 1$ enhanced performance (3.45 MOPS under the same latency target), allowing more SSDs to be in read-mode because of a reduction in writes. $k = 2$ achieved the highest performance of 4.75 MOPS by balancing the amount of writes and high flexibility for reads.

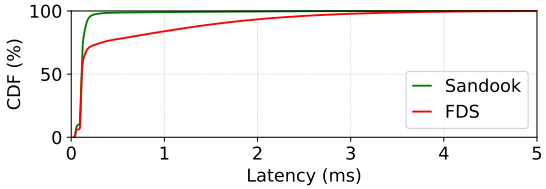


Figure 9: Sandook drastically reduces storage block access latency across a broad range of percentiles, outperforming FDS in a synthetic workload with 10% writes and 3 MOPS offered load.

The impact of write replication grew as writes became more prevalent. In a balanced read-write condition (50% writes, Figure 8b), $k = 1$ emerged as the top performer, benefiting from the absence of replicated writes, and achieving 3.75 MOPS under the sub-millisecond target. $k = 2$ followed, with 2.7 MOPS, outperforming $k = 3$, which reaches 1.9 MOPS.

Similarly, in a write-heavy scenario (70% writes, Figure 8c), higher replication yielded lower performance, with $k = 1, 2, 3$ achieving 3.1, 2.1, and 1.4 MOPS, respectively.

These results show that $k = 2$, our default setting, offers the best performance in typical read-heavy scenarios in datacenters. Meanwhile, for write-intensive scenarios, $k = 1$ delivers better performance but sacrifices fault tolerance.

9.7 Latency Results of Other Percentiles

While previous results focused on 90th-percentile latency, we investigated performance at other percentiles for a comprehensive view. We repeated the synthetic microbenchmark (§9.4) with 10% writes at a constant 3 MOPS load.

Figure 9 compares the latency CDFs for Sandook and FDS. At the 50$^{\text{th}}$ percentile, Sandook and FDS achieved 0.11 ms. However, at 95$^{\text{th}}$ and 99$^{\text{th}}$ percentiles, Sandook maintained sub-millisecond latencies of 0.19 ms and 0.90 ms respectively, significantly better than FDS's 2.27 ms and 3.6 ms.

10 Conclusion

Existing SSD disaggregation systems often neglect SSD performance variability, causing underutilization and inefficiency. Sandook addresses this by using real-time performance monitoring to adaptively steer I/O requests, unlocking full SSD potential. Compared to prior systems, it delivers 1.7× storage throughput, 1.12–1.94× application throughput, 71–88% lower latency, and 23% higher GPU utilization—without specialized hardware or application code changes.

Acknowledgments

We thank our shepherd, Erci Xu, and anonymous reviewers for their insightful suggestions and feedback. We also thank members of the Parallel and Distributed Systems (PDOS) research groups at MIT for their helpful feedback. This research was supported by CNS-2104398 and CNS-2212099, and by ACE, one of seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. CORFU: A shared log design for flash clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 1–14, San Jose, CA, April 2012. USENIX Association.

[2] Jayanta Basak and Madhumita Bharde. Dynamic Provisioning of Storage Workloads. In *29th Large Installation System Administration Conference (LISA15)*, pages 13–24, Washington, D.C., November 2015. USENIX Association.

[3] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[4] Laurent Bindschaedler, Ashvin Goel, and Willy Zwaenepoel. Hailstorm: Disaggregated Compute and Storage for Distributed LSM-based Databases. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 301–316, New York, NY, USA, 2020. Association for Computing Machinery.

[5] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, February 2017. USENIX Association.

[6] L.S. Brakmo and L.L. Peterson. TCP Vegas: end to end congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

[7] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, oct 2016.

[8] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. *SIGMETRICS Perform. Eval. Rev.*, 37(1):181–192, jun 2009.

[9] Zongzhi Chen, Xinjun Yang, Feifei Li, Xuntao Cheng, Qingda Hu, Zheyu Miao, Rongbiao Xie, Xiaofei Wu, Kang Wang, Zhao Song, Haiqing Sun, Zechao Zhuang, Yuming Yang, Jie Xu, Liang Yin, Wenchao Zhou, and Sheng Wang. CloudJump: optimizing cloud databases for cloud storages. *Proc. VLDB Endow.*, 15(12):3432–3444, aug 2022.

[10] Tzi-cker Chiueh, Weafon Tsao, Hou-Chiang Sun, Ting-Fang Chien, An-Nan Chang, and Cheng-Ding Chen. Software Orchestrated Flash Array. In *Proceedings of International Conference on Systems and Storage*, SYSTOR 2014, page 1–11, New York, NY, USA, 2014. Association for Computing Machinery.

[11] Özgün Çiçek, Ahmed Abdulkadir, Soeren S. Lienkamp, Thomas Brox, and Olaf Ronneberger. 3D U-Net: Learning Dense Volumetric Segmentation from Sparse Annotation. *CoRR*, abs/1606.06650, 2016.

[12] Alex Conway, Ainesh Bakshi, Yizheng Jiao, William Jannen, Yang Zhan, Jun Yuan, Michael A. Bender, Rob Johnson, Bradley C. Kuszmaul, Donald E. Porter, and Martin Farach-Colton. File Systems Fated for Senescence? Nonsense, Says Science! In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 45–58, Santa Clara, CA, February 2017. USENIX Association.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[14] Jonathan Corbet. Four-level page tables. https://lwn.net/Articles/106177, 2004.

[15] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[16] Cheng Peng Fu and S.C. Liew. TCP Veno: TCP enhancement for transmission over wireless access networks. *IEEE Journal on Selected Areas in Communications*, 21(2):216–228, 2003.

[17] Persistent Disk at Google Cloud. https://cloud.google.com/persistent-disk?hl=en.

[18] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. Performance Characterization of NVMe-over-Fabrics Storage Disaggregation. *ACM Trans. Storage*, 14(4), dec 2018.

[19] Gabriel Haas and Viktor Leis. What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines. *Proc. VLDB Endow.*, 16(9):2090–2102, may 2023.

[20] Shujie Han, Patrick P. C. Lee, Fan Xu, Yi Liu, Cheng He, and Jiongzhou Liu. An In-Depth Study of Correlated Failures in Production SSD-Based Data Centers. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 417–429. USENIX Association, February 2021.

[21] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 168–183, New York, NY, USA, 2017. Association for Computing Machinery.

[22] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 263–276, Santa Clara, CA, February 2016. USENIX Association.

[23] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 173–190. USENIX Association, November 2020.

[24] Tejun Heo, Dan Schatzberg, Andrew Newell, Song Liu, Saravanan Dhakshinamurthy, Iyswarya Narayanan, Josef Bacik, Chris Mason, Chunqiang Tang, and Dimitrios Skarlatos. IOCost: block IO control for containers in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 595–608, New York, NY, USA, 2022. Association for Computing Machinery.

[25] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, 2012.

[26] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery.

[27] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*. USENIX Association, June 2010.

[28] Lokesh N. Jaliminche, Chandranil Nil Chakraborttii, Changho Choi, and Heiner Litz. Enabling Multi-tenancy on SSDs with Accurate IO Interference Modeling. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 216–232, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. FusionRAID: Achieving Consistent Low Latency for Commodity SSD Arrays. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 355–370. USENIX Association, February 2021.

[30] Ziyang Jiao and Bryan S. Kim. Asymmetric RAID: Rethinking RAID for SSD Heterogeneity. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '24, page 101–107, New York, NY, USA, 2024. Association for Computing Machinery.

[31] The Linux Kernel. Userspace block device driver (ublk driver). https://docs.kernel.org/block/ublk.html.

[32] Giryoung Kim and Dongkun Shin. Performance analysis of SSD write using TRIM in NTFS and EXT4. In *2011 6th International Conference on Computer Sciences and Convergence Information Technology (ICCIT)*, pages 422–423, 2011.

[33] Hyojun Kim and Seongjun Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, February 2008. USENIX Association.

[34] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays. In *USENIX Annual Technical Conference (ATC)*, pages 799–812, Renton, WA, July 2019. USENIX Association.

[35] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. Harmonia: A globally coordinated garbage collector for arrays of Solid-State Drives. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2011.

[36] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash Storage Disaggregation. In *European Conference on Computer Systems (EuroSys)*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash ≈ Local Flash. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '17, page 345–359, New York, NY, USA, 2017. Association for Computing Machinery.

[38] Diego Kreutz, Fernando M. V. Ramos, Paulo Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-Defined Networking: A Comprehensive Survey, 2014.

[39] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanael Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony Rowstron. Understanding Rack-Scale Disaggregated Storage. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association.

[40] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. LeanStore: In-Memory Data Management beyond Main Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 185–196, 2018.

[41] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 591–605, New York, NY, USA, 2020. Association for Computing Machinery.

[42] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. IODA: A Host/Device Co-Design for Strong Predictability Contract on Modern Flash Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 263–279, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Jinhong Li, Qiuping Wang, Patrick P. C. Lee, and Chao Shi. An In-depth Comparative Analysis of Cloud Block Storage Workloads: Findings and Implications. *ACM Trans. Storage*, 19(2), mar 2023.

[44] Nanqinqin Li, Mingzhe Hao, Huaicheng Li, Xing Lin, Tim Emami, and Haryadi S. Gunawi. Fantastic SSD internals and how to learn and use them. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, SYSTOR '22, page 72–84, New York, NY, USA, 2022. Association for Computing Machinery.

[45] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, Huayong Wang, Shanyang Liu, Lulu Chen, Zhiwu Wu, Haonan Qiu, Derui Liu, Gexiao Tian, Chao Han, Shaozong Liu, Yaohui Wu, Zicheng Luo, Yuchao Shao, Junping Wu, Zheng Cao, Zhongjie Wu, Jiaji Zhu, Jinbo Wu, Jiwu Shu, and Jiesheng Wu. More Than Capacity: Performance-oriented Evolution of Pangu in Alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 331–346, Santa Clara, CA, February 2023. USENIX Association.

[46] LWN. Ringing in a new asynchronous I/O API. https://lwn.net/Articles/776703.

[47] LZ4. https://lz4.org.

[48] Liuying Ma, Zhenqing Liu, Jin Xiong, Yue Wu, Renhai Chen, Xi Peng, Ying Zhang, Gong Zhang, and Dejun Jiang. zQoS: Unleashing full performance capabilities of NVMe SSDs while enforcing SLOs in distributed storage systems. In *Proceedings of the 53rd International Conference on Parallel Processing*, ICPP '24, page 618–628, New York, NY, USA, 2024. Association for Computing Machinery.

[49] Peter Mattson, Vijay Janapa Reddi, Christine Cheng, Cody Coleman, Greg Diamos, David Kanter, Paulius Micikevicius, David Patterson, Guenther Schmuelling, Hanlin Tang, Gu-Yeon Wei, and Carole-Jean Wu. MLPerf: An Industry Standard Benchmark Suite for Machine Learning Performance. *IEEE Micro*, 40(2):8–16, 2020.

[50] James Mickens, Edmund B. Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 257–273, Seattle, WA, April 2014. USENIX Association.

[51] Introduction to Azure managed disks. https://learn.microsoft.com/en-us/azure/virtual-machines/managed-disks-overview.

[52] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 106–122, 2021.

[53] Sparsh Mittal and Jeffrey S. Vetter. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2016.

[54] Mihir Nanavati, Jake Wires, and Andrew Warfield. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 17–33, Boston, MA, March 2017. USENIX Association.

[55] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, February 2008. USENIX Association.

[56] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat Datacenter Storage. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–15, Hollywood, CA, October 2012. USENIX Association.

[57] NVIDIA A100 Tensor Core GPU Datasheet, 2024.

[58] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '14, page 471–484, New York, NY, USA, 2014. Association for Computing Machinery.

[59] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. *PyTorch: an imperative style, high-performance deep learning library*, page 12. Curran Associates Inc., Red Hook, NY, USA, 2019.

[60] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi Noghabi, and Jian Huang. BlockFlex: Enabling Storage Harvesting with Software-Defined Flash in Modern Cloud Platforms. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–33, Carlsbad, CA, July 2022. USENIX Association.

[61] Benjamin Reidys, Yuqi Xue, Daixuan Li, Bharat Sukhwani, Wen-Mei Hwu, Deming Chen, Sameh Asaad, and Jian Huang. RackBlox: A Software-Defined Rack-Scale Storage System with Network-Storage Co-Design. In *SOSP '23*, page 182–199, New York, NY, USA, 2023. Association for Computing Machinery.

[62] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured Memory for DRAM-based Storage. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 1–16, Santa Clara, CA, February 2014. USENIX Association.

[63] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[64] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009.

[65] Samsung PM1725a NVMe SSD. https://download.semiconductor.samsung.com/resources/brochure/Brochure_Samsung_PM1725a_NVMe_SSD_1805.pdf, 2018.

[66] Sandook GitHub Repository. https://github.com/mit-sandook/sandook.

[67] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. Xoring elephants: Novel erasure codes for big data. *arXiv preprint arXiv:1301.3791*, 2013.

[68] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Gecko: contention-oblivious disk arrays for cloud storage. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, page 285–298, USA, 2013. USENIX Association.

[69] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on Rails: Consistent Flash Performance through Redundancy. In *USENIX Annual Technical Conference (ATC)*, pages 463–474, Philadelphia, PA, June 2014. USENIX Association.

[70] Storage Performance Development Kit. https://spdk.io/doc.

[71] Robbert Van Renesse and Fred B Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, volume 4, 2004.

[72] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building An Elastic Query Engine on Disaggregated Storage . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 449–462, Santa Clara, CA, February 2020. USENIX Association.

[73] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 15–28, Santa Clara, CA, February 2017. USENIX Association.

[74] Weidong Zhang, Erci Xu, Qiuping Wang, Xiaolu Zhang, Yuesheng Gu, Zhenwei Lu, Tao Ouyang, Guanqun Dai, Wenwen Peng, Zhe Xu, Shuo Zhang, Dong Wu, Yilei Peng, Tianyun Wang, Haoran Zhang, Jiasheng Wang, Wenyuan Yan, Yuanyuan Dong, Wenhui Yao, Zhongjie Wu, Lingjun Zhu, Chao Shi, Yinhu Wang, Rong Liu, Junping Wu, Jiaji Zhu, and Jiesheng Wu. What's the Story in EBS Glory: Evolutions and Lessons in Building Cloud Block Store. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 277–291, Santa Clara, CA, February 2024. USENIX Association.

[75] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. Tencent block storage traces (SNIA IOTTA trace 27920). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, September 2018.

[76] Mark Zhao, Niket Agarwal, Aarti Basant, Buğra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, Sundaram Narayanan, Jack Langman, Kevin Wilfong, Harsha Rastogi, Carole-Jean Wu, Christos Kozyrakis, and Parik Pol. Understanding data storage and ingestion for large-scale deep recommendation model training: industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 1042–1057, New York, NY, USA, 2022. Association for Computing Machinery.

[77] Mark Zhao, Satadru Pan, Niket Agarwal, Zhaoduo Wen, David Xu, Anand Natarajan, Pavan Kumar, Shiva Shankar P, Ritesh Tijoriwala, Karan Asher, Hao Wu, Aarti Basant, Daniel Ford, Delia David, Nezih Yigitbasi, Pratap Singh, and Carole-Jean Wu. Tectonic-Shift: A Composite Storage Fabric for Large-Scale ML Training. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 433–449, Boston, MA, July 2023. USENIX Association.

[78] Yue Zhu, Weikuan Yu, Bing Jiao, Kathryn Mohror, Adam Moody, and Fahim Chowdhury. Efficient User-Level Storage Disaggregation for Deep Learning. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12, 2019.

A Appendix

A.1 Handling I/O

We list the steps taken by the Sandook Client when handling read and write I/O operations in Algorithms 1 and 2.

Algorithm 1 Read I/O Path

Require: Logical block b_l
Ensure: Data from physical block
$P \leftarrow$ MappingTable$[b_l]$
if $P = \emptyset$ **then**
  return error
end if
$p_{selected} \leftarrow$ SelectReplica$(P, weights)$
$data \leftarrow$ ReadFromRemoteSSD$(p_{selected})$
return $data$

Algorithm 2 Write I/O Path

Require: Logical block b_l, data d, replication factor k
Ensure: Data written to k physical blocks
$P \leftarrow \emptyset$ ▷ Set of physical blocks
for $i = 1$ to k **do**
  $p_i \leftarrow$ SelectFromFreeList$(weights)$
  $P \leftarrow P \cup \{p_i\}$
end for
$old\_map \leftarrow$ MappingTable$[b_l]$
if $old\_map \neq$ null **then**
  Invalidate(old_map)
end if
MappingTable$[b_l] \leftarrow P$
for each $p_i \in P$ **do**
  WriteToRemoteSSD(p_i, d)
end for

A.2 Trim

We list the steps taken by the Sandook Controller when coordinating trim operation across SSDs in Algorithm 3.

Algorithm 3 Trim Coordination (Controller)

Require: Set of SSDs S, token ring order R
Ensure: coordinated trim operations across SSDs
$token \leftarrow 0$ ▷ Token position in ring
while true **do**
  $InvalidatedBlocks \leftarrow$ PollClients() ▷ Gather invalidated blocks
  $current\_ssd \leftarrow R[token]$
  $blocks\_to\_trim \leftarrow$ FilterBlocks$(InvalidatedBlocks, current_ssd)$
  if $|blocks\_to\_trim| > 0$ **then**
    $w_{old} \leftarrow weights[current\_ssd]$
    $weights[current\_ssd] \leftarrow w_{old} \times \alpha$ ▷ Reduce weight, $\alpha < 1$
    BroadcastWeights(weights) ▷ Notify clients
    IssueTrim$(current_ssd, blocks_to_trim)$
    WaitForCompletion$(current_ssd)$
    $weights[current\_ssd] \leftarrow w_{old}$ ▷ Restore weight
    BroadcastWeights(weights)
  end if
  $token \leftarrow (token + 1) \mod |R|$ ▷ Move token to next SSD
  Sleep$(T_{interval})$ ▷ Wait before next iteration
end while