Gimbal: Enabling Multi-tenant Storage Disaggregation on SmartNIC JBOFs

Jaehong Min
University of Washington and Samsung Electronics

Ming Liu
University of Wisconsin-Madison and VMware Research

Tapan Chugh
University of Washington

Chenxingyu Zhao
University of Washington

Andrew Wei
University of Washington

In Hwan Doh
Samsung Electronics

Arvind Krishnamurthy
University of Washington

Abstract

Emerging SmartNIC-based disaggregated NVMe storage has become a promising storage infrastructure due to its competitive IO performance and low cost. These SmartNIC JBOFs are shared among multiple co-resident applications, and there is a need for the platform to ensure fairness, QoS, and high utilization. Unfortunately, given the limited computing capability of the SmartNICs and the non-deterministic nature of NVMe drives, it is challenging to provide such support on today's SmartNIC JBOFs.

This paper presents Gimbal, a software storage switch that orchestrates IO traffic between Ethernet ports and NVMe drives for co-located tenants. It enables efficient multi-tenancy on SmartNIC JBOFs using the following techniques: a delay-based SSD congestion control algorithm, dynamic estimation of SSD write costs, a fair scheduler that operates at the granularity of a virtual slot, and an end-to-end credit-based flow control channel. Our prototyped system not only achieves up to x6.6 better utilization and 62.6% less tail latency but also improves the fairness for complex workloads. It also improves a commercial key-value store performance in a multi-tenant environment with x1.7 better throughput and 35.0% less tail latency on average.

CCS Concepts

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

Keywords

congestion control, disaggregated storage, SSD, fairness

ACM Reference Format:
Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: Enabling Multi-tenant Storage Disaggregation on SmartNIC JBOFs . In *ACM SIGCOMM 2021 Conference (SIGCOMM '21), August 23–28, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3452296.3472940

SIGCOMM '21, August 23–28, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8383-7/21/08…$15.00
https://doi.org/10.1145/3452296.3472940

1 Introduction

Storage disaggregation has gained significant interest recently since it allows for independent scaling of compute/storage capacities and achieves high resource utilization [1–4, 48]. As datacenter network speeds have transitioned to 100/200 Gbps and the NVMe-oF specification [15] that enables flash-based SSDs to communicate over a network is becoming widely adopted, a disaggregated NVMe SSD can provide microsecond-scale access latencies and millions of IOPS.

Recently, SmartNIC-based disaggregated storage solutions, such as Mellanox BlueField and Broadcom Stingray [8, 11], have emerged and become increasingly popular because of their low deployment costs and competitive IO performance compared to traditional server-based approaches. Such a storage node usually comprises a commercial-of-the-shelf high-bandwidth SmartNIC, domain-specific accelerators (like RAID), a PCIe-switch, and a collection of NVMe SSDs, supported by a standalone power supply. Consider the Broadcom Stingray solution as an example. Compared with a conventional disaggregated server node, the Stingray PS1100R storage box is much cheaper and consumes up to 52.5W active power while delivering 1.4 million 4KB random read IOPS at 75.3μs unloaded latency.

Disaggregated storage is shared among multiple tenants for running different kinds of storage applications with diverse IO access patterns. An efficient multi-tenancy mechanism should maximize NVMe SSD usage, ensure fairness among different storage streams, and provide QoS guarantees without overloading the device. However, today's SmartNIC JBOFs[1] lack such essential support, which is non-trivial to build. First, the unpredictable performance characteristics of NVMe SSDs (which vary with IO size, read/write mix, random/sequential access patterns, and SSD conditions) make it extremely hard to estimate the runtime bandwidth capacity and per-IO cost of the storage device. For example, as shown in Section 2.3, a fragmented SSD can only achieve 16.9% write bandwidth of a clean SSD; adding 5% writes to a read-only stream could cause a 42.6% total IOPS drop. Second, an SSD has a complex internal architecture, and its controller does not disclose the execution details of individual IO commands. This complicates the IO service time estimation of a tenant as well as the fair scheduler design. Finally, SmartNICs are wimpy computing devices. Besides driving

[1] JBOF = Just a Bunch of Flash

the full networking and storage bandwidth, the amount of available computation per-IO is bounded, i.e., 1μs and 5μs for a 4KB and 128KB read, respectively.

To address these challenges, we design and implement Gimbal, a *software storage switch* that orchestrates NVMe-oF commands among multiple co-located tenants. Gimbal borrows ideas from traditional networking and applies them to the domain of managing storage resources. First, Gimbal views the SSD device as a networked system and applies a *delay-based congestion control* mechanism to estimate its runtime bandwidth headroom. Second, it introduces the virtual slot concept and the write cost estimation logic to enable online characterization of the *per-IO cost*. Finally, Gimbal exposes an SSD virtual view via an end-to-end *credit-based flow control* and *request priority tagging* so that applications are able to design flexible IO prioritization, rate-limiting, and load-balancing mechanisms.

We prototype Gimbal on Broadcom Stingray PS1100R SmartNIC JBOFs and compare with previously proposed multi-tenant storage solutions with isolation mechanisms (i.e., Reflex [49], Parda [38], FlashFQ [70]). Our experiments with synthetic workloads show that Gimbal not only achieves up to x6.6 better utilization and 62.6% less tail latency but also improves the fairness for various complex workloads. We also ported a commercial key-value store (i.e., RocksDB) over a blobstore file system implemented on our disaggregated storage platform. Our implementation employs a hierarchical blob allocator to fully utilize a pool of storage nodes. It manages the storage load and steers read requests based on the runtime loads of the storage devices using an IO rate limiter and a load balancer. Our evaluations show that Gimbal can improve application throughput by x1.7 and reduce tail latency by 35.0% on average.

2 Background and Motivating Experiments

2.1 NVMe-over-Fabrics Protocol

NVMe-over-Fabrics (NVMe-oF) [15] is an emerging storage protocol to support disaggregation of modern memory devices (e.g., NAND, persistent RAM). It defines a common architecture that supports the NVMe block storage protocol over a range of storage network fabrics (e.g., RDMA, TCP, Fiber Channel). NVMe-oF extends the NVMe base specification [14] and the controller interface. A storage client (i.e., NVMe-oF initiator) first attaches to a storage server (also known as NVMe-oF target) and then issues NVMe commands to the remote controller. The NVMe-oF target comprises two main components: NVMe target core and fabric transport. After setting up a connection with the initiator, it creates a one-to-one mapping between IO submission queues and IO completion queues.

NVMe-over-RDMA relies on memory-mapped IO for all operations. Both the host and the device perform memory read/write of the host memory to modify the related data structures (including submission queue, completion queue, data buffer). NVMe-over-RDMA uses different RDMA verbs for initiating and fulfilling IO flows. Specifically, *RDMA_SEND* is used to issue a submission capsule to the target and a completion capsule back to the host. All data transfers are performed at the NVMe-oF target using *RDMA_READ* and *RDMA_WRITE* verbs. Thus, the data transfer phase requires no host-side computing cycles. Further, unlike the NVMe specification, NVMe-oF does not introduce an interrupt mechanism for the storage controller. Instead, host interrupts are generated by the host fabric interface (e.g., host bus adapter, RDMA NIC).

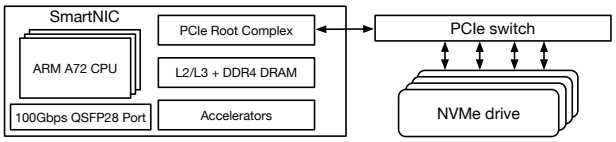


Figure 1: Architectural block diagram of the Stingray PS1100R SmartNIC-based disaggregated storage.

Concretely, the request flow of a read/write under NVMe-over-RDMA is as follows: (a) a client host sends an NVMe command capsule (including the NVMe submission queue entry and the scatter-gather address list) to an NVMe-oF target using *RDMA_SEND*; (b) the target process picks up commands from the submission queue. Under a write, it fetches client data via the *RDMA_READ*; (c) the target storage controller then performs a read or write IO execution on the SSDs; (d) in case of reads, the NVMe-oF target issues a *RDMA_WRITE* to transmit data from a local buffer back to the client host memory; (e) the NVMe-oF target process catches the completion signal, builds a response capsule (which contains the completion queue entry), and sends this completion capsule via *RDMA_SEND*. Some NVMe-oF implementations allow for inlining small data blocks (e.g., 4KB) into the capsule, reducing the number of RDMA messages and improving the IO latency.

2.2 SmartNIC JBOF

SmartNICs [5, 8, 10–13] have emerged in the datacenter recently, not only for accelerating packet manipulations and virtual switching functionalities [6, 36], but also offloading generic distributed workloads [35, 54, 58, 59]. Typically, a SmartNIC comprises general-purpose computing substrates (e.g., ARM or FPGA), an array of domain-specific accelerators (e.g., crypto engine, reconfigurable match-action table), onboard memory, and a traffic manager for packet steering. Most SmartNICs are low-profile PCIe devices that add incremental cost to the existing datacenter infrastructure and have shown the potential to expand the warehouse computing capacity cheaply.

Lately, hardware vendors have combined a SmartNIC with NVMe drives as *disaggregated storage* to replace traditional server-based solutions for cost efficiency. Figure 1 presents the Broadcom Stingray solution [8]. It encloses a PS1100R SmartNIC, a PCIe carrier board, a few NVMe SSDs, and a standalone power supply. The carrier board holds both the SmartNIC and NVMe drives and an on-board PCIe switch connecting the components. The SmartNIC has 8 × 3.0GHz ARM A72 CPU, 8GB DDR4-2400 DRAM (along with 16MB cache), FlexSPARX [7] acceleration engines, 100Gb NetXtreme Ethernet NIC, and PCIe Gen3 root complex controllers. The PCIe switch offers ×16 PCIe 3.0 lanes (15.75GB/s theoretical peak bandwidth), and can support either 2 × 8 or 4 × 4 PCIe bifurcation. A Stingray disaggregated storage box with four Samsung DCT983 960GB SSDs, is listed for $3228.0, with likely much lower bulk prices, and is thus much cheaper than a Xeon-based one with similar IO configurations. Unsurprisingly, it also consumes lower power than a Xeon

disaggregated server node. The power consumption of a Stingray PS1100R storage node is 52.5W at most, nearly one-fourth of the Xeon one (192.0W). Section 5.1 describes the hardware configurations. We use a *Watts Up Pro* meter [17] to measure the wall power. The software stack on a SmartNIC JBOF works similar to the Xeon-based server JBOF, except that the NVMe-oF target runs on the SmartNIC wimpy cores instead of the x86 cores. The IO request processing also includes five steps as described above.

SmartNIC JBOFs achieve performance that is competitive to server JBOFs. In terms of unloaded read/write latency, we configure `fio` [9] with one outstanding IO and measure the average latency as we increase the request size (Figure 2). When serving random reads, the SmartNIC solution adds 1.0% latencies on average across five cases where the request size is no larger than 64KB. The latency differences rise to 20.3% and 23.3% if the IO block size is 128KB and 256KB, respectively. For sequential writes, SmartNIC execution adds only 2.7μs compared with the server, on average across all cases. We further break down the latency at the NVMe-oF target and compare the server and SmartNIC cases. We find that the most time-consuming part for both reads and writes is the NVMe command execution phase (including writing into the submission queue, processing commands within the SSD, and catching signals from the completion queue). This explains why the latencies on SmartNIC and server JBOFs are similar. For a 4KB/128KB random read, it contributes to 92.4%/86.1% and 88.8%/92.2% for server and SmartNIC, respectively.

Considering bandwidth, SmartNIC JBOF is also able to saturate the storage limit but using more cores. This experiment measures the maximum 4KB random read and sequential write bandwidth as we increase the number of cores. For each FIO configuration, we increase the IO depth to maximize throughput. As depicted in Figure 3, the server achieves 1513 KIOPS and 1316 KIOPS using two cores, respectively. In the case of the SmartNIC, it is able to serve similar read and write traffic with 3 ARM cores. One core is enough to achieve the maximum bandwidth under a large request size (i.e., 128KB).

2.3 Multi-tenant Disaggregated Storage

Different kinds of storage applications share disaggregated storage nodes, and therefore, we need to provide support for multi-tenancy and isolation between different workloads. Today's NVMe SSDs provide some isolation support. For example, an NVMe namespace is a collection of logical block addresses that provides independent addressing and is accessed via the host software. However, namespaces do not isolate SSD data blocks physically, and requests to access different namespaces can still interfere.

An ideal mechanism should achieve the following goals: (1) provide fairness across tenants; (2) maintain high device utilization; (3) exhibit low computation overheads and predictable delays at the storage node. Our characterizations show that existing SmartNIC JBOFs present limited support for multi-tenancy, as multiple aspects of IO interference cause unfair resource sharing of storage throughput. In Figure 4, the victim flow issues random 4KB reads with 32 concurrent I/Os, and we inject a neighboring flow with various IO sizes, intensity, and patterns. Overall, a flow with high intensity always obtains more bandwidth regardless of the IO size and pattern. For example, the bandwidth of the neighboring flow with random 128KB reads is 377MB/s and 58.4% less than the victim's when it has only one concurrent IO. But, it dramatically rises to 1275MB/s as the concurrent IO is increased to 8 and obtains 3.1x higher bandwidth than the victim. In addition, the bandwidth of the victim decreases significantly when the neighboring flow uses a write pattern. The victim shows 59.1% less bandwidth when the neighbor has the same IO size and intensity but performs writes. (Appendix D describes more characterizations.) We summarize below three challenges to realizing an efficient multi-tenancy mechanism for JBOFs.

Issue 1: IO throughput capacity varies unpredictably with workload characteristics (e.g., random v.s. sequential, read v.s. write, IO size) and SSD conditions. Modern NVMe SSDs favor sequential access due to request coalescing and write-ahead logging [21, 29]. Under the disaggregated setting, the overall IO access patterns become much more random due to request interleaving (i.e., IO blender effect) and impact the IO throughput. SSDs also experience read/write interference. To minimize NAND access contention, an SSD distributes reads/writes across different data channels and NAND chips [31, 34]. Writes are usually slower than reads (unless they are directly served from the SSD controller write buffer) and incur the erase-before-write penalty. A flash page has to clear its data blocks before accepting new values. This causes the write amplification problem [21], where the actual amount of data written to the storage media is more than the intended size. Further, SSDs present asymmetric performance under different IO sizes due to NAND access granularity and die-level parallelism. For example, on a Samsung DCT983 960GB SSD, 128KB read could achieve 3.2GB/s, while the 4KB one maxes out at 1.6GB/s.

The SSD condition, typically characterized by the number of clean flash pages and their location distribution, depends on the previous write history. An SSD uses a flash translation layer (FTL) that maps logical blocks to physical blocks. A new write appends to a pre-erased block first. To improve the write performance and endurance, it also employs wear leveling (i.e., balancing the program/erasure cycles among all data blocks) [21, 81] and garbage collection [21, 68] (i.e., replenishing valid blocks), which complicates estimating the SSD condition. When the SSD is highly fragmented, there are fewer free blocks, and garbage collections are often triggered, which hurts both write and read performance (described in Appendix A).

Issue 2: Per-IO cost changes drastically with not only read and write mix but also IO concurrency and IO size. An IO will be delayed when there is execution contention at the SSD controller, head-of-line blocking at the device queue, access contention at the NAND channel/chip/die, or SSD internal activities (e.g., garbage collection). SSD vendors are hesitant to disclose the device execution statistics, making it hard to identify the per-IO cost. This complicates fair IO resource allocations among multiple tenants. Previous studies [49, 60, 73] apply approximate offline-profiled IO cost models, which cannot accurately capture the SSD's dynamic runtime state.

Issue 3: Fairness in a multi-tenant disaggregated storage means that each tenant should receive the same amount of request service time from an SSD controller. The controller's request service time is opaque to the host software stack, complicating the IO scheduler design. Even though modern NVMe SSDs apply a multi-queue interface [26, 47, 75] for multiplexing and multicore

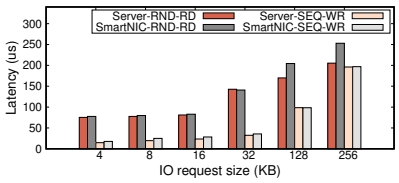


Figure 2: Read/write latency comparison between SmartNIC and Server JBOFs.

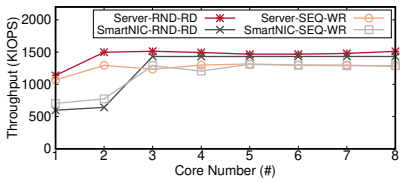


Figure 3: Read/write throughput as increasing the number of cores on server and SmartNIC JBOFs.

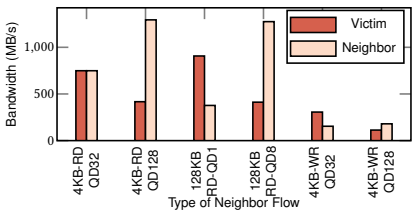


Figure 4: Multi-tenant interference in different workloads. (RD:Read, WR:Write, QD:Number of Concurrent I/Os)

scalability, the device usually schedules requests among IO queues in a simple round-robin fashion, impeding the fairness support. Prior works [49, 60, 64, 70, 70] apply deficit/weighted round-robin scheduling or its variants to issue IOs from different tenants. In such approaches, determining an operation's deficit value/weight is vital to achieving fairness. However, as discussed above, simply using IOPS, bandwidth (bytes/s), or some other synthetic static metrics (such as virtual IOPS [73], approximate IO cost mode [60], or SLO-aware token [49]), cannot capture the exact IO execution time. For example, consider a 4KB random read stream mixed with a 64KB one with the same type and IOPS (Figure 20 in Appendix). If we use IOPS as the metric, these two streams achieve 91.0MB/s and 1473.0MB/s, indicating that larger IOs dominate the SSD execution; if we use bandwidth instead, smaller IOs could submit four times more requests. Some other works proposed timeslice-based IO schedulers (e.g., Argon [77], CFQ [25], FIOS [70]) that provide time quanta with exclusive device access. These approaches not only violate the responsiveness under high consolidation but also ignore the fact that the IO capacity is not constant (as discussed above).

2.4 Challenges of SmartNIC-based Disaggregation

SmartNICs have wimpy computing cores compared with the server case. When achieving the same storage load, the request tail latency on SmartNIC JBOFs is higher. For example, the 99.9th latency is 34/66μs on server/SmartNIC if serving ~3000MB/s 4KB sequential writes. To fully drive the storage read/write bandwidth, SmartNICs have little computing headroom for each IO request. We evaluate the achieved bandwidth of 4KB/128KB random read and sequential write as we add to the per-IO processing cost (Figure 16 in Appendix). We use all NIC cores in this case. The maximum tolerable latency limit is 1μs and 5μs for 4KB read and write requests, respectively. However, if the request size is 128KB, one can add at most 5μs and 10μ of execution cost for reads and writes without bandwidth loss. Thus, we can only add minimal computation for each storage IO, and the amount of offloading depends on the storage traffic profile.

Fortunately, as we demonstrate in this work, the limited SmartNIC computing capabilities are sufficient to realize a *software storage switch* and equip it with QoS techniques along the ingress/egress data planes for IO orchestration.

3 SmartNIC as a Switch

This section describes our design and implementation of Gimbal, a software storage switch with efficient multi-tenancy support (i.e., high utilization, low latency, and fairness) for SmartNIC JBOFs.

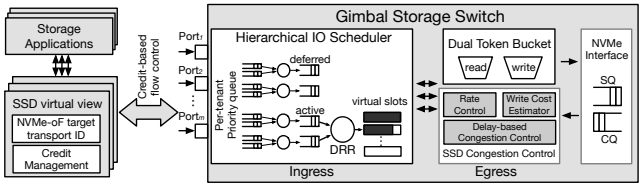


Figure 5: An overview of the Gimbal storage switch architecture. On the Broadcom Stingray PS1100R, there are at most 4 NIC ports along with 4 NVMe SSDs.

We first describe its high-level architecture and then discuss each system component in detail.

3.1 Overview

Figure 5 presents the overall architecture of the software storage switch, inspired by today's SAN switches [18, 19]. It comprises per-SSD pipelines that orchestrate IO flows between NIC ports and NVMe SSDs. Each pipeline is equipped with three major components: (1) *IO scheduler* at the ingress, which provides per-tenant priority queueing and executes IOs in a deficit round-robin fashion (DRR) using a normalized IO unit (called a virtual slot). It exposes a fair queueing abstraction; (2) *delay-based congestion control mechanism* at the egress, which measures the storage bandwidth availability and monitors the SSD load status using IO completion time at runtime. In addition, it employs a rate pacing engine to mitigate congestion and IO burstiness during submission; (3) *write-cost estimator*, dynamically calibrating the SSD write cost based on latency and providing this information to other system components. It implements an approximate performance model that adapts to the workload and SSD conditions.

The switch runs across one or several dedicated SmartNIC cores and listens on active NIC ports. Similar to other works [49], a tenant contains an RDMA qpair (for request send/receive) and an NVMe qpair (for storage command submission/completion). Gimbal focuses on enabling efficient SSD sharing and relies on the remote transport protocol (e.g., RDMA) to address in-network contention.

3.2 Delay-based SSD Congestion Control

An SSD has a complex internal architecture. To estimate its instantaneous capacity headroom, we take a black-box approach and borrow the congestion control mechanism from the networking domain. Specifically, we view the SSD as a networked system, where the controller, circuitry, and NAND chips behave as a router, connection pipes, and end-hosts, respectively. Thus, one can use a TCP-like probing technique to measure its available bandwidth. However,

this is non-trivial because (1) the SSD internal parallelism is unknown due to the FTL mapping logic and IO request interference; (2) housekeeping operations (such as garbage collection) are unpredictable and consume a non-deterministic amount of bandwidth when triggered; (3) an SSD is a lossless system without request drops. Therefore, we develop a customized delay-based congestion control algorithm to address these challenges.

Traditional delay-based algorithms (e.g., TCP Vegas [27]) use the measured RTT to calculate the actual bandwidth and take the bandwidth difference (between actual and expected) as a congestion signal to adjust its transmission window. However, the bandwidth metric is ineffective for SSDs because of their opaque parallelism [28, 45]. An SSD employs multiple NAND channels, planes, and dies to improve its bandwidth so that concurrent IO requests could execute in parallel and complete independently. Thus, the IO latency might not be indicative of the consumed SSD bandwidth. Further, a modern SSD usually applies a 4KB-page based mapping mechanism. It splits a large IO into multiple 4KB chunks and then spreads them to different channels as much as possible. Consequently, the latency is not linear with the IO size, and different sized IOs would achieve different maximum bandwidths.

Instead, we explore the direct use of the IO latency as the feedback (like Swift [50]) and take the latency difference between measured and target levels as a congestion signal. This is motivated by our observation that the SSD access latency is very sensitive to the device load and has an impulse response to the congestion (see Figure 17 in Appendix). A key question to realizing this mechanism is determining the latency threshold. We start with a fixed value (e.g., 2ms) and measure the average achieved latency using EWMA (Exponential Weighted Moving Average), where α_D denotes the weight. We find that 2ms fixed threshold is only effective for large IOs (like 64/128KB) but cannot capture the congestion for small IOs promptly. Reducing the threshold (e.g., <1ms) would also not work because it hurts the device utilization. Therefore, we propose a dynamic latency threshold scaling method. It works similar to Reno's congestion control logic [63] for the latency threshold. Specifically, we set up the minimum and maximum threshold and adjust the value based on the EWMA IO latency using,

$$Thres(t) = Thres(t-1) - \alpha_T \times (Thres(t-1) - Latency_{ewma})$$

When the EWMA latency approaches the threshold, it promptly detects a latency increase. Once the EWMA IO latency exceeds the current threshold, it generates a congestion signal, and the threshold is increased to the midpoint of the current and the maximum threshold by $(Thres(t) = (Thres(t-1)+Thresh_{max})/2$. Consequently, Gimbal gets the congestion signal more frequently if the EWMA latency is close to the maximum threshold or grows rapidly. Tuning the min/max threshold should consider the efficiency of congestion detection, device utilization, convergence time, as well as the flash media characteristics (i.e., SLC/MLC/TLC).

3.3 Rate Control Engine

Under a congestion signal, Gimbal applies a rate pacing mechanism to decide the IO submission rate. Traditional congestion window-based approaches are ineffective due to the following reasons. First, since a storage stream contains IOs of various sizes and types, the same window size (i.e., outstanding bytes) would result in different bandwidths for different IO patterns. Second, a short burst of write IOs absorbed by the SSD internal write buffer would cause a significant increase of the available window size. As a result, more IOs would be submitted, overwhelming the device's capability, and the congestion control system would suffer from performance fluctuations. We instead use a rate pacing mechanism with a token bucket algorithm to address this issue. Further, since the single bucket approach would submit write IOs at a wrong rate (i.e., the read rate) and cause severe latency increments, Gimbal employs a dual token bucket algorithm that consists of separate buckets for reads and writes (see Appendix C.1).

We define four congestion states based on the latency thresholds ($Thresh_{max}$, $Thresh_{min}$, $Thresh_{cur}$) and the measured EWMA latency (Lat_{ewma}), and adjust the target submission rate upon each IO completion. Specifically, these four states are: *overloaded* ($Lat_{ewma} \geq Thresh_{max}$), *congested*($Thresh_{cur} \leq Lat_{ewma} < Thresh_{max}$), *congestion avoidance*($Thresh_{min} \leq Lat_{ewma} < Thresh_{cur}$), and *under-utilized*($Lat_{ewma} < Thresh_{min}$).

For the congestion avoidance and congested states, Gimbal incrementally changes the target rate to either probe for more bandwidth or lower the offered load. The rate is increased/decreased by the IO completion size for the congestion avoidance/congested state. For the overloaded state, the rate is immediately adjusted to be lower than the IO completion rate, and Gimbal discards the remaining tokens in the buckets to avoid a bursty submission. Gimbal periodically measures the completion rate for this. In addition, Gimbal increases the target rate at a faster rate when it observes the underutilized state (parameter β in Algorithm 1).

Gimbal handles the overloaded and under-utilized states in the above manner because the incremental adjustments are meaningful only when the IO pattern doesn't change; incremental adjustments will not converge fast enough for dynamic IO patterns. Specifically, the maximum bandwidth of the SSD may differ dramatically according to the read and write mixed ratio. On a fragmented SSD (defined in Section 5.1), the random write bandwidth is about 180MB/s while the random read bandwidth is over 1600MB/s. Consequently, the rate converges slowly if the pattern shifts from write-heavy to read-heavy. Gimbal, inspired by CUBIC[40] and TIMELY[62], adapts an aggressive probing strategy for identifying the desired operating point. This condition only appears when the target rate is insufficient to submit IO before the SSD drains most of the IOs in the internal queue. On the other hand, the target rate may exceed the maximum bandwidth enormously when the pattern shifts in the opposite direction (i.e., from read-heavy to write-heavy), resulting in the SSD performing at its peak bandwidth but with a latency higher than $Thresh_{max}$. In this case, Gimbal first identifies the current completion rate and sets the target rate to the completion rate. It then further reduces the target rate by an amount equal to that of the size of the completed IO. As a consequence, Gimbal holds the target rate to be lower than that of the SSD's peak capacity until the SSD drains the queued IOs and starts exhibiting a normal latency.

Algorithm 1 depicts the congestion control logic. The submission function is invoked on each request arrival and completion so that it works in a self-clocked manner. With the congestion control mechanism, the SSD maintains an average delay in a stable range providing performance comparable to the device maximum. It adjusts the IO request rate of each SSD while eliminating unnecessary

waiting time in the device's internal queue. Crucially, it allows us to determine the performance headroom of an SSD during runtime.

3.4 Write Cost Estimation

Performance asymmetry of reads and writes is a well-known issue for NAND devices. Typically, a write consumes more resources than the same-sized read due to the write amplification caused by garbage collection. To capture this, we introduce the *write cost* parameter – the ratio between the achieved read and write bandwidths. For example, let the maximum read and write bandwidths (measured independently) of a given SSD be 1000MB/s and 300MB/s, respectively. The write cost is 3.3 in this case, where we assume that 700MB/s might be used to run internal tasks for the write. We cannot obtain this value directly from the SSD. Instead, we use a baseline setting as the worst case and dynamically calibrate the write cost according to the current SSD state. One can obtain $write\ cost_{worst}$ via a pre-calibration or from the SSD specification, so it is a fixed parameter for a specific SSD.

We update the write cost periodically in an ADMI (Additive-Decrease Multiplicative-Increase) manner. The write cost decreases by δ if the write EWMA latency is lower than the minimum latency threshold and increases to $(write\ cost + write\ cost_{worst})/2$ otherwise. This allows Gimbal to quickly converge to the worst-case when we observe latency increases. By using the latency for adjusting the write cost, Gimbal takes the SSD device optimization for writes into consideration. Specifically, an SSD encloses a small DRAM write buffer and stores user data in the buffer first before flushing it in a batch to the actual NAND at the optimal time [46]. When the write submission rate is lower than the write buffer consuming capability, writes are served immediately with consistent low latency. In this case, unlike other schemes that have only a fixed cost ratio between read and write, Gimbal reduces the cost down to 1 (i.e., $write\ cost = 1$), same as the read cost. When the write rate rises beyond the write buffer serving capacity, its latency and write cost increase.

3.5 Two-level Hierarchical IO Scheduler and Virtual slot

We define the per-IO cost of NVMe SSDs as the average occupancy of operations within the NAND per byte of transmitted data. It is not constant and is affected by numerous factors (Section 2.3). This metric reflects how an SSD controller executes different types of NAND commands, such as splitting a large request into multiple small blocks, blocking in an internal queue due to access contention, etc. The IO cost should be considered as a key evaluation parameter for determining fairness since two storage streams would consume significantly different amounts of resources in an SSD even if they achieve the same bandwidth.

IO cost is hard to measure because SSDs do not disclose detailed execution statistics of the NAND and its data channels. IO costs can also be biased by operation granularity. For instance, a large 128KB IO might be decomposed into individual 4KB requests internally and deemed complete only when all individual requests have been processed. In contrast, if we were to pipeline a sequence of 32 × 4KB operations, issuing a new one after each completion, the SSD internal queue occupancy would increase even though the observable outstanding bytes is the same with 128KB IO. We, therefore, use the notion of a *virtual slot*, which is a group of IOs

Algorithm 1 Congestion Control with Rate Pacing

```
1:  procedure SUBMISSION()
2:      update_token_buckets()
3:      req = DRR.dequeue()
4:      bucket ← dual_token_bucket[req.io_type]
5:      if bucket.tokens ≥ req.size then
6:          io_outstanding += 1
7:          submit_to_ssd(req)
8:      return
1:  procedure COMPLETION()
2:      state = update_latency(cpl.io_type, cpl.latency)
3:      if state == overloaded then
4:          target_rate = completion_rate
5:          discard_remain_tokens(dual_token_bucket)
6:      if state == congested or overloaded then
7:          target_rate −= cpl.size
8:      else if state == congestion_avoidance then
9:          target_rate += cpl.size
10:     else
11:         target_rate += β× cpl.size
12:     return
1:  procedure UPDATE_LATENCY(IO_TYPE, LATENCY)
2:      lat_mon ← latency monitor for io_type
3:      lat_mon.ewma = (1-α_D) lat_mon.ewma_lat + α_D× latency
4:      if lat_mon.ewma > thresh_max then
5:          lat_mon.thresh = thresh_max
6:          state = overloaded
7:      else if lat_mon.ewma > lat_mon.threshold then
8:          lat_mon.thresh = (lat_mon.thresh + thresh_max)/2
9:          state = congested
10:     else if lat_mon.ewma > thresh_min then
11:         lat_mon.thresh −= α_T× (lat_mon.thresh - lat_mon.ewma)
12:         state = congestion avoidance
13:     else
14:         lat_mon.thresh −= α_T× (lat_mon.thresh - lat_mon.ewma)
15:         state = underutilized
16:     return state
```

up to 128KB in total (e.g., it might contain up to 1 × 128KB or 32 × 4KB IO commands) and manage IO completion in the granularity of virtual slots. A virtual slot completes when all operations in the slot complete. Each tenant always has the same number of virtual slots. If a tenant runs out of its virtual slots, the IO scheduler defers following IOs until one of its virtual slots completes and becomes available. Gimbal maintains the number of virtual slots per tenant at a minimum and adapts the IO cost variance according to sizes.

The virtual slot mechanism provides an upper bound on the submission rate and guarantees that any sized IO pattern obtains a fair portion of the SSD internal resource. It also addresses the *deceptive idleness* issue [44] (found in many work-conserving fair queuing schedulers) because an allocated slot cannot be stolen by other streams. Gimbal sets the threshold for the number of virtual slots in a single tenant to the minimum number to reach the device's maximum bandwidth if there is only one active tenant. Virtual slots are equally distributed when more active tenants contend for the storage. Since each tenant should have at least one virtual slot to perform IOs, the total number of virtual slots may exceed the threshold under high consolidation.

Gimbal integrates the virtual slot concept into the DRR scheduler and ensures the number of slots for each tenant is the same. Similar to the fair queueing mechanism [33, 37], the DRR scheduler divides all tenants that have requests into two lists: *active*, where each one in the list has assigned virtual slots; *deferred*, where its tenants have no available virtual slots and wait for outstanding requests to be completed. Also, the scheduler sets the deficit count to zero when a tenant moves to the deferred list and does not increase the deficit counter of the tenant. Once the tenant receives a new virtual slot, it moves to the end of the active list, and the scheduler resumes increasing the deficit count. Gimbal uses a cost-weighted size for a write IO instead of the actual size ($write\ cost \times IO\ size$) to capture the write cost in the virtual slot. An IO can only be submitted if its deficit count is larger than the weighted size. For example, if the tenant has a 128KB write request when the system is operating under a write cost of 3, the tenant would be allowed to issue the operation only after three round-robin rounds of satisfying tenants in the active list (with each round updating the deficit count associated with the tenant).

Per-tenant priority queues. The ingress pipeline of Gimbal maintains priority queues for each tenant. The priority is tagged by clients and carried over NVMe-oF requests. When a tenant is being scheduled within an available virtual slot, the scheduler cycles over these priority queues in a round-robin manner, uses each queue's weights to selects IO requests from the queue, and constructs the IO request bundle. This mechanism allows clients to prioritize a latency-sensitive request over a throughput-oriented request.

3.6 End-to-End Credit-based Flow Control

The switch applies an end-to-end credit-based flow control between the client-side and the target-side per-tenant queue. This controls the number of outstanding IOs to a remote NVMe SSD and avoids queue buildup at the switch ingress. Unlike the networking setting, where a credit means a fixed-size data segment or a flit [23, 30], the number of credits in our case represents the amount of IO regardless of the size that a device could serve without hurting QoS. It is obtained from the congestion control module (described above). The total credit for the tenant is the number of allotted virtual slots times the IO count of the latest completed slot. A tenant submits IO if the total credit is larger than the amount of outstanding IO.

Instead of using a separate communication channel for credit exchange, we piggyback the allocated credits into the NVMe-oF completion response (i.e., the first reservation field). Similar to the traditional credit-based flow control used for networking, our credit exchange/update scheme (like N23 [51, 52]) also minimizes the credit exchange frequency and avoids credit overflow. However, it differs in that our protocol (1) works in an end-to-end fashion, not hop-by-hop; (2) targets at maximizing remote SSD usage with a QoS guarantee. Algorithm 3 (in Appendix) describes the details.

3.7 Per-SSD Virtual View

Our switch provides a managed view of its SSDs to each tenant that indicates how much read/write bandwidth headroom is available at the target so that clients can use the SSD resource efficiently in a multi-tenant environment. In addition, it also supports IO priority tagging, which allows applications to prioritize storage operations based on workload requirements. Such a view enables applications to develop flexible mechanisms/policies, like application-specific IO scheduler, rate limiter, IO load balancer, etc. We later discuss how we integrate it into a log-structured merge-tree key-value store.

4 Implementation

4.1 Switch Pipeline

We built Gimbal using Intel SPDK [16]. It targets the RDMA transport and extends the basic NVMe-over-Fabric target application by adding three major components: DRR IO scheduler, write cost estimator, and SSD congestion control. The overall implementations follow the shared-nothing architecture and rely on the reactor framework of SPDK. On the Broadcom Stingray platform, we find that one SmartNIC CPU (ARM A72) core is able to fully drive PCIe Gen3 × 4-lanes SSD for any network and storage traffic profile. Therefore, Gimbal uses one dedicated CPU core to handle a specific SSD. Each switch pipeline is dedicated to a specific SSD and shares nothing with other pipelines that handle different SSDs. It runs as a reactor (i.e., kernel-bypass executor) on a dedicated SmartNIC core asynchronously and will trigger the ingress handler if incoming events come from RDMA qpair (or egress handler when an event is from the NVMe qpair). A reactor uses one or more pollers to listen to incoming requests from the network/storage. Our implementations are compatible with the existing NVMe-oF protocol.

4.2 Parameters of Gimbal

Gimbal is affected by the following parameters:

- **Max/Min Delay threshold**. $Thresh_{min}$ is an upper bound of "congestion-free" latency. It should be larger than the highest latency when there is only one outstanding IO, which is 230us on our SSD. We set $Thresh_{min}$ to 250us. $Thresh_{max}$ should ensure that SSDs achieve high utilization for all cases and provide sufficient flexibility to tune other parameters. We characterized this parameter by configuring different types of storage profiles and found that when saturating the bandwidth, the lowest latency is between 500us and 1000us, depending on the workload. Further, to minimize the frequency that Gimbal enters the overloaded state (Section 3.3), we set $Thresh_{max}$ to a slightly higher value (i.e., 1500us). Since the minimum latency of the SSD is highly correlated with the NAND characteristics, the parameter values we determine generally apply to other TLC-based SSDs as well. However, 3DXP, SLC, or QLC have completely different device characteristics, and we need to adjust these thresholds appropriately for such devices.
- α_T, α_D **and** β. α_T decides how frequently the congestion signal is generated. Although the higher value (e.g., 2^{-8}) has a negligible impact on the result, we set α_T to 2^{-1} and speculatively generate a signal to minimize the trigger rate of the "overloaded" state. α_D is used to tolerate occasional latency spikes and capture recent trends. We set it to 2^{-1}. β determines how fast the target rate is increased in the underutilized state. We set the value to 8. Gimbal can increase the target rate to a peak value within a second (Section 3.3).
- **Number of virtual slots and the size:** Gimbal uses 128KB as the slot size as it is the de facto maximum IO size of the NVMe-oF implementation. If the system supports a larger size, the value may be increased accordingly. However, such a larger virtual slot degrades fairness. The threshold of the number of virtual

slots for a single tenant is highly related to the outstanding bytes required for the maximum sequential read bandwidth. We found that 8 × 128KB sequential read I/Os is the minimum to reach 3.3GB/s bandwidth. Gimbal uses the value 8 for the threshold.

- **Write cost and decrement factor δ:** $Write\ cost_{worst}$ describes the maximum cost of a write IO on the SSD. It can be measured via a microbenchmark or simply calculated by using the maximum random read and write IOPS from the datasheet. We choose the latter way and set the parameter to 9 for the Samsung DCT983 NVMe SSD. The additive decrements factor decides how fast Gimbal reacts to observed write latencies. It should reduce the write cost only when writes are served fast from the SSD (not intermediate low write latency). We set δ to 0.5 empirically.

4.3 Case Study: RocksDB over Gimbal

This section describes how we support a log-structured merge-tree (LSM) key-value store (i.e., RocksDB) in a multi-tenant environment. We show how to optimize its various components using the per-SSD virtual view exposed by the storage switch.

Overview. RocksDB [20] is based on an LSM-tree data structure (Appendix E). We run the RocksDB over a blobstore file system in an NVMe-oF aware environment. Our modifications include a hierarchical blob allocator over a pool of NVMe-oF backends, an IO rate limiter to control outstanding read/write IOs, and a load balancer that steers read requests based on the runtime loading factor of a storage node.

Hierarchical blob allocator. We allocate storage blobs across a pool of remote storage nodes. Upon an allocation request, it chooses a blob from an available NVMe SSD and then updates blob address mappings. The blob address in our case includes *<NVMe transport identifier, start LBA number, LBA amount, LBA sector size>*. A free operation then releases allocated sectors and puts them back into the pool. Such an allocator should (1) fully use the storage capacity and provide the appropriate granularity to reduce the storage waste; (2) use a small amount of metadata for block management.

We apply a hierarchical blob allocator (HBA) to satisfy these requirements. First, there is a global blob allocator at the rack-scale that divides total storage into *mega blobs* (i.e., a large chunk of contiguous storage blocks, 4GB in our current implementation), and uses a bitmap mechanism to maintain availability. Within the RocksDB remote environment layer, there is a local agent performing allocation in the granularity of *micro blobs* (i.e., a small chunk of contiguous storage blocks, 256KB in our case). It also maintains a free micro blob list based on allocated mega blobs. A file blob allocation request is served by the local allocator first and will trigger the global allocator when the local free pool becomes empty. To minimize the IO interference impact, we employ a load-aware policy to choose a free mega/micro blob: selecting the one with the maximum credit (i.e., the least load) of the NVMe SSD.

IO rate limiter. Since our RocksDB is built on top of the SSD virtual view, which applies the credit-based flow control with the NVMe-oF target, it automatically supports an IO rate limiter. Specifically, a read/write request is issued when there are enough credits; otherwise, it is queued locally. Under an NVMe-oF completion response, if the credit amount is updated in the virtual view, the database will submit more requests from the queue.

Replication and load balancing We also built a replication mechanism to tolerate flash failures [61, 69] and a load balancer to improve remote read performance. Each file has a primary and a shadow copy that is spread across different remote backends. When there is an allocation request (during file creation/resize), we will reserve primary and secondary microblobs from two different backends in the HBA. If any of their local pools run out of microblobs, a megablob allocation request is triggered. A write operation will result in two NVMe-oF writes and is completed only when the two writes finish. In terms of read, since there are two data copies, RocksDB will issue a read request to the copy whose remote SSD has the least load. We simply rely on the number of allocated credits to decide the loading status on the target. As described before, since credit is normalized in our case, the one with more credits is able to absorb more read/write requests.

5 Evaluation

5.1 Experiment Methodology

Testbed setup. Our testbed comprises a local rack-scale RDMA-capable cluster with x86 servers and Stingray PS1100R storage nodes, connected to a 100Gbps Arista 716032-CQ switch. Each server has two Intel Xeon processors, 64/96GB memory, and a 100Gbps dual-port Mellanox ConnectX-5 NIC via PCIe 3.0 × 16-lanes. Section 2.2 describes the Stingray setup, and we configure it to use 4 × NVMe SSDs. We use Samsung DCT983 960GB NVMe SSDs for all experiments unless otherwise specified. We run CentOS 7.4 on Intel machines and Ubuntu 16.04 on Stingray nodes, where their SPDK version is 19.07 and 19.10, respectively.

Comparison Schemes. We compare Gimbal against the following multi-tenancy solutions designed for shared storage. Since none of these systems target NVMe-oF, we ported these systems onto SmartNIC JBOFs and tailored them to our settings.

- **ReFlex** [49], enables fast remote Flash access over Ethernet. It applies three techniques: kernel-bypass dataplane executor for IO request processing, SLO-based request management, and DRR-like QoS-aware scheduler. We implemented their scheduler model within the SPDK NVMe-oF target process and used the proposed curve-fitting method to calibrate the SSD cost model.
- **Parda** [38], enforces proportional fair share of remote storage among distributed hosts. Each client regulates the IO submission based on the observed average IO latency. We emulated Parda at the NVMe-oF client-side and applied a similar flow control. To obtain the IO RTT, we encode a timestamp into the NVMe-oF submission command and piggy it back upon completion.
- **FlashFQ** [70], is a fair queueing IO scheduler. It applies the start-time fair queueing mechanism [37] and combines two techniques to mitigate IO interference and deceptive idleness: throttled dispatching and dynamically adjusting the virtual start time based on IO anticipation. It uses a linear model to calculate the virtual finish time. We implemented these techniques and calibrated the parameters (including the dispatching threshold and model) based on our SSD.

SSD and Workloads. We emulate two SSD conditions: *Clean-SSD*, pre-conditioned with 128KB sequential writes; *Fragment-SSD*, pre-conditioned with 4KB random writes for multiple hours. They present different "bathtub" characteristics (Figure 14). We use a

variety of synthetic FIO [9] benchmarks to evaluate different aspects of Gimbal and compare with the other three approaches. We set the maximum outstanding IOs (i.e., queue depth or QD) 4 and 32 to 128KB and 4KB workloads, respectively. All read workloads in the microbenchmark are random, while 128KB write is sequential and 4KB write is random. We re-condition the SSD with the same pattern before each test. We use YCSB [32] for the RocksDB evaluation.

Evaluation metric. When multiple applications run simultaneously over an SSD, the bandwidth allocated to each application depends on its storage profile and may differ from each other significantly. To examine fairness quantitatively, we propose the term *fair utilization* or f*-Util* as a worker-specific normalized utilization ratio. As shown below, it is calculated by dividing the per-worker achieved bandwidth over its standalone maximum bandwidth when it runs exclusively on the SSD, and the ideal ratio is 1.

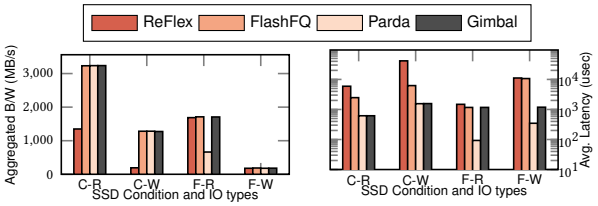
$$f\text{-}Util(i) = \frac{per\_worker\_bw(i)}{standalone\_max\_bw(i)/total\_\#\_of\_workers}$$

5.2 Utilization

We run 16 workers of the same workload (i.e., standalone benchmark) to evaluate the utilization of each scheme. Figure 6 describes the standalone bandwidth of workloads and their average latency. Gimbal performs similarly to the FlashFQ on both Clean-SSD and Fragment-SSD but outperforms ReFlex on Clean-SSD by x2.4 and x6.6 for read and write, respectively. It also outperforms Parda on Fragment-SSD by x2.6 in the read utilization. This is mainly because our SSD congestion control mechanism can estimate the accurate bandwidth headroom so that the scheduler submits just the right amount of IOs. The offline-profiled cost model (used by ReFlex) overestimates the IO cost for writes and large IOs, resulting in lower throughput. Parda fails to reach the maximum bandwidth in the 4KB read workload on Fragment-SSD because the end-to-end RTT between clients and NVMe-oF targets in Parda is too large to detect both the throughput capacity and congestion. In terms of latency, FlashFQ is a work-conserving SFQ scheduler and will issue much more IOs to the SSD without considering the request QoS. Gimbal keeps the latency low by employing the credit-based flow control and performs similar to the Parda for Clean-SSD. The write latency of Gimbal on Fragment-SSD is higher than Parda by x3.4, but it outperforms other schemes by x9 on average. Note that Gimbal does not improve the read latency for Fragment-SSDs because the maximum number of outstanding IOs in the workload is the same as the total credit count for the tenant.

5.3 Fairness

Different IO Sizes. On our testbed, the maximum 128KB random read performance on Clean-SSD (3.16GB/s) is 89% higher than the 4KB read performance (1.67GB/s). We run the benchmark with 16-workers of 4KB read and 4-workers of 128KB read. Figure 7a and 7d present the bandwidth of one worker for each IO size and the f*-Util* for each scheme. We define the utilization deviation as $\frac{|actual\_Util - ideal\_Util|}{ideal\_Util}$ to identify how each tenant achieves its fair bandwidth share. Gimbal shows the closest bandwidth to the ideal value and the utilization deviation of the 128KB IO case is x2.1, x8.7, and x6.4 less than ReFlex, FlashFQ, and Parda, respectively (x1.9, x4.1 and x7.1 for 4KB). Unlike other schemes, Gimbal considers the cost difference in the virtual slot mechanism so that it is able



(a) Bandwidth **(b) Avg. Latency (Log-scale)**

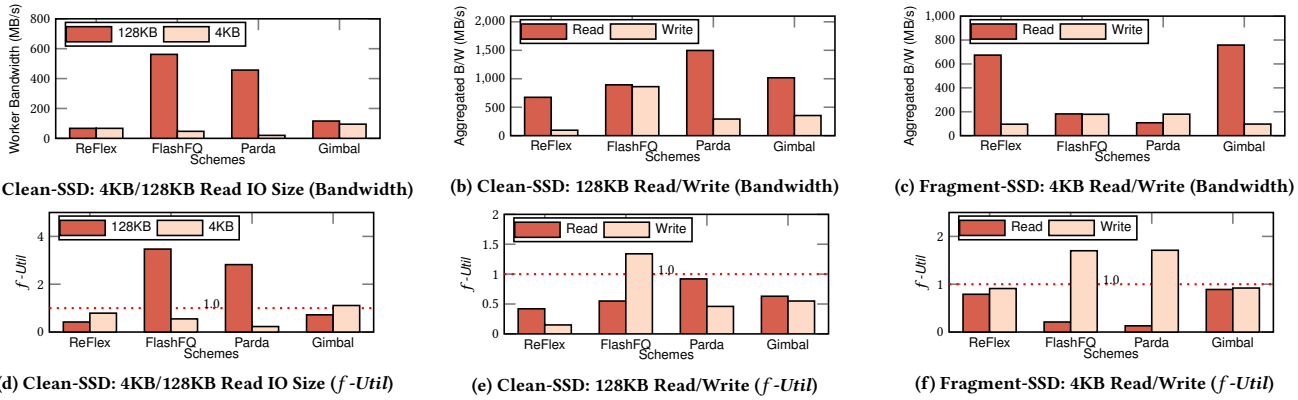
Figure 6: Device utilization on different schemes. IO size is 128KB and 4KB for Clean-SSD and Fragment-SSD, respectively. (C:Clean-SSD, F:Fragment-SSD, R:Read, W:Write)

to allocate 22.0% more bandwidth for 128KB IO in the experiment. The IO cost in ReFlex is proportional to the request size and shows the same bandwidth for both cases.

Different IO Types. In this experiment, we run 16 workers for each read and write and compare the f*-Util* of each scheme. The Clean-SSD case executes 128KB sequential read and 128KB random write, while the Fragment-SSD one contains 4KB random read and 4KB random write. Figure 7b, 7c, 7e and 7f describe the aggregated read/write bandwidth and f*-Util*. Gimbal only shows a difference of 13.8% for Clean-SSD and 3.8% for Fragment-SSD between read and write f*-Util*, and outperforms ReFlex, FlashFQ, and Parda by x12.8, x10.4 and x7.5 on Clean-SSD (x4.2, x184.2 and x330.2 on Fragment-SSD), respectively. Gimbal improves the fair bandwidth allocation for read/write cases because it addresses the cost of different IO types and the SSD conditions with the virtual slot and the write cost mechanism. Note that Gimbal shows a lower utilization on Clean-SSDs because the congestion control prevents the latency from growing beyond the maximum threshold. Parda fails to allot the read bandwidth on Fragment-SSD since the write latency for small IO size is not correlated to the IO cost (possibly lower than the same-sized read latency). The linear model of FlashFQ does not provide fairness in modern SSDs, and the read and write bandwidths are the same on both Clean-SSD and Fragment-SSD. ReFlex has a fixed pre-calibrated model, irrespective of SSD conditions, and limits the write bandwidth substantially on Clean-SSD. As a result, it only works on Fragment-SSD; it would require re-calibration of the model in an online manner to adapt to SSD conditions.

5.4 Latency

Figures 8 presents the average and tail latency of different IO types from the previous experiment (i.e., read and write mixed workload). We report the end-to-end average, 99th and 99.9th latency including in-network queueing delay along the data path as well as the request execution time. Gimbal not only maximizes the SSD usage but also provides the best service guarantee compared with Parda. The benefits mainly come from the fact Gimbal can (1) balance the number of outstanding reads and writes to mitigate the IO interference at the device; (2) use credits to control the request rate. On average, compare with Parda, Gimbal reduces the 99th read and write latency by 48.6% and 57.1% for a Clean-SSD (57.5% and 62.6% for a Fragment-SSD). Parda shows an average latency lower than Gimbal on Fragment-SSD, but it suffers from poor utilization and imbalanced resource allocation. The IO RTT observed by the client alone is not sufficient to avoid the congestion on the shared SSD, and Gimbal outperforms Parda in the 99th and 99.9th latency across



(a) Clean-SSD: 4KB/128KB Read IO Size (Bandwidth)

(b) Clean-SSD: 128KB Read/Write (Bandwidth)

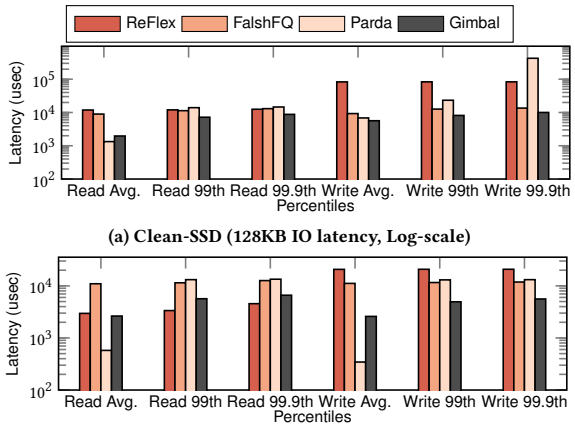
(c) Fragment-SSD: 4KB Read/Write (Bandwidth)

(d) Clean-SSD: 4KB/128KB Read IO Size (f-*Util*)

(e) Clean-SSD: 128KB Read/Write (f-*Util*)

(f) Fragment-SSD: 4KB Read/Write (f-*Util*)

Figure 7: Fairness in various mixed workloads on different schemes and SSD conditions



(a) Clean-SSD (128KB IO latency, Log-scale)

(b) Fragment-SSD (4KB IO latency, Log-scale)

Figure 8: Read/Write IO latency comparison. 16 workers each for Read and Write.

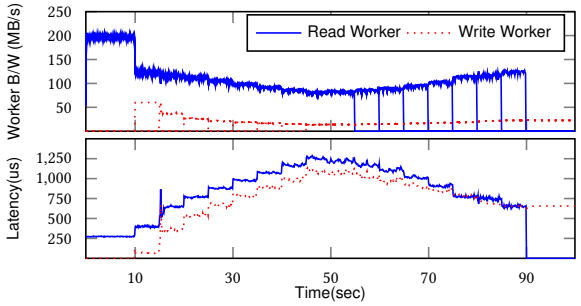


Figure 9: Performance over time as the number of workers changes (the latency is a raw device latency measured directly in Gimbal and averaged out every 100ms)

all workloads. FlashFQ and ReFlex have no flow control mechanisms and incur high latencies under high worker consolidation or a large number of outstanding IOs.

5.5 Dynamic Workloads

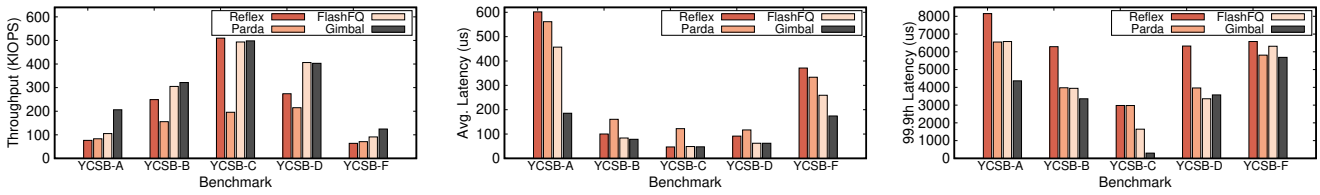
This experiment changes the workload dynamically and demonstrates the importance of estimating the dynamic write cost. We initiate 8 read workers at the beginning of the experiment and add a write worker at a 5-second interval until the number of read and write workers becomes the same (i.e., 8 read and 8 write workers run simultaneously at the maximum congestion). We then drop one read worker at a time at the same interval. Additionally, we limit the maximum IO rate of the single worker to 200MB/s and 60MB/s for read and write, respectively. This simulates an application with a specific IO rate.

Figure 9 shows the bandwidth over time for each worker and the latency for each IO type. Gimbal shows that it adapts to the workload characteristics and can accelerate write IOs accordingly. The first write worker benefits from the internal write buffer so that most IOs complete immediately. The latency thus is about 70 usec (i.e., less than the minimum latency threshold) on average, while the average read latency is about 1000 usec at the same moment. As discussed in Section 3.4, Gimbal reduces the write cost to 1 and benefits from the SSD device optimization for writes in this case. After the second writer arrives, the write rate starts to exceed the capability of the internal buffer and the latency grows more than 10 times. Gimbal detects such a latency change at runtime and increases the write cost. As a result, the bandwidth for write workers converges to the fair bandwidth share.

5.6 Application Performance

In this experiment, we run 24 RocksDB instances over three SmartNIC JBOFs that are configured with different mechanisms on fragmented SSDs. As described before, we enhanced RocksDB with a rate limiter and a read load balancer. We report the aggregated throughput and average/99.9th latency (Figure 10). On average, across five benchmarks, Gimbal outperforms ReFlex, Parda, and FlashFQ by x1.7, x2.1, x1.3 in terms of throughput, and reduces the average/99.9th latency by 34.9%/48.0%, 54.7%/30.2%, 20.1%/26.8%, respectively. Among them, YCSB-A and YCSB-F (i.e., update-heavy and read-modify-write) workloads benefit the most while YCSB-C (i.e., read-only) observes the least performance improvements. This is because Gimbal can schedule a mix of read/write IOs more efficiently to maximize the SSD performance.

Scalability. We used the same RocksDB configuration and scaled the number of RocksDB instances over three SmartNIC JBOFs (as above). Figures 11 and 12 present throughput and average read latency, respectively. YCSB-A, YCSB-B, and YCSB-D max out their performance with 20 instances, while YCSB-F achieves the max throughput under 16 instances. For example, the average read latency of YCSB-F with 24 DB instances increases by 38.1% compared with the 16 instance case. YCSB-C is a read-only workload, where 24



(a) Throughput. (b) Average read latency. (c) 99.9th read latency.

Figure 10: RocksDB performance comparison over four approaches. We configure the YCSB to generate 10M 1KB key-value pairs with a Zipfian distribution of skewness 0.99 for each DB instance.

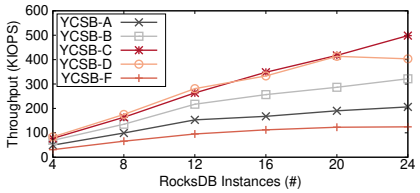


Figure 11: Throughput as increasing the number of RocksDB instances.

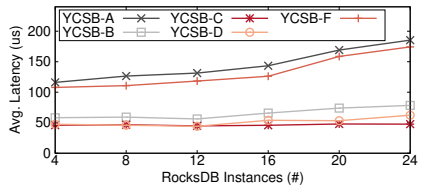


Figure 12: Average read latency as increasing the number of RocksDB instances.

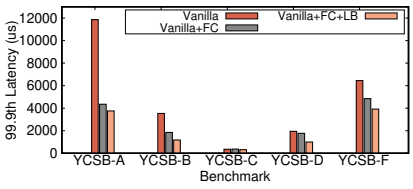


Figure 13: Performance improvement via the SSD virtual view enabled optimizations.

instances still cannot saturate the NVMe-oF target read bandwidth, and the average read latency varies little.

Virtual view enabled optimizations. This experiment examines how the virtual view provided by Gimbal could help improve the application performance. We followed the above RocksDB setting and ran 8 DB instances (from two client servers) over one SmartNIC JBOF. Figure 13 presents the read tail latency of five benchmarks comparing three cases (i.e., vanilla w/o optimizations, w/ flow control, w/ flow control and load balancing). On average, across the five workloads, the IO rate limiter enabled via our credit scheme reduces the 99.9th latencies by 28.2% compared with the vanilla one. The request load balancer, which could choose one replica that has more bandwidth, further reduces the tail by 18.8%.

5.7 Overheads

We evaluate the overhead of Gimbal in two ways. First, we compare the average CPU cycles of the submission and completion procedure with a vanilla SPDK NVMe-oF target on SmartNIC. As shown in Table 1a, Gimbal adds 37.5-62.5% more cycles to realize the storage switch. Although Gimbal adds some overhead on average in the pipeline, it does not hurt the performance for PCIe Gen3 SSD as discussed in Section 2.4. Future PCIe Gen4 SSDs could achieve ~7GB/s bandwidth or 1 MIOPS. We run the 4KB read benchmark with a NULL device (which does not perform actual IO and returns immediately) to measure the maximum IOPS of Gimbal on SmartNIC. Gimbal performs 821 KIOPS with 1 SmartNIC core. Gimbal could support a multi-core implementation if one can distribute the active tenants to each core in a balanced manner. For example, we extend the experiment to the 4-core case and find that Gimbal achieves 2446 KIOPS, indicating that SmartNIC-based Gimbal can support next-generation SSDs. We also expect Gimbal to scale up with future powerful ARM cores or specialized engines.

5.8 Generalization

This experiment evaluates how Gimbal performs on a different type of SSD. We run the same microbenchmark (Section 5.3) using Intel DC P3600 1.2TB model. This SSD uses 2-bit MLC NAND, presenting

		Vanilla SPDK	Gimbal
1 workers (QD1)	Submit	32	52 (+62.5%)
	Complete	16	22 (+37.5%)
16 workers (QD32)	Submit	21	30 (+42.9%)
	Complete	17	25 (+47.1%)

(a) CPU cycle comparison (4KB Read, QD=Queue Depth, 125cycles=1usec)

	Vanilla SPDK	Gimbal
1 CPU core, 1 worker	937 KIOPS	821 KIOPS (-12.4%)
4 CPU cores, 8 workers	2692 KIOPS	2446 KIOPS (-9.2%)

(b) The maximum IOPS with NULL device (4KB Read IO)

Table 1: Overhead comparison with vanilla SPDK

33.5% lower 128KB read (2.1GB/s) and 35.0% higher 4KB random write (243MB/s) in terms of bandwidth. We tune the $Thresh_{max}$ to 3ms for better read utilization as it achieves higher tail latency than DCT983 for 128KB read. Gimbal adapts the characteristics of the SSD and performs similarly to the DCT983 case in terms of the f-*Util*. Specifically, it shows 0.63 and 0.72 of f-*Util* for read and write under the clean condition and 0.58 and 0.90 for read and write under the fragmented condition.

We also run Gimbal on Xeon E5-2620 v4 CPU and compare the overhead with the vanilla SPDK. Gimbal performs 10.8% lower (1368 KIOPS) than the vanilla SPDK (1533 KIOPS) for 4KB read performance with the NULL device, similar to the result on SmartNIC.

5.9 Summary

Table 2 summarizes the high-level differences between Gimbal and the other approaches. Gimbal outperforms the other approaches because it dynamically estimates the available bandwidth and IO costs for each storage device based on current conditions and workloads, performs fine-grained fair queueing at the NVMe-oF target across tenants, and uses credit-based flow control to adjust the client behavior. The other approaches lack one or more such support. For example, ReFlex and FlashFQ use an approximate offline-profiled SSD cost model that hurts scheduling efficiency. They also don't

	ReFlex	Parda	FlashFQ	Gimbal
BW estimation	Static	Dynamic	✗	Dynamic
IO cost & WR tax	Static	✗	Static	Dynamic
Fair queueing	@Target	@Client	@Target	@Target
Flow control	✗	✓	✗	✓

Table 2: Comparison of four multi-tenancy mechanisms.

regulate client-side IOs, causing significant delays. Parda employs a client-side mechanism, which uses the end-to-end delay as the congestion signal for its flow control, and performs limited fair queueing within a server without coordination with other hosts. Such a high latency feedback control loop is unsuitable for low-latency high-bandwidth NVMe SSDs.

6 Related Work and Discussion

Shared storage with QoS guarantee. Prior work has explored how to achieve high device utilization and fairness for local and remote shared storage [38, 42, 49, 60, 64, 65, 70, 73, 76, 77]. As discussed in Section 5, these approaches cannot precisely estimate the IO capacity (under different SSD conditions) and per-IO cost (under various mixed workloads), causing inefficiencies when applied to the NVMe-oF based disaggregated setting. There are also some timeslice-based IO schedulers (e.g., CFQ [25], Argon [77]) that allocate time quanta for fairness and provide tenant-level exclusive access. They usually target millisecond-scale slow storage and would hurt responsiveness, utilization, and fairness when used with fast NVMe SSDs. Recently, researchers have tried to leverage ML techniques (e.g., a light neural network) to predict the per-IO performance [41]. The predicted result is a binary model (indicating if an IO is fast or slow) and can only help improve the admission control of storage applications for predictable performance, which is insufficient to achieve fairness. IOFlow [76] proposes a software-defined architecture for enforcing end-to-end policies of storage IOs running in the data center. It adds queueing abstractions along the data plane and relies on a centralized controller for translating policies to queuing rules. We believe Gimbal can serve as one of its execution stages at the storage server.

Remote storage IO stacks. Researchers [48] have characterized the performance of iSCSI-based disaggregated storage and proposed optimization opportunities, such as parallel TCP protocol processing, NIC TSO offloading, jumbo frames, and interrupt affinity pinning. ReFlex [49] provides a kernel-bypass data-plane to remove the NVMe storage processing stack, along with a credit-based QoS scheduler. Also, i10 [43] is an efficient in-kernel TCP/IP remote storage stack using dedicated end-to-end IO paths and delayed doorbell notifications. Researchers [39] have also characterized the server-based NVMe-oF performance. Our work targets SmartNIC JBOFs that use NVMe-oF protocols.

Disaggregated storage architecture. New hardware designs have been proposed to address the limitations of existing disaggregation designs. Sergey et al. [53] proposes four kinds of in-rack storage disaggregation (i.e., complete, dynamic elastic, failure, and configuration disaggregation) and explores their tradeoffs. Shoal [72] is a power-efficient and performant network fabric design for disaggregated racks built using fast circuit switches. LeapIO [55] provides a uniform address space across the host Intel x86 CPU and ARM-based co-processors in the runtime layer and exposes the virtual NVMe driver to an unmodified guest VM. As a result, ARM cores can run a complex cloud storage stack. We focus on emerging SmartNIC-based disaggregated storage solutions.

Programmable packet scheduling. Researchers have explored new HW/SW programmable packet schedulers. PIFO [74] proposes a push-in first-out queue that enables packets to be pushed into an arbitrary position based on their calculated rank and dequeued from the head. PIEO [71] then improves PIFO scalability and expressiveness via two capabilities: providing each scheduling element a predicate; dequeuing the packet with the smallest index from a subset of the entire queue. SP-PIFO [22] further approximates the PIFO design via coarse-grained priority levels and strict FIFO queues. It dynamically adjusts the priority range of individual queues by modifying the queuing threshold. PANIC [56] combines PIFO with a load balancer and a priority-aware packet dropping mechanism to explore efficient multi-tenancy support on SmartNICs. Carousel [66] and Eiffel [67] are two efficient software packet schedulers, which rely on a timing wheel data structure and bucketed integer priority for fast packet operations. Compared with these studies, a key difference is that Gimbal focuses on scheduling storage IO requests among NVMe SSDs instead of individual packets. Exposing programmability from the Gimbal traffic manager to realize flexible scheduling policies will be our future work.

Emerging storage media. QLC NAND has gained significant attention because of its cost and capacity advantage over TLC. However, its performance characteristic is worse than TLC, presenting a higher degree of read and write asymmetry [57]. We expect the techniques introduced by Gimbal could also apply to QLC SSDs. Gimbal could also support the 3DXP device. However, it is suitable for a local cache rather than the disaggregated storage [80]. 3DXP also provides a similar performance of read and write with in-place update [79] and might not benefit from Gimbal as much as NAND devices.

Hardware-accelerated Gimbal. The pipelined architecture of Gimbal has a standard interface for both ingress and egress and can be plugged into any hardware-accelerated NVMe-oF implementation. In addition, Gimbal itself is portable to a hardware logic with ease using a framework such as Tonic [24].

7 Conclusion

This paper presents Gimbal, a software storage switch that enables multi-tenant storage disaggregation on SmartNIC JBOFs. Gimbal applies four techniques: a delay-based congestion control mechanism for SSD bandwidth estimation, a new IO cost measurement scheme, a two-level hierarchical I/O scheduling framework, and an end-to-end credit-based flow control along with an exposed SSD virtual view. We design, implement, and evaluate Gimbal on the Broadcom Stingray PS1100R platforms. Our evaluations show that Gimbal can maximize the SSD usage, ensure fairness among multiple tenants, provide better QoS guarantees, and enable multi-tenancy aware performance optimizations for applications. This work does not raise any ethical issues.

Acknowledgments

This work is supported by Samsung and NSF grants CNS-2028771 and CNS-2006349. We would like to thank the anonymous reviewers and our shepherd, Brent Stephens, for their comments and feedback.

References

[1] 2017. Disaggregated or Hyperconverged, What Storage will Win the Enterprise? https://www.nextplatform.com/2017/12/04/disaggregated-hyperconverged-storage-will-win-enterprise/.

[2] 2018. Free Your Flash And Disaggregate. https://www.lightbitslabs.com/blog/free-your-flash-and-disaggregate/.

[3] 2018. Industry Outlook: NVMe and NVMe-oF For Storage. https://www.iol.unh.edu/news/2018/02/08/industry-outlook-nvme-and-nvme-storage.

[4] 2019. What is Composable Disaggregated Infrastructure? https://blog.westerndigital.com/what-is-composable-disaggregated-infrastructure/.

[5] 2020. Alpha Data ADM-PCIE-9V3 FPGA SmartNIC. https://www.alpha-data.com/dcp/products.php?product=adm-pcie-9v3.

[6] 2020. AWS Nitro System. https://aws.amazon.com/ec2/nitro/.

[7] 2020. Broadcom FlexSPARX acceleration subsystem. https://docs.broadcom.com/doc/1211168571391.

[8] 2020. Broadcom Stingray PS1100R SmartNIC. https://www.broadcom.com/products/storage/ethernet-storage-adapters-ics/ps1100r.

[9] 2020. Flexible I/O Tester. https://github.com/axboe/fio.

[10] 2020. Marvell LiquidIO SmartNICs. https://www.marvell.com/products/ethernet-adapters-and-controllers/liquidio-smart-nics/liquidio-ii-smart-nics.html.

[11] 2020. Mellanox BlueField-2 SmartNIC. https://www.mellanox.com/products/bluefield2-overview.

[12] 2020. Mellanox Innova-2 Flex Open Programmable SmartNIC. https://www.mellanox.com/products/smartnics/innova-2-flex.

[13] 2020. Netronome Agilio SmartNIC. https://www.netronome.com/products/agilio-cx/.

[14] 2020. NVM Express Base Specification. https://nvmexpress.org/developers/nvme-specification/.

[15] 2020. NVM Express over Fabrics Specification. https://nvmexpress.org/developers/nvme-of-specification/.

[16] 2020. The Intel Storage Performance Development Kit (SPDK). https://spdk.io/.

[17] 2020. Watts up? PRO. http://www.idlboise.com/sites/default/files/WattsUp_Pro_ES.pdf.

[18] 2021. Brocade G620 Switch. https://www.broadcom.com/products/fibre-channel-networking/switches/g620-switch.

[19] 2021. Cisco MDS 9000 Series Multilayer Switches. https://www.cisco.com/c/en/us/products/storage-networking/mds-9000-series-multilayer-switches/index.html.

[20] 2021. RocksDB. https://rocksdb.org.

[21] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark S Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance.. In *USENIX Annual Technical Conference*.

[22] Albert Gran Alcoz, Alexander Dietmüller, and Laurent Vanbever. 2020. SP-PIFO: approximating push-in first-out behaviors using strict-priority queues. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*.

[23] Thomas E Anderson, Susan S Owicki, James B Saxe, and Charles P Thacker. 1993. High-speed switch scheduling for local-area networks. *ACM Transactions on Computer Systems (TOCS)* 11, 4 (1993), 319–352.

[24] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling programmable transport protocols in high-speed NICs. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*. 93–109.

[25] Jens Axboe. 2004. Linux block IO—present and future. In *Ottawa Linux Symp.*

[26] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux Block IO: Introducing Multi-Queue SSD Access on Multi-Core Systems. In *Proceedings of the 6th International Systems and Storage Conference*.

[27] Lawrence S. Brakmo and Larry L. Peterson. 1995. TCP Vegas: End to end congestion avoidance on a global Internet. *IEEE Journal on selected Areas in communications* 13, 8 (1995), 1465–1480.

[28] Feng Chen, Binbing Hou, and Rubao Lee. 2016. Internal parallelism of flash memory-based solid-state drives. *ACM Transactions on Storage (TOS)* 12, 3 (2016), 1–39.

[29] Feng Chen, David A Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. *ACM SIGMETRICS Performance Evaluation Review* 37, 1 (2009), 181–192.

[30] Eric Chung, Andreas Nowatzyk, Tom Rodeheffer, Chuck Thacker, and Fang Yu. 2014. An3: A low-cost, circuit-switched datacenter network. *Technical Report MSR-TR-2014-35, Microsoft Research* (2014).

[31] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. 2009. A survey of flash translation layer. *Journal of Systems Architecture* 55, 5-6 (2009), 332–343.

[32] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*.

[33] A. Demers, S. Keshav, and S. Shenker. 1989. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*.

[34] Peter Desnoyers. 2012. Analytic modeling of SSD write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*.

[35] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. 2019. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*.

[36] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*.

[37] Pawan Goyal, Harrick M Vin, and Haichen Chen. 1996. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*.

[38] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. 2009. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *7th USENIX Conference on File and Storage Technologies (FAST 09)*.

[39] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. 2017. NVMe-over-Fabrics Performance Characterization and the Path to Low-Overhead Flash Disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference*.

[40] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review* 42, 5 (2008), 64–74.

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

[42] Mohammad Hedayati, Kai Shen, Michael L Scott, and Mike Marty. 2019. Multi-queue fair queuing. In *2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*. 301–314.

[43] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10 . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*.

[44] Sitaram Iyer and Peter Druschel. 2001. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*.

[45] Myoungsoo Jung and Mahmut Kandemir. 2013. Revisiting widely held SSD expectations and rethinking system-level implications. *ACM SIGMETRICS Performance Evaluation Review* 41, 1 (2013), 203–216.

[46] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. 2014. Durable write cache in flash memory SSD for relational and NoSQL databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 529–540.

[47] Tae Yong Kim, Dong Hyun Kang, Dongwoo Lee, and Young Ik Eom. 2015. Improving performance by bridging the semantic gap between multi-queue SSD and I/O virtualization framework. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*.

[48] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*.

[49] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*.

[50] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*.

[51] HT Kung and Alan Chapman. 1993. The FCVC (flow-controlled virtual channels) proposal for ATM networks: A summary. In *1993 International Conference on Network Protocols*. IEEE, 116–127.

[52] H. T. Kung, Trevor Blackwell, and Alan Chapman. 1994. Credit-Based Flow Control for ATM Networks:Credit Update Protocol, Adaptive Credit Allocation, and Statistical Multiplexing. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications*.

[53] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanaël Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony Rowstron. 2017. Understanding Rack-Scale Disaggregated Storage. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*.

[54] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*.

[55] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. 2020. 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*.

[56] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. 2020. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*.

[57] Chun-Yi Liu, Yunju Lee, Myoungsoo Jung, Mahmut Taylan Kandemir, and Wonil Choi. 2021. Prolonging 3D NAND SSD lifetime via read latency relaxation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 730–742.

[58] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications onto SmartNICs Using IPipe. In *Proceedings of the ACM Special Interest Group on Data Communication*.

[59] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*.

[60] Hui Lu, Brendan Saltaformaggio, Ramana Kompella, and Dongyan Xu. 2015. vFair: latency-aware fair storage scheduling via per-IO cost-based differentiation. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*.

[61] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. 2015. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*.

[62] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-Based Congestion Control for the Datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*.

[63] Jeonghoon Mo, Richard J La, Venkat Anantharam, and Jean Walrand. 1999. Analysis and comparison of TCP Reno and Vegas. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, Vol. 3. 1556–1563.

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

[65] Stan Park and Kai Shen. 2012. FIOS: a fair, efficient flash I/O scheduler.. In *FAST*.

[66] Ahmed Saeed, Nandita Dukkipati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. 2017. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*.

[67] Ahmed Saeed, Yimeng Zhao, Nandita Dukkipati, Ellen Zegura, Mostafa Ammar, Khaled Harras, and Amin Vahdat. 2019. Eiffel: Efficient and Flexible Software Packet Scheduling. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[68] Mark R Schibilla and Randy J Reiter. 2012. Garbage collection for solid state disks. US Patent 8,166,233.

[69] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash Reliability in Production: The Expected and the Unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*.

[70] Kai Shen and Stan Park. 2013. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *2013 {USENIX} Annual Technical Conference ({USENIX} {ATC} 13)*. 67–78.

[71] Vishal Shrivastav. 2019. Fast, scalable, and programmable packet scheduler in hardware. In *Proceedings of the ACM Special Interest Group on Data Communication*.

[72] Vishal Shrivastav, Asaf Valadarsky, Hitesh Ballani, Paolo Costa, Ki Suh Lee, Han Wang, Rachit Agarwal, and Hakim Weatherspoon. 2019. Shoal: A Network Architecture for Disaggregated Racks. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*.

[73] David Shue and Michael J Freedman. 2014. From application requests to Virtual IOPs: Provisioned key-value storage with Libra. In *Proceedings of the Ninth European Conference on Computer Systems*.

[74] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. 2016. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*.

[75] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. 2018. MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*.

[76] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*.

[77] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R Ganger. 2007. Argon: Performance Insulation for Shared Storage Servers.. In *FAST*.

[78] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Manu Awasthi, Tameesh Suri, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. 2015. Performance characterization of hyperscale applicationson on nvme ssds. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*.

[79] Jinfeng Yang, Bingzhe Li, and David J Lilja. 2020. Exploring Performance Characteristics of the Optane 3D Xpoint Storage Technology. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 5, 1 (2020), 1–28.

[80] Jinfeng Yang, Bingzhe Li, and David J Lilja. 2021. HeuristicDB: a hybrid storage database system using a non-volatile memory block device. In *Proceedings of the 14th ACM International Conference on Systems and Storage*. 1–12.

[81] Yiying Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2012. De-indirection for flash-based SSDs with nameless writes.. In *FAST*.

Appendices are supporting material that has not been peer-reviewed.

Appendix A Additional SSD characteristics

Figure 14 demonstrates the performance impact of write amplification by comparing two conditions: SSDs pre-conditioned with large sequential (clean) and random IO (fragmented) patterns. Adding 5% writes reduces the overall IOPS by 42.6% in the fragmented case. Compared to the clean case, the fragmented one achieves 16.9% and 17.8% of the throughput of write-only and 90/10 W/R scenarios, respectively.

Figure 15 compares the unloaded latency in a clean SSD v.s. three other scenarios for different IO sizes. On average, across these cases, fragmented SSD, 70/30 read/write mix, and 8 concurrent IOs cause 52.0%, 83.6%, and 80.5% latency increase, respectively. Larger IOs (e.g., 128/256KB) observe more degradation as they are more likely to contend with other requests.

Appendix B Dynamics of Congestion Control

In this section, we provide experimental results that illustrate the following aspects of SSD behavior: (a) delay as a function of SSD load, (b) dynamic latency threshold determined by the congestion control algorithm, and (c) bandwidth capability estimated by the congestion control algorithm.

As shown in Figure 17, the observed latency increases dramatically when the load exceeds the throughput capacity. With the congestion control algorithm, the SSD maintains an average delay in a stable range, providing throughputs close to the device maximum. We plot the typical behavior of the congestion control algorithm as it adapts the delay threshold based on observed latencies. Figure 18 shows the latency threshold according to the EWMA latency. As the number of outstanding IO increases, the EWMA latency begins to exceed the threshold, and hit the threshold more frequently as expected.

Appendix C Additional Details of the Gimbal Switch

Algorithm 2 Gimbal IO scheduler

```
1:  procedure SCHED_SUBMIT
2:      virt_slot = tenant.curr_virt_slot
3:      io.tenant = tenant
4:      io.virt_slot = virt_slot
5:      virt_slot.submits += 1
6:      virt_slot.size += io.weighted_size
7:      if virt_slot.size > 128KB then
8:          virt_slot.is_full = true
9:          close(virt_slot)
10:         if open_virt_slot(tenant) == fail then
11:             move_to_deferred(tenant)
12: procedure SCHED_COMPLETE
13:     tenant = io.tenant
14:     v_slot = io.virt_slot
15:     v_slot.completions += 1
16:     if v_slot.is_full AND v_slot.submits == v_slot.completions then
17:         reset(v_slot)
18:         if tenant.deferred AND open_slot(tenant) == success then
19:             move_to_active(tenant)
```

We now describe some more implementation details of the software storage switch. We describe how to manage virtual slots, how to handle heterogeneity in request bundle sizes caused by the write tax and how the system maintains lists of tenants with requests. We then provide the pseudocode of the scheduling algorithm.

Active and deferred list. Our scheduler calls a function on every IO submission and is also triggered as a callback function when there is an IO completion event. The whole logic is described in Algorithm 2. Generally, the scheduler applies the DRR service discipline for inter-tenant fair storage resource allocation. For each slot in use, the scheduler tracks if its outstanding I/Os are completed and free the slot when the operations are done. When the chosen tenant is in the deferred list then the callback function tries to open new virtual slot. If it succeeds, then the tenant is removed from the deferred list, added to the active list. For each tenant in the active list, the scheduler will accumulate the deficit value, and move to the deferred list when there are no more available virtual slots. Here, the basic quantum for a deficit is 128KB (the maximum IO size). As described before, it also takes the write cost into considerations and uses the weighted IO size.

Credit-based Flow Control Our protocol is integrated with the NVMe-oF submission and completion command processing (Algorithm 3). Upon issuing a request, if there are enough credits, the command is sending to the target side, and the inflight count is increased; otherwise, applications receive a busy device signal, and storage I/Os will be scheduled later. When receiving a NVMe-OF completion request, it first processes the NVMe-oF command, extracts the credit value, updates the latest credit amount, decrease the inflight count, and then resumes storage workloads via callbacks. Note that all credit manipulation operations are atomic.

Algorithm 3 Credit-based flow control inlined with the NVMe-oF request processing

```
1:  procedure NVMEOF_REQ_SUBMIT(req)
2:      target_ssd = req.ssd
3:      if target_ssd.credit_tot > target_ssd.inflight then
4:          ret = submit_nvmeof_req(req)
5:          if ret is success then
6:              target_ssd.inflight = target_ssd.inflight + 1
7:      else
8:          ret = device.busy
        return ret
9:  procedure NVMEOF_REQ_COMPLETE(req)
10:     target_ssd = req.ssd
11:     ret = complete_nvmeof_req(req)
12:     if ret is success then
13:         target_ssd.credit_tot = credit_obtain(req)
14:     target_ssd.inflight = target_ssd.inflight − 1
15:     completion_callback(req)                ▷ Application specific
16:     return ret
```

C.1 Dual Token Bucket for Read and Write

We describe the detail algorithm of the dual token bucket here (Algorithm 4). A burst submission of write I/Os should be avoided since it causes a latency spike even under moderate bandwidth. Gimbal employs a *token bucket* algorithm in the rate pacing module to mitigate the burstiness. A single bucket for both read and write

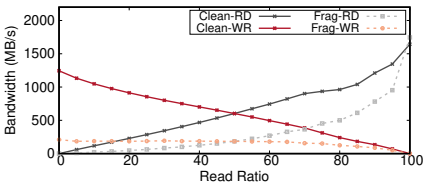


Figure 14: 4KB IO performance as increasing the read ratio in clean and fragmented conditions.

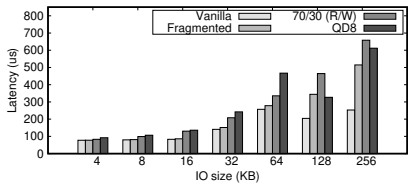


Figure 15: Random read latency varying IO request size under different scenarios. QD=queue depth.

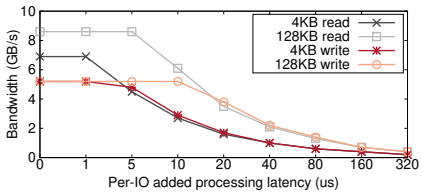


Figure 16: 4KB/128KB read/write bandwidth as increasing the per-IO processing cost on SmartNIC JBOFs.

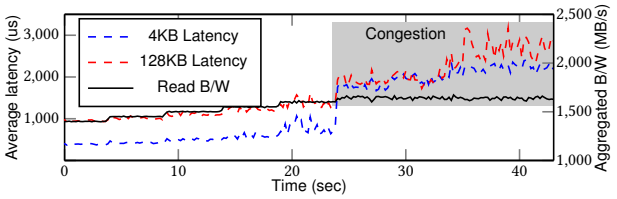


Figure 17: Latency increases over time under the 4KB/128KB read mixed workload. Left and right Y-axis represent latency and aggregated bandwidth, respectively.

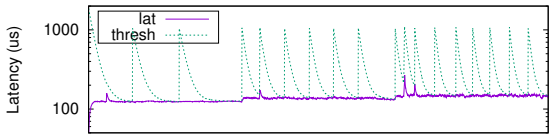


Figure 18: Dynamic Latency Threshold (128KB Random Read)

does not prevent the burst submission because the DRR IO scheduler in Gimbal does not reorder read and write I/Os so that it is possible that only a single kind of IO operations may be dequeued in a series. Gimbal has only one target rate which is the sum of read and write bandwidth and it is significantly higher than a desirable write bandwidth. Therefore, the single bucket approach would submit write I/Os at a wrong rate and cause severe latency increments in this case. Hence, Gimbal employs a *dual token bucket* algorithm. There are two buckets for each read and write. Tokens are generated by the target rate and then distributed to each bucket according to the IO cost. Specifically, out of the total generated tokens, the amount of $\frac{write\_cost}{write\_cost+1}$ is given to the read bucket and the write bucket receives the remainder or the amount of $\frac{1}{write\_cost+1}$. Lastly, we allow overflowed tokens to transfer between each other. Thus, *dual token bucket* offers the flexibility in rate pacing mechanism while avoiding the burst submission of write I/Os for write intensive workloads. Our dual token bucket works globally and applies to all tenants. Algorithm 4 in Appendix describes the token update procedure in detail.

Token bucket size. Gimbal does not reorder I/Os when dequeued from the DRR scheduler. As a result, one bucket needs to wait for the next IO if (1) the dequeued IO is not for the bucket and (2) the other bucket does not have sufficient tokens to submit the IO. A smaller bucket size would cause one bucket drops lots of tokens during the wait time. Since the read bucket typically waits for the write bucket, a small-sized one would hurt read bandwidth. On the other hand, a large bucket size allows the read bucket to accumulate tokens during the wait time, increasing the read bandwidth.

Algorithm 4 Dual Token Bucket

```
1:  procedure UPDATE_TOKEN_BUCKETS()
2:      max_tokens ← 256KB
3:      avail_tokens = target_rate × time_since_last_update
4:      read_tokens += avail_tokens × write_cost / (1+write_cost)
5:      write_tokens += avail_tokens × 1 / (1+write_cost)
6:      if read_tokens > max_tokens then
7:          write_tokens += read_tokens - max_tokens
8:          read_tokens = max_tokens
9:      if write_tokens > max_tokens then
10:         read_tokens += write_tokens - max_tokens
11:         read_tokens = min(read_tokens, max_tokens)
12:         write_tokens = max_tokens
```

We evaluated the range from 128KB to 512KB and set the size to 256KB empirically, which provides a fair bandwidth allocation for read/write under a mixed-IO workload.

Appendix D Characterizing JBOF Multi-tenancy

This section characterizes the multi-tenancy support for today's SmartNIC JBOFs. In the disaggregated storage environment, different IO streams interact with each other along the IO path, and will impact their performance. We categorize interference factors into three types, i.e., IO intensity, IO size, and IO pattern (read v.s. write, random v.s. sequential). We then use controlled experiments to demonstrate how each factor affects a storage stream performance and causes unfair resource sharing. Our experiment setup is described in Section 5.1 and we use the *fio* [9] utility plus the SPDK fio plugin. Note that (1) fio NVMe-oF clients run on different client servers, and read/write to the same NVMe SSD; (2) we use dedicated NVMe-oF target cores to handle different fio streams.

IO intensity: indicates how frequent a storage stream issues IO requests to the remote storage. We configure two competing fio storage streams with the same IO request size and read/write type. Both streams can maximize the SSD bandwidth alone, but they differ in the iodepth (i.e., the number of outstanding IO requests) where stream1 issues twice the number of requests as stream2. Figure 19 reports the results for 4KB random read and 16KB sequential write, respectively. On average across different IO sizes, stream1 achieves 2.0× and 1.8× more bandwidth compared with stream2 for two cases, respectively. This is because the NVMe-oF target receives more requests from stream1, and submits more NVMe commands to SSD device queues. As a result, stream1 obtains more bandwidth share than stream2.

IO size: presents the read/write block size of an NVMe-oF request within a stream. Again, we configure two competing fio storage streams with the same read/write type and iodepth (which

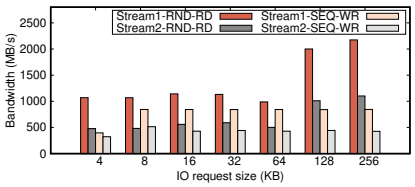


Figure 19: Bandwidth of two competing storage streams with different IO depth varying the IO request size.

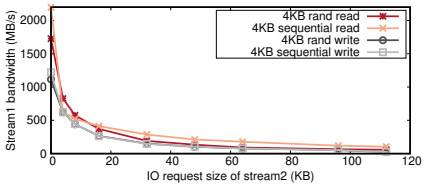


Figure 20: Stream1 bandwidth (4KB random/sequential read/write) varying the IO request size for stream2.

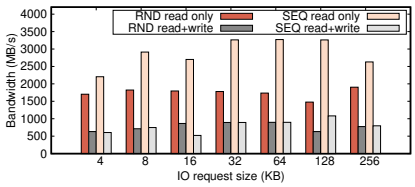


Figure 21: Stream1 bandwidth compared between standalone v.s. mixed cases varying the IO request size.

is large enough to saturate the remote bandwidth). The IO size of stream1 is 4KB, and stream2 gradually increase its size. We report the achieved bandwidth of stream1 under four cases in Figure 20. Even though two storage streams would submit the similar amount of requests into the NVMe drive, apparently, large IO occupies more bandwidth for any of the random/sequential read/write scenarios. For example, in the case of random read, when stream1 and stream2 are both 4KB, each stream takes around 850.0 MB/s. When a 4KB stream1 mixes with 64KB stream2, stream1 only uses 91.0MB/s, while stream2 achieves 1473.0MB/s.

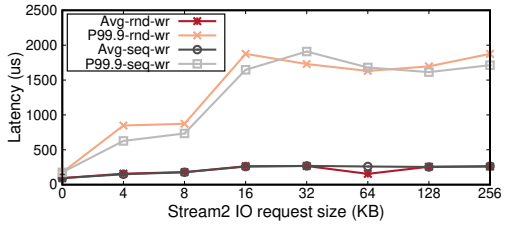


Figure 22: Avg./P99.9 latency of 4KB random read mixed with write traffic varying its IO request size.

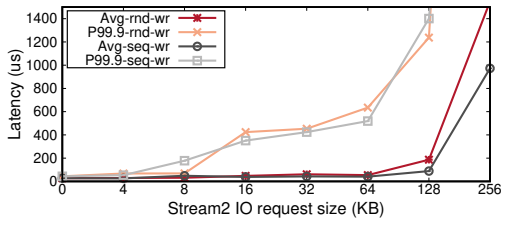


Figure 23: Avg./P99.9 latency of 4KB sequential write mixed with read traffic varying its IO request size.

IO pattern: refers to the type of IO request (e.g., read or write, random or sequential) of a storage stream. Since the NVMe SSD presents distinct execution costs for different typed IOs, when they mix, one would observe significant performance interference in terms of latency and bandwidth. We set up two competing fio streams with the same adequate io depth and io size where stream1/stream2 performs read/write, respectively. As Figure 21 shows, compared with the standalone mode, stream1 only achieves 38.9% and 27.3% bandwidth (on average across different IO sizes) for random and sequential cases when mixed IO happens. This is mainly because the read/write request handling within the SSD has lots of overlapping [75, 78], such as device-level IO request queue, FTL engine, write cache, flash chip controller for accessing the planes, etc.

Next, in terms of latency, we run a 4KB random read stream, and couple with another stream issuing random/sequential writes, varying its IO size (Figure 22). We also repeat the same experiment by mixing a 4KB sequential write stream with a random/sequential read streams (Figure 23). Adding background traffic (stream2) definitely hurts the average and tail latency of the frontend one (stream1). This is due to the head-of-line blocking impact coming from interleaved different typed IOs. The larger the IO size is, the more latency degradation one would observe. For example, considering random read under sequential write, the average/p99.9 latency of the 128KB case is 1.7× and 2.6× higher than the 4KB case, respectively. Further, the curve becomes flat in Figure 22 after 16KB because stream2 has saturated the maximum write bandwidth. Therefore, to isolate different storage streams, one should also take the read/write distribution and carefully monitor its dynamic execution costs at runtime.

Appendix E RocksDB LSM-tree

RocksDB [20] is based on an LSM-tree data structure, consisting of two key components: *Memtable*, an in-memory data structure that accumulates recent updates and serves reads of recently updated value; *SSTable*, collections of sorted key-value pairs maintained in a series of levels. When the memtable reaches the size limit, it is persisted as an SSTable by flushing the updates in sequential batches and merging with other overlapping SSTables. Also, low-level SSTables are merged into high-level ones via compaction operations. L_0 SSTables contain the latest data, while $L_1..L_n$ contain the older data. Files within each level are maintained in a sorted-order, with a disjoint key-range for each SSTable file (except in L_0, where each SSTable file can span the entire key-range). Data retrieval starts from the Memtable and will look up multiple SSTables (from level 0 to high levels) until finding the key.