Building an Elastic Block Storage over EBOFs Using Shadow Views

Sheng Jiang[1] and Ming Liu[2]

[1]Carnegie Mellon University [2]University of Wisconsin-Madison

Abstract

The EBOF (Ethernet-Bunch-Of-Flash) has emerged as an enticing and promising disaggregated storage platform due to its streamlined I/O processing, high scalability, and substantial energy/cost-efficiency improvement. An EBOF applies a smart-sender dumb-receiver design philosophy and provides backward-compatible storage volumes to expedite system deployment. Yet, the static and opaque internal I/O processing pipeline lacks resource allocation, I/O scheduling, and traffic orchestration capabilities, entailing bandwidth waste, workload non-adaptiveness, and performance interference.

This paper presents the design and implementation of a distributed telemetry system (called **shadow view**) to tackle the above challenges and facilitate the effective use of an EBOF. We model an EBOF as a two-layer multi-switch architecture and develop a view development protocol to construct the EBOF running snapshot and expose internal execution statistics at runtime. Our design is motivated by the observation that fast data center networks make the overheads of inter-server communication and synchronization negligible. We demonstrate the effectiveness of shadow view by building a block storage (dubbed **Flint**[1]) atop EBOFs. The enhanced I/O data plane allows us to develop three new techniques–an elastic volume manager, an eIO scheduler, and a view-enabled bandwidth auction mechanism. Our evaluations using the Fungible FS1600 EBOF show that a Flint volume achieves 9.3/9.2 GB/s read/write bandwidth with no latency degradation, significantly outperforming the defacto EBOF volume. It achieves up to 2.9× throughput improvements when running an object store. Flint is tenant-aware and remote target-aware, delivering efficient multi-tenancy and workload adaptiveness.

1 Introduction

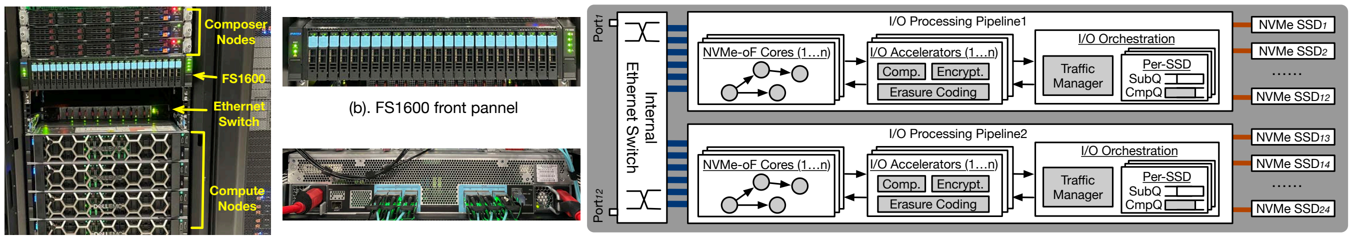
Storage disaggregation has gained increasing attraction in the past few years because of its independent scaling, high utilization, and cost-efficiency improvement. With the advent of 400+GbE network and fast remote storage protocols, a remote storage server can deliver millions of IOPS at tens to hundreds of microseconds, approaching the performance of a direct-attached storage system. We have seen a rising number of deployments [15, 52, 72, 76] at public clouds, enterprise on-premise clusters, and edge data centers.

Ethernet-Bunch-Of-Flash (EBOF) is an emerging disaggregated storage solution that packs an Ethernet switch with NVMe drives into one system SoC (System-on-a-Chip). Compared with today's server JBOFs (Just-a-Bunch-Of-Flash), an EBOF implements hardware-assisted remote I/O processing pipelines, elides the power-hungry general-purpose CPUs on the I/O path, and increases the I/O scalability by eliminating the DRAM and PCIe subsystems, significantly improving the system energy efficiency. For example, the Fungible FS1600 (our evaluation target) achieves 200K and 5.9K IOPS per Joule for reads and writes, nearly outperforming existing server storage appliances by one order of magnitude!

However, there is no well-trodden understanding of the EBOF capabilities and limitations, hindering deploying this new storage platform. An EBOF applies the *smart-sender dumb-receiver* design philosophy and provides backward-compatible volume-oriented storage functionalities. This simplifies the system integration and expedites platform deployment but imposes several inefficiencies (§2.2). First, an EBOF volume is statically mapped to an NVMe drive and embodies location oblivious block placement, failing to harness the massive internal I/O bandwidth. Second, an EBOF volume employs size-dependent bandwidth allocation, whose performance only scales with the volume capacity, unadapted to what the actual workload I/O needs. Third, an EBOF is tenant-unconscious, unaware of the device condition, and incurs substantial interference among concurrent I/O streams.

The root cause is that its static I/O processing pipelines only expose block read/write access interfaces, but lack resource allocation, I/O scheduling, and traffic orchestration capabilities. To address this, our idea is to develop a distributed telemetry system (dubbed **shadow view**) that continuously monitors the EBOF running condition. The shadow view serves as the intermediate layer between an EBOF and storage clients,

[1]Flint is available at https://github.com/netlab-wisconsin/Flint.



(a). Cluster setup (b). FS1600 front pannel (c). FS1600 back pannel (d). FS1600 Internal Architecture

Figure 1: The system deployment and internal architecture of the Fungible FS1600 EBOF platform. (a) depicts our evaluation testbed. (b)/(c) show the front/back panels of the chassis. (d) details the internal architecture of an FS1600.

which exposes the EBOF's internal execution characteristics and assists clients in making efficient I/O execution (such as block placement, bandwidth allocation, and scheduling). The key observation behind this is that the rising data center network speeds make fast (single-digit microsecond) inter-server communication and data synchronization possible. Such a marginal latency to develop the shadow view will then have a negligible impact on the storage applications.

Realizing this vision is non-trivial due to the inherent opaqueness and non-programmable interface of EBOF internals. Towards this, we model an EBOF as a *two-layer multi-switch architecture* that bridges frontend Ethernet ports and backend NVMe drives. Based on the model, we develop a view construction scheme by enabling performance monitoring at multiple vantage points along the I/O path, tracing I/O execution from storage clients, and collectively analyzing the performance characteristics of different components. Our shadow view service runs at view agents (co-located with clients) and a view controller, continuously synchronized via a managed protocol to ensure an up-to-date view is shared among clients. The view controller employs a back-propagation bottleneck analysis mechanism to detect internal EBOF congestions and affected storage streams.

We then build **Flint**–an elastic block storage for an EBOF–atop the shadow view to demonstrate its effectiveness. With the enhanced EBOF I/O data plane, Flint develops three techniques leveraging the shadow view's capabilities. The first one is an elastic volume manager that places storage blocks across all internal NVMe drives on-demand, guided by a runtime EBOF execution profile. The second one is an eIO scheduler that dynamically computes per-IO priority as rank and uses it to orchestrate concurrent I/O submission. The third one is a view-enabled bandwidth auction mechanism, which partitions available bandwidth among competing volumes in a max-min fairness manner at fine granularity. In tandem, Flint achieves high throughput with tolerable latency cost, maximizes device utilization, and ensures efficient multi-tenancy.

We evaluate Flint on a real-system testbed comprising commodity Dell servers and Fungible FS1600 EBOFs. Using FIO-based synthetic micro-benchmarks, we show that one elastic volume over Flint achieves up to 9.3/9.2 GB/s read/write bandwidth, outperforming the defacto EBOF volume by 14.5/13.6×, whose I/O latencies are similar. We further configure various experiments and demonstrate that Flint is tenant-aware and remote target-aware, which can mitigate I/O interference, deliver fair bandwidth allocation, and offer workload adaptiveness. When deploying an object store over Flint, one can achieve up to 2.9× throughput improvements with 66.4%/74.6% average read/write latency savings.

2 Understanding EBOF

This section provides some necessary background about EBOFs and presents a characterization study.

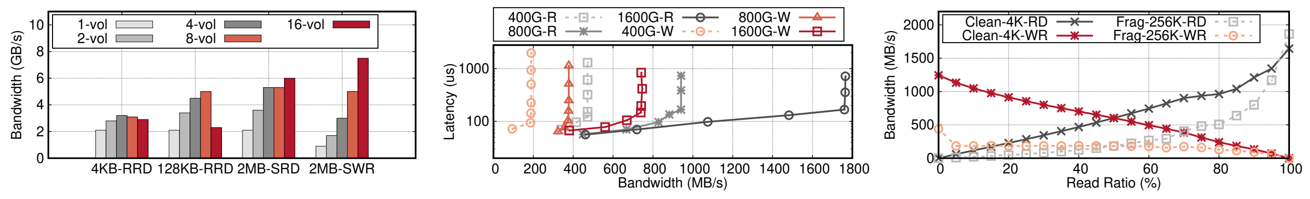
2.1 Ethernet Bunch of Flash (EBOF)

EBOFs are an emerging disaggregated storage solution that holds a number of NVMe drives using an Ethernet switch-integrated SoC (System-on-a-Chip). Compared with today's server/SmartNIC-based JBOFs (Just-A-Bunch-Of-Flash), an EBOF (a) increases the I/O scalability since it elides system-level bottlenecks due to the constrained CPU processing speeds and DRAM/PCIe bandwidths; (b) improves the overall resource utilization and mitigates the resource stranding issue as it eschews the fixed ratio between CPU and SSDs induced by the number of PCIe lanes; (c) achieves high energy/cost-efficiency because it eradicates the hefty computational resources at the storage target. There are some early engineering prototypes being developed, tested, and sampled, such as Ingrasys ES2000 [7] and Fungible FS1600 [5].

FS1600 EBOF. An EBOF is a standalone 2U chassis resembling commodity servers in physical appearance (Figure 1-a). Take the Fungible FS1600 EBOF as an example (which is our development and evaluation target). It encloses 12× 100GbE ports (Figure 1-c) and 24 NVMe drives (Figure 1-b), capable of processing 1.2Tbps storage traffic, sustaining 15M IOPS small random reads and 4.4M IOPS large sequential writes at most, aggregated across all the drives. The typical active power consumption is 750W (850W peak), translating to 200K and 5.9K IOPS per Joule for reads and writes, respectively, significantly outperforming existing JBOF boxes.

Figure 1-d presents the FS1600's internal architecture, consisting of an integrated Ethernet switch, two I/O processing pipelines, and 24 NVMe SSDs. First, the internal switch has 12 ports for external connectivity and delivers network traffic to two pipelines based on the storage targets. Next, packets undergo a sequence of processing stages along the pipeline:

- *Stage 1: NVMe-oF protocol processing.* This includes (i)



(a) Static mapping and location oblivious. **(b) Size-dependent and workload-agnostic.** **(c) Tenant-unconscious and device unaware.**

Figure 2: Performance characteristics of EBOF volumes on FS1600. (a) compares the throughput of four I/O workloads under five logical volume configurations. Physical volumes are from different SSDs. A logical volume uses the striped type (with the 128KB stripe size) where the striping width equals the number of physical volumes. We max out the bandwidth using multiple I/O streams with adequate queue depth. RRD=Random Read. SRD=Sequential read. SWR=Sequential Write. (b) presents the latency-throughput of 4KB random reads and 4KB sequential writes for 400GB, 800GB, and 1600GB volumes, respectively. The y-axis is log scale. (c) shows read/write throughput varying with the read ratio of a mixed workload. The 4KB I/O runs over a clean drive, while the 256KB one runs on a fragmented SSD (preconditioned with 128KB random writes).

networking stack handling and (ii) NVMe-oF capsules and NVMe I/O commands conversion, which are executed over a chain of specialized NVMe-oF cores;

- *Stage 2: Per-IO acceleration.* An EBOF is equipped with several domain-specific accelerators for compute-intensive tasks, such as compression, encryption, and erasure coding;
- *Stage 3: I/O orchestration.* It delivers I/O commands to each SSD via a traffic control module. Akin to an Ethernet switch, there is a traffic manager and a queueing subsystem for both submission and completion.

Last, an EBOF usually supports NVMe drives in U.2 and the emerging compacted EDSFF form factors [3].

EBOF Deployment. An EBOF imposes two differences compared with deploying a traditional JBOF. First, it requires several redundant dedicated servers (called composer nodes in Figure 1-a) to hold the management plane, which (a) configures the EBOF system (e.g., adding/removing forwarding rules for the internal switch, formatting the hot-plugged NVMe drives); and (b) provides a remote control panel for users. Clients interact with these composers on creating/deleting/modifying volumes, monitoring the EBOF running status, and alerting the health condition. Second, each pipeline of an EBOF organizes its Ethernet ports (6 in our case) as one trunk port, with a L3 IP address assigned. The ToR switch, connecting to the EBOF, should then create LACP (Link Aggregation Control Protocol) [6] trunk groups for the corresponding ports, and configure the fallback mode.

An EBOF system provides the physical volume abstraction as today's storage disaggregation solutions. To use it, one should first create a fixed-sized physical volume, which is mapped to one specific SSD. Clients then instantiate an NVMe-oF session, mount the remote volume locally, and make it an NVMe block device. Users can further create logical volumes for capacity/bandwidth expansion and fault tolerance. On our FS1600 testbed, a physical volume, consisting of a sequence of extents, can also be compressed and encrypted if enabled during the volume initialization phase. It primarily supports the NVMe-over-TCP protocol [8].

2.2 Characterizing an EBOF

An EBOF system applies the *smart-sender dumb-receiver* design philosophy and provides backward-compatible volume-oriented storage functionalities. It exposes physical volumes with basic block I/O read/write capabilities, and delegates advanced storage support (such as load balancing and replication) to clients. This drastically expedites the system integration and platform deployment because one can seamlessly transition to an EBOF setup from traditional server-based storage appliances. However, such simplicity comes at a cost. Below we empirically characterize an EBOF and identify several limitations that hinder its performance and efficacy.

Issue #1: Location oblivious block placement. An EBOF volume is statically mapped to an NVMe drive and cannot harness the massive internal I/O bandwidth. Its data placement strategy is straightforward: (i) a newly created volume is placed at one SSD with enough available space; (2) when there are multiple candidates, it chooses one randomly. This simplifies the hardware logic and reduces the in-memory metadata footprint, but bounds the maximum performance of a volume to just one drive, far below the bandwidth capacity of an EBOF. As shown in Figure 2-a, the bandwidth of random reads and sequential writes under large blocks of a physical volume is capped at 2.1GB/s and 0.9GB/s, respectively, approaching the NVMe SSD device limit.

A workaround solution is to create multiple physical volumes (from different SSDs), combine them as a logical volume, and format it into the striped type. This could benefit large sequential I/Os. For example, the 2MB sequential write (Figure 2-a) achieves nearly twice the bandwidth when doubling the number of physical volumes because each I/O hits the strip group, yielding multiple concurrent I/Os to different SSDs. However, for many other I/O profiles, one would observe marginal benefits because the inherent extent layout of a logical volume is location oblivious, unable to harness the parallel I/O paths. As depicted in Figure 2-a, the 4KB/128KB random read maxes out with 4/8 physical volumes, achieving 3.2GB/s and 5.0GB/s. Similarly, the 2MB sequential read shows inferior performance and only sustains 6.0GB/s, less than one-fifth of the throughput in the 16-vol case.

Takeaways. Flexible and location-aware data placement is essential to unleash the EBOF bandwidth capacity. Ideally, a volume can distribute its extents to one or several bandwidth-capable drives based on online profiling.

Issue #2: Size-dependent bandwidth allocation. The performance of an EBOF volume only scales its capacity and is inadaptable to the actual workload requirements. An EBOF takes the *IOPS/GB* as the key metric when reserving per-volume bandwidth on the control plane upon initialization. The data plane then ensures that the actual bandwidth usage doesn't exceed its allocated share. Such a design only requires a static traffic policer installed at the processing pipeline without employing some dynamic mechanisms like the water-filling algorithm [66]. This completely moves the I/O resource allocation and scheduling out of the performance path, entailing a streamlined I/O pipeline. Take the FS1600 as an example. It configures the QoS upper bound of a volume via $\frac{EBOF\ Read/Write\ IOPS}{EBOF\ Capacity} \times Volume_{size}$.

Figure 2-b depicts the latency-throughput of two workloads running under different-sized volumes. The latency of 4KB random reads (sequential writes) rises drastically when the bandwidth approaches 472 (188) MB/s, 941 (378) MB/s, and 1760 (741) MB/s for 400GB, 800GB, and 1600GB volumes, respectively. This matches the bandwidth upper bound prescribed as above. Such a size-dependent EBOF volume, benefits workloads embodying the same requirements on capacity and throughput, but conversely, contradicts many other application usage patterns. For example, a throughput-oriented workload with a small size requirement, like a metadata service in a distributed blob store [38, 57, 59], has to request a large volume to ensure enough bandwidth is provisioned, entailing storage capacity waste. Similarly, a capacity-oriented application with low throughput needs, like a logging service, allocates a high-capacity volume with excessive throughput reserved implicitly, causing bandwidth waste.

Takeaways. Decoupling capacity allocation and bandwidth reservation functionalities of an EBOF helps improve its efficiency when holding I/O workloads with incongruent size and throughput requirements. This necessitates the I/O processing pipeline to control bandwidth on-demand at the data plane.

Issue #3: Interference heavy. An EBOF volume is tenant-unconscious and device condition unaware, entailing great interference among concurrent I/O streams. An EBOF system enforces performance isolation at the volume granularity. There is a per-volume rate limiter inside the traffic manager that ensures a busy or noisy volume cannot exceed its bandwidth share. This mechanism mitigates inter-volume I/O interference when requests transmit within the pipeline, but completely ignores interference at the NVMe drive. As an example, we co-locate one victim volume with four adversarial volumes on one SSD. The victim volume runs a 4KB random read with QD (queue depth) =1, and we measure

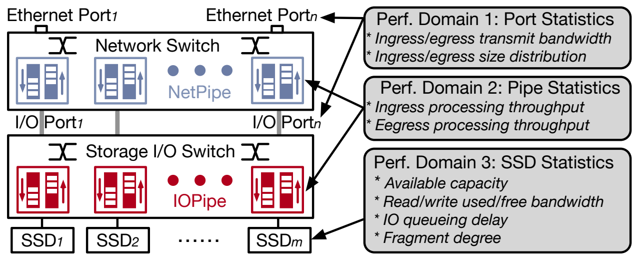

Ethernet Port1
Ethernet Portn
Network Switch
NetPipe
I/O Port1
I/O Portn
Storage I/O Switch
IOPipe
SSD1
SSD2
......
SSDm
Perf. Domain 1: Port Statistics
* Ingress/egress transmit bandwidth
* Ingress/egress size distribution
Perf. Domain 2: Pipe Statistics
* Ingress processing throughput
* Eegress processing throughput
Perf. Domain 3: SSD Statistics
* Available capacity
* Read/write used/free bandwidth
* IO queueing delay
* Fragment degree

Figure 3: The hardware model of an EBOF and the performance monitor domains of the shadow view.

its latency. The rest volumes run other throughput-intensive read/write streams. We observe that the P999 latency of the victim volume increases from 169us to several milliseconds!

Further, intra-volume performance isolation is not supported on the EBOF. As shown in Figure 2-c, a 4KB read-only stream achieves 439.0 MB/s higher bandwidth than the mixed case with a 50/50 read/write ratio on a clean SSD. When the drive becomes fragmented, even with a large 256KB block size, it sustains 59.5% and 63.4% less read and write throughput on average across all scenarios. The fundamental issue is that an EBOF schedules I/Os without considering the request cost and SSD condition, yielding inefficient multi-tenancy. As shown in prior studies [11, 37, 50, 54, 74], the per-I/O execution cost is determined by read/write mix ratio, I/O block size, and I/O concurrency. When contention happens at the SSD controller, the device queue, or NAND channels/chips/dies, an I/O would be stalled and incur a high execution cost. Capturing the SSD condition is non-trivial because (a) the number of clean flash pages and their location distribution hinge on the previous write history, which is not exposed; (b) the SSD internal parallelism is unknown due to the FTL mapping and private proprietary internal architecture; (c) housekeeping operations (like garbage collection and wear leveling) are unpredictable and thereby consume a non-deterministic amount of internal bandwidth when triggered.

Takeaways. Mitigating I/O interference requires an EBOF to monitor its end-to-end bandwidth availability (including network, I/O pipeline, and SSD) at the runtime and prudently schedule individual IO in a tenant-aware manner.

3 Shadow View: A EBOF Telemetry System

This section introduces the shadow view, a distributed telemetry system that enhances the EBOF data-plane capabilities.

3.1 Key Idea

The root cause that yields EBOF inefficiencies is the static I/O processing pipelines that only provide block read/write access functionalities, but lack resource allocation, I/O scheduling, and traffic orchestration capabilities. Our key observation is that the rising data center network speeds make fast (single-digit microsecond) inter-server communication and data synchronization possible. Such a marginal latency overhead will have a negligible impact on the storage applications because the per-I/O latency is in the order of tens to hundreds of microseconds. Therefore, our idea is to develop a software-based telemetry system (called **shadow view**) that continuously monitors the EBOF running condition. It serves as an intermediate layer between an EBOF and storage clients, which exposes EBOF runtime internal statistics and assists clients in efficient I/O execution (including block placement, bandwidth allocation, and scheduling).

3.2 Definition and Capabilities

The shadow view is a software-managed distributed telemetry system that profiles and reports the computation and I/O resource availability of an EBOF at runtime. It decouples from the EBOF hardware box and works as a standalone distributed service, deployed over a bunch of general server hosts. The key challenge to realizing it is the black box nature of EBOF internals that exposes little interaction and programmable interfaces. Hence, we develop a reactive approach that tracks, collects, and analyzes the end-to-end per-I/O statistics from all mounted volumes to construct an EBOF running snapshot.

To facilitate the view development, we model an EBOF as a two-layer multi-switch architecture (Figure 3) that bridges frontend Ethernet ports and backend NVMe drives.

- The **upper half** is a network switch, connecting N Ethernet ports on one side and an equal amount of I/O backplane ports on the other side, which forwards incoming/outgoing Ethernet packets. The switch internally comprises N bidirectional $NetPipe$s. For each pipe, we associate a virtual packet queue with each direction;
- The **bottom half** is a storage I/O switch that connects N I/O ports and M NVMe drives. Except for the basic forwarding functionality, it converts between NVMe-oF capsules and NVMe commands and accesses the SSD drive. Similarly, we model the switch as M bidirectional $IOPipe$s, staying close to the SSD, where each is equipped with a virtual I/O queue on the submission and completion paths.

Note that (a) unlike traditional Ethernet switches, both switches are asymmetric with two distinct sets of operating ports; (b) our hardware model is generic, only requiring minimal knowledge (like port #) of the EBOF specification.

Based on the proposed hardware model, we then organize the shadow view into three performance monitoring domains. The first one is port statistics, where we capture the traffic usage of Ethernet/IO backplane ports at packet/flow/session granularities, including used/available bandwidth and size distribution. The second one is pipe statistics that focus on reporting the processing throughput and queueing delay of $NetPipe$s and $IOPipe$s. We separate ingress and egress metrics to capture I/O submission and completion paths. The last one is SSD statistics. Due to the inherent opaqueness, except for the SSD's available space, we estimate the available read/write bandwidth headroom, I/O delay, and NAND fragment degree (an indicator to determine the I/O write cost) following prior studies [11, 37, 50, 54].

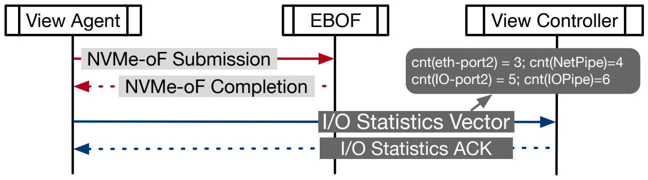


Figure 4: The view construction procedure.

Akin to other telemetry systems, our shadow view behaves as a database for an EBOF. It imposes two main capabilities. One is the runtime statistics query in which one can flexibly integrate the shadow view into a system and explore EBOF-conscious optimizations. The other is event notification, where applications can register callbacks to trigger proactive resource management. We will show how to use the shadow view to develop a block storage (§4).

3.3 View Construction

We construct the shadow view by tracing I/O execution from storage clients, collectively analyzing the performance characteristics of different components, and mapping them to the underlying hardware model. Our shadow view service is distributed across two entities: view controller and view agent. The view controller is integrated into the central arbiter and maintains the entire picture of an EBOF. It receives the per-I/O running statistics from view agents, performs collaborative analyses, and exposes RPC interfaces for interaction. The view agent, colocated with storage clients, monitors the I/O execution of its mounted block devices and only holds a partial view of an EBOF. We employ a hybrid pull/push approach to synchronize the view between these two (§3.4).

An EBOF embodies two types of execution paths, described as follows: (a) NVMe-oF submission: $Ethernet\ Port \Rightarrow NetPipe_{ingress} \Rightarrow I/O\ Port \Rightarrow IOPipe_{ingress} \Rightarrow SSD$; (b) NVMe-oF completion: $SSD \Rightarrow IOPipe_{egress} \Rightarrow I/O\ Port \Rightarrow NetPipe_{egress} \Rightarrow Ethernet\ Port$. Based on this, we show how the view is developed across the following two entities:

- **View Agent.** It takes the timestamp when submitting the I/O request and receiving the response. By subtracting these two, we know the end-to-end round-trip time (RTT), i.e., the sum of NVMe-oF submission and NVMe-oF completion. Upon completion (Figure 4), the agent (a) forwards the I/O statistic vector–<NVMe-oF session, target SSD, I/O type, I/O size, RTT>–to the controller; (b) updates corresponding metrics for its partial view. The I/O statistics vectors are batched before sending to the controller to reduce network communications. This indicates that the view development process is out of the system's performance path;
- **View Controller.** We maintain an in-memory representation of the hardware model (Figure 3) at the controller. It covers all system components from three performance domains. We employ a window-based approach to profile each metric to report the current and EWMA (exponentially

weighted moving average) value. Upon receiving an I/O vector message, the controller first dissects its traversed path within an EBOF based on the NVMe-oF session and target SSD, including *NetPipe*, *IOPipe*, and *I/O Port*. It then updates the corresponding metrics of the full view.

Bookkeeping Statistics. Our agent and controller use the same methods for aggregating running statistics. The packet and I/O distribution are captured via a typical counting array. Regarding bandwidth and throughput, we consider both bit-per-second and packet (I/O)-per-second because the processing capacity of an EBOF is size-dependent. Except for the available capacity, SSD statistics are generally challenging to obtain due to the opaqueness. Following prior work [37, 54, 70], we combine an offline profiling and an online estimation strategy that uses the end-to-end I/O delay as an indicator to decide the queueing delay at the controller and bandwidth headroom. The fragment degree is approximated using the dynamic write cost [54]. Due to the window-based profiling scheme, the view service supports aging automatically. Future commodity EBOFs could do fine-grained timestamping at different stages across the pipeline, which would give us more internal execution visibility. Our shadow view leaves the extension interface to integrate them later.

3.4 View Synchronization

The shadow view is collaboratively built by the view controller and view agents. We develop a view synchronization protocol to update and synchronize the view. Our design is inspired by prior distributed cache-coherence protocols [40, 42]. The key idea is to use a monotonically increasing counter to represent the view recency. Thus, each entity of the hardware model (like port, pipe, and SSD) is assigned a dedicated counter at both agent and controller. We don't use a shared counter because the view can be updated partially, i.e., an I/O request only traverses a subset of an EBOF.

The view controller works as a serialization point of an EBOF to order all issued I/Os. Upon receiving the statistic vector of a completed I/O request, it performs three operations atomically: updates the corresponding view entries, increments their counters by one, and returns an acknowledgment message that attaches the changed counters for the agent's partial view (Figure 4). The controller also maintains a directory of mounting storage clients who hold the partial view. Next, when the view agent receives the response, it first compares the attached counter (X') with the previously held one (X) to decide if the view is obsolete as follows:

- If |X'-X| = 1, this indicates the client is the latest EBOF user and its partial view is update-to-date;
- If |X'-X| > 1, it means there are multiple concurrent storage clients issue I/Os and its partial view is outdated;

The view agent always overwrites the counter with the newly received value because only the controller can update the counter. A careful reader might notice that the message order of the I/O statistics vector received by the controller might be different from the I/O completion order of an EBOF. For example, a client might complete an I/O request earlier than another one, while its statistics update message comes late. We resort to the EBOF timestamp information of the statistics message to resolve the issue. Our controller connects to EBOF composer nodes (§ 2.1) via the management network and has time synchronized. When updating each component of the shadow view, it uses the timestamp to decide if the statistics stay within the profiling window.

Based on the above support, our shadow view service provides two types of synchronization. In the *PUSH* mode, the view controller periodically publishes its latest shadow view to its view agents. If a client registers a particular callback function to monitor an event (such as if the SSD available bandwidth has dropped to a threshold), the controller then copies the view proactively. In the *PULL* model, the view agent eagerly fetches its accessed partial view if its own is obsolete. This is realized via the controller RPC.

Algorithm 1 Back-propagation Bottleneck Analysis.

```
1:  procedure ANALYZE_BOTTLENECK                ▷ Executed per-epoch
2:      for SSD in all_SSDs do
3:          if $SSD.avail\_bw \leq Threshold$ then
4:              Congested_SSDs.add(SSD);
5:              Congested_IOPipes.add(SSD.IOPipe);
6:      for Session in all_NVMeoF_Sessions do
7:          if $Session.target\_SSD$ in $Congested\_SSDs$ then
8:              Congested_NetPipes.add(Session.NetPipe);
9:              Victim_Sessions.add(Session);
10:     for Session in all_NVMeoF_Sessions do
11:         if $Session.NetPipe$ in $Congested\_NetPipes$ then
12:             if $Session$ not in $Victim\_Sessions$ then
13:                 Victim_Sessions.add(Session);
```

3.5 Bottleneck Analysis

When an internal component of an EBOF reaches its processing limit, one would observe queue buildup along the EBOF I/O processing pipeline, entailing head-of-line blocking and jeopardizing I/O latency. The shadow view offers two possible solutions to identify the bottleneck: one is examining the queueing delay at the *NetPipe*, *IOPipe*, and *SSD*; another one is comparing the achieved bandwidth with the device limit (obtained from the EBOF specification). Based on our operating and development experience, we find that (a) *SSD*s are always the first to hit the system bottleneck; (b) the processing capability of the *NetPipe* and *IOPipe* is generally over-provisioned. Because of the path symmetry between I/O submission and completion within an EBOF, congestion only happens at the ingress path, not egress.

The shadow view employs a back-propagation analysis scheme to dissect the congestion region and identify victim NVMe-oF sessions (ALG1). When an NVMe drive becomes over-subscribed, the shadow view can quickly locate the affected *IOPipe* because there is a 1-to-1 mapping between

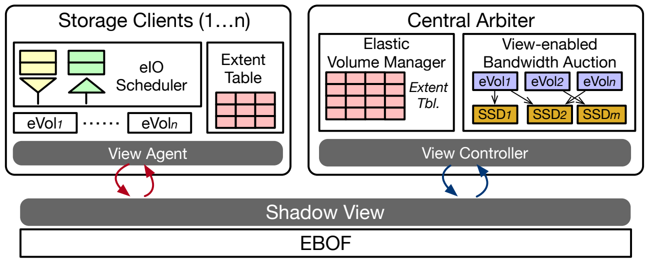


Figure 5: The system architecture of Flint.

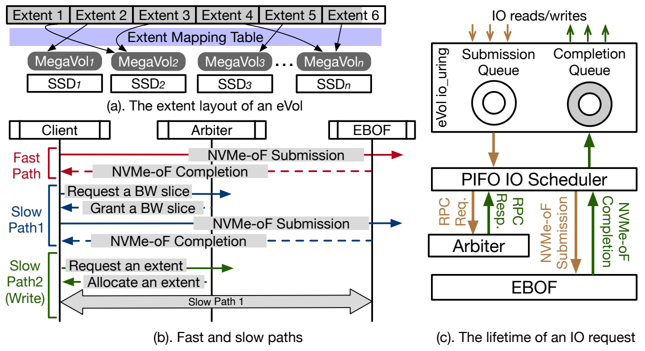


Figure 6: *eVol* extent layout, I/O lifetime and I/O path.

IOPipe and *SSD* (ALG1 L1-5). Next, since the mapping between *NetPipe* and *IOPipe* is n-to-m, to identify victim *NetPipe*s, we scan through active NVMe-oF sessions whose destination is the congested SSD (ALG1 L6-9). Last, for each victim *NetPipe*, the algorithm reexamines all other NVMe-oF sessions and finds out the ones that share the congested pipe (ALG1 L10-13). The bottleneck analysis process is launched periodically at the end of each profiling epoch.

4 Flint: an Elastic Block Storage

This section presents Flint that overcomes the drawbacks (§2.2) of existing EBOF volumes. Our design goals are:

- **High throughput with tolerable latency cost.** Massive I/O bandwidth is the crucial capability of an EBOF compared with conventional JBOFs. Flint aims to fully utilize the storage bandwidth across all NVMe drives and eliminate any potential I/O stalls along the pipeline;
- **High utilization and deployment flexibility.** The volume provided by Flint should use any available space from any SSDs, regardless of whether the storage blocks are continuous on one SSD or fragmented across multiple drives. Its performance should scale with the application demands instead of being determined by volume sizes;
- **Efficient multi-tenancy.** Flint should provide the max-min fairness guarantee when partitioning bandwidth among co-located volumes. It should estimate the device bandwidth availability at runtime and admit the right amount of I/Os to avoid oversubscription. The head-of-line blocking issue should be tackled in an end-to-end fashion.

4.1 Overview

Flint comprises two types of nodes (Figure 5): storage client and central arbiter, running atop the shadow view. The client, co-locating with a view agent, allocates/deletes/manipulates volumes, submits read/write I/Os, and orchestrates requests through **an eIO scheduler** (§4.3 and §4.5) to mitigate the head-of-line blocking. The central arbiter, cooperating with a view controller, acts as the EBOF resource manager. It encloses two key system components: (1) **an elastic volume manager** (§4.2), which places data blocks, maintains an extent mapping table, and serves volume management requests; (2) **a view-enabled bandwidth auction mechanism** (§4.6), partitioning available bandwidth among competing volumes in a max-min fairness manner at the fine granularity. A volume can be replicated using chain replication for high availability (§4.4). The arbiter is not replicated because it is not in the data path and clients can continue performing I/Os on an open volume even if the arbiter is down, although it can be made reliable. All the cross-node communications are realized via RPCs, and we implement them atop eRPC [35].

4.2 Elastic Volume and Volume Manager

Flint provides an elastic volume (*eVol*) to address the limitations of the defacto EBOF volume. An *eVol* consists of a sequence of fixed-sized extents (2MB in our case), spreading across more than one SSD. To enable flexible data placement over all EBOF drives, Flint employs a two-level hierarchical volume manager at the arbiter. As depicted in Figure 6-a, the bottom layer creates one mega-volume (*MegaVol*) for each SSD covering the entire storage space through the default EBOF control plane. The upper layer then performs block allocation and places extents to different mega-volumes based on the given policy. Each *eVol* has an extent mapping table, where our volume manager uses an aggregated one for the whole EBOF. Each mapping entry has 16 bytes with the following format: SSD index, physical extent number, and replication node. We implement the extent table using an in-memory hashtable. It takes only 8MB to address a 1TB *eVol*, which is feasible on commodity servers.

A storage client requests a new volume via the `create_vol` API. It takes the volume size and allocation policies as parameters. We apply a lazy allocation scheme. Upon the `create_vol` request, the volume manager only checks whether there is enough capacity to hold the volume, reserves the space, and sends back the acknowledgment. A successful return doesn't indicate the actual data placement is done. When an I/O write happens and there is no corresponding mapping entry in the local extent table, clients would interact with our volume manager to allocate a new extent from one SSD (mega-volume), and then update extent tables on both sides. The extent tables are stored in RocksDB [1] backed by a replicated file system (such as Ceph [77]), and all updates are executed in a transactional manner.

We use a weighted score function to realize different data placement policies. Under the extent allocation request from an *eVol*, our volume manager calculates the intended score for each SSD and chooses the one with the lowest score. The function (shown below) balances a slew of factors: (a) prior allocation history, i.e., how many extents have been placed; (b) available capacity, where we aim to balance the SSD utilization; (c) recent busy status from the bandwidth perspective; (d) fragment (or wear-leveling) degree of the SSD; (e) user-defined preferences. We obtain the parameters of (c) and (d) from the shadow view.

$$Score(eVol_i, SSD_j) = \alpha \times Extent_{\#} + \beta \times Capacity_{used} + \gamma \times Busy_{degree} + \delta \times Fragment_{degree} + \eta \times Preference$$

The volume manager maintains a score matrix for each *eVol* and updates it periodically. Clients set the initial placement policy in `create_vol` and can alter the strategy on the fly via `update_vol`. When a volume is destroyed (`destroy_vol`), the volume manager frees the space, updates the metadata from the extent table, and deletes the related score matrix.

4.3 I/O Path

In Flint, a storage client submits I/O requests first to a co-located block I/O scheduler, then to a per-thread `io_uring` engine. We choose the `io_uring` interface [2, 10, 22] due to its efficient asynchronous support, rich feature set, and high scalability. The I/O scheduler (§4.5) leverages the shadow view to determine the I/O issuing order, mitigating the head-of-line blocking. Finally, Flint uses the NVMe-over-TCP protocol [8] to carry I/O reads/writes between clients and an EBOF. Figure 6-c depicts the lifetime of an I/O.

Flint introduces fast and slow paths when accessing the EBOF because its bandwidth resources are managed by the central arbiter (Figure 6-b). A read can be issued to the EBOF directly only if the allocated bandwidth slice (§4.6) can hold the request size (fast path). Otherwise, it should request a new slice from the arbiter and then submit the I/O (slow path1). A write behaves similarly except that it may introduce one more slow path–asking the arbiter to allocate a new extent. However, in practice, we noticed that this extra RPC call takes only a few hundred microseconds and incurs negligible performance overhead, given that: 1) a typical write I/O to consume an extent (2MB) takes thousands of microseconds; 2) this slow path can only happen for write I/Os; 3) write I/Os are directed to the most performance-capable extent that is dynamically allocated based on the current system condition.

4.4 Replication

Flint employs the chain replication [65] as an optional configuration to improve system availability. It replicates data blocks at the extent granularity. When enabled, given an extent allocation, the volume manager chooses three different SSDs with the lowest scores for placement. Then in the extent mapping table, each entry records all the replicas along with the ordering in the replication chain.

Flint follows standard protocol execution logics as chain replication prescribes. I/O writes are issued to the head, which are then passed along the chain. We pass down commands instead of actual updates because the target LBAs (logical block addresses) are different. I/O reads are handled by the tail of the chain. Chain replication achieves strong consistency naturally because requests are ordered by the tail, which operates as the primary in a traditional primary-backup system. There are two ways to facilitate request passing over the chain. One is using the EBOF's recirculation capability, where a completed I/O can be reinserted into the I/O processing pipeline with some modifications. Such an intra-EBOF coordination is an ideal solution, reducing the replication latency and saving external network bandwidth. The other one is implementing the chain replication coordinator at the client host. Essentially, after receiving the I/O completion from the predecessor, it refabricates the I/O submission request and sends it to the successor. This incurs multiple round trips between the client and EBOF. Flint takes the latter approach because our FS1600 box doesn't expose the recirculation directly. We are currently working with the vendor on realizing the former approach.

Algorithm 2 eIO Scheduler.

```
1:  procedure EIO_ENQUEUE
2:      io = pending_queue.poll();
3:      compute_rank(io);
4:      sort_window.binary_insert(io);
5:  procedure EIO_DEQUEUE
6:      for io IN sort_window do
7:          if io.id == io.stream.sq.front().id then
8:              io.admit();
9:              sort_window.remove(i);
10:             io.stream.sq.pop();
11:             aging(); break;
```

4.5 eIO PIFO Scheduler

Flint runs an I/O scheduler at each client to improve the *eVol* performance. To capture the dynamic I/O execution cost and mitigate the resulting head-of-line blocking, Flint develops a priority-based I/O scheduler based on the ranking concept. The rank of an I/O determines its issuing order.

Our eIO PIFO scheduler uses a scheduling tree for the implementation. It encompasses a priority queue for submission and a FIFO queue for completion. Upon an incoming I/O, the scheduler computes its rank and inserts it into the corresponding location. Rank, representing the I/O cost, takes I/O characteristics (size and type), queueing time, and allocated bandwidth into account. A lower-ranked I/O would be dequeued first. The completion side delivers I/O simply following the arrival order. When the client receives a bottleneck report (§3.5), all victim I/Os will experience a rank increase due to the rising cost, thereby hedged from submission.

Algorirthm 2 presents the pseudo-code of our eIO scheduler. It polls I/Os issued from different threads, calculates their ranks based on the current shadow view, and inserts them into a binary search tree that is sorted using the rank value. The *sort_window* uses binary search since requests

Algorithm 3 Client-Side Bandwidth Auction.

```
 1: procedure IO_SUBMISSION(io)
 2:     ssd_nqn = io.ssd_nqn;
 3:     avail_slices = shadow_view[ssd_nqn].avail_slices;
 4:     slice = avail_slices.back();
 5:     if slice.is_full() then
 6:         slot = RPC_request_slice(avail_slice);
 7:     slice.consume_io(io);
 8:     io.admit();
 9: procedure IO_COMPLETION
10:     slice = io.slice
11:     slice.completes += 1;
12:     if slice.is_full() AND slice.submits = slice.completes then
13:         slice.close();
14:     RPC_report_completion();
```

Algorithm 4 Arbiter-Side Bandwidth Auction.

```
 1: procedure ARBITER_BW_AUCT
 2:     while true do
 3:         epoch_advance(); estimate_avail_bw();
 4:         reassign_quota();
 5:         while not epoch_expired() do
 6:             refresh_drr_lists();
 7:             for tenant in active_list do
 8:                 tenant.deficit += Quantum;
 9:                 if tenant.deficit > DEF_THRESHOLD then
10:                     tenant.state = Idle; tenant.deficit = 0;
11:                     idle_list.add(tenant);
12:                     active_list.remove(tenant);
13:                     reassign_quota();
```

are sorted by their ranks. During dequeue, Flint iterates over the *sort_window* until an I/O is in the front of the submission queue of an I/O stream and admits it. Hence, Flint ensures: (1) no I/O reordering happens within a stream, therefore following their submission order; (2) requests belonging to different streams are submitted following their ranks. Finally, we apply aging for the pending requests to prevent starvation.

4.6 Bandwidth Auction

Flint allocates bandwidth fairly among competing storage streams using an RTS/CTS-like mechanism [17]. An NVMe-oF client can only issue an I/O when granted enough bandwidth slices. Hence, each active client periodically asks the arbiter for available bandwidth, and the arbiter sends back grant messages after the auction. The challenge to realizing it is that the bandwidth allocation of an EBOF should holistically consider each component between Ethernet ports and SSDs, instead of only targeting one entity. For example, a storage stream might receive adequate bandwidth from an SSD but then be bottlenecked by the I/O processing pipeline (like *NetPipe*), causing bandwidth waste.

Our arbiter combines deficit round-robin (DRR) [71] and gang scheduling [23,24] techniques to address the issue. Each active NVMe-oF session–that has issued a bandwidth allocation request–is assigned a three-tuple deficit counter vector, representing resource requirements for *NetPipe*, *IOPipe*, and *SSD*. We then divide all sessions into different groups based on their target SSD. The shadow view reports the bandwidth headroom of each component, which we translate to the total amount of I/Os that can be admitted. For each group, the arbiter walks through its associated sessions whose deficit counter vector is accumulated with a prescribed quantum (whose value is in proportion to the session's priority) at each round. A bandwidth slice is granted when (a) all the deficit counters of a session have reached their target thresholds, and (b) there are adequate resources indicated by the shadow view. After that, the deficit counter vector is reset and the session waits for the next allocation. Users can specify their performance targets and Flint will then calculate the quantum to ensure the total bandwidth of a SSD is partitioned accordingly. By default the bandwidth is equally shared.

Algorithms 3 and 4 detail the bandwidth auction mechanism on the client and arbiter sides. When a client issues an I/O, it looks up the current bandwidth slice from the local shadow view and checks its availability. If the slice has no room, it asks for a new one from the arbiter. The slice consumes an I/O based on its weighted cost (profiled by the shadow view) and then calls *admit*() to notify the I/O executors for submission (through the `io_uring`). Upon completion, the client reports the I/O latency to the arbiter. If the slice becomes empty and there are no outstanding I/Os, it requests for a new bandwidth slice, which is piggybacked through the report RPC (ALG 3 L12–14).

On the arbiter's side, there is a DRR routine running for each SSD device. Each DRR routine maintains three different lists: (a) an active list containing tenants that are actively issuing I/Os, (b) a deferred list containing tenants that have run out of the virtual slice quota, and (c) an idle list that includes tenants that do not issue I/Os within a time window. At the beginning of a new epoch, Flint estimates the available bandwidth of the device and reassigns the bandwidth slices to each tenant. Flint then runs the DRR loop until the current epoch expires. In each loop, we refresh the lists in case their states have changed and iterate tenants over the *active_list* to increase their deficit value. If the deficit exceeds a threshold, we set its state to *Idle*, update the corresponding lists, and reassign quotas (ALG 4 L10–L13).

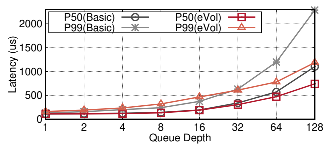
5 Evaluation

Our evaluations aim to answer the following questions:

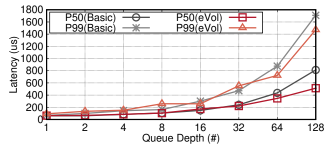
- How does Flint perform compared with the default EBOF block storage in terms of latency and throughput? (§5.2)
- How does Flint address the three performance inefficiencies of an EBOF volume? (§5.3, §5.4, §5.5)
- What are the considerations when deploying Flint? (§5.6)

5.1 Experimental Methodology

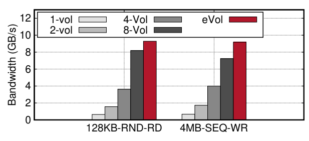
Testbed. We evaluated Flint using a small-scale rack cluster, comprising 2U commodity servers as storage clients, a Dell



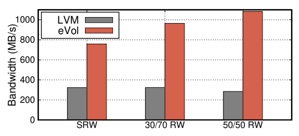
(a) 4KB Random Read.



(b) 4KB Sequential Write.



(c) Bandwidth.



(d) Congested Bandwidth.

Figure 7: Performance comparison between eVol and the defacto physical/logical volume on EBOF and LVM. (a)/(b) report the P50 and P99 latency of 4KB random read and 4KB sequential write when increasing the queue depth. (c) shows the maximum bandwidth of 128KB random read and 4MB sequential write comparing eVol and different types of logical volumes. (d) shows the bandwidth of LVM and eVol of 1MB sequential write, 30/70 mixed read/write, and 50/50 mixed read/write, where certain drives are congested.

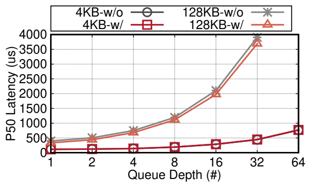
Z9264F-ON ToR switch (w/ 64× 100GbE ports), and the Fungible FS1600 EBOF. Each server is a Dell R7525 box, enclosing two AMD 7302 processors (running at 3.0GHz), 256GB DDR4 memory, a Broadcom 25GbE dual-port NIC, a Mellanox 100GbE dual-port CX6 NIC, and two Samsung NVMe SSDs. All clients run Ubuntu 22.04.

Experiment setup. We build Flint from scratch in about 7600 lines of C++ code. Flint provides eVol management and both sync/async I/O interfaces to the client. We primarily compared with the defacto volume service provided by EBOF. Our experiments first use FIO-based synthetic workloads [4] to explore the benefits of Flint proposed techniques and the capabilities of the shadow view. We then take an object store as a storage application to showcase the overall benefits. We use the YCSB benchmark [20] in this case. We report throughput and average/tail per-I/O latencies as the performance metrics.

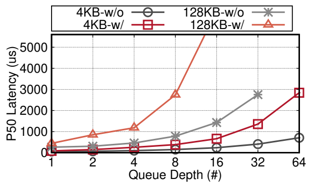
5.2 eVol Performance

We first examine the basic performance of an eVol. In terms of small I/Os, when we increase the same queue depth for an I/O stream, eVol achieves the same P50 latency as a physical volume (Figures 7-a and b). This is because the majority of I/Os traverse the fast path (§4.3), yielding one RTT between the client and an EBOF. An eVol places data blocks over all the SSDs in an EBOF. As a result, given the same queue depth, each SSD will receive less amount of I/Os than the physical volume. Thus, the eVol reduces the P99 latency by 48.1% and 13.4% in terms of 4KB random read and 4KB sequential write, respectively. Regarding large I/Os, eVol can harness the massive I/O bandwidth of an EBOF easily because of its flexible data layout (Issue #1 in §2.2). As shown in Figure 7-c, it achieves 9.3/9.2 GB/s when running 128KB random read/4KB sequential write, outperforming a logical volume with 1, 2, 4, and 8 physical volumes by 14.5/13.6×, 5.9/5.3×, 2.6/2.3×, 1.2/1.3×, respectively.

Replication. Flint uses chain replication (§4.4) to achieve high availability. Figure 8 reports the P50 latency of I/O reads and writes with the replication enabled/disabled. We use three replicas in this experiment. For 4KB and 128KB random reads, with replication, they achieve the same performance as the disabled case when varying the queue depth because requests are served by the tail node. However, in terms of writes, since an I/O should pass through the chain from the



(a) I/O read under replication



(b) I/O write under replication

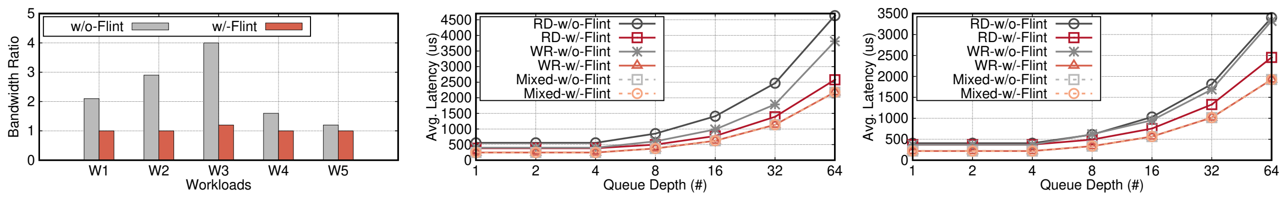
Figure 8: Latency varying the I/O queue depth when enabling and disabling the replication.

head to the tail, one would experience a considerable latency increase. For example, on average across all cases, the P50 latency of a 4KB and 128KB write is worsened by 2.9× and 3.5×, respectively. With future EBOF recirculation support (where a request can be reinjected into the I/O processing pipeline), we expect the latency can be further reduced.

5.3 Flint Decouples Capacity and Bandwidth Allocation

Next, we evaluate how Flint addresses the second issue (§2.2) via the bandwidth auction technique. We configure five scenarios with two types of contending random I/O streams, where each is issued from different-sized volumes: W1 (16× 4KB reads *v.s.* 8× 128KB reads), W2 (8× 4KB reads *v.s.* 8× 4KB writes), W3 (8× 4KB writes *v.s.* 8× 128KB reads), W3 (8× 4KB reads *v.s.* 8× 4KB writes), W4 (8× 128KB reads *v.s.* 8× 128KB writes), and W5 (8× 4KB 70/30 reads/writes *v.s.* 8× 4KB 30/70 reads/writes). We max out the bandwidth of each stream using a large queue depth (QD).

Figure 9-a reports their aggregated relative bandwidth ratio. For all scenarios, Flint ensures I/O streams with the same demand receive similar bandwidth regardless of their capacity. This is because (a) our arbiter distributes equal bandwidth slices to each active tenant in a deficit round-robin manner; (b) a client can only submit an I/O when its granted slice still has room. Thus, a small volume can still receive more bandwidth if it has more pending I/Os, whereas a large volume cannot take more than its (weighted) equal share. Under contention, no matter how many outstanding I/Os a stream issued, it can only receive more bandwidth when all contending streams operate in a similar phase. When disabling our mechanism, one can observe great unfair allocation. For example, in W3, the 128KB read ones take 4× more than the 4KB write ones.



(a) Bandwidth Allocation. **(b) Perf. under a read-congested SSD.** **(c) Perf. under a write-congested SSD.**

Figure 9: We compare two cases that enable (w/) and disable (w/o) the proposed Flint technique to evaluate the multi-tenancy support. (a) presents the achieved bandwidth ratio between two categories of contending I/O streams. (b) and (c) show the average latency of a 32KB read, write, mixed (70% read) I/O stream when running under a read/write-congested SSD.

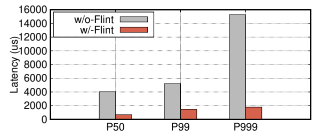
5.4 Flint Mitigates I/O Interference

This experiment demonstrates the effectiveness of the Flint I/O scheduler. We set up two categories of I/O streams on one SSD: the victim ones that issue 4KB read/write I/Os with a small QD; and the background ones that send different types of I/Os with a large QD. Because Flint computes the I/O rank (priority) on the fly and dynamically determines the request issuing order, a low-ranked I/O would then be inserted in the front of the queue, which is dequeued first, mitigating the head-of-line blocking issue. Figure 10 reports the average P50/P99/P999 latency of the victim streams. When 4KB random read co-locating with 128KB random read (Figure 10), Flint helps reduce the P50, P99, and P99 latencies by 4.8×, 2.6×, and 7.5×, respectively. We observe similar trends in the other three cases. Under read/write mixed cases, the latency savings are related to the write cost estimation, which determines the accuracy of computing the write I/O rank. For example, when 4KB writes interleave with 128KB writes, one can achieve 36.4%, 17.7%, and 18.9% lower P50, P99, and P999 latency when enabling Flint, respectively.

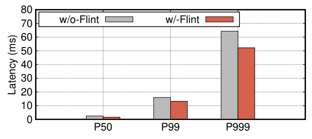
5.5 Flint Avoids Busy EBOF SSDs

Flint is conscious of the device running condition due to the shadow view's capabilities. To demonstrate this, we set up an experiment that pollutes an EBOF SSD using 4 128KB random reads/writes. We then create an eVol over the SSD and launch a 32KB read, write, and mixed (70% read) stream. We disable/enable the dynamic extent remapping technique and I/O scheduler and measure the average latency when increasing the QD. As shown in Figures 9-b and c, Flint consistently reduces latency for scenarios because it eschews the congested SSD and directs I/Os to others. For example, under the read/write congestion, it reduces the average latency by 40.1%/29.8%. Therefore, based on §5.4 and §5.5, one can see that Flint is capable of tackling the third issue (§2.2) and achieves effective multi-tenancy.

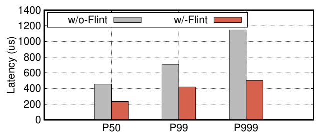
We also compare the performance of eVol and Logical Volume Management (LVM). Figure 7-d shows their bandwidth under three different workloads, where some SSDs are congested. Compared with the workload oblivious placement in LVM, eVol is aware of the device's runtime condition and places extents in a less-loaded drive, yielding 2.3×, 3.0×, 3.8× throughput improvement, respectively.



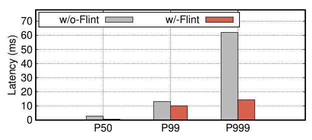
(a) 4KB RD v.s. 128KB RD



(b) 4KB WR v.s. 128KB WR



(c) 4KB RD v.s. 4KB WR



(d) 4KB RD v.s. 128 WR

Figure 10: Latency comparison when consolidating different types of I/O streams. RD/WR=Read/Write.

Primitive	Throughput (MRPS)	P50 (us)	P99 (us)
`view_query`	21.5	24	31
`view_sync`	5.8	38	67

Table 1: The shadow view performance on a storage client.

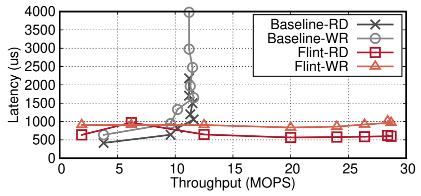
5.6 System deployment

Application integration. Flint is backward-compatible, provides the volume interface, and supports different storage workloads. As a case study, we deploy an object store [12,29] over Flint and see how much benefits it can bring to the application. We use one storage client in this experiment. Figure 11 presents the latency v.s. throughput when running YCSB workloads. When running over an eVol, one can achieve 2.8×, 2.8×, 2.9× throughput improvements, with 66.4(74.6)%, 63.7(73.8%), and 61.9% read(write) latency reduction for YCSB-A, YCSB-B, and YCSB-C, respectively.

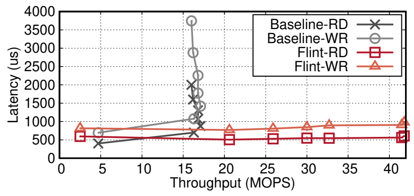
Shadow view. `view_query` and `view_sync` are the two major RPC primitives provided by our shadow view service. Flint uses the Protobuf [9] for serialization, incurring some data processing overheads on both the client and arbiter sides. For example, as shown in Table 1, the average/tail latency of a `view_query` is 24/31us, which could be optimized using hardware-assisted serialization techniques [63].

6 Related Work

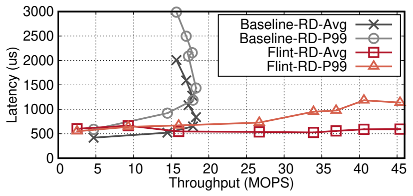
Block Storage. Researchers have built a plethora of distributed block storage systems in different settings. For example, Petal [41] realizes an early vision of network-attached disks through ATM networks that enable adaptive capac-



(a) YCSB-A.



(b) YCSB-B.



(c) YCSB-C.

Figure 11: Performance comparison between the EBOF volume (Baseline) and Flint when holding an object store. We use the YCSB benchmark [20] with the default parameters. We report the average latency for A/B and average/P99 latency for C.

ity/bandwidth scaling. Parallax [51] harnesses the characteristics of a virtualized environment to implement advanced storage features and provides a block interface over storage VMs. Strata [21] develops a scalable storage system over network-attached flash using global data address translation and SDN-assisted storage protocol virtualization. Blizzard [53], built atop FDS [59], provides a cloud-scale virtual drive with some optimizations, such as nested striping and out-of-order write commits. Decibel [58] exposes the *dVols* abstraction and employs a shared-nothing data-plane design, allowing efficient sharing of disaggregated storage class memories (SCMs).

Object Storage. It organizes data as objects and uses a global identifier for addressing. NASD (network-attached secure disks) [27] builds an early version of object store and enables scalable disk bandwidth. OceanStore [39] develops a data object infrastructure that spans the globe and provides continuous access to persistent information. Ceph [77] distributes data replication, failure detection, and recovery to semi-autonomous object storage devices that run a specialized local object file system. Haystack [16] is an object store for the Facebook's photos application and changes costly disk metadata operations into in-memory ones. FDS [59] proposes locality-oblivious blobs over full bisection CLOS networks. F4 [57] designs a hierarchical tier architecture and improves the warm object access performance. Pocket [38] is a distributed object store for serverless applications with fast response, auto-scaling, and smart data placement capabilities.

Distributed File Systems. There is a long history of building distributed file systems. xFS [13] proposes a serverless principle and distributes management and data access services over multiple machines. Frangipani [75] takes a similar design but relies on the underlying Petal [41] to improve system scalability. GFS [26] is a fault-tolerant distributed file system targeting large-scale data processing workloads. pNFS [32] extends NFSv4 with a layout driver, an I/O driver, and a file layout retrieval interface such that clients can perform direct access to remote storage. HDFS [73] serves as the storage backend for the Hadoop data processing framework. NOVA [78] optimizes the file system performance on hybrid memory systems (DRAM+NVM) using a log-structured design. Assise [14] materializes the client-local persistent memory module as a linearizable and crash-recoverable cache and improves the performance of distributed file systems.

Programmable Networks. An EBOF is partially motivated by reconfigurable switches [18, 31, 44, 48, 68, 69] and SmartNICs [19, 25, 28, 46, 49, 60–62, 67], which realize some functionalities of its data path (§3.2). Akin to an in-network telemetry system, our shadow view captures EBOF runtime statistics and serves as an assistant building block for others.

Storage Disaggregation. People have explored disaggregated storage extensively given the rising networking bandwidth and fast remote storage protocol [29, 33, 34, 36, 37, 43, 45, 47, 54–56, 64, 79]. Ana Klimovic *et al.* characterize the performance of iSCSI-based disaggregated storage [36]. Reflex [37] develops a customized kernel-bypass data-plane for remote NVMe accesses. i10 [34] develops an efficient in-kernel TCP/IP remote storage based on dedicated end-to-end IO paths and delayed doorbell notifications. LeapIO [43] unifies the address space across X86 CPUs and ARM coprocessors, exposing a virtual NVMe to an unmodified guest VM. Researchers [30] also report the performance characteristics of server-based NVMe-oF boxes. Gimbal [54] designs an efficient multi-tenancy mechanism to share a SmartNIC JBOF via traditional networking techniques. LEED [29] builds a fast and replicated key-value store over an array of SmartNIC-based storage appliances to improve energy efficiency. RackBlox [64] co-designs the software-defined networking and storage stacks, dividing storage functionalities between the programmable switch and the host I/O stack.

7 Conclusion

This paper explores how to use emerging EBOFs efficiently to serve storage applications. The linchpin is a distributed telemetry system (shadow view) that continuously monitors and reports the EBOF running condition. It enhances the static and opaque EBOF I/O processing pipelines with performance monitoring, resource accounting, and central and global coordination capabilities. Using it, we then design and implement an elastic block storage system over EBOFs and demonstrate considerable benefits compared with defacto schemes from performance, utilization, and multi-tenancy perspectives.

Acknowledgement

We would like to thank the anonymous reviewers and our shepherd, Kang Chen, for their comments and feedback. This work is supported in part by NSF grants CNS-2106199, CNS-2212192, and CAREER-2339755.

References

[1] Rocksdb. https://rocksdb.org/.

[2] Efficient IO with io_uring. https://kernel.dk/io_uring.pdf, 2025.

[3] Enterprise and Datacenter Standard Form Factor. https://www.snia.org/forums/cmsi/knowledge/formfactors#EDSFF, 2025.

[4] Flexible I/O Tester. https://github.com/axboe/fio, 2025.

[5] Fungible FS1600 Platform. https://www.servethehome.com/fungible-dpu-based-storage-cluster-hands-on-look/fungible-fs1600-capacity-and-performance/, 2025.

[6] IEEE 802.3ad Link Aggregation Configuration. https://www.ibm.com/docs/en/aix/7.1?topic=aggregation-ieee-8023ad-link-configuration, 2025.

[7] Ingrasys ES2000/ES2100 Platform. https://www.ingrasys.com/es2000, 2025.

[8] NVMe Express TCP Transport Specification. https://nvmexpress.org/specification/tcp-transport-specification/, 2025.

[9] Protocol Buffers. https://protobuf.dev, 2025.

[10] The rapid growth of io_uring. https://lwn.net/Articles/810414/, 2025.

[11] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for {SSD} performance. In *2008 USENIX Annual Technical Conference (ATC'08)*, 2008.

[12] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (SOSP'09)*, pages 1–14, 2009.

[13] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless Network File Systems. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP'95)*, page 109–126, 1995.

[14] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-Local NVM in a Distributed File System. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*, 2020.

[15] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering Azure Storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI'23)*, pages 49–67, 2023.

[16] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a Needle in Haystack: Facebook's Photo Storage. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*, 2010.

[17] Vaduvur Bharghavan, Alan Demers, Scott Shenker, and Lixia Zhang. MACAW: A Media Access Protocol for Wireless LAN's. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM'94)*, page 212–225, 1994.

[18] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM'13)*, page 99–110, 2013.

[19] Xuzheng Chen, Jie Zhang, Ting Fu, Yifan Shen, Shu Ma, Kun Qian, Lingjun Zhu, Chao Shi, Yin Zhang, Ming Liu, et al. Demystifying datapath accelerator enhanced off-path smartnic. In *2024 IEEE 32nd International Conference on Network Protocols (ICNP'24)*, pages 1–12, 2024.

[20] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC'10)*, pages 143–154, 2010.

[21] Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, Geoffre Lefebvre, Daniel Ferstay, and Andrew Warfield. Strata: High-Performance Scalable Storage on Virtualized Non-volatile Memory. In *12th USENIX Conference on File and Storage Technologies (FAST'14)*, 2014.

[22] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage (Systor'22)*, pages 120–127, 2022.

[23] Dror G Feitelson. Packing schemes for gang scheduling. In *workshop on job scheduling strategies for parallel processing*, pages 89–110, 1996.

[24] Dror G Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and distributed Computing*, 16(4):306–318, 1992.

[25] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure accelerated networking:{SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 51–66, 2018.

[26] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 20–43, 2003.

[27] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, page 92–103, 1998.

[28] Zerui Guo, Jiaxin Lin, Yuebin Bai, Daehyeok Kim, Michael Swift, Aditya Akella, and Ming Liu. LogNIC: A High-Level Performance Model for SmartNICs. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'23)*, page 916–929, 2023.

[29] Zerui Guo, Hua Zhang, Chenxingyu Zhao, Yuebin Bai, Michael Swift, and Ming Liu. LEED: A Low-Power, Fast Persistent Key-Value Store on SmartNIC JBOFs. In *Proceedings of the ACM SIGCOMM 2023 Conference (SIGCOMM'23)*, page 1012–1027, 2023.

[30] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. NVMe-over-Fabrics Performance Characterization and the Path to Low-Overhead Flash Disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference (Systor'17)*, 2017.

[31] Yongchao He, Wenfei Wu, Yanfang Le, Ming Liu, and ChonLam Lao. A Generic Service to Provide In-Network Aggregation for Key-Value Streams. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23), Volume 2*, page 33–47, 2023.

[32] Dean Hildebrand and Peter Honeyman. Exporting storage systems in a scalable manner with pNFS. In *22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'05)*, pages 18–27, 2005.

[33] Wentao Hou, Jie Zhang, Zeke Wang, and Ming Liu. Understanding Routable PCIe Performance for Composable Infrastructures. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI'24)*, pages 297–312, 2024.

[34] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP $\approx$ RDMA: CPU-efficient Remote Storage Access with i10 . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, 2020.

[35] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI'19)*, pages 1–16, 2019.

[36] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (Eurosys'16)*, 2016.

[37] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, page 345–359, 2017.

[38] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 427–444, 2018.

[39] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna

Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*, page 190–201, 2000.

[40] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*, page 302–313, 1994.

[41] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, page 84–92, 1996.

[42] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA'92)*, page 92–103, 1992.

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

[44] Ming Liu. *Building Distributed Systems Using Programmable Networks*. University of Washington, 2020.

[45] Ming Liu. Fabric-Centric Computing. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS'23)*, page 118–126, 2023.

[46] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*, page 318–333, 2019.

[47] Ming Liu, Arvind Krishnamurthy, Harsha V. Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, and Anoop Jawahar. Fine-Grained Replicated State Machines for a Cluster Storage System . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 305–323, 2020.

[48] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, page 795–809, 2017.

[49] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*, pages 363–378, 2019.

[50] Hui Lu, Brendan Saltaformaggio, Ramana Kompella, and Dongyan Xu. vFair: latency-aware fair storage scheduling via per-IO cost-based differentiation. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC'15)*, pages 125–138, 2015.

[51] Dutch T. Meyer, Gitika Aggarwal, Brendan Cully, Geoffrey Lefebvre, Michael J. Feeley, Norman C. Hutchinson, and Andrew Warfield. Parallax: Virtual Disks for Virtual Machines. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys'08)*, page 41–54, 2008.

[52] Rui Miao, Lingjun Zhu, Shu Ma, Kun Qian, Shujun Zhuang, Bo Li, Shuguang Cheng, Jiaqi Gao, Yan Zhuang, Pengcheng Zhang, Rong Liu, Chao Shi, Binzhang Fu, Jiaji Zhu, Jiesheng Wu, Dennis Cai, and Hongqiang Harry Liu. From Luna to Solar: The Evolutions of the Compute-to-Storage Networks in Alibaba Cloud. In *Proceedings of the ACM SIGCOMM 2022 Conference (SIGCOMM'22)*, page 753–766, 2022.

[53] James Mickens, Edmund B Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, pages 257–273, 2014.

[54] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. Gimbal: enabling multi-tenant storage disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*, page 106–122, 2021.

[55] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: An elastic zoned namespace for commodity ZNS SSDs. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI'23)*, pages 461–477, 2023.

[56] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. eZNS: Elastic Zoned Namespace for Enhanced Performance Isolation and Device Utilization. *ACM Trans. Storage*, 20(3), June 2024.

[57] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's Warm BLOB Storage System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 383–398, 2014.

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

[59] Edmund B Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*, pages 1–15, 2012.

[60] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 663–679, 2018.

[61] Yiming Qiu, Qiao Kang, Ming Liu, and Ang Chen. Clara: Performance Clarity for SmartNIC Offloading. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets'20)*, page 16–22, 2020.

[62] Yiming Qiu, Jiarong Xing, Kuo-Feng Hsu, Qiao Kang, Ming Liu, Srinivas Narayana, and Ang Chen. Automated SmartNIC Offloading Insights for Network Functions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, page 772–787, 2021.

[63] Deepti Raghavan, Shreya Ravi, Gina Yuan, Pratiksha Thaker, Sanjari Srivastava, Micah Murray, Pedro Henrique Penna, Amy Ousterhout, Philip Levis, Matei Zaharia, and Irene Zhang. Cornflakes: Zero-Copy Serialization for Microsecond-Scale Networking. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, page 200–215, 2023.

[64] Benjamin Reidys, Yuqi Xue, Daixuan Li, Bharat Sukhwani, Wen-Mei Hwu, Deming Chen, Sameh Asaad, and Jian Huang. RackBlox: A Software-Defined Rack-Scale Storage System with Network-Storage Co-Design. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*, page 182–199, 2023.

[65] Robbert Van Renesse and Fred B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *6th Symposium on Operating Systems Design & Implementation (OSDI'04)*, 2004.

[66] Jordi Ros and Wei Kang Tsai. A theory of convergence order of maxmin rate allocation and an optimal protocol. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 2, pages 717–726, 2001.

[67] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-Accelerated Distributed Transactions. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*, page 740–755, 2021.

[68] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, pages 1–16, 2018.

[69] Naveen Kr. Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable Calendar Queues for High-speed Packet Scheduling . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 685–699, 2020.

[70] Kai Shen and Stan Park. FlashFQ: A fair queueing I/O scheduler for Flash-Based SSDs. In *2013 USENIX Annual Technical Conference (USENIX ATC'13)*, pages 67–78, 2013.

[71] M. Shreedhar and George Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM'95)*, page 231–242, 1995.

[72] Junyi Shu, Kun Qian, Ennan Zhai, Xuanzhe Liu, and Xin Jin. Burstable Cloud Block Storage with Data Processing Units. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI'24)*, pages 783–799, 2024.

[73] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[74] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. {MQSim}: A framework for enabling realistic studies

of modern {Multi-Queue}{SSD} devices. In *16th USENIX Conference on File and Storage Technologies (FAST'18)*, pages 49–66, 2018.

[75] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP'97)*, page 224–237, 1997.

[76] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI'20)*, pages 449–462, 2020.

[77] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006.

[78] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST'16)*, pages 323–338, 2016.

[79] Chenxingyu Zhao, Tapan Chugh, Jaehong Min, Ming Liu, and Arvind Krishnamurthy. Dremel: Adaptive Configuration Tuning of RocksDB KV-Store. *Proc. ACM Meas. Anal. Comput. Syst.*, 6(2), June 2022.