RackBlox: A Software-Defined Rack-Scale Storage System with Network-Storage Co-Design

Benjamin Reidys
University of Illinois
Urbana-Champaign, USA

Yuqi Xue
University of Illinois
Urbana-Champaign, USA

Daixuan Li
University of Illinois
Urbana-Champaign, USA

Bharat Sukhwani
IBM T. J. Watson Research Center,
Yorktown Heights, USA

Wen-mei Hwu
University of Illinois
Urbana-Champaign, USA

Deming Chen
University of Illinois
Urbana-Champaign, USA

Sameh Asaad
IBM T. J. Watson Research Center,
Yorktown Heights, USA

Jian Huang
University of Illinois
Urbana-Champaign, USA

Abstract

Software-defined networking (SDN) and software-defined flash (SDF) have been serving as the backbone of modern data centers. They are managed separately to handle I/O requests. At first glance, this is a reasonable design by following the rack-scale hierarchical design principles. However, it suffers from suboptimal end-to-end performance, due to the lack of coordination between SDN and SDF.

In this paper, we co-design the SDN and SDF stack by re-defining the functions of their control plane and data plane, and splitting up them within a new architecture named RackBlox. RackBlox decouples the storage management functions of flash-based solid-state drives (SSDs), and allow the SDN to track and manage the states of SSDs in a rack. Therefore, we can enable the state sharing between SDN and SDF, and facilitate global storage resource management. RackBlox has three major components: (1) coordinated I/O scheduling, in which it dynamically adjusts the I/O scheduling in the storage stack with the measured and predicted network latency, such that it can coordinate the effort of I/O scheduling across the network and storage stack for achieving predictable end-to-end performance; (2) coordinated garbage collection (GC), in which it will coordinate the GC activities across the SSDs in a rack to minimize their impact on incoming I/O requests; (3) rack-scale wear leveling, in which it enables global wear leveling among SSDs in a rack by periodically swapping data, for achieving improved device lifetime for the entire rack. We implement RackBlox using programmable SSDs and switch. Our experiments demonstrate that RackBlox can reduce the tail latency of I/O requests by up to 5.8× over state-of-the-art rack-scale storage systems.

CCS Concepts: • **Networks** → **Programmable networks**; • **Computer systems organization** → **Cloud computing**; **Secondary storage organization**.

Keywords: Network/Storage Co-Design, Software-Defined Networking, Software-Defined Flash, Rack-Scale Storage.

ACM Reference Format:
Benjamin Reidys, Yuqi Xue, Daixuan Li, Bharat Sukhwani, Wen-mei Hwu, Deming Chen, Sameh Asaad, and Jian Huang. 2023. RackBlox: A Software-Defined Rack-Scale Storage System with Network-Storage Co-Design. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3600006.3613170

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10…$15.00
https://doi.org/10.1145/3600006.3613170

1 Introduction

The software-defined infrastructure has become the new standard for managing data centers, as it provides flexibility and agility for platform operators to customize hardware resources for applications [15, 35, 65]. As the backbone technology, software-defined networking (SDN) allows network operators to configure and manage network resources through programmable switches [14, 33, 34, 48]. Since SDN has demonstrated its benefits, software-defined storage (SDS) [65, 79, 90] has also been developed. A typical example is software-defined flash (SDF) [28, 53, 65, 71].

Similar to SDN, SDF enables upper-level software to manage the low-level flash chips for improved performance and resource utilization [28, 44, 65]. Since the cost of flash chips has dramatically decreased while offering orders of magnitude better performance than conventional hard disk drives (HDDs), they are becoming the mainstream choice in large-scale data centers [25, 38, 43].

Both SDN and SDF have their own control plane and data plane, and provide programmability for developers to define and implement their policies for resource management and scheduling. However, SDN and SDF are managed separately in modern data centers. At first glance, this is reasonable by following the rack-scale hierarchical design principles. However, it suffers from suboptimal end-to-end performance, due to the lack of coordination between SDN and SDF.

Although both SDN and SDF can make the best effort to achieve their quality of service, they do not share their states and lack global information for storage management and scheduling, making it challenging for applications to achieve predictable end-to-end performance. Prior studies [6, 79] have proposed various software techniques such as token bucket and virtual cost for enforcing performance isolation across the rack-scale storage stack. However, they treat the underlying SSDs as black boxes, and cannot capture their hardware events, such as garbage collection (GC) and I/O scheduling in the storage stack. Thus, it is still hard to achieve predictable performance across the entire rack.

In this paper, we propose a new software-defined architecture, named RackBlox, to exploit the capabilities of SDN and SDF in a coordinated fashion. As both SDN and SDF share a similar architecture–the control plane is responsible for managing the programmable devices, and the data plane is responsible for processing I/O requests–we can integrate and co-design both SDN and SDF, and redefine their functions to improve the efficiency of the entire rack-scale storage system. RackBlox does not require new hardware changes, as both SDN and SDF today have offered the flexibility to redefine the functions of their data planes.

To develop RackBlox, we first decouple the functions of the storage management (i.e., flash translation layer) of SDF, and integrate appropriate functions such as garbage collection and wear leveling into the control plane of top-of-rack (ToR) switches in the SDN. Such a new software-defined architecture enables state sharing between SDN and SDF, and facilitates the global storage resource management in a rack. This is compatible with storage virtualization by enabling the state tracking of virtualized SSD instances in SDN. Therefore, we can coordinate the efforts of I/O scheduling across the entire rack. RackBlox tracks the elapsed time in the programmable switches with In-band Network Telemetry (INT) [1], adapts the I/O scheduling in the data plane of SDF, and predicts the response time from the storage devices back to the client. Therefore, RackBlox can manage the end-to-end latency and offer predictable performance.

RackBlox further enables coordinated GC among the rack of SSDs to minimize the impact of GC on application performance. RackBlox has the global information of the storage states, which provides the convenience to coordinate GC events among all the SSDs in a rack. Upon GC events, RackBlox takes advantage of the data replicas in the same rack, and enables the ToR switch to redirect I/O requests to the other data replicas. Therefore, the expensive GC activities can be alleviated from the critical path. RackBlox employs different GC policies for different performance isolation guarantees of virtualized SSD instances.

RackBlox also enables rack-scale wear leveling to ensure a uniform lifetime of SSDs in a rack. As the write traffic to each SSD can be different, it will cause wear imbalance between SSDs. In addition, platform operators have to replace unhealthy or failed SSDs with new SSDs, making the wear imbalance even harder to manage. RackBlox develops a two-level wear leveling mechanism. It balances wear within each individual SSD in a storage server as well as across SSDs in the rack. Instead of swapping SSDs frequently, RackBlox periodically swaps the SSD that has incurred the maximum wear with the SSD that has the minimum rate of wear.

We implement RackBlox with a programmable Tofino switch and programmable SSDs (i.e., open-channel SSDs). We evaluate RackBlox with network traces collected from various data centers and a variety of data-intensive applications. Our experiments show that RackBlox reduces the tail latency of end-to-end I/O requests by up to 5.8×, and can achieve a uniform lifetime for a rack of SSDs without introducing much additional performance overhead. In summary, we make the following contributions in this paper.

- We propose a new software-defined rack-scale storage system by decoupling the storage management functions of SDF, and co-designing them with SDN.
- We enable state sharing between SDN and SDF, and coordinate the efforts of I/O request scheduling across the full rack for achieving predictable end-to-end performance.
- We present a coordinated GC mechanism for a rack of SSDs, it enables SDN to redirect I/O requests to data replicas to minimize the GC impact on storage performance.
- We develop a rack-scale wear leveling mechanism for ensuring the uniform lifetime of a rack of SSDs.
- We show the benefits of RackBlox by developing a real system prototype with programmable switch and SSDs.

2 Background and Motivation

We first introduce the background of SDN and SDS, then the motivation for software-defined network/storage co-design.

2.1 Software-Defined Networking

Modern data centers have seen a trend that software-defined networking (SDN) has become the new standard for network management, in which the programmable switch is the backbone technology that allows platform operators to define their own packet formats and functions for processing network traffic without affecting the line-rate throughput [34, 36]. SDN has been deployed in real data centers such as Alibaba cloud [47, 66] and Google data centers [5].

SDN has a control plane and data plane. The control plane is in charge of network management and protocol definition,

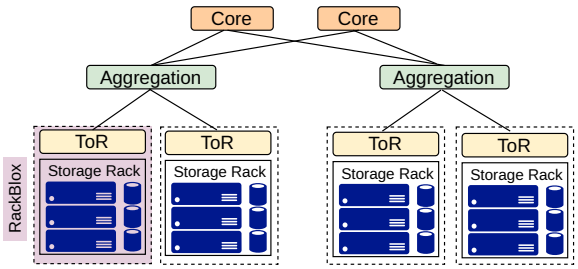


Figure 1. System overview of a rack-scale storage system.

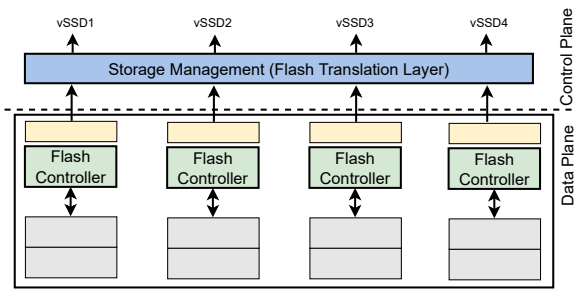


Figure 2. System architecture of software-defined flash.

while the data plane is responsible for data transfer and runtime statistics collection. The programmable switch usually has reconfigurable hardware such as a programmable ASIC that supports domain-specific languages like P4 [14]. It supports various network flow scheduling policies for flexible traffic management and performance isolation [4, 27, 89]. As shown in Figure 1, all the servers in the same rack are connected by a Top-of-Rack (ToR) switch. These ToR switches are connected with aggregation switches and core switches in a hierarchical manner. In this paper, we focus on the ToR switch, and co-design network and storage stack in a rack.

2.2 Software-Defined Storage

Recent studies have shown that making software aware of the underlying storage devices can significantly improve the storage performance and resource efficiency [28, 65, 84]. This is known as software-defined storage (SDS), which enables data centers to unlock the potential of storage devices by enabling the software to directly interact with storage devices and control their internal operations. Software-defined flash (SDF), which is built on SSDs, is a typical example of SDS, and has seen deployment in industry data centers [18, 49].

In this paper, we focus on SDF, because flash-based SSDs are becoming indispensable parts of modern computer systems. An SSD has three major components: a set of flash memory packages, an SSD controller having an embedded processor with device memory, and flash controllers. As shown in Figure 2, each SSD has multiple channels and each channel can receive and process I/O commands independently. Each channel is shared by multiple flash memory packages. Each package is made of multiple chips. Each chip has multiple flash blocks. With SDF, an SSD can be virtualized into multiple virtual SSD instances (vSSDs), and each can be mapped to a set of SSDs, flash channels, or flash chips.

Due to the nature of flash memory, when a free page is written, that page is no longer available for future writes until that page is erased. However, erase operations can be performed only at block granularity, which are time-consuming. Thus, writes are issued to free pages erased in advance (i.e., out-of-place write) rather than waiting for the expensive erase operation for every write. And garbage collection (GC) will be performed later to erase the stale data on SSDs. Since an SSD channel cannot issue new I/O requests during GC, minimizing the negative impact of GC events is critical to storage performance. In addition, as each flash block has limited endurance, it is important for the blocks to age uniformly (i.e., wear leveling).

SSDs have the Flash Translation Layer (FTL) to manage flash blocks and maintain the logical-to-physical address mappings. Unlike conventional SSDs that implement the FTL in the device firmware, SDF exposes the FTL to the upper-level software, and enables the software to manage the flash chips (see Figure 2).

2.3 Why Network-Storage Co-Design

In modern data centers, the SDN and SDF are managed separately to handle I/O requests across the network and storage stack, respectively. Such an architecture suffers from suboptimal performance and misses the opportunities offered by the software-defined rack for three major reasons.

First, as SDN and SDF are deployed as independent components, achieving predictable end-to-end performance is challenging, due to the lack of coordination between the two components. SDN and SDF have redundant control plane policies, such as I/O scheduling, which may contradict between the network and storage stack and break service-level objectives (SLOs). And optimizing such policies without coordination is suboptimal due to incomplete knowledge and redundant effort. Ideally, as we forward I/O requests in SDN, with the knowledge of the storage status (e.g., busy, idle, or predicted performance), it can make smarter decisions (e.g., early redirection to data replicas). Similarly, as SDF schedules the received I/O requests, the measured network latency of these I/O requests can help the SDF to adjust the I/O scheduling to meet the SLO for end users.

Second, although prior studies such as IOFlow [79] and VDC [6] proposed software-based methods like token bucket rate limiting to enforce the performance isolation between I/O flows, they cannot capture the underlying hardware events such as GC and I/O scheduling in SSDs, due to the lack of state sharing between SDN and SDF. And software-based coordination incurs extra network round-trip delay and host software overhead (see our evaluation in §4).

Third, it is feasible to co-design and coordinate the network and storage stack today, as both programmable switches and programmable SSDs have enabled developers to program and configure the network and storage stack respectively.

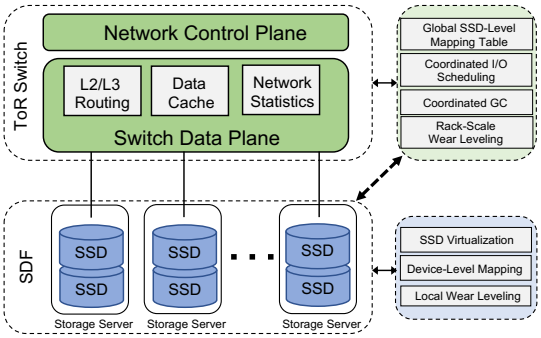


Figure 3. System overview of RackBlox.

In this work, we integrate the storage management of SDF into SDN as shown in Figure 3, while preserving the programmability and simplicity for the new infrastructure.

3 RackBlox Design and Implementation

RackBlox provides a holistic approach that can achieve predictable end-to-end performance and improve storage management at scale. As we develop RackBlox based on modern SDN and SDF, we have to overcome the following challenges.

3.1 Design Challenges of RackBlox

- It is unclear how the functions of storage management should be decoupled and placed across SDN and SDF. The control plane of SDF has many functions, including wear leveling, GC, block allocation, and block management. Placing all the storage functions into SDN will inevitably increase the burden of SDN. Thus, we have to carefully decide the partitioning and placement of SDF functions.
- The hardware resources of programmable devices are limited. Specifically, the on-chip memory (tens of MBs) and compute resource are limited in programmable switches and SSD controllers, due to the hardware cost and power budget. Thus, we have to carefully define the data structures for the network/storage co-design.
- As we enable the coordinated storage management between SDN and SDF, we must preserve their programmability, ease-of-use, and original advantages. Thus, RackBlox should be compatible with hardware upgrades.

3.2 RackBlox Overview

We rethink the software-defined network and storage hierarchy (see Figure 3), and propose a new software-defined architecture, RackBlox. We first decouple the functions of the storage management (i.e., flash translation layer) of SSDs, and integrate the appropriate functions such as GC and wear-leveling into the SDN (§3.3). Such a new architecture enables state sharing between SDN and SDF. It utilizes the capability of SDN to enable global storage resource management in a rack. Thus, we can coordinate the efforts of I/O scheduling across the entire rack. Henceforth, the SDN and SDF can manage the end-to-end request delay, and provide precise feedback to the I/O scheduler on the storage servers (§3.4). The coordinated I/O scheduling mechanism improves the

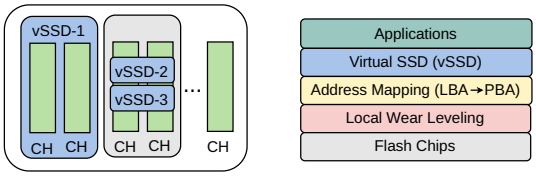


Figure 4. The structure of virtualized SSDs. RackBlox supports both hardware-isolated and software-isolated vSSDs.

end-to-end I/O performance and enables intelligent decision-making in advance. To alleviate the performance interference caused by the GC, RackBlox enables coordinated GC by exploiting the data replicas in a rack (§3.5). As the ToR switch has the global states of the SSDs in a rack, it can redirect I/O requests to the corresponding replica upon GC. Similarly, RackBlox enables rack-scale wear leveling, as it has the knowledge of the wear of SSDs in a rack (§3.6). It has a two-level wear leveling mechanism: a local wear balancer for ensuring the wear balance in each storage server, and a global wear balancer that reduces the wear variance across the entire rack. These wear balancers work at different levels and cooperate to ensure rack-scale wear leveling.

RackBlox manages SSDs at rack-scale for three major reasons. First, storage systems are commonly deployed at rack scale, making this a natural granularity for storage management [7, 23]. Second, rack-scale management is facilitated by the programmable ToR-switch with the capability to observe the rack's network and I/O traffic. Third, existing rack-aware replica placement schemes make it a natural choice for coordinating the GC of SSDs across the rack. We now discuss each proposed technique in RackBlox as follows.

3.3 Decoupling the Storage Management

When decoupling storage management, we need to consider two factors: (1) whether integrating an SDF function into SDN will benefit from the coordination or not; and (2) if yes, the integration should consume minimum precious hardware resources in the switch. We now discuss how RackBlox decouples storage management between SDF and SDN to maximize the benefits of co-design while retaining the original flexibility and modularity of SDN and SDF.

Storage management in SDF. As the ToR switch has limited hardware resources, we keep the essential functions for the vSSD management locally on storage servers (see Figure 3). They include SSD virtualization, device-level mapping, and local wear leveling for SSDs in a server.

With SSD virtualization, a programmable SSD can be virtualized into two types of vSSDs: hardware-isolated vSSDs, and software-isolated vSSDs. A hardware-isolated vSSD instance is mapped to a set of flash channels, as the channel-level parallelism in SSDs provides the strongest performance isolation (vSSD-1 in Figure 4). A software-isolated vSSD is mapped to a set of flash chips, and it will share the flash channels with other software-isolated vSSDs, such as vSSD-2 and

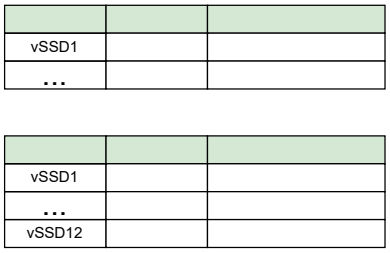


Figure 5. RackBlox tables placed in the ToR switch.

vSSD-3 shown in Figure 4. It relies on the software-isolation techniques such as token bucket rate limiting to offer relatively weaker performance isolation. RackBlox supports both hardware-isolated and software-isolated vSSD instances to support different cloud storage services.

For each vSSD, it has its own address mapping table (device-level mapping) and local wear leveling (i.e., the default wear leveling) for flash block management, as shown in Figure 4b. We keep these functions in the SDF stack, as they are more convenient when handled by storage servers. As for other FTL functions, such as bad block management and error correction code (ECC) of an SSD, we leave them to the SSD firmware, as the hardware engine in SSD controllers is more efficient in managing them.

RackBlox enables data replication at vSSD granularity. This is a natural design choice, as the vSSD abstraction has been shown to simplify the storage management of flash blocks, offer flexibility for mapping vSSD instances to underlying flash chips, and incur limited metadata overhead [28].

Storage management in ToR switch. To make the ToR switch aware of the states of SSDs in a rack, we maintain a vSSD-level mapping table in its data plane, as shown in Figure 5. The states tracked in the vSSD-level mapping tables provide the essential knowledge for the coordinated I/O scheduling and coordinated GC.

Specifically, RackBlox maintains two tables in the ToR switch as shown in Figure 5: **1)** replica table, which tracks the GC status (1 byte) of each vSSD and its replica vSSD ID (4 bytes); **2)** destination table, which mainly tracks the corresponding server IP (4 bytes) of each vSSD, and the GC status (1 byte) of the vSSD. As this mapping table is managed at vSSD granularity, its storage cost is small, which can be stored in the on-chip memory of the programmable switch. Given that a rack usually has 64 servers or less, each server has 16 SSDs, and each SSD can be virtualized into 128 vSSDs, we will have up to 64K vSSDs in a rack[1]. The maximum size of each table is 1.3MB. The total size of these tables for RackBlox is much less than the available SRAM capacity (tens of MBs) in modern programmable switches.

[1]A typical server in data centers today has 16 PCIe slots, it can host 16 SSDs. Assume each SSD has 4TB, the minimum size of a vSSD is 32GB, therefore, each SSD can host up to 128 vSSDs.

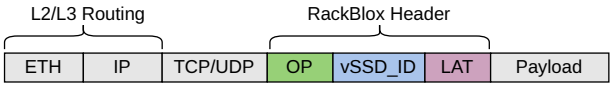


Figure 6. The network packet format in RackBlox.

Table 1. Network protocols used in RackBlox.

Operation Name	Description
`create_vssd`	Register a newly created vSSD in the ToR switch.
`del_vssd`	Remove a registered vSSD from the tables.
`write`	Write issued by client.
`read`	Read issued by client.
`gc_op`	Packet to update GC for vSSD.

State communication between SDN and SDF. To facilitate the state communication between the ToR switch and storage servers in the same rack, RackBlox leverages the programmability of SDN, and has its own network packet format based on regular network protocols, as shown in Figure 6. The packet has one 1-byte *OP* field to indicate different operations as shown in Table 1, one 4-byte field to indicate the target vSSD ID, and one 4-byte field (*LAT*) for storing the measured network latency for the packets transferred through the data center network. The payload will be filled with different values, according to the operation specified in RackBlox header. We will discuss the purpose of each operation throughout the paper.

The RackBlox header is part of the L4 payload. RackBlox uses existing L2/L3 routing protocols to route packets. As such, switches can forward RackBlox packets normally, and RackBlox is compatible with flow/congestion control and other network functionalities in the transport layer. We differentiate RackBlox packets in the ToR switch through a reserved TCP/UDP port.

To initialize RackBlox tables in the ToR switch, the storage servers send a packet that contains the *create_vssd* operation to the switch upon creating a new vSSD. The *vSSD_ID* field will store the ID of the newly created vSSD, the payload will include the 4-byte server IP, its replica vSSD ID, and the server IP of the replica vSSD. The replica vSSD ID and IP are allocated with the vSSD–following the rack-aware replica placement scheme in rack-scale storage systems. The ToR switch will insert a new entry in the replica table and destination table, with GC states initialized as 0 (idle). Upon vSSD deletion, the storage server sends the *del_vssd* packet to the ToR switch to remove the corresponding entries in the RackBlox tables. As we serve I/O requests at runtime, RackBlox tables will be updated depending on the events.

3.4 Coordinated I/O Scheduling

Although both SDN and SDF can make the best effort to achieve their quality of service (QoS), the lack of state sharing and coordination will cause suboptimal end-to-end performance and wasted effort on I/O scheduling. For instance, as SDN and SDF are independent, they do not share states of I/O requests, therefore, SDN may forward I/O requests to busy storage servers, although it is apparent that their

end-to-end service-level objective is likely to be violated. This will exacerbate network congestion and increase the pressure of processing I/O requests on storage servers.

RackBlox enables the state sharing of I/O requests across SDN and SDF, and develops a coordinated I/O scheduling mechanism to improve end-to-end performance. It tracks the elapsed time in the programmable switches, adapts the I/O scheduling in the data plane of SDF to control the end-to-end delay for the request, and predicts the time it would take to transmit the response from the storage server to the client.

RackBlox tracks I/O requests across the entire stack: (1) Net_{time}: the elapsed time in the network stack since the I/O request is issued until it reaches the storage server; (2) $Storage_{time}$: the delayed time in the I/O queue of the storage stack; (3) $Predict_{time}$: the time it takes to transfer the response back to the client over the network. To manage I/O scheduling in SDF, RackBlox uses $Prio_{sched}$ = (Net_{time} + $Storage_{time}$+ $Predict_{time}$) as the scheduling priority. As RackBlox issues I/O requests from the queue in the storage stack, it selects the request with the maximum $Prio_{sched}$ value. RackBlox differs from state-of-the-art storage I/O scheduling schemes by considering the network latency to make the best effort to reduce the end-to-end latency [17, 19].

In order to track Net_{time} with low overhead, we use the In-band Network Telemetry (INT) available in programmable switches [1]. It enables the network state collection in the data plane without intervention from the control plane. RackBlox uses INT to compute the sum of per-hop latency in the switches, since the routing and queuing latencies dominate the network latency [24, 29]. It embeds the measured network latency in the network packet being transferred to the storage server, following the network format in Figure 6. As for $Storage_{time}$, RackBlox tracks the queuing delay for each I/O request in the queue of the storage stack.

To predict the time it will take to return the response to the client ($Predict_{time}$), we develop a predictor using a simple yet effective sliding window algorithm. We track one sliding window for each vSSD with the average network latency of the 100 most recent incoming packets. We choose 100 packets because it is small enough to quickly detect changes in the network (e.g., network congestion), but large enough to smoothen outlier requests. We use incoming packets because they can better capture the factors causing network delays. We maintain separate windows for reads and writes as their outgoing packet sizes are different [58, 69, 92].

Our experiments with a variety of network traces in data centers [32, 59, 67] (see §3.7 for details) show that this approach effectively predicts the return latency. The predicted latency is within 25μs of the correct value 95% of the time (across all distributions) and 86.6% of the time in the worst case. The predictions are within 10% of the true latency in the worst case. Mispredictions primarily occur at the begin/end of congestion or with highly variables network patterns. We show the benefit of the coordinated I/O scheduling in §4.

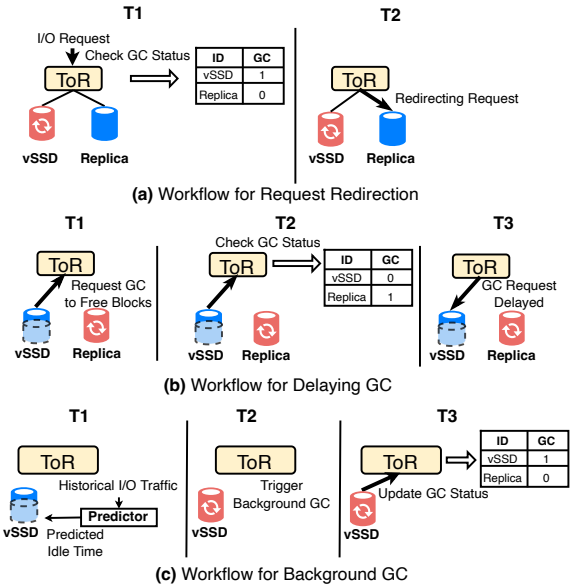


Figure 7. Coordinated GC optimizations in RackBlox.

3.5 Coordinated Garbage Collection

The GC overhead of SSDs is significant, as it blocks incoming I/O requests and seriously harms end-to-end latency [50, 64]. For instance, a 4KB read request in SSDs can be completed in under 100 μs, but it may wait for a few milliseconds due to the GC. This is critical in data centers, where many applications have strict performance requirements [13].

The fundamental issue is that the upper-level system software fails to consider the underlying SSD behavior. Without coordination at rack scale, it is hard to optimize GC across replicas or redirect requests away from replicas executing GC, even though SDF exposes the underlying storage behavior. Since the ToR switch will forward each storage request entering the rack, it has the states of the SSDs in the rack, it is natural to coordinate GC across the SSDs with the switch.

Prior work has explored various techniques for coordinating GC between SSDs within servers [39–41, 45, 75, 78, 88]. These studies managed SSDs either spatially by reserving spare SSDs to serve requests or temporally by scheduling GC to ensure predictable latency for read requests. However, they did not enable GC coordination across servers at rack scale. Industry has been developing rack-scale storage solutions [16, 26, 68, 76], however, to the best of our knowledge, they also lack GC coordination across data replicas.

As different levels of vSSD isolation (software vs. hardware) have different challenges, we will begin with coordinating GC for hardware-isolated vSSD instances and then extend it to software-isolated vSSDs.

3.5.1 Coordinated GC for Hardware-Isolated vSSDs.

RackBlox coordinates GC between the replicas of each vSSD, as shown in Figure 7. Since each hardware-isolated vSSD is mapped to one or more flash channels that run GC independently, GC in other vSSDs does not affect its performance.

Thus, by coordinating GC per vSSD, we can achieve predictable performance. Coordinating GC at vSSD granularity also aligns with the granularity of data replication. If we keep at least one vSSD replica idle, we can ensure that one data replica can be accessed with predictable performance.

For each vSSD, we achieve predictable performance for storage requests by enabling request redirection (Figure 7a). While requests are directed to idle replicas, other replicas of the vSSD may still run GC. Thus, the switch delays GC for one replica until another is ready to serve requests (Figure 7b). Also, RackBlox enables background GC to utilize the idle cycles of SSDs (Figure 7c).

We outline the packet processing workflow of SDN and SDF in Algorithm 1 and Algorithm 2. We examine RackBlox with two replicas in a single rack and one replica in another following the common rack-scale storage systems [7–9, 23, 56]. RackBlox can be extended to any number of replicas.

Request Redirection. Upon receiving a packet, RackBlox queries the Replica Table (see §3.3) to get the *gc_status* and *replica* for *vssd_id*. Writes are not redirected (Line 2-3 in Algorithm 1), but issued to all replicas for reliability and consistency [7, 22, 23, 56]. RackBlox supports different consistency models, and our implementation uses Hermes [37] to ensure strong consistency between replicas and correctness when redirecting requests. We avoid long tail latencies for writes by utilizing existing DRAM caches in data center servers to absorb writes during GC [22, 45]. This follows the durability semantics of existing systems that primarily rely on replicas to ensure data durability [46, 63]. Writes are considered complete when all replicas have a DRAM copy and are flushed in the background (Line 2-4 in Algorithm 2).

For read requests, if the *gc_status* is set for the *vssd_id*, we query the Destination Table to get the *gc_status* for the replica vSSD (T1 in Figure 7a). If the vSSD is not executing GC or if both the vSSD and its replica are executing GC, we forward the packet as is. Otherwise, we redirect the request to the replica using its destination IP in the Destination Table (T2 in Figure 7a, Line 4-9 in Algorithm 1). I/O requests are scheduled locally with coordinated I/O scheduling (Line 5-6 in Algorithm 2). By redirecting requests, requests are served by replicas without suffering from the GC overhead.

While RackBlox maximizes the chance that at least one replica is available, it is possible that both replicas are executing GC. The techniques that submit requests to another rack in parallel can be applied to ensure high performance [20]. In this paper, we focus on the intra-rack I/O scheduling.

Delaying GC. Since all replicas receive the same writes, replicas may execute GC at the same time [45]. Therefore, naive request redirection cannot alleviate the GC overhead. To overcome this issue, we leverage the shared states in the switch and empower the switch to delay the GC of a replica.

While delaying GC can ensure that two replicas do not execute their GC simultaneously, we cannot delay indefinitely. GC is typically executed when the available free blocks fall below a fixed *gc_threshold* (e.g., 25%). This is a hard threshold to free flash blocks for future use. To make room for delaying GC, we configure a relaxed *soft_threshold* (35% by default). Instead of having the SDF notify the switch when it must do GC, it requests GC once its free block ratio falls below the *soft_threshold*. The switch can use its shared GC state to delay GC until the replica finishes GC.

Storage servers will periodically (every 30 seconds by default) check the free block ratio for each vSSD (Line 9-19 in Algorithm 2). If any GC condition triggers, the SDF will send a *gc_op* packet (T1 in Figure 7b). If the free block ratio falls below the *gc_threshold*, the *gc* field in the payload is set to *regular* (value of 1) to indicate that the replica must execute GC. GC requests with *regular* will not be denied as the GC has been delayed as much as possible. If *regular* GC requests are not acknowledged due to link or switch failure, the vSSD will execute GC after retrying (3 retries by default). If the free block ratio only falls below the *soft_threshold*, the *gc* field in the payload is set to *soft* (value of 0).

The logic for *accepting* or *delaying* GC requests in the switch is shown in Line 10-25 of Algorithm 1. The switch begins with updating the GC status in the Replica Table to

Algorithm 1: RackBlox workflow in SDN

```
Input: pkt ← RackBlox packet
       gc_status ← GC status
       dst ← table mapping vssd_id to its destination IP
       replica ← replica for this vssd_id
1  Function process_packet(pkt, gc_status, replica):
2      if pkt.op = write then
3          forward(pkt)
4      if pkt.op = read then
5          if gc_status[pkt.vssd_id] = 1 then
6              if gc_status[replica] = 0 then
7                  pkt.dst ← dst[replica]
8                  pkt.vssd_id ← replica
9          forward(pkt)
10     if pkt.op = gc_op then
11         gc_status[pkt.vssd_id] ← 1
12         if pkt.gc = soft then
               // requires recirculation
13             if gc_status[replica] = 1 then
14                 pkt.gc ← delay
15                 gc_status[pkt.vssd_id] ← 0
16             else
17                 pkt.gc ← accept
18                 dst_gc_status[pkt.vssd_id] ← 1
19         else if pkt.gc = finish then
20             gc_status[pkt.vssd_id] ← 0
21         else
22             dst_gc_status[pkt.vssd_id] ← 1
23             pkt.gc ← accept
24         pkt.dst ← pkt.src
25         forward(pkt)
```

Algorithm 2: RackBlox workflow in SDF

```
   Input: pkt ← RackBlox packet
1  Function process_packet(pkt):
2      if pkt.op = write then
3          if cache is full then
4              flush DRAM cache with write data
5      if pkt.op = read then
6          schedule local read with coordinated I/O
7      if pkt.op = gc_op and pkt.gc = accept then
8          begin GC
   // Periodic GC Monitoring
   Input: vssd ← vSSD being checked for GC
          soft_threshold ← soft GC threshold
          gc_threshold ← regular GC threshold
          bg_pred ← idle prediction for background GC
9  Function trigger_gc(vssd, bg_pred):
10     gc_type ← none
11     if vssd.free_blocks < gc_threshold then
12         gc_type ← regular
13     else if vssd.free_blocks < soft_threshold then
14         gc_type ← soft
15     else if bg_pred = True then
16         gc_type ← bg
17     if gc_type ≠ none then
18         pkt ← new pkt
19         pkt.dst, pkt.op, pkt.gc ← switch, gc_op, gc_type
```

1. If the request is *regular*, the switch also updates the GC status in the Destination Table to 1, sets the *gc* field in the payload to *accept* (value of 3), and sends the reply back to the server. For *soft* requests, the switch will check the GC status of the replica (T2 in Figure 7b). If the replica is executing GC, the switch will *delay* (value of 4) the request (T3 in Figure 7b). Otherwise, the switch will *accept* it. Both the Replica and Destination Tables have a GC status that must be consistent. The *soft* requests that must check the replica's GC status in the Destination Table cannot also update the GC status of the vSSD due to the memory limitations of programmable switches. Therefore, we recirculate the packet once to ensure consistency. The SDF sends a final *gc_op* packet when the vSSD has finished GC with the *gc* field set to *finish* (value of 5) in the payload. The switch uses this to clear the GC status in both tables (Line 19-20 in Algorithm 1).

Background GC. Delaying GC enables the switch to reduce overlapping GC. RackBlox also opportunistically utilizes idle cycles to free blocks. Background GC requests are labeled as *bg* (value of 2) in the *gc* field of the payload. Since background GC is performed during idle cycles, the SDF executes it without approval from the ToR switch. To facilitate background GC, RackBlox predicts the next idle time for a given vSSD based on the last interval between I/O requests [12, 55, 70, 85], as shown in T1 in Figure 7c: $T_i^{predict} = \alpha * T_{i-1}^{real} + (1-\alpha) * T_{i-1}^{predict}$, where $T_{i-1}^{predict}$ is the idle time of the last prediction, and α is the exponential smoothing parameter ($\alpha = 0.5$ by default). Once $T_i^{predict}$ is larger than a defined threshold (30 milliseconds by default), the storage server will execute GC and update the GC status in the switch (T2 and T3 in Figure 7c).

3.5.2 Coordinated GC for Software-Isolated vSSDs.

Unlike hardware-isolated vSSDs, software-isolated vSSDs can share channels with other software-isolated vSSDs. As they rely on software techniques to offer performance isolation, software-isolated vSSDs provide relaxed isolation guarantees. Thus, request redirection may not guarantee predictable storage performance for those vSSDs. Even if one replica is not executing GC, a collocated vSSD may execute GC, resulting in significant interference.

RackBlox enables simple management of software-isolated vSSDs by grouping them into channel groups in the SDF. Each channel group is a set of software-isolated vSSDs that span the same set of channels and all vSSDs of the channel group will perform GC simultaneously. Intuitively, if one vSSD must perform GC and each vSSD will be affected anyway, then all vSSDs should perform GC to reduce GC frequency. This simplifies coordination and reduces overhead.

The channel group is managed exclusively by the SDF and is not exposed to the switch. Multiple software-isolated vSSDs sharing the same channels may have diverse GC behavior. To ensure all vSSDs of the channel group can execute GC together, we allow a vSSD that has exhausted its free blocks to transparently borrow free blocks from collocated vSSDs. Blocks are borrowed in groups (1GB by default) and transferred between the free block lists of vSSDs. Thus, we can delay GC until the channel group's free block ratio falls below the *gc_threshold*. The borrowed blocks will be erased (for security) and returned to the original vSSD after the GC. The coordinated GC will not worsen the write amplification, as it makes the best effort to avoid unnecessary GC operations. When sending *gc_op* packets to the switch, the storage server generates a separate packet for each vSSD in the channel group. Note that a *delay* response (i.e., the corresponding replica vSSD is executing GC) from any vSSD will delay the GC of the channel group.

3.6 Rack-Scale Wear Leveling

The limitation of the SSD lifetime has created complexity for their use and management in practice [51, 52]. This is especially true in large-scale data centers. First, as different applications have different workload patterns, the write traffic to each SSD can be different, causing wear imbalance between SSDs in a rack. Second, platform operators have to replace unhealthy or failed SSDs with new SSDs frequently, making the wear management of SSDs across the entire rack even harder. Third, modern cloud infrastructures mostly consider the load balance rather than the wear balance across SSDs. Therefore, the wear-leveling management of SSDs has become a fundamental challenge in data centers today.

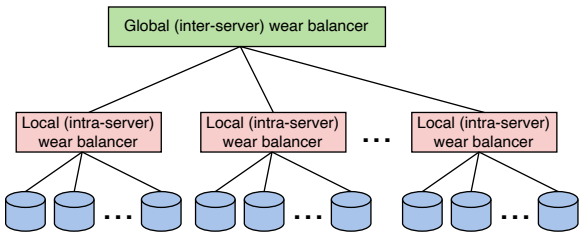


Figure 8. Two-level rack-scale wear leveling in RackBlox.

Premature death or removal of an SSD not only increases the operation cost, but causes an opportunity loss of other hardware components, given that others like CPU, network, and memory do not prematurely lose capability. Thus, it is desirable to ensure SSDs in a rack are aging at the same rate.

To extend the lifetime of a rack of SSDs, we propose a two-level wear leveling mechanism (see Figure 8). It consists of two parts: a local (intra-server) wear balancer processes the local wear balance between SSDs in a storage server, and a global (inter-server) wear balancer reduces the wear variance for SSDs in a rack. The wear balancers work at different level and cooperate to ensure rack-scale wear leveling.

As for the local wear balancer, to obtain the uniform lifetime among SSDs in a storage server, we track the average erase count for an SSD[2], and ensure the wear balance for all the SSDs in a storage server. Let φ_i denote the wear (average erase count of all the blocks to date) of the i^{th} SSD. $\lambda = \varphi_{max}/\varphi_{avg}$ denotes the wear imbalance across SSDs, which must not exceed $1 + \gamma$, where $\varphi_{max} = Max(\varphi_1, ..., \varphi_N)$, $\varphi_{avg} = Avg(\varphi_1, ..., \varphi_N)$, N is the total number of SSDs, and γ represents the maximum permitted imbalance. Instead of swapping SSDs frequently, RackBlox periodically swaps the SSD that has incurred the maximum wear with the SSD that has the minimum rate of wear, following the relaxed wear leveling approach developed in [28]. Given $\gamma = 0.1$, each server can have 16 SSDs, and each SSD can last five years, RackBlox can achieve uniform lifetime for SSDs in a storage server by swapping once per 12 days for the worst case [28]. Assume each flash block can endure 30K writes, this swapping consumes only 0.5% of its lifetime.

SSDs are swapped atomically by pausing operations for the chosen blocks, reading them into memory, writing them to their new locations, updating the mapping tables, and serving the paused requests. As the swapping occurs infrequently, it does not affect the tail latency. To further minimize its impact on application performance, RackBlox assigns higher priority to regular I/O requests during swapping.

Similarly, we can quantify the wear imbalance between storage servers in a rack by using the wear (average erase count of all the SSDs to date) of a server. However, different from the swapping of SSDs in a single server, the swapping cost between storage servers is more expensive, due to the networking overhead. Therefore, we relax the swapping frequency (8 weeks by default). This is less of a concern, as modern storage infrastructures have employed the load balance (e.g., round-robin vSSD allocation) across servers. Since RackBlox does not swap SSDs across servers frequently, we do not implement the rack-scale wear leveling in the switch to keep our design simple. Our experiments (see §4.6) show that the relaxed wear leveling will ensure near-ideal wear balance for datacenter workloads.

[2]For the programmable SSDs, we can track the erase count of each flash block. Therefore, we can obtain the average erase count of a flash channel as well as an SSD.

3.7 Implementation Details

Testbed. Our experiments are conducted on a testbed of five servers connected to a 6.5Tbps Tofino switch [60]. Each server is equipped with a 24-core Intel Xeon E5-2687W processor running at 3.00GHz, 108GB DRAM, and 1TB programmable SSD. Each server has a Mellanox ConnectX-4 50Gb NIC connected to the programmable switch.

Network implementation. To implement the custom packets described in Figure 6, we use DPDK (v22.11.1) [30]. If the packet type is *gc_op*, the payload contains a *gc* field (1 byte) storing the necessary type of GC request. When the packet is a *create_vssd* packet, we include the server IP, vSSD_ID, and server IP of the replica in the payload.

We develop the switch data plane in P4 [14] and run it on an Intel Tofino ASIC [31]. The control plane is implemented in Python and interacts with the switch data plane through Thrift APIs [10] using Intel's P4 SDE 9.10.0. We implement the tables as described in §3.3 using 1.3MB SRAM in the switch. The GC states of the replica and destination table use registers, such that they can be updated in the data plane, consuming a total of 128KB of stateful memory.

Since we do not have access to a real data center, we emulate datacenter network traffic in our cluster using traces and released network traffic distributions [32, 59, 67]. The traces include delays between VMs in cloud data centers. To simulate the variations of network latency, we scale the trace in [67] following the latency patterns and distributions in [32, 59]. The latency is associated with each request and stored in the LAT field when the packet is generated (see Figure 6). When the request traverses the switch, we add the per-hop latency as described in §3.4. The end-to-end latency is computed by adding the time spent at the storage server and the final LAT value in the return packet.

Storage implementation. We build the SDF (SSD virtualization) stack on top of programmable SSDs. By default we implement a greedy, threshold-based GC. We specify the GC thresholds used in each experiment in §4. In our testbed, we use one server as clients, and others to host vSSDs.

Emulation. Since we only have one type of programmable SSD, we build an SSD emulator using Python to test RackBlox against different SSD device performance (see §4.5.3). We validate the emulator with our programmable SSD. For these experiments, we use the same implementation, but issue requests to the emulated SSDs.

Table 2. Workloads used in our evaluation.

Workload	Description	Write%
YCSB [87]	Cloud data serving queries.	0-100%
TPC-H [82]	Business-oriented ad-hoc queries.	2.27%
Seats [57]	Airline ticketing system queries.	10.34%
AuctionMark [83]	Activity queries in an auction site.	53.76%
TPC-C [81]	Online transaction queries.	59.95%
Twitter [54]	Micro-blogging website queries.	97.86%

Others. Similar to modern storage systems [7, 23], RackBlox leverages heartbeats to detect failures. On link failure, it redirects requests to replicas in the rack. On server failure, RackBlox replicates the replicas to other servers and updates their switches. Upon data recovery, it updates stale data from replica vSSDs before serving requests [37]. On switch failure, RackBlox relies on replicas in another rack to serve requests. The ToR switch is repopulated on switch recovery. RackBlox focuses on storage management of a rack. As future work, we wish to extend it to multiple racks by modifying Algorithm 1 to keep GC states consistent among switches.

4 Evaluation

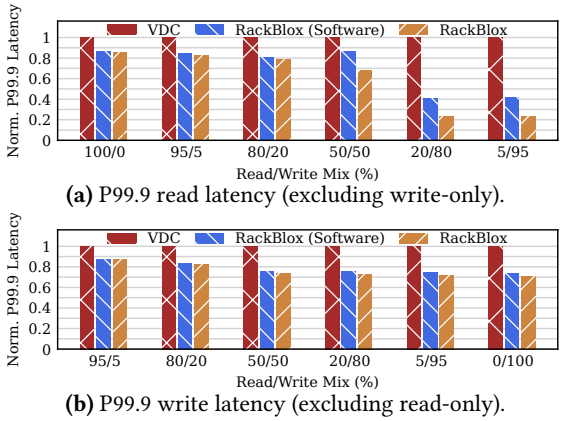
Our evaluation shows that: (1) RackBlox reduces the tail latency of I/O requests by up to 5.8× for data-center applications with network-storage coordination (§4.2 and §4.3); (2) RackBlox works with various storage and network scheduling policies (§4.5.1 and §4.5.2); (3) RackBlox benefits various SSD devices and network latency distributions (§4.5.3); and (4) RackBlox extends the lifetime of a rack of SSDs (§4.6).

4.1 Experimental Setup

To examine RackBlox's performance under different workload patterns, we use YCSB with different read/write ratios [87] and various workloads from BenchBase [21]. These represent common data center applications sensitive to network and storage performance (Table 2). All workloads run on hardware-isolated vSSDs. The datasets range in 50-100GB, so we allocate vSSD capacity accordingly (64-128GB). We set *soft_threshold* to 35%, and set *gc_threshold* to 25%. Before each experiment, we run a subset of the workloads to trigger GC and consume 50% of the free blocks. RackBlox uses Linux's Kyber scheduler [42] by default, as it performs the best across various settings (see §4.5.1). Kyber uses 750μs for reads and 3 millisecs for writes as target 95-th percentile (P95) latencies. When enabling coordinated I/O, we use 1.75 millisecs and 4 millisecs to account for P95 network delay. We use the default priority-based isolation in the switch.

To show the performance benefits of network-storage co-design, we compare RackBlox with state-of-the-art software-defined storage architecture designs at datacenter scale.

VDC: Virtual datacenter (VDC) [6] enables end-to-end isolation between multiple tenants sharing the same physical network and storage. It implements a logically centralized controller that allocates resources to each tenant's VDC as well as each tenant's I/O flows [79]. We run the controller on a separate server updating flow demand and allocations. VDC enforces end-to-end isolation for each flow with multi-resource token bucket rate limiting.



(a) P99.9 read latency (excluding write-only).

(b) P99.9 write latency (excluding read-only).

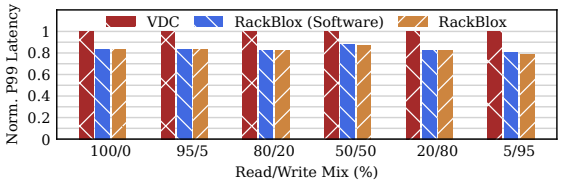
Figure 9. RackBlox's benefits for P99.9 end-to-end latency.

RackBlox (Software): Although RackBlox is developed with a programmable switch and SSDs, its core ideas can be implemented in the software stack. To evaluate this, we extend VDC by adding software-based coordinated I/O scheduling and GC. We make the VDC controller GC-aware by tracking the GC state of vSSDs, and implementing the coordinated GC (§3.5) in software. When the controller grants the vSSD's request to perform GC, it also returns the location of a replica not performing GC. Therefore, storage servers can redirect requests when the vSSD is performing GC.

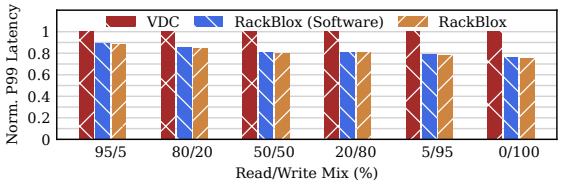
4.2 End-to-End Performance Benefits

To evaluate the end-to-end performance of RackBlox, we run YCSB benchmarks with the zipfian request distribution, and vary the write ratio from 0% (read-only) to 100% (write-only). With network-storage co-design, RackBlox improves the 99.9-th percentile (P99.9) read latency by up to 4.4× (12.4 millisecs vs. 2.8 millisecs), the P99.9 write latency by up to 1.4× (4.3 millisecs vs. 3.0 millisecs), as shown in Figure 9. We show the detailed results in Figure 16.

Although VDC ensures flow isolation across network and storage stack in software, it performs worse than RackBlox (Figure 9a), due to the lack of the coordination between the network and storage stack. RackBlox (Software) enables GC redirection in software, and reduces the overhead when requests are blocked by GC. For read-heavy workloads, the performance of RackBlox (Software) is similar to that of RackBlox, as they both conduct the coordinated I/O scheduling. However, for write-heavy workloads, which cause more intensive GC, RackBlox (Software) can improve VDC's performance by to 2.4× (12.4 millisecs vs. 5.2 millisecs). However, since RackBlox (Software) incurs additional network overhead, it remains suboptimal. RackBlox outperforms VDC and RackBlox (Software) by enabling coordinated I/O scheduling and coordinated GC in the network by 4.4× and 1.84× (5.2 millisecs vs. 2.8 millisecs) respectively.

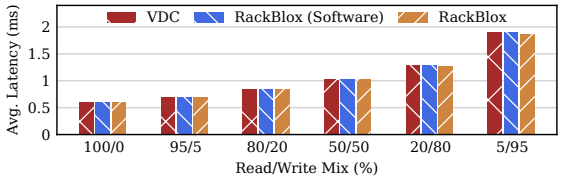


(a) P99 read latency (excluding write-only).

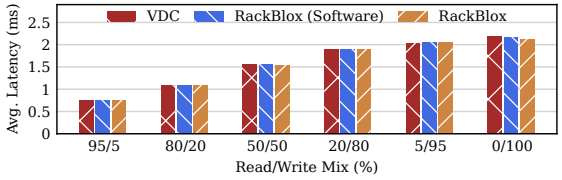


(b) P99 write latency (excluding read-only).

Figure 10. RackBlox's benefits for P99 end-to-end latency.



(a) Average read latency (excluding write-only).



(b) Average write latency (excluding read-only).

Figure 11. Comparing RackBlox's average end-to-end latency against VDC and RackBlox (Software).

Writes are hardly affected by GC because of the write cache in storage servers, as shown in Figure 9b. Thus, RackBlox and RackBlox (Software) have similar performance, as the coordinated I/O scheduling improves the 99.9th percentile write tail latency by up to 1.4×. We show RackBlox's benefit for the 99th percentile (P99) latency in Figure 10. The read latency is improved by up to 2.1× (5.3 millisecs vs. 2.6 millisecs) and the write latency is improved by up to 1.3× (3.7 millisecs vs. 2.8 millisecs). This demonstrates that RackBlox can achieve benefit at lower tails as well.

RackBlox does not negatively affect the average latency, as shown in Figure 11. As we increase the write ratio in the workloads, the average latency of reads/writes is gradually increased, due to the read/write interference, and the write latency is longer than read latency in the storage stack.

We show the average throughput of YCSB benchmarks in Figure 12. RackBlox does not negatively affect throughput, as RackBlox targets improved tail latency. Similar to the average latency, higher write rates lead to lower IOPS since writes have higher device latency.

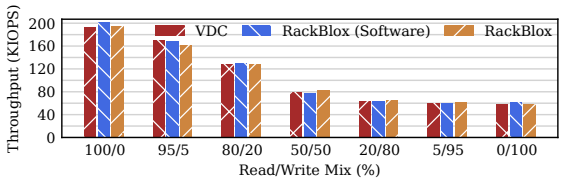
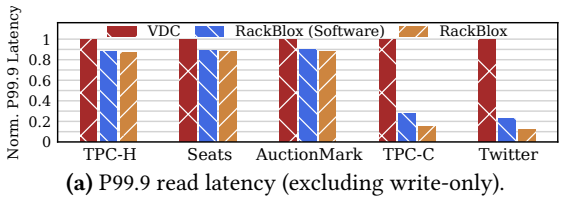
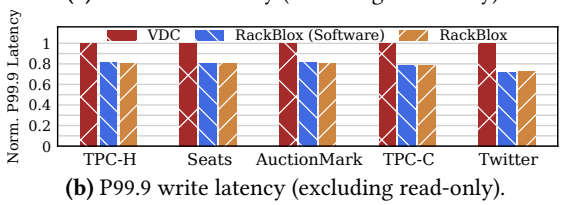


Figure 12. RackBlox's impact on throughput.



(a) P99.9 read latency (excluding write-only).



(b) P99.9 write latency (excluding read-only).

Figure 13. Comparing RackBlox's P99.9 end-to-end latency against VDC and RackBlox (Software) for various workloads.

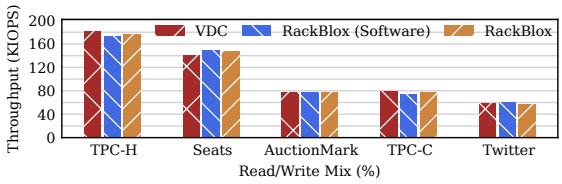
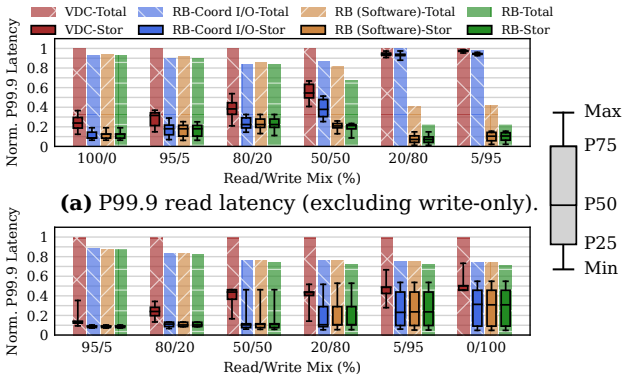


Figure 14. Throughput for various workloads.

4.3 End-to-End Performance of Various Workloads

We further evaluate RackBlox on various workloads (see Table 2), following the setup described in §4.1. Figure 13a shows that RackBlox improves the P99.9 read latency of various workloads by up to 7.9× (23.8 millisecs vs. 3.0 millisecs), in comparison with VDC. For P99 read tail latency, RackBlox achieves up to 2.9× improvement (8.1 millisecs vs. 2.8 millisecs). Compared to the YCSB experiments, we observe similar correlation between write ratio and read tail latency improvement in various workloads. For read-intensive workloads like TPC-H, RackBlox and RackBlox (Software) improve mainly via coordinated I/O scheduling. For write-intensive ones like Twitter, RackBlox improves performance mainly by alleviating GC interference. AuctionMark has less benefit than YCSB with 50% writes, although it has slightly higher write ratio. This is because AuctionMark has a different I/O request pattern (e.g., a long sequence of writes followed by a sequence of reads, rather than mixed reads and writes in YCSB), it has fewer I/O requests affected by the GC. RackBlox (Software) performs worse than RackBlox due to the additional networking overhead for coordinated GC. The end-to-end write tail latency, shown in Figure 13b, demonstrates a similar trend and improvement to YCSB. For throughput (see Figure 14) and average read/write latency, we observe the similar trend as YCSB benchmarks (see §4.2).



(a) P99.9 read latency (excluding write-only).

(b) P99.9 write latency (excluding read-only).

Figure 15. P99.9 latency breakdown. RB is RackBlox. Stor is the storage latency and Total is the end-to-end latency.

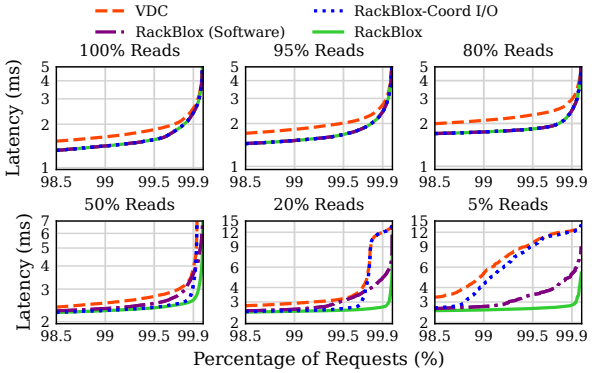


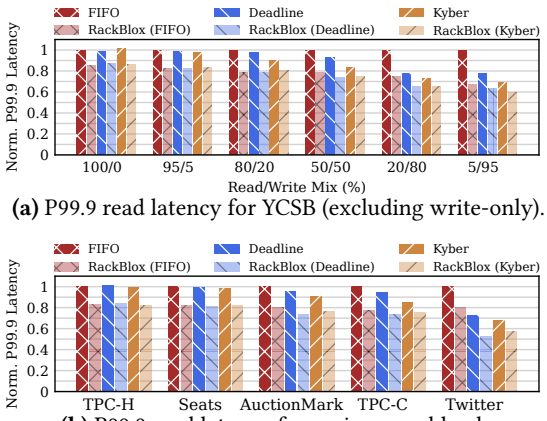
Figure 16. Cumulative distribution of read latency.

4.4 Performance Benefit Breakdown in RackBlox

To break down the performance improvements in RackBlox, we evaluate **RackBlox-Coord I/O**, in which we enable coordinated I/O scheduling between the network and storage stack, but disable coordinated GC in RackBlox.

We show the comparison in Figure 15 with latency breakdowns. By coordinating I/O between network and storage, RackBlox-Coord I/O reduces the P99.9 read latency by up to 1.1-1.23× (3.9 millisecs vs. 3.1 millisecs) and write latency by 1.1-1.4× compared to VDC. With increased write ratios, RackBlox-Coord I/O brings more benefits for the tail latency, because the potential delay of a request in the storage queues increases, as writes have higher device latency than reads. Thus, prioritizing requests in the storage queue leads to more obvious effects on the end-to-end latency and coordinated I/O scheduling provides greater speedup.

However, for write-dominant workloads (e.g., more than 50%), the read tail latency improvement of coordinated I/O scheduling diminishes no matter how we schedule as shown in Figure 15a, because intensive writes incur high GC overhead. With high write ratios, the coordinated I/O scheduling brings more benefits to the tail latency of writes than that of reads, since the write cache helps alleviate the GC overhead. As we further increase the write ratio (i.e., above 50%),



(a) P99.9 read latency for YCSB (excluding write-only).

(b) P99.9 read latency for various workloads.

Figure 17. Comparing RackBlox's P99.9 end-to-end latency with different storage I/O schedulers.

the tail latency of both VDC and RackBlox-Coord I/O is increased, due to the increased storage queue delay (see the storage latency distribution in Figure 15b). Compared to VDC, the normalized tail latency reduction of coordinated I/O scheduling is almost the same as shown in Figure 15b.

The coordinated GC mechanism in RackBlox will further improve the read tail latency (by up to 4.3×), as shown in Figure 15a and Figure 16. Both RackBlox and RackBlox (Software) implement coordinated GC, but RackBlox provides more speedup with the programmable switch, as it alleviates the unnecessary networking round-trip delays. The coordinated GC does not benefit writes, as we need to issue writes to all replicas for data consistency (as discussed in §3.5.1).

4.5 Sensitivity Analysis

We demonstrate that RackBlox retains the flexibility and modularity of the original SDN/SDF design by evaluating different scheduling policies and system configurations.

4.5.1 Varying storage I/O scheduling policies. We now examine the benefit of the coordinated I/O scheduling under different storage I/O scheduling policies. In particular, we implement Linux's storage schedulers for SDF: no-op (**FIFO**), **Deadline**, and **Kyber** [42]. No-op is the default on NVMe devices, while both Kyber and Deadline target predictable latency. Deadline splits requests into read and write queues, and prioritizes requests when they reach their respective deadlines. Kyber also splits requests into read and write queues, and throttles each queue to meet the latency targets. To enable coordinated I/O scheduling, RackBlox reorders requests in each queue using network latencies. We use 0.5 millisecs and 1.75 millisecs as deadlines for reads and writes in Deadline and 1.5 millisecs and 2.75 millisecs in RackBlox (Deadline). We use 0.75 millisecs and 3 millisecs for reads and writes in Kyber and 1.75 millisecs and 4 millisecs for RackBlox (Kyber). RackBlox (Deadline) and RackBlox (Kyber) use increased parameter values, as RackBlox incorporates the network latency in its coordinated I/O scheduling, based on the distribution of network latencies in data centers [67].

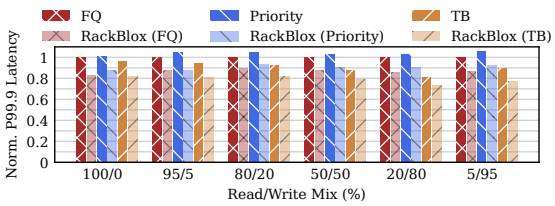


Figure 18. Comparing P99.9 read latency for RackBlox with different network scheduling policies.

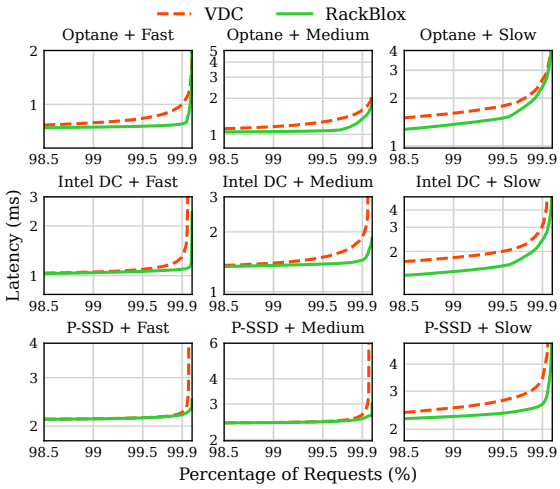


Figure 19. End-to-end read latency distributions of RackBlox on YCSB-A with varying SSDs and network latencies.

We show the results in Figure 17. As expected, coordinated I/O scheduling always outperforms its baseline scheduler. RackBlox (FIFO) achieves the greatest speedup over its baseline scheduler (1.5×). RackBlox (Kyber) and RackBlox (Deadline) have fewer opportunities to reorder requests when splitting reads and writes into separate queues, but still benefit from coordination (1.24× and 1.36× respectively).

4.5.2 Varying network scheduling policies. We now evaluate the performance of RackBlox under different network scheduling policies in the switch. Besides the Token Bucket rate limiting (**TB**) policy that ensures isolation between flows (similar to VDC), we examine the fair queuing (**FQ**) and priority based network scheduling (**Priority**) policies. For FQ, we have four client servers competing for one storage server with each receiving a fair share of the network bandwidth. In Priority, we periodically create higher priority traffic using [72], which delays lower-priority requests.

We show the results in Figure 18. Coordinated I/O scheduling can benefit all the underlying network schedulers. FQ and Priority result in higher latency as requests are delayed in the network. This provides increased opportunities for reordering, which allows RackBlox to achieve up to 1.21× and 1.15× performance improvement on average, respectively.

4.5.3 Varying the Network/Storage Latency. The coordinated I/O scheduling works by hiding higher network latency with lower storage latency, or vice versa. Therefore, if the network latency overwhelms storage latency or vice versa, RackBlox helps less to improve the end-to-end latency.

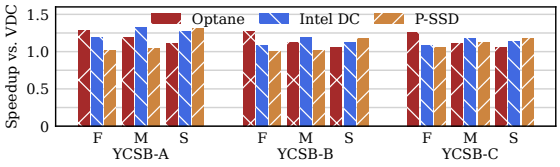


Figure 20. P99.9 read latency improvements of RackBlox with different storage and network latencies.

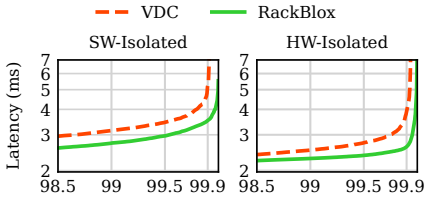


Figure 21. Read tail latency with different isolation.

We analyze the sensitivity of RackBlox by evaluating the YCSB benchmarks with emulated devices of different latencies (see §3.7). We evaluate three SSDs from fastest to slowest: **Optane** [3], **Intel DC** [2], and **P-SSD** (programmable SSD) [53]. We evaluate networks with **Fast** [67], **Medium** [59], and **Slow** [32] latencies (see §3.7). The resulting end-to-end latencies of YCSB-A (50% reads) are shown in Figure 19.

Varying the SSD performance. For RackBlox, the marginal benefit on end-to-end tail latency by upgrading SSD is low when the SSD already outperforms the network. For example, upgrading the SSD from Intel DC to Optane under Slow network brings little benefit to the P99.9 latency. Thus, the performance improvement of RackBlox over VDC is also low. In contrast, the benefits of upgrading SSD are more obvious when the network outperforms SSDs, which brings overall performance benefit for RackBlox.

Varying the networking performance. Similar conclusions are drawn as we vary network latencies. For example, upgrading the network from Slow to Fast with the slowest P-SSD hardly improves read tail latency in RackBlox, because SSD latency dominates the end-to-end latency. In contrast, by upgrading the network with the fastest Optane SSD, we significantly improve the read tail latency in RackBlox.

Our findings are consistent across various YCSB benchmarks, as shown in Figure 20. The fastest Optane SSD matches best (i.e., RackBlox achieves the most benefit) with Fast network, the slower Intel DC SSD matches with Medium network, and the slowest P-SSD matches with Slow network. The reduced benefit for RackBlox under unmatched latencies is a potential limitation, but this is less of a concern, as modern data centers usually upgrade network and storage hardware together for best resource efficiency (e.g., using slow storage with fast RDMA network is impractical in the real world). Therefore, pairing the storage stack with the network stack fully unleashes the potential of RackBlox.

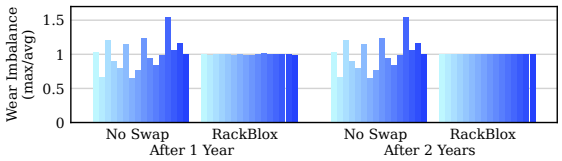


Figure 22. Benefits of RackBlox on each server's wear balance (different colors represent different SSDs in one server).

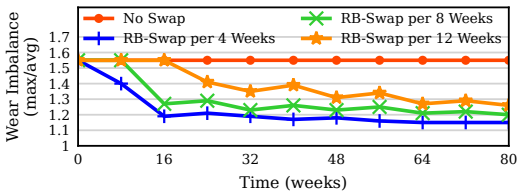


Figure 23. Benefits of RackBlox on rack-scale wear balance (the lower is better).

4.5.4 Software-Isolated vSSDs vs. Hardware-Isolated vSSDs.

To examine RackBlox for software-isolated vSSDs, we run two software-isolated vSSDs on the same flash channels (SW-Isolated). These vSSDs are isolated using token bucket rate limiting and both run YCSB with 50% writes. We compare SW-Isolated with a hardware-isolated vSSD (HW-Isolated) that has the full ownership of the flash channels.

RackBlox reduces the P99.9 latency by 1.47× in comparison with VDC for SW-Isolated vSSDs and by 1.51× for HW-Isolated vSSDs, as shown in Figure 21. With hardware-isolated vSSDs, RackBlox brings marginally more benefit, since the hardware-isolated vSSD minimizes the interference from colocated workloads. Thus, RackBlox can improve the performance for both software-isolated and hardware-isolated vSSDs with the coordinated I/O scheduling and GC.

4.6 Benefits of Rack-Scale Wear Leveling

To evaluate the rack-scale wear leveling of RackBlox, we simulate the effects of running real workloads. We configure a rack with 32 servers, each server has 16 SSDs, and each SSD hosts 4 vSSDs. Each vSSD runs one workload (see Table 2), which may cause wear imbalance across different SSDs, since the workloads have diverse erase frequency. Each SSD is well balanced at the device level as it has its own device-level wear leveling. Following the load balancing of modern storage infrastructures, we assign the vSSDs across servers using round robin [61]. We evaluate RackBlox's hierarchical wear leveling against modern storage infrastructure which does not swap across SSDs and servers (No Swap) [51, 52].

Local wear balancer. Figure 22 demonstrates that RackBlox's local wear balancer effectively maintains wear balance across different SSDs. While No Swap has significant wear imbalance, RackBlox can ensure near-optimal wear balance across the SSDs in each server with periodic swapping.

Global wear balancer. Local wear balancing suffers wear-imbalance at rack scale. Figure 23 shows that RackBlox's global wear balancer effectively maintains rack-scale wear balance, despite reduced swapping frequency (e.g., 8 weeks).

5 Related Work

Software-defined networking. Recent studies investigated SDN systems, including networking abstraction, packet processing and scheduling, QoS, SDN programming, performance, and fault tolerance [11, 14, 73, 74, 77]. Programmable schedulers and frameworks have been proposed to allow developers to develop a variety of scheduling algorithms [77, 86]. With these efforts, the community has produced a set of open-sourced frameworks such as OpenFlow [62], and the programming language P4 [14], as well as the hardware devices like Intel Tofino [80]. Recent work [33, 34, 48] also demonstrates that SDN can benefit distributed storage systems. However, none studied the codesign of SDN and SDS. RackBlox makes an initial effort in this, and shows the benefits of the new software-defined rack-scale storage system.

Software-defined storage. Researchers proposed techniques like SDF [44, 65, 84] and open-channel SSDs [53], so upper-level system software can exploit the intrinsic properties of flash memory. As the cost of flash-based SSDs approaches that of HDDs and their performance has improved, SDF is a compelling solution for storage management in data centers [28, 65]. However, no previous study focused on the integration of SDN and SDF.

Network/storage co-scheduling. To improve the end-to-end performance for data center applications, IOFlow [79] and VDC [6] enforced policies for I/O requests in centralized servers or hypervisors. However, they treated the SDN and SDF as black boxes without considering the underlying hardware opportunities. Recently, researchers leveraged programmable switches to fulfill system functions like data caching [34], consistency protocols [33], and task scheduling [91], showing that it is feasible to integrate system functions into programmable switches. We integrate storage functions into SDN and show the benefits of this design.

6 Conclusion

We present RackBlox, a new rack-scale storage system by co-designing the software-defined networking and storage stack. RackBlox integrates essential storage functions into the programmable switch, and enables the state sharing between the network and storage stack. With coordinated I/O scheduling, GC, and rack-scale wear leveling, RackBlox achieves improved end-to-end storage performance, while ensuring near-ideal lifetime for SSDs in a rack.

Acknowledgments

We thank the anonymous reviewers and our shepherd Simon Peter for their insightful comments and feedback. We thank Yiqi Liu for proofreading an early version of this paper. This work was partially supported by NSF grant CCF-1919044, CCF-2107470, and the Hybrid Cloud and AI program at the IBM-Illinois Discovery Accelerator Institute.

References

[1] In-band Network Telemetry (INT) Dataplane Specification. https://github.com/p4lang/p4-applications/blob/master/docs/INT.pdf.

[2] Intel ssd data center family. https://www.intel.com/content/www/us/en/products/details/memory-storage/data-center-ssds.html.

[3] Intel® Optane™ SSD 900P Series. https://www.intel.com/content/www/us/en/products/sku/123623.

[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pFabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM*, August 2013.

[5] Amin Vahdat. Jupiter evolving: Reflecting on Google's data center network transformation. https://cloud.google.com/blog/topics/systems/the-evolution-of-googles-jupiter-data-center-network, 2022.

[6] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, October 2014.

[7] Apache Hadoop. HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[8] Apache Hadoop. HDFS Block Placement. https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsBlockPlacementPolicies.html.

[9] Apache Kafka. Apache Kafka Documentation. https://kafka.apache.org/documentation/.

[10] Apache Thrift. Apache thrift, https://thrift.apache.org/, 2022.

[11] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. SNAP: Stateful Network-Wide Abstractions for Packet Processing. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*, Florianopolis, Brazil, August 2016.

[12] Eitan Bachmat and Jiri Schindler. Analysis of methods for scheduling low priority disk drive tasks. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'02)*, 2002.

[13] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Communications of the ACM (CACM)*, mar 2017.

[14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3), July 2014.

[15] David Clark, Jennifer Rexford, and Amin Vahdat. A Purpose-Built Global Network: Google's Move to SDN. *Communications of ACM*, 59(3), March 2016.

[16] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, 2015.

[17] Complete Fairness Queueing (CFQ). https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[18] Data Center News. Alibaba launches Dual-mode SSD to optimize hyper-scale infrastructure, https://datacenternews.asia/story/alibaba-launches-dual-mode-ssd-optimize-hyper-scale-infrastructure, 2018.

[19] Deadline IO Scheduler Tunables. https://www.kernel.org/doc/Documentation/block/deadline-iosched.txt.

[20] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM (CACM)*, feb 2013.

[21] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[22] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, Fei Feng, Yan Zhuang, Fan Liu, Pan Liu, Xingkui Liu, Zhongjie Wu, Junping Wu, Zheng Cao, Chen Tian, Jinbo Wu, Jiaji Zhu, Haiyong Wang, Dennis Cai, and Jiesheng Wu. When cloud storage meets {RDMA}. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21)*, 2021.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03)*, 2003.

[24] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. Association for Computing Machinery, 2015.

[25] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principle (SOSP'17)*, 2017.

[26] Hitachi. Hitachi Virtual Storage Platform 5000 Series Datasheet,https://www.hitachivantara.com/en-us/pdf/datasheet/virtual-storage-platform-5000-series-datasheet.pdf, 2023.

[27] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. In *ACM SIGCOMM*, August 2012.

[28] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*, Santa Clara, CA, 2017.

[29] Improving Network Monitoring and Management with Programmable Data Planes. https://p4.org/p4/inband-network-telemetry/.

[30] Intel. Intel data plane development kit (dpdk), 2022. http://dpdk.org/.

[31] Intel Tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[32] Hassan Iqbal, Anand Singh, and Muhammad Shahzad. Characterizing the availability and latency in aws network from the perspective of tenants. *IEEE/ACM Transactions on Networking (TONS)*, 2022.

[33] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*, Renton, WA, April 2018.

[34] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principle (SOSP'17)*, Shanghai, China, 2017.

[35] Joey Benamy. Software-defined infrastructure: What is it, and why is it the future of it? https://stratacloud.com/blog/software-defined-infrastructure-what-is-it-\and-why-is-it-the-future-of-it, 2016.

[36] John P John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. Consensus routing: The Internet as a distributed system. In *USENIX NSDI*, April 2008.

[37] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. Association for Computing Machinery, 2020.

[38] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'18)*, 2018.

[39] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. Alleviating garbage collection interference through spatial separation in all flash arrays. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)*, Renton, WA, July 2019. USENIX Association.

[40] Youngjae Kim, Junghee Lee, Sarp Oral, David A. Dillow, Feiyi Wang, and Galen M. Shipman. Coordinating garbage collectionfor arrays of solid-state drives. *IEEE Transactions on Computers*, 2014.

[41] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST '11)*, 2011.

[42] Kyber I/O Scheduler. https://lwn.net/Articles/720675/.

[43] Laura Caulfield. Project denali to define flexible ssds for cloud-scale applications. https://www.opencompute.org/files/2018-03-OCP-Denali.pdf, 2018.

[44] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-Managed Flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, Santa Clara, CA, February 2016.

[45] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. loda: A host/device co-design for strong predictability contract on modern flash storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, 2021.

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

[47] Yifan Li, Jiaqi Gao, Ennan Zhai, Mengqi Liu, Kun Liu, and Hongqiang Harry Liu. Cetus: Releasing p4 programmers from the chore of trial and error compiling. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI'22)*, Renton, WA, April 2022.

[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)*, Xian, China, 2017.

[49] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, and Jiesheng Wu. Perseus: A Fail-Slow detection framework for cloud storage systems. In *21st USENIX Conference on File and Storage Technologies (FAST'23)*, Santa Clara, CA, 2023. USENIX Association.

[50] Martin Maas, Krste Asanovic, Tim Harris, and John Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, 2016.

[51] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. A study of ssd reliability in large scale enterprise storage deployments. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*, Santa Clara, CA, February 2020.

[52] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. Operational characteristics of SSDs in enterprise storage systems: A Large-Scale field study. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST'22)*, Santa Clara, CA, February 2022.

[53] Matias Bjorling and Javier Gonzalez and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proc. USENIX FAST'17*, Santa Clara, CA, February 2016.

[54] Meeyoung Cha, Hamed Haddadi, Fabrício Benevenuto, and Krishna P. Gummadi. Measuring user influence in Twitter: The million follower fallacy. In ICWSM, May 2010.

[55] Ningfang Mi, Alma Riska, Qi Zhang, Evgenia Smirni, and Erik Riedel. Efficient management of idleness in storage systems. *ACM Trans. Storage*, 5(2), 2009.

[56] 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)*. Association for Computing Machinery, 2022.

[57] Michael Stonebraker and Andy Pavlo. The SEATS Airline Ticketing Systems Benchmark. http://hstore.cs.brown.edu/projects/seats.

[58] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *ACM SIGCOMM Computer Communication Review (SIGCOMM'15)*, 2015.

[59] Jeffrey C Mogul and Ramana Rao Kompella. Inferring the network latency requirements of cloud tenants. In *15th Workshop on Hot Topics in Operating Systems (HotOS '15)*, 2015.

[60] EdgeCore Networks. DCS801 6.4T Programmable Data Center Switch, 2022.

[61] NGinx Load Balancing. https://www.nginx.com/resources/glossary/load-balancing/.

[62] OpenFlow. https://en.wikipedia.org/wiki/OpenFlow.

[63] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The Case for RAMCloud. *Commun. ACM*, 54(7):121–130, July 2011.

[64] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, pages 293–307, 2015.

[65] Jian Ouyang, Shiding Lin, Song Jiang, Yong Wang, Wei Qi, Jason Cong, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *Proc. ACM ASPLOS*, Salt Lake City, UT, March 2014.

[66] Tian Pan, Nianbing Yu, Chenhao Jia, Jianwen Pi, Liang Xu, Yisong Qiao, Zhiguo Li, Kun Liu, Jie Lu, Jianyuan Lu, Enge Song, Jiao Zhang, Tao Huang, and Shunmin Zhu. Sailfish: Accelerating cloud-scale multi-tenant multi-service gateways with programmable switches. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (SIGCOMM'21)*, Virtual Event, USA, 2021.

[67] Diana Andreea Popescu and Andrew W Moore. A first look at data center network condition through the eyes of ptpmesh. In *2018 Network Traffic Measurement and Analysis Conference (TMA)*. IEEE, 2018.

[68] Pure Storage. Pure Storage Flash Array//XL, https://www.purestorage.com/docs.html?item=/type/pdf/subtype/doc/path/content/dam/pdf/en/datasheets/ds-flasharray-xl.pdf, 2023.

[69] Lenin Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP'13)*, 2013.

[70] Benjamin Reidys, Peng Liu, and Jian Huang. Rssd: Defend against ransomware with hardware-isolated network-storage codesign and post-attack analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*, Lausanne, Switzerland, 2022.

[71] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi Noghabi, and Jian Huang. BlockFlex: Enabling storage harvesting with Software-Defined flash in modern cloud platforms. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*, Carlsbad, CA, 2022.

[72] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. Inside the social network's (datacenter) network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*, pages 123–137, 2015.

[73] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. PISCES: A Programmable, Protocol-Independent Software Switch. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*, Florianopolis, Brazil, August 2016.

[74] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *NSDI'18*, Renton, WA, 2018.

[75] Ji Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Gecko: Contention-Oblivious disk arrays for cloud storage. In *11th USENIX Conference on File and Storage Technologies (FAST '13)*, San Jose, CA, February 2013. USENIX Association.

[76] Silk. Silk All-Flash Array Architecture,https://silk.us/wp-content/uploads/2020/10/Silk-All-Flash-Array-Architecture.pdf, 2020.

[77] Anirudh Sivaraman, Suvinay Subramanian, Mohammad Alizadeh, Sharad Chole, Shang-Tse Chuang, Anurag Agrawal, Hari Balakrishnan, Tom Edsall, Sachin Katti, and Nick McKeown. Programmable Packet Scheduling at Line Rate. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*, Florianopolis, Brazil, August 2016.

[78] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. Flash on rails: consistent flash performance through redundancy. In *Proc. USENIX ATC'14*, Philadelphia, PA, June 2014.

[79] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A Software-Defined Storage Architecture. In *Proceedings of the SOSP'13*, Farmington, PA, November 2013.

[80] ToFino: Word's fastest P4-programmable Ethernet Switch ASICs. https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/tofino.html.

[81] TPC-C Benchmark. http://www.tpc.org/tpcc/.

[82] TPC-H Benchmark. http://www.tpc.org/tpch/.

[83] Visawee Angkanawaraphan and Andy Pavlo. AuctionMark: A Benchmark for High-Performance OLTP Systems. http://hstore.cs.brown.edu/projects/auctionmark.

[84] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Effective Design and Implementation of LSM-Tree based Key-Value Store on Open-Channel SSD. In *Proceedings of the European Conference on Computer Systems (EuroSys'14)*, Amsterdam, the Netherlands, April 2014.

[85] Xiaohao Wang, Yifan Yuan, You Zhou, Chance Coats, and Jian Huang. Project Almanac: A Time-Traveling Solid-State Drive. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*, Dresden, Germany, 2019.

[86] Joseph Y.-T.Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4, 1989.

[87] Yahoo! Cloud Serving Benchmark. https://github.com/brianfrankcooper/YCSB/wiki.

[88] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail flash: Near-Perfect elimination of garbage collection tail latencies in NAND SSDs. In *15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, CA, February 2017. USENIX Association.

[89] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the flow completion time tail in datacenter networks. In *ACM SIGCOMM*, August 2012.

[90] Ning Zhang, Junichi Tatemura, Jignesh M. Patel, and Hakan Hacigumus. Re-evaluating Designs for Multi-Tenant OLTP Workloads on SSD-based I/O Subsystems. In *Proc. SIGMOD'14*, Snowbird, UT, June 2014.

[91] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*, Virtual Event, November 2020.

[92] Timothy Zhu, Alexey Tumanov, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. Prioritymeister: Tail latency qos for shared networked storage. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'14)*, 2014.