FleetIO: Managing Multi-Tenant Cloud Storage with Multi-Agent Reinforcement Learning

Jinghan Sun
UIUC
js39@illinois.edu

Benjamin Reidys
UIUC
breidys2@illinois.edu

Daixuan Li
UIUC
daixuan2@illinois.edu

Jichuan Chang
Google
jichuan@google.com

Marc Snir
UIUC
snir@illinois.edu

Jian Huang
UIUC
jianh@illinois.edu

Abstract

Cloud platforms have been virtualizing storage devices like flash-based solid-state drives (SSDs) to make effective use of storage resources. They enable either software-isolated instance or hardware-isolated instance for facilitating the storage sharing between multi-tenant applications. However, for decades, they have to combat the fundamental tussle between the performance isolation and resource utilization. They suffer from either long tail latency caused by weak isolation or low storage utilization caused by strong isolation.

In this paper, we present FleetIO, a learning-based storage virtualization framework that employs reinforcement learning (RL) for managing virtualized SSDs. FleetIO explores the unique features of RL to handle the dynamic changes of application workloads and storage states, and integrates the storage scheduling into the RL decision-making process. It achieves both performance isolation and improved storage utilization by enabling dynamic fine-grained storage harvesting across collocated application instances, while minimizing its negative impact on their service-level objectives (SLOs). FleetIO clusters workloads into different types (e.g., latency-sensitive and bandwidth-intensive) based on the collected I/O traces at runtime, and fine-tunes the RL reward functions for each type of workloads. We implement FleetIO on a real programmable SSD board and evaluate it with diverse cloud applications. We show that FleetIO improves the overall storage utilization of the shared SSD by up to 1.4×, and decreases the tail latency of I/O requests by 1.5× on average, compared to the state-of-the-art storage sharing approaches.

CCS Concepts: **• Hardware → External storage**; **• Theory of computation → Reinforcement learning**; **• Information systems → Storage virtualization**.

Keywords: Cloud storage, reinforcement learning, storage virtualization

ACM Reference Format:
Jinghan Sun, Benjamin Reidys, Daixuan Li, Jichuan Chang, Marc Snir, and Jian Huang. 2025. FleetIO: Managing Multi-Tenant Cloud Storage with Multi-Agent Reinforcement Learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3669940.3707229

1 Introduction

In cloud platforms today, storage devices such as flash-based SSDs are shared by multiple applications via storage virtualization [2, 22, 36, 38, 46]. They slice SSDs into multiple virtual instances (i.e., virtual SSDs or vSSDs) and map each vSSD to the underlying flash chips according to its demands of storage capacity and bandwidth [17, 27, 42, 54]. This not only enables storage sharing among cloud virtual machines (VMs) but also facilitates cloud storage management. However, it has been a longstanding challenge to develop efficient SSD virtualization, due to the fundamental tussle between the performance isolation and storage utilization.

As we collocate multiple cloud applications on a shared SSD, two approaches have been developed to enforce the storage isolation: software-isolated approach [3, 16, 42, 47] and hardware-isolated approach [9, 17, 38]. The software-isolated approach enables sharing among multi-tenant applications using I/O throttling [3]. It can maximize the overall storage utilization, but offers limited performance isolation due to the interference between application instances, resulting in long tail latency and SLO violations. The hardware-isolated approach takes advantage of the software-defined flash architecture [17, 34], and maps each vSSD to the flash channels by utilizing the internal parallelism of SSDs (Figure 1). As each instance fully owns the flash channels, it provides the strongest isolation but at the sacrifice of utilization. Our study with cloud applications shows that the storage utilization of the hardware-isolated approach can be 1.5× worse than the software-isolated one, while its tail latency can be 2.0× less than the software-isolated approach (see §2.2).

Ideally, we wish to achieve both strong performance isolation and maximum storage utilization simultaneously. Most recently, researchers [26] used pre-trained deep neural networks (DNNs) to learn workload patterns, and decided an optimized number of flash channels that should be allocated. And heuristic-based approaches [2, 38, 55] have been developed in cloud platforms to predict the availability of idle resources that can be harvested by low-priority VMs such as spot VMs [5, 6]. However, all these approaches cannot quickly capture the dynamic changes of workload patterns at runtime, and the updates to the storage states of the shared SSD after each I/O activity (e.g., read, write, and garbage collection). Thus, they still result in suboptimal efficiency. Moreover, they mainly target storage utilization without considering the performance isolation (i.e., SLO violation and tail latency reduction) in their decision-making process. Thus, it is difficult to rely on them to achieve the ideal goal.

To this end, we present FleetIO, an RL-based storage virtualization framework that employs RL techniques to manage virtualized SSD instances. We argue that RL is a natural fit for managing virtualized SSDs and their shared storage states, thanks to its unique advantages of taking actions to meet defined purposes in a dynamic environment (see Figure 4).

Developing FleetIO with RL techniques is not easy. We first need to extract the essential storage states of virtualized SSDs that can be represented as the RL states for assisting RL agents in making decisions. These states include the measured average I/O latency and throughput, measured tail latency, I/O queue delays, available storage capacity, and active garbage collection (GC) events. To support multiple vSSDs concurrently, FleetIO deploys one RL agent in each vSSD, and allows each agent to make independent actions.

FleetIO reflects the goal of improving both performance isolation and storage utilization in its RL actions and reward function. Specifically, FleetIO defines three types of actions in each agent: (1) set the priority for I/O requests, which enables flexible I/O scheduling in FleetIO to help each vSSD meet the specified SLOs; (2) harvest storage resource from collocated vSSDs, which enables FleetIO to improve the overall storage utilization by harvesting idle storage resources across collocated vSSDs; (3) decide how much storage resource are harvestable for collocated vSSDs, which offers the flexibility for each vSSD to balance storage performance and utilization. To facilitate fine-grained storage harvesting, we rethink the structure of virtualized SSDs and develop a new abstraction named ghost superblock (gSB) to track the blocks that can be harvested across flash channels (see §3.6). The gSB abstraction simplifies the allocation and deallocation of harvestable flash blocks in vSSDs and eases the execution of harvesting actions for RL agents.

FleetIO formulates the RL reward function based on the measured I/O throughput and the SLO violations of issued I/O requests, as the former reflects how much bandwidth of a vSSD has been utilized, and the latter indicates the level of performance isolation. To improve the overall storage utilization, instead of having complicated coordination between RL agents, FleetIO uses a coefficient factor to balance the rewards of collocated RL agents. Thus, each agent takes actions independently while aiming to meet the overall goal.

To further improve the effectiveness of FleetIO, we fine-tune the reward function for different types of workloads (e.g., latency-sensitive vs. bandwidth-intensive). This is because different types of workloads are in favor of different performance metrics. Instead of having a unified reward function, FleetIO categorizes the workloads into different types based on their I/O access patterns. FleetIO only needs to fine-tune the reward function for each type of workloads, based on our study that workloads with similar I/O access patterns can be clustered into the same type (see Figure 6). For workloads that cannot be categorized into an existing type, FleetIO uses a unified reward function for simplicity.

We implement the SSD virtualization framework of FleetIO on a real programmable SSD (i.e., open-channel SSD) using 3.5K lines of code with C/C++ based on the SDK library of the storage device, and 2K lines of code using Python. We develop the RL model using RLlib [25] and PyTorch, and pre-train the model offline on Ray [32] using a set of workloads. The RL model is 2.2MB. Our experiments with diverse cloud applications show that FleetIO can improve the overall utilization of the shared SSD by up to up to 1.4×, and decrease the tail latency of I/O requests by 1.5× on average, in comparison with the state-of-the-art storage sharing approaches. FleetIO still outperforms these existing approaches significantly as we scale the number of collocated vSSDs on a shared SSD. Overall, we make the following contributions:

- We propose an RL-based SSD virtualization framework FleetIO that can achieve both performance isolation and improved storage utilization with automated decision-making process for storage harvesting and I/O scheduling.
- We extend the virtualized SSD abstraction for supporting RL in FleetIO and facilitating the fine-grained storage harvesting between collocated application instances.
- We study the impact of workload types on the RL learning efficiency, based on which we fine-tune the RL reward function for different types of workloads.
- We develop a system prototype of FleetIO on a real virtualized SSD platform and demonstrate its effectiveness and scalability with diverse cloud applications.

2 Background and Motivation

2.1 SSD Virtualization

Flash-based SSDs have been widely deployed in cloud platforms for their high performance and low cost [36, 38]. As

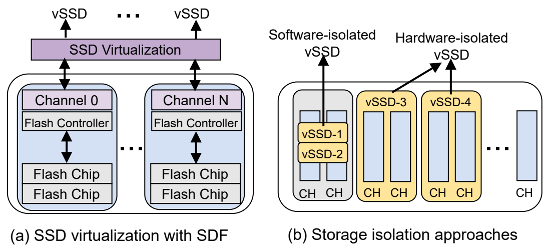


Figure 1. SSD virtualization that supports software-isolated and hardware-isolated approaches.

shown in Figure 1 (a), an SSD has multiple flash channels. Each flash channel has a set of flash chips, and can issue I/O commands independently with the flash controller. Each flash chip has multiple flash blocks. SSDs employ out-of-place update to tolerate expensive block erase operations, have the flash translation layer (FTL) to handle the logical-to-physical address mapping, and perform garbage collection (GC) to reclaim flash blocks to free up space for future use.

SSDs are usually virtualized and shared by multiple virtual machines (VMs) or containers for utilization improvement. Thanks to the development of software-defined flash (SDF) [8, 17, 34, 39] that allows the upper-level software to directly manage and access the underlying flash chips, SSD virtualization can map each virtual SSD (vSSD) to a set of flash channels and chips, and collocate multiple vSSDs on the shared SSD, as shown in Figure 1 (b). Cloud platforms can follow the pay-as-you-go model to allocate flash channels and chips for each vSSD. Such an SSD virtualization solution offers sufficient flexibility for resource management and has become popular in cloud platforms today [8, 18, 30, 44, 45].

As we collocate multiple application instances or vSSDs on the shared SSD, two approaches have been developed to enforce their performance isolation: software-isolated approach and hardware-isolated approach. As shown in Figure 1 (b), vSSD-1 and vSSD-2 are software-isolated vSSDs, they are mapped to the same flash channels, and use I/O throttling techniques like token bucket rate limiting [3, 46] to achieve the performance isolation. The software-isolated vSSDs can maximize the storage utilization of shared channels, but offer poor performance isolation due to the interference between vSSDs. The hardware-isolated vSSDs, such as vSSD-3 and vSSD-4 in Figure 1 (b), fully own the flash channels and leverage their device-level independence to obtain the strongest isolation. However, they suffer from low storage utilization, especially for applications that have dynamic demands on storage resources.

2.2 Storage Utilization vs. Performance Isolation

To further understand the trade-off between performance isolation and storage utilization, we conduct a quantitative study with a variety of cloud workloads (see §4.1 for the detailed experimental setup). In each experiment, we collocate

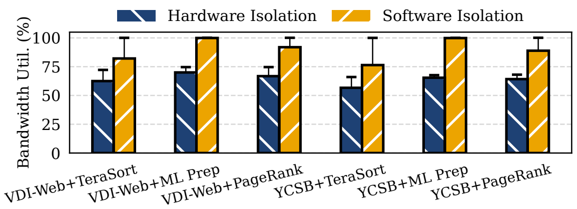
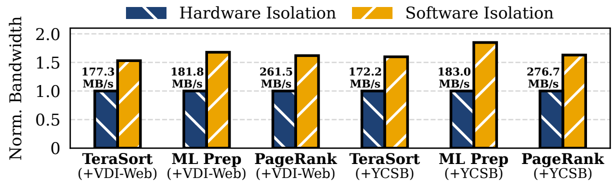
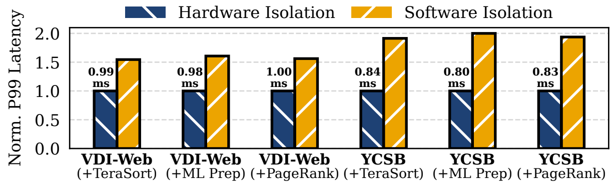


Figure 2. The SSD utilization with different isolation approaches. We show both the average and P95 utilization (error bars) during the execution of different workloads.



(a) I/O bandwidth of the bandwidth-intensive workloads (TeraSort, ML Preparation, and PageRank).



(b) P99 latency of latency-sensitive workloads (VDI-Web & YCSB).

Figure 3. Storage performance of the collocated workloads with different performance isolation approaches.

two different types of workloads : (1) bandwidth-intensive workloads (e.g., TeraSort, PageRank, and ML Preparation) and (2) latency-sensitive workloads (e.g., YCSB and VDI-Web). As shown in Figure 2, the software isolation approach can improve the average bandwidth utilization by up to 1.52× (1.39× on average). We also plot the P95 bandwidth utilization in Figure 2, and show that the hardware isolation always cannot fully utilize the SSD bandwidth.

We also present the throughput and latency of the two collocated workloads in Figure 3 (a) and Figure 3 (b), respectively. The software isolation delivers up to 1.84× (1.64× on average) higher bandwidth for bandwidth-intensive applications, but causes up to 2.02× higher tail latency for latency-sensitive applications, in comparison with the hardware isolation approach. As the software isolation approach allows bandwidth-intensive workloads to utilize the available SSD bandwidth, therefore, it benefits the storage utilization improvement (e.g., YCSB+ML Prep in Figure 2 and Figure 3). However, the read/write interference between collocated applications will cause long tail latency. Since hardware-isolated vSSDs are fully isolated, even though a vSSD has idle storage bandwidth, it does not allow the collocated vSSDs to utilize it, resulting in resource underutilization.

In summary, neither software isolation nor hardware isolation can achieve both performance isolation and improved storage utiliztion at the same time.

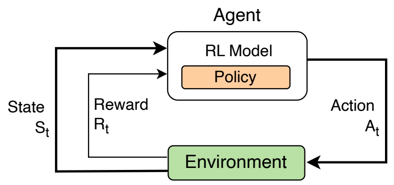


Figure 4. An example of the learning process of RL.

2.3 ML for Storage Resource Management

Most recently, researchers proposed using machine learning (ML) techniques for storage management [2, 26, 38, 49]. For instance, SSDKeeper [26] developed a DNN model to learn the demanded number of flash channels for a vSSD based on its workload patterns. BlockFlex [38] used a heuristic-based approach to predict the availability of storage resources that can be harvested by low-priority VMs like spot VMs [5, 6].

Unfortunately, these approaches fall short of achieving both performance isolation and improved storage utilization simultaneously for three reasons. First, as cloud applications would change their data access patterns and resource demands at runtime, existing approaches cannot capture their changes in a timely manner, which misses potential optimization opportunities. Second, the storage states of the shared SSD will be updated at runtime after each I/O activity. These state updates have a significant impact on storage performance, e.g., a read/write request would delay the incoming requests, and a GC event will stop the SSD from issuing new I/O requests. However, none of existing approaches considers the runtime changes of storage state in their decision-making process. Third, all these existing approaches mainly targeted storage utilization improvement, they did not treat the performance isolation as the first-class citizen.

Why RL? With the above reasons, we argue that RL is a natural fit to meet our goal in the multi-tenant cloud setting, because of its unique advantages of taking actions to meet defined purposes in a dynamic environment. As shown in Figure 4, a standard RL setting has an agent interacting with a given environment repeatedly. The agent obtains a state (S_t) from the environment and then takes an action (A_t) based on its policy in a model. The policy maps the current state to an action from an action set (i.e., $policy\ (state) \rightarrow action$). After the selected action is conducted, the agent will receive a reward (R_t), and the next state will be generated.

For virtualized SSDs in the multi-tenant cloud setting, the workload patterns and SSD states change dynamically, forming the dynamic environment. The interaction between each vSSD and the environment is similar to that of an RL agent: taking actions against the environment, and receiving rewards depending on their impact on the defined optimization goals. Thus, we can map the virtualized SSD management and I/O scheduling problem to the RL learning process. To the best of our knowledge, this is the first work to investigate RL in virtualized storage resource management.

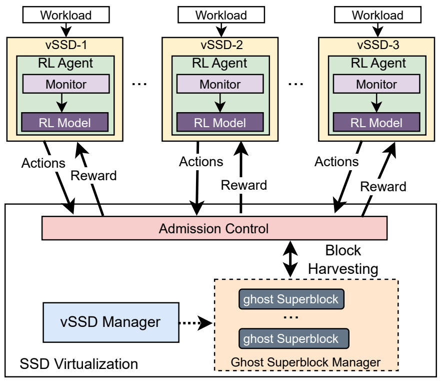


Figure 5. System architecture of FleetIO.

3 Design and Implementation

Exploring RL to manage virtualized SSDs is not easy, we have to overcome the following research challenges.

3.1 Research Challenges

- First, we need to abstract the virtualized SSD management into the RL process, and define corresponding RL actions and reward functions for achieving the specific goals (i.e., performance isolation and improved storage utilization).
- Second, we need to rethink the design and implementation of the virtualized SSD framework for facilitating the employment and deployment of RL techniques.
- Third, as we support multiple virtualized SSD instances on a shared SSD, the RL deployment should be able to scale well to achieve improved overall efficiency.
- Fourth, the RL deployment should be lightweight, which can make smart decisions instantly while introducing minimum overheads to the virtualized SSD infrastructure.

In the following, we will address these challenges through the design and implementation of FleetIO.

3.2 System Overview of FleetIO

We show the system architecture of FleetIO in Figure 5. FleetIO deploys an RL agent in each vSSD, enabling scalability to support multiple vSSDs. Each agent will monitor the I/O traffic of the vSSD, extract the essential storage states (e.g., I/O latency, throughput, and queue delay), and transfer them into RL states that are the input for the RL model. An agent will select the RL action that earns the highest predicted reward and issue the action to the SSD virtualization platform. The RL actions are validated by admission control before they are executed on the shared SSD.

FleetIO optimizes its decision-making process through the RL reward function. As each workload typically optimizes for a combination of I/O latency and bandwidth, FleetIO defines reward functions that balance the workload bandwidth while minimizing the SLO violations. Different applications may have different trade-offs for these metrics. For instance, TeraSort [12] is hungry for bandwidth while key-value stores are sensitive to latency. FleetIO enables fine-tuning the reward function for each RL agent to maximize its effectiveness. To improve the storage utilization, it extends the SSD virtualization and enables automated fine-grained storage harvesting among collocated vSSDs by defining it as an RL action.

3.3 RL Formulation for Multi-Tenant vSSDs

FleetIO formulates the multi-tenant vSSD management and scheduling into a multi-agent RL problem. It defines the RL states, actions, and reward for each agent.

3.3.1 Representation of Storage States in RL. The state space of our RL model is generated from the observable virtualized SSD environment. The RL states include the key storage performance indicators, storage states, and workload patterns. We list the RL states in Table 1.

As for the performance indicators of each vSSD, we maintain performance counters to keep track of the average vSSD bandwidth (Avg_BW), the average IOPS (Avg_IOPS), the average vSSD latency (Avg_Lat), and the percentage of SLO violations (SLO_Vio) of the past I/O requests.

Cloud platforms usually offer predefined tail latency SLO for vSSD instances. This can be specified in different ways, such as the 99th percentile (P99) latency and a few times the median latency with offline profiling. By default, we set the SLO to the P99 tail latency of I/O requests in a hardware-isolated vSSD. Instead of using the tail latency to indicate the performance isolation level of a vSSD, we use the percentage of SLO violations, as it provides a more reasonable way to reflect the performance isolation level in real scenarios.

We also consider the workload characteristics in the RL states. They include the read/write ratio (RW_Ratio), and a vSSD's I/O request delay ($QDelay$). The read/write ratio of the I/O requests is an indicator of the current I/O request distribution. We quantify the I/O request delay by creating a dynamic virtual queue in each vSSD to track all the pending I/O requests. The $QDelay$ is an indicator of the number of outstanding requests and their queuing delay. A larger number of outstanding I/O requests means that the vSSD needs more storage resources (e.g., more flash channels to serve I/O requests). These RL states will help the model learn the workload patterns so that it can adapt quickly to its changes.

To represent the vSSD states in the RL model, we include the available storage capacity ($Avail\_Capacity$), garbage collection status (In_GC), and I/O request priority ($Cur\_Priority$). $Avail\_Capacity$ represents the free storage capacity of a vSSD. In_GC is a boolean state indicating if the vSSD is performing GC. As GC has a great impact on vSSD performance, we include it in the RL state to help optimize the I/O scheduling. $Cur\_Priority$ represents the current I/O request priority of a vSSD. The RL model uses it to adjust the priority of I/O requests to meet the specified SLOs.

Table 1. Definition of RL states in FleetIO.

Symbol	Definition
Avg_BW_t	Average I/O bandwidth of the vSSD (MB/s)
Avg_IOPS_t	Average IOPS of the workload ($reqs/s$)
Avg_Lat_t	Average request latency of the vSSD (us)
SLO_Vio_t	Percentage of vSSD SLO violations (%)
$QDelay_t$	I/O request delay in a vSSD
RW_Ratio_t	Read/write ratio of the workload (%)
$Avail\_Capacity_t$	Available storage capacity (GB)
In_GC_t	Whether the vSSD is performing GC
$Cur\_Priority_t$	The current I/O request priority

To help the agent make better decisions , we also include two states that are shared among RL agents: (1) the sum of average IOPS (Avg_IOPS) of collocated agents and (2) the sum of SLO violations (SLO_Vio) of collocated agents. These shared states inform the agent about other agents, which can help avoid conflicts. For example, the agent can harvest fewer resources when the other agents have high SLO violations. FleetIO does not explicitly include system-level information, as it is transparent to upper-level VMs and applications.

We use a time window (2 seconds by default) to track the RL states. There is a trade-off for deciding the time window length. Using a large time window would reduce the opportunity for fine-grained harvesting, and using a small time window will increase the decision frequency and introduce more performance overheads. Each time window has 11 states (9 states in Table 1 and 2 shared states). To make accurate decisions, we concatenate states from three prior time windows together for capturing dynamic changes in storage states. We do not concatenate too many time windows, since this will enlarge the state space, which further increases the training overhead.

3.3.2 Definition of RL Actions for Each vSSD. We define the RL actions in FleetIO based on our optimization goals: (1) performance isolation guarantee and (2) storage utilization improvement. For the former one, it takes the best-effort approach to minimize the SLO violations for each vSSD by adjusting its I/O request priority. Optimizing I/O scheduling can enforce performance isolation by prioritizing requests from the vSSDs that have difficulty in meeting the tail latency SLOs. For the latter one, FleetIO enables fine-grained storage harvesting from collocated vSSDs. For example, one vSSD can temporarily use the idle resources of another vSSD to gain additional I/O bandwidth and increase storage utilization. Therefore, FleetIO defines three actions as shown in Table 2.

- $Harvest(gsb\_bw)$ decides how much storage resource a vSSD should harvest from others, where gsb_bw is a value that specifies the storage bandwidth needed, and FleetIO turns it into the demand of superblocks (see §3.6). To simplify the decision process, we combine the read and write bandwidth in gsb_bw.
- $Make\_Harvestable(gsb\_bw)$ decides how much storage resource a vSSD makes harvestable for other collocated

Table 2. RL actions defined in FleetIO.

Symbol	Definition
$Harvest(gsb\_bw)$	Storage bandwidth to harvest
$Make\_Harvestable(gsb\_bw)$	Storage bandwidth harvestable for other vSSDs
$Set\_Priority(level)$	Set the I/O request priority

vSSDs. It offers the flexibility for each vSSD to decide the harvestable bandwidth depending on its current state.

- $Set\_Priority(level)$ decides the I/O request scheduling priority (i.e., low/medium/high) of a vSSD. It helps each vSSD meet the performance isolation goal.

Each RL agent learns to take appropriate actions to benefit the overall storage utilization and performance isolation, based on the RL states (§3.3.1). An agent may take the $Harvest()$ action to gain more bandwidth from its collocated vSSDs with low Avg_IOPS. It may also take the $Make\_Harvestable()$ action to make its resource harvestable if it has low bandwidth utilization. If a vSSD runs GC frequently, FleetIO will reduce its harvestable storage resource in $Make\_Harvestable()$ action. If a vSSD experiences high SLO violations or high I/O queuing delay, the RL agent will increase the priority level for its requests using $Set\_Priority()$. We do not call $Set\_Priority()$ in other actions for simplifying the management and reasoning of the RL action space.

FleetIO provides a device-agnostic design, it does not explicitly include device-level information in the RL model. Specifically, the RL does not specify the flash channels/chips and storage capacity that will be harvested. Instead, the RL agent will only decide the storage bandwidth to harvest (or make harvestable), as it is the most critical factor for the optimization goals. We extend the SSD virtualization framework to manage the harvestable blocks corresponding to the underlying storage architecture in §3.6. Note that RL actions are executed in the background, and each vSSD issues storage I/O requests normally.

3.3.3 Definition of Reward Functions.

We also reflect the optimization goals of the RL model in its reward function. The most critical performance metrics of a vSSD are the storage bandwidth and tail latency. The storage bandwidth of a vSSD reflects how much it can utilize the provided storage bandwidth. The tail latency of a vSSD reflects the performance isolation level it can obtain. Therefore, we define the reward function for each vSSD as follows:

$$R_{single} = (1-\alpha) * \frac{Avg\_BW_{RL}}{Avg\_BW_{guar}} - \alpha * \frac{SLO\_Vio_{RL}}{SLO\_Vio_{guar}} \quad (1)$$

Avg_BW_{RL} and SLO_{RL} represent the bandwidth and the percentage of SLO violations. FleetIO can directly obtain them from the RL states. Avg_BW_{guar} represents the I/O bandwidth of the allocated storage resources. It can be measured by counting the number of allocated flash channels, as the aggregated storage bandwidth increases linearly with the number of flash channels (i.e., N channels can deliver $N \times bandwidth\_per\_channel$ aggregated bandwidth). We

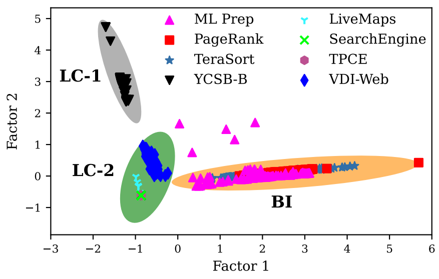


Figure 6. A clustering of typical cloud workloads.

empirically set the bandwidth per channel depending on the SSD hardware used in the deployment. SLO_{guar} is the guaranteed SLO violation, which is usually set by the cloud vendors. In our study, we set the target percentage of SLO violations to be 1%. FleetIO uses Avg_BW_{guar} and SLO_{guar} as the baselines because we want to maximize bandwidth utilization while minimizing SLO violations. The parameter α manages the tradeoff between performance isolation and storage utilization. A smaller α prioritizes storage utilization, and a larger α prioritizes performance isolation.

Reward for Multiple Agents. While RL agents optimize their reward, they should account for other agent's rewards to improve the overall storage efficiency. To achieve this, we use a coefficient β to balance the rewards among collocated RL agents. We define the reward function for a $vSSD_i$ as:

$$R_{i,\ multi\_agent} = \beta * R_{i,\ single} + (1-\beta) * \frac{\sum_{v \in V}^{v \neq i} R_{v,\ single}}{|V|-1} \quad (2)$$

where V is the set of collocated vSSDs and $|V|$ is its cardinality. With larger β, each agent will care more about its own reward. Otherwise, it will emphasize the rewards of other agents. We set $\beta = 0.6$ by default based on our study of its impact on diverse cloud applications. We will demonstrate the effectiveness of the multi-agent reward function in §4.4.

3.4 RL Fine-Tuning for Different Workload Types

Each vSSD may host workloads with various storage demands that prefer optimizations for different metrics (e.g., latency-sensitive and bandwidth-intensive). Therefore, a unified reward function for all collocated vSSDs may result in overall inefficiency. In FleetIO, we fine-tune each RL agent for different workload characteristics. For example, bandwidth-intensive workloads prefer high bandwidth, so we can customize its reward function with a lower α.

However, it is challenging to fine-tune the RL model for each individual workload. Instead, FleetIO categorizes the workloads into different types, and fine-tunes the reward function for each type. FleetIO uses a clustering approach to learn the workload type with collected block I/O traces from the vSSD.

FleetIO divides the I/O traces into small windows (10K requests per window). For each window, we extract four I/O features: read bandwidth, write bandwidth, LPA entropy, and average I/O size. We sample windows from 9 typical cloud workloads and cluster them with k-means, using 70% of workload traces as the training set and 30% for testing. We use Principal Component Analysis (PCA) to present the result in two dimensions. Figure 6 shows that we can separate the bandwidth-intensive workloads (e.g., TeraSort, PageRank, and ML Prep) from the latency-sensitive ones (e.g., TPCE, SearchEngine, and VDI-Web) from their I/O patterns. Compared to other latency-sensitive workloads, YCSB-B has lower LPA entropy (i.e., better locality), so it has its own cluster. 98.4% of the testing data points fall into the ground-truth clusters, showing that clustering has high accuracy.

To fine-tune the reward function for each workload type, we use the workload closest to the center of each cluster. We examine the percentage of SLO violations and bandwidth utilization of the selected workload using different reward functions by binary searching α between 0 and 1. We select the optimized reward function that ensures the workload does not exceed the SLO violation threshold (5% by default) while delivering the highest bandwidth improvement.

FleetIO uses the trained clusters to decide the optimized reward function for new workloads. If the I/O pattern falls into an existing cluster, FleetIO will use the pre-tuned reward function. If not, FleetIO will use the unified reward function (with $\alpha = 0.01$, see §3.3.3), mark this workload for offline tuning, and utilize the aforementioned fine-tuning method to learn an optimized α. In FleetIO, we perform offline tuning periodically (1 hour by default). As FleetIO aggregates more of such workloads, a new cluster will be formed, such that we can reuse the learned experience.

3.5 Admission Control for RL Actions

Although RL agents make decisions independently, it is necessary to execute them in a coordinated fashion on the shared SSD. It enables cloud providers to customize their permission checking for each vSSD's actions. For example, cloud providers may prevent high-priority VMs from making their resources harvestable, even if doing so would benefit overall resource utilization. Alternatively, cloud providers may prevent low-priority VMs (e.g., Spot VMs) from harvesting at all. Thus, we design an admission control mechanism (Figure 5) that ingests each RL agent's *Harvest()* and *Make_Harvestable()* actions, filters out inadmissible actions, and submits the remaining actions to the gSB manager (§3.6) which is in charge of the harvestable storage.

Instead of processing each action individually, FleetIO processes them in batches and reorders each batch to process *Make_Harvestable()* actions first. This executes the actions that provide the harvestable resource before the ones that consume, which maximizes harvestable resources availability and avoids immediate reclamation. By default, we select a batch size of every 50 milliseconds without sacrificing the benefit of executing actions promptly.

```
typedef struct gSB {
        int n_chls;      //number of channels
        int capacity;    //capacity of GSB (in GB)
        boolean in_use;  //whether the gSB is harvested
        vssd_t *home_vssd; //vSSD that owns the resource
        vssd_t *harvest_vssd; //vSSD harvesting the gSB
} gSB_t;
```

Figure 7. The metadata of a ghost superblock (gSB).

FleetIO ranks the *Harvest()* actions when it detects the demand for harvestable resources exceeds the supply. FleetIO provides the flexibility for cloud providers to specify the policies to handle such contention (e.g., priority-based or fair-sharing). To support this, FleetIO stores additional metadata in each vSSD. The admission control can query the metadata to implement its custom policies. By default, the admission control serves *Harvest()* actions using the first-come-first-serve policy. If the harvest actions need to harvest more storage resources than available gSBs, vSSDs with less harvested resources will have higher priority to harvest the available storage resources.

Note that the admission control is off the critical path of storage I/O requests and its batch processing of actions also incurs trivial performance overheads (see §4.7).

3.6 A New Abstraction in SSD Virtualization for RL

As FleetIO aims to improve the overall storage utilization of the shared SSD by enabling storage resource harvesting among collocated vSSDs (§3.3.2), it requires a simple way to interact with the SSD virtualization framework for accessing the harvestable storage resources. Instead of relying on the RL agents to manage the harvestable resources, FleetIO proposes a new abstraction named ghost superblock (gSB) and integrates it into the SSD virtualization framework.

3.6.1 Ghost Superblock. We show the metadata of a gSB in Figure 7. Each gSB consists of a harvestable superblock that stripes across one or more channels (i.e., *n_chls*). The free blocks of the harvestable superblock are evenly striped across all the chips in each channel. The *capacity* represents the number of blocks in the gSB. By default, we harvest a fixed number of blocks from each channel. For example, in our SSD, the minimum size of a superblock that stripes across a single channel is 16 flash blocks (64MB). The *home_vssd*, *harvest_vssd* and *in_use* are used during the harvesting process to indicate which vSSD gave up the resources for the gSB, which vSSD harvests the gSB (if any), and if the gSB is currently harvested, respectively. All metadata is stored in the gSB and is initialized when the gSB is created.

3.6.2 gSB Management. Figure 8 shows an overview of the gSB management. We now discuss the gSB management and how it works with the RL actions in detail.

Managing gSBs. The gSB manager manages the gSBs in the gSB pool using a set of linked lists. Each linked list stores all gSBs with the same number of channels (i.e., *n_chls*). To simplify best-fit searching, we index these lists by the *n_chls*

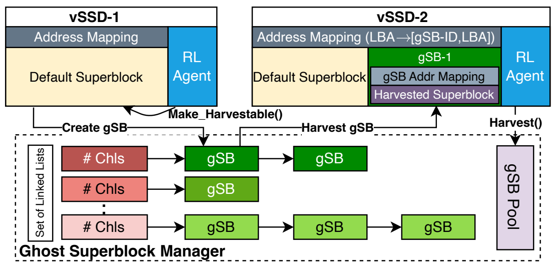


Figure 8. gSB management in FleetIO.

of the gSBs they store and sort the lists by their gSBs' *n_chls*. The total number of lists in the pool corresponds to the total number of physical channels in the SSD (e.g., up to 32 channels in many SSDs). To support high concurrency, the gSB pool is implemented with lock-free linked lists [14].

Creating gSBs. FleetIO creates a new gSB when an RL agent takes the *Make_Harvestable()* action. The action specifies the harvestable bandwidth (i.e., *gsb_bw*) that the new gSB should offer. To decide *n_chls* for the gSB, we divide the harvestable bandwidth by the maximum bandwidth of a single channel (e.g., 64 MB/s), rounding down. The *capacity* is set to *n_chls* multiplied by the minimal superblock size (i.e., 64MB). The *home_vssd* is initialized with a pointer to the vSSD providing the harvestable resources. The *harvest_vssd* and *in_use* are set to null and 0, respectively. The gSB manager will then obtain free blocks from the *home_vssd* and initialize a block-level mapping table for them. Note that we do not create new gSBs on channels with less than 25% free blocks (see §3.7). The created gSB is inserted into the head of the list indexed by its *n_chls*. We also add the gSB to the harvestable gSB list maintained in the *home_vssd* metadata for future reference.

Harvesting gSBs. An RL agent harvests a gSB when the agent takes the *Harvest()* action. Similar to gSB creation, *n_chls* for the gSB is determined by the storage bandwidth (i.e., *gsb_bw*) specified in the *Harvest()* action. To obtain a gSB with the desired *n_chls* from the pool, we find the list indexed by the same *n_chls* and then get the first gSB from the list. If the list is empty, we first search the lists with smaller *n_chls* until we can obtain a gSB, before searching lists with larger *n_chls*. Once a suitable gSB is found, the *harvest_vssd* is set to point to the vSSD that is harvesting and *in_use* is set to 1 to indicate the gSB is harvested. After a gSB is harvested, the home vSSD and the vSSD harvesting the gSB will share the storage bandwidth of the corresponding channels. Note that the gSB manager prevents a vSSD from harvesting its own gSBs by checking the *home_vssd*.

gSBs are harvested transparently to workloads in the harvest vSSD. To support this, the address mapping in the vSSD is extended to support an additional indirection layer that maps the vSSD logical block addresses (LBAs) into the LBAs of the gSB (see vSSD-2 in Figure 8). The gSB translates the LBA to the physical block address (PBA). This also enables each vSSD to harvest multiple gSBs to improve its bandwidth.

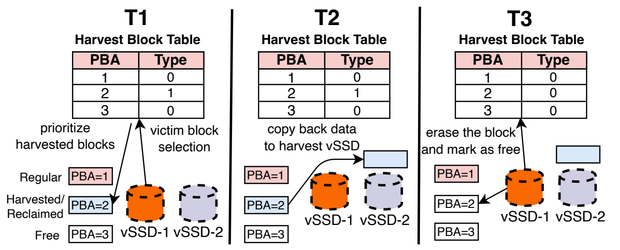


Figure 9. Garbage collection in FleetIO.

Reclaiming gSBs. FleetIO checks whether it should also reclaim gSBs when an RL agent takes the *Make_Harvestable()* action. The gSB manager will scan the harvestable gSB list maintained in the metadata of the *home_vssd* for any gSBs with more *n_chls* than the *n_chls* specified by *Make_Harvestable()*. For each such gSB, the gSB manager checks its *in_use* flag. gSBs with *in_use* set to 0 are removed from their list in the pool, their free blocks are returned to the *home_vssd*, and the gSB is destroyed. gSBs with *in_use* set to 1 will be reclaimed in a lazy manner. GC will migrate the valid data in those channels to blocks owned by the *harvest_vssd* (§3.7).

3.7 Exception Handling

We discuss how FleetIO gracefully handles the exceptions during its execution.

Garbage Collection. As a vSSD offers gSBs for harvesting, the free blocks in its channels may be consumed more quickly, triggering more frequent GCs to erase the invalid data, and even worse, may exhaust the capacity of the vSSD.

To solve this problem, FleetIO migrates data from blocks harvested by other vSSDs (i.e., harvested blocks) to blocks owned by the *harvest_vssds*. As shown in Figure 9, FleetIO prioritizes selecting the blocks reclaimed by the *home_vssd* (i.e., reclaimed blocks) and harvested blocks over the vSSD's regular blocks, and writes their valid data to free blocks of the *harvest_vssds*. The harvested blocks may have a slightly higher valid page ratio than those of the home vSSD. However, we observe that this does not introduce much write amplification (< 5% in our experiments).

FleetIO tracks the reclaimed and harvested blocks with a Harvested Block Table (HBT), it tracks the block type of each PBA. FleetIO uses one bit to indicate the two block types: regular (0) or harvested/reclaimed (1), consuming at most 0.5MB storage space (assuming a 1TB SSD with a 4MB block size). We do not differentiate between harvested and reclaimed blocks as GC will treat them the same. The gSB manager can detect both cases by marking the PBAs of all harvested blocks as harvested/reclaimed (1) when it creates the gSB. Blocks are marked as regular (0) after erased by GC.

vSSD Deallocation. When a vSSD is deallocated, all of its data is marked as invalid, such that it will be erased in the next GC. To support the harvesting of deallocated flash blocks, FleetIO uses a placeholder vSSD that owns the free resources and makes them available for harvesting.

Table 3. Experimental parameters in FleetIO.

RL Parameters	Value	SDF Parameters	Value
Decision interval	2 secs	Capacity	1TB
Reward coefficient β	0.6	# Channels	16
Learning rate	10^{-4}	# Chips/Channel	4
Discount factor γ	0.9	Page Size	16KB
Hidden layer sizes	[50, 50]	Max queue depth	16
Batch size	32	Overprovisioning ratio	20%

3.8 Implementation Details

FleetIO Framework. We implement FleetIO on a real open-channel SSD. We list the core configuration parameters for the open-channel SSD in Table 3. It provides the programmability for developers to implement their own flash management layer in the host. We follow the prior studies [9, 17, 38] to implement the basic vSSD abstraction on the open-channel SSD, with 3.5K lines of C code with the SDK library of the device. After that, we integrate our RL-based vSSD management into it with 2K lines of Python code.

FleetIO Training. We develop our RL model based on Proximal Policy Optimization (PPO) [40] using RLlib [25] and PyTorch [35]. We list the hyper-parameters of the model in Table 3. We pre-train the RL model offline using a set of storage workloads (e.g., LiveMaps, TPCE, SearchEngine, and Batch Analytics) that are not used in the evaluation. We perform pre-training on a server configured with 24 Intel Xeon CPU (E5-2687W v4) processors running at 3.0GHz and 96GB DRAM. We pre-train the model using Ray [32]. Since we have a limited number of open-channel SSDs, we collect training datasets by running multiple workloads on the SSD simulator WiscSim in parallel to accelerate the training process [15]. We pre-train the model with 2,000 iterations using a batch size of 256. With the fine-tuning method discussed in §3.4, for the workload clusters LC-1, LC-2, and TO in Figure 6, the values of their coefficients α are 2.5×10^{-2}, 5×10^{-3}, and 0, respectively.

FleetIO Deployment. We deploy the pre-trained model in FleetIO framework. Upon the creation of a vSSD, an RL agent will be deployed. Its model size is 2.2MB with 9K parameters. Each RL agent will monitor the vSSD states at runtime (§3.3), collect storage I/O traces at the block level periodically to learn the workload type (§3.4), and take actions accordingly. FleetIO does not introduce much overheads to the SSD virtualization (see our evaluation in §4.7).

4 Evaluation

Our evaluation shows that: (1) FleetIO achieves both improved resource utilization and performance isolation (§4.2); (2) the multi-agent design of FleetIO demonstrates its effectiveness as we scale the number of vSSDs (§4.3); (3) the fine-tuned reward function of FleetIO shows its positive impact on RL decisions (§4.4); (4) FleetIO can benefit both hardware-isolated and software-isolated vSSDs (§4.5); (5) FleetIO is resilient to workload changes in the collocated vSSDs (§4.6); (6) FleetIO introduces negligible overhead to the SSD virtualization (§4.7).

Table 4. Workloads used in our evaluation.

Category	Workload	Description
Bandwidth-intensive	TeraSort [12]	Sort large datasets using Hadoop.
	ML Prep [1]	Image preprocessing for machine learning.
	PageRank [11]	Compute a graph's pagerank.
Latency-sensitive	VDI-Web [23]	Enterprise virtual desktop running interactive applications.
	YCSB [51]	YCSB workloads backed by SQLite.

4.1 Experimental Setup

We describe the hardware setting for the experiments in §3.8. We use 5 typical cloud workloads listed in Table 4. These workloads cover both bandwidth-intensive workloads and latency-sensitive workloads. For each experiment, we run a combination of the workloads on the shared SSD. In the experiments, we use all 16 channels of the SSD and allocate them to vSSDs depending on the experimental setting. Before the experiments, we run a mix of different workloads to warm up the vSSDs to consume at least 50% of the free blocks to ensure that GC will be executed during the experiments. The GC uses a lazy algorithm with a 20% free block threshold. By default, each vSSD when using FleetIO starts with hardware isolation unless otherwise specified. We compare FleetIO with the following state-of-the-art approaches.

- **Hardware Isolation**: Each vSSD owns an equal share of hardware-isolated flash channels. Hardware isolation delivers the strongest performance isolation (see §2).
- **SSDKeeper**: SSDKeeper [26] uses a deep neural network (DNN) to decide the hardware-isolated static resource partitioning for vSSDs that minimizes average latency.
- **Adaptive**: The number of flash channels allocated to vSSDs in each time window is proportional to their bandwidth utilization in the prior time window [31].
- **Software Isolation**: All vSSDs have shared access to all the allocated flash channels. We use a token bucket I/O rate limiter to throttle I/O requests from different vSSDs [46]. To further minimize interference, we use stride scheduling [48, 52] to ensure that workloads with high I/O intensity do not starve workloads with low I/O intensity. Software isolation has the best overall storage utilization.

4.2 Resource Utilization and Performance Isolation

FleetIO achieves both improved storage utilization and performance isolation. To demonstrate this, we collocate two vSSDs, one running a bandwidth-intensive workload and one running a latency-sensitive workload (see Table 4).

We present the tradeoff between improved storage utilization and P99 tail latency in Figure 10. FleetIO improves the bandwidth utilization over Hardware Isolation by up to 1.39× (1.30× on average) while reducing the P99 tail latency over Software Isolation by up to 1.58× (1.47× on average).

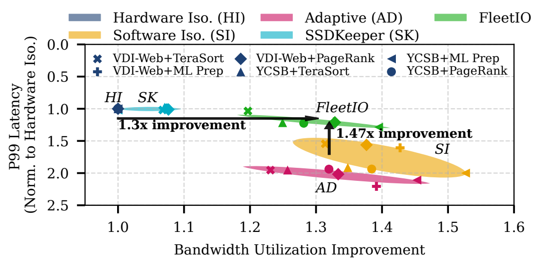


Figure 10. Tradeoff between overall storage utilization and P99 tail latency of latency-sensitive workloads.

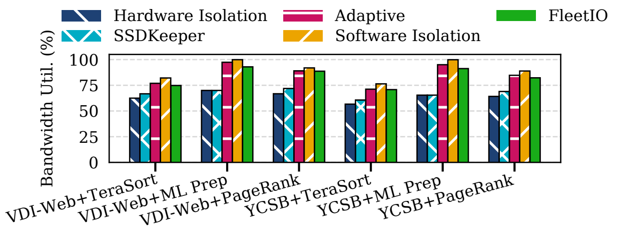


Figure 11. FleetIO's benefit for storage utilization.

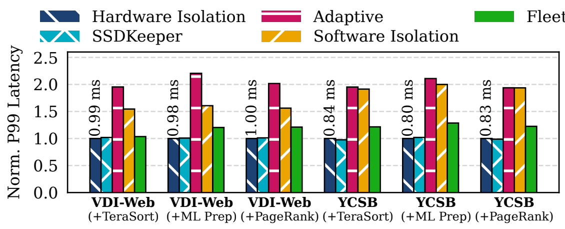


Figure 12. Normalized P99 latency of the latency-sensitive workloads (VDI-Web and YCSB).

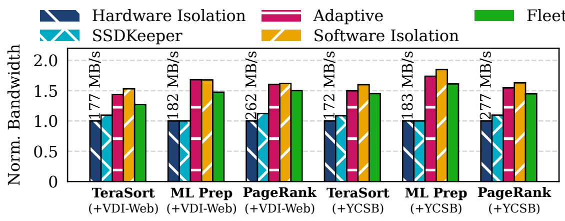


Figure 13. I/O bandwidth of the bandwidth-intensive workloads (TeraSort, ML Prep, and PageRank).

No other approach achieves this tradeoff. Hardware Isolation and SSDKeeper achieve high performance isolation, but deliver at most 1.08× storage utilization improvement. Software Isolation and Adaptive achieve high storage utilization improvement as they allow collocated vSSDs to share flash channels from each other, but they increase the P99 latency by 1.76× and 2.03×, respectively, due to the I/O interference.

Hardware Isolation and SSDKeeper cannot achieve both high storage utilization and strong performance isolation, because they statically allocate hardware-isolated channels. This can guarantee low tail latency but neglect the dynamically changing bandwidth of the collocated vSSDs. FleetIO improves the storage utilization over both policies up to 1.39× (see the detailed results in Figure 11), which is 93% of the best storage utilization. This is because FleetIO enables the bandwidth-intensive workload to dynamically harvest resources as the bandwidth demand fluctuates.

Software Isolation and Adaptive cannot achieve both objectives either. Even though they enable a more flexible channel allocation to the vSSDs, they do not emphasize the performance interference suffered by collocated vSSDs. FleetIO incorporates both the average bandwidth and the SLO violations into the reward function, so RL agents consider both metrics in their decisions. Therefore, FleetIO achieves 1.29-1.89× lower P99 tail latency (see detailed results in Figure 12), and is within 1.2× of the strongest performance isolation. Compared to Hardware Isolation, FleetIO increases the P95 and P99.9 tail latency by only 3% and 8%, respectively. This is expected, as the bandwidth-intensive workload harvests storage bandwidth to meet its needs, the collocated vSSD hosting the latency-critical workload will dynamically take *Set_Priority* action to enforce higher priority to meet its SLO.

To further understand the storage utilization improvement, Figure 13 shows the I/O bandwidth of the bandwidth-intensive vSSD normalized to the bandwidth of Hardware Isolation. FleetIO improves the bandwidth over Hardware Isolation by 1.27-1.61× (1.46× on average) and over SSDKeeper by 1.37× on average. FleetIO achieves up to 93% of Software Isolation's bandwidth (89% on average) and 91% of Adapative's bandwidth on average. This shows that the RL agent effectively learns the workload patterns and harvests spare storage resources from the collocated vSSDs. Since the bandwidth-intensive workload dominates the bandwidth utilization of the SSD, its bandwidth improvement is reflected in the storage utilization improvement of FleetIO. Note that we do not expect FleetIO to reach the ideal performance isolation or storage utilization. FleetIO employs a best-effort approach to achieve improved performance isolation and storage utilization via fine-grained storage harvesting.

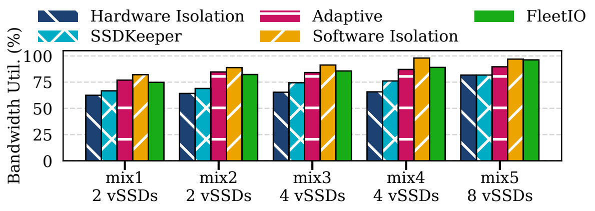
4.3 Scalability of FleetIO

FleetIO improves resource utilization while ensuring performance isolation, as we scale the number of vSSDs. We run mixes of 4-8 workloads shown in Table 5.

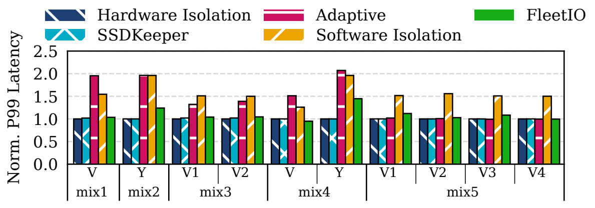
As shown in Figure 14 (a), FleetIO consistently improves the overall bandwidth utilization as we increase the number of vSSDs, achieving 1.33× and 1.18× improvement over Hardware Isolation for 4-vSSD and 8-vSSD mixes, respectively. These improvements are up to 94% and 99% of the best storage utilization (i.e., Software Isolation), indicating that FleetIO obtains more benefit as we scale the number of vSSDs. This is because FleetIO can match the diverse harvestable resources from the latency-sensitive vSSDs with the diverse harvest storage demand from the bandwidth-intensive vSSDs.

Table 5. Workload combinations for scalability experiments.

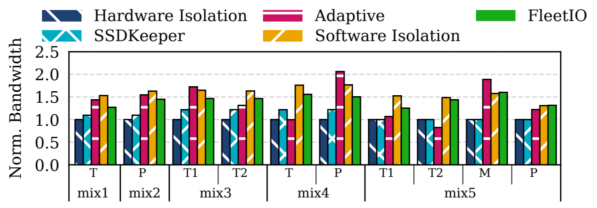
#vSSDs	Label	Workload Combinations
2	mix1	VDI-Web, TeraSort
	mix2	YCSB, PageRank
4	mix3	2 VDI-Web, 2 TeraSort
	mix4	VDI-Web, YCSB, TeraSort, PageRank
8	mix5	4 VDI-Web, 2 TeraSort, PageRank, MLPrep



(a) The average storage utilization.



(b) P99 tail latency of latency-sensitive workloads.



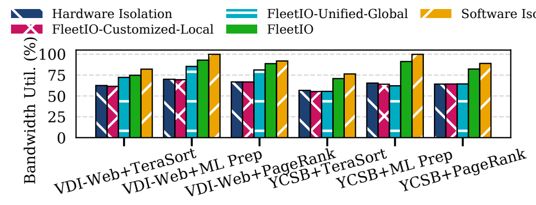
(c) I/O bandwidth of bandwidth-intensive workloads.

Figure 14. Scalability of FleetIO on the number of vSSDs.

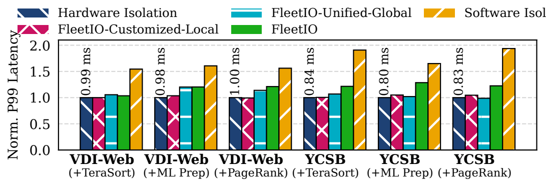
FleetIO ensures high performance for each vSSD as we scale the number of vSSDs. Figure 14 (b) shows that FleetIO ensures performance isolation–it keeps the P99 latency increase over Hardware Isolation to less than 10%. FleetIO ensures performance isolation because of the greater availability of harvestable resources. Figure 14 (c) shows that FleetIO improves the bandwidth-intensive vSSDs' performance by 1.45× on average. FleetIO improves the bandwidth of each vSSD by at least 1.25×. Meanwhile, other policies even decrease the bandwidth. FleetIO achieves uniform benefit, because its multi-agent reward function can effectively balance the improvement for each vSSD while considering the overall storage utilization.

4.4 Benefits of an Optimized Reward Function

FleetIO benefits from a customized reward function for each workload type and balancing the reward of each agent with its collocated agents. We analyze the performance benefit of FleetIO's reward function by selectively disabling each optimization: *FleetIO-Unified-Global* selects a unified $\alpha = 0.01$ optimized for all agents instead of the customized reward



(a) The average storage utilization.



(b) P99 tail latency of latency-sensitive workloads.

Figure 15. Effectiveness of optimized reward function.

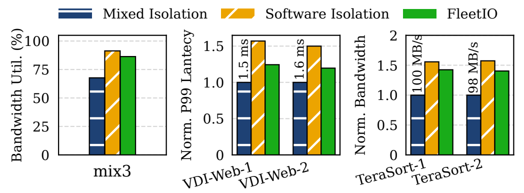


Figure 16. Storage utilization and performance isolation of FleetIO with mixed software- and hardware-isolated vSSDs.

(see §3.4). *FleetIO-Customized-Local* uses the custom α but each agent only optimizes for its reward ($\beta = 1$). FleetIO uses both optimizations.

Figure 15 shows the bandwidth utilization and P99 latency against Hardware and Software Isolation. FleetIO-Customized-Local shows that despite using customized reward functions, without using a β to balance the reward of collocated agents, there is little incentive for RL agents to make resources harvestable and FleetIO has similar performance to Hardware Isolation. Alternatively, FleetIO-Unified-Global using a unified reward function for all workloads can be effective but inconsistent. In particular, workload combinations with VDI-Web improve bandwidth, while those with YCSB show little improvement. Instead, FleetIO uses both and can improve the storage utilization while ensuring performance isolation.

4.5 Mixed Hardware and Software Isolation

FleetIO can benefit both software- and hardware-isolated vSSDs. To demonstrate this, we evaluate FleetIO with mix3 (see Table 5) on a mix of hardware- and software-isolated vSSDs. The VDI-Web workloads each run in a 4-channel hardware-isolated vSSD, and TeraSort each run in an 8-channel software-isolated vSSD. *Mixed Isolation* represents the strongest performance isolation in this experiment.

Figure 16 shows that FleetIO achieves 1.27× improved storage utilization compared to Mixed Isolation and 1.42× bandwidth improvement for bandwidth-intensive workloads. FleetIO also achieves more than 94% of Software Isolation's

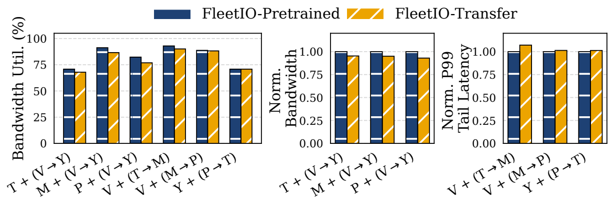


Figure 17. Storage performance of FleetIO after changing the collocated workload. T=Terasort, V=VDI-Web, P=PageRank, Y=YCSB, and M=ML Prep. T + (V⟶Y) indicates the model is tuned with T+V and evaluated on T+Y.

storage utilization and more than 90% of its I/O bandwidth for TeraSort. Furthermore, FleetIO only increases the tail latency over Mixed Isolation by 1.19×. It shows that FleetIO can improve the storage performance across different vSSD isolation approaches.

4.6 Robustness of FleetIO

FleetIO is resilient to changes in the collocated vSSDs. We evaluate FleetIO by changing the collocated workload halfway through the experiment, shown in Figure 17. *FleetIO-Transfer* reports the storage performance measured after switching the collocated workload (e.g., V+T) to the evaluated workload combination (e.g., V+Y). *FleetIO-PreTrained* reports the storage performance of only tuning FleetIO on the evaluated workload combination. Across all workload combinations, the performance of FleetIO-Transfer is within 5% of the performance of FleetIO-PreTrained. It shows that FleetIO does not overfit to the workload patterns of the specific collocated vSSDs and it can benefit storage utilization in the dynamic settings of a real deployment.

4.7 Overhead Sources in FleetIO

FleetIO is lightweight and introduces minimum overhead to cloud platforms. FleetIO takes 2 hours to pre-train the model offline on a server with 24 Intel Xeon CPU processors. During deployment, as we run each RL agent on a single CPU core, each model incurs a 51.2 millisecond fine-tuning overhead every 10 time windows (2 seconds per window) and a 1.1 millisecond inference overhead in each time window. Both fine-tuning and inference are off the critical path of I/O requests. To store the model and model inputs, FleetIO incurs only 2.2 MB storage cost per vSSD.

FleetIO's gSB management and admission control are also lightweight. Since gSB creation only involves metadata operations, it takes <1 μs. Similarly, admission control takes only 0.8 milliseconds to process a batch of 1,000 actions. Both gSB creation and admission control are performed in the background, off the critical path of I/O requests.

5 Discussion

Benefit Sources. It is challenging to tune heuristic-based policies and manually identify their optimal thresholds. This is because, to identify an optimal threshold, we need to consider the correlation of the storage states, the SSD performance, and the workload characteristics. FleetIO performs this tuning implicitly through RL by enabling fine-grained storage harvesting from collocated vSSDs at runtime, while having the SLO violations as a strict condition for its actions. Furthermore, its benefits for storage utilization improvement depend on the opportunities available for harvesting. As cloud platform today has placed diverse workloads on the shared storage, we expect that FleetIO will deliver more benefits (similar to our scalability evaluation in §4.3). In contrast, if the collocated workloads have conflicting goals and offer few opportunities for storage harvesting, the capability of FleetIO will be limited. However, FleetIO will not perform worse than state-of-the-art approaches.

Mispredictions of FleetIO. We do not need to manually correct mispredictions for FleetIO. This is because RL will inherently minimize the impact of mispredictions. It will identify the optimized actions for maximizing the rewards by default, a higher reward value indicates better resource utilization and performance isolation.

Generalizability of FleetIO. FleetIO supports both public and private cloud. For public cloud, since FleetIO significantly improves storage utilization, it can introduce considerable savings for cloud providers that can be passed to the users as a discounted VM price [4]. For private cloud, FleetIO could provide even more benefits using workload insights from tenants. Furthermore, FleetIO can be applied to different cloud storage platforms, because we employ a device-agnostic approach in our design. We can map the gSB abstraction to different types of SSD devices, such as Zoned Namespace (ZNS) SSDs.

Bandwidth and Capacity Harvesting. As for the storage utilization improvement, FleetIO currently focuses on the storage bandwidth rather than the capacity. This is because storage bandwidth is the most critical factor that determines the performance of data-intensive applications. And cloud applications usually have sufficient storage capacity, as the storage capacity cost is much lower than the bandwidth cost.

Security Implications. As we share physical flash channels and blocks across collocated vSSDs, security concerns may arise, for example, data leakage may happen as vSSDs reclaim blocks. This is less of a concern in FleetIO, as all the harvested and reclaimed blocks will be physically erased before returning them to their owners, and the storage harvesting and reclamation procedure is transparent to end users. An adversarial workload may intentionally trigger resource harvesting. For this case, the RL agents will not make their resource harvestable if they realize that such actions constantly hurt the performance of corresponding vSSDs.

6 Related Work

Storage Virtualization. Recent studies have shown the software-defined storage that makes software manage underlying hardware resources can improve storage performance and efficiency [17, 34, 41]. They can be virtualized as system-wide shared resources to provide storage services. Virtualized SSDs make efficient use of storage capacity and performance by slicing resources among multi-tenant applications [20, 27, 42, 43, 54]. However, it has traditionally been a challenge due to the fundamental tussle between the performance isolation and resource utilization (§2). FleetIO exactly tackles this challenge and demonstrates its capability.

Cloud Storage Efficiency. Recent studies [2, 3, 19, 24, 27, 33, 38, 46] reported that the cloud storage utilization is low, although storage virtualization has been employed. Prior studies proposed heuristic-based approaches to allow low-priority VMs to harvest unused resources [2, 38, 49, 50]. However, due to their limitations of capturing real-time changes of workloads and storage states, there is still much space for improvement. Moreover, due to the lack of system support for fine-grained storage harvesting, it is still challenging to maximize the storage utilization. FleetIO tackles these problems with the first RL-based virtualized SSD framework.

Machine Learning for Systems. Most recently, researchers started to leverage learning techniques to improve the task scheduling [37, 49, 56], cluster resource management [2, 7, 10, 29, 53], and performance optimizations [13, 21, 28, 57]. They have demonstrated the capability of ML techniques. Our work shares a similar purpose, but with a focus on the multi-tenant cloud storage. We carefully study our targeted problem and realize that RL is a natural fit for multi-tenant cloud storage management. We develop FleetIO and show its efficiency. We wish FleetIO would inspire future studies on devleoping RL-based computing systems.

7 Conclusion

We present FleetIO, an RL-based storage virtualization framework, which utilizes RL techniques to automate the storage I/O scheduling and harvesting for virtualized SSDs. Our experiments with various cloud applications show that FleetIO can achieve both improved performance isolation and storage utilization for virtualized SSDs, which is impossible with state-of-the-art storage sharing approaches.

Acknowledgments

We thank the anonymous reviewers and our shepherd Huaicheng Li for their insightful comments and feedback. We thank Yiqi Liu from the Systems Platform Research Group at the University of Illinois Urbana-Champaign for discussing and proofreading this paper. We also thank Wanyu Zhao for the discussion at the early stage of this work. This work was partially supported by NSF CAREER Award 2144796 and a gift grant from Google.

References

[1] Albumentations. 2021. Albumentations Image Processing. https://github.com/albumentations-team/albumentations.

[2] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*.

[3] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. 2014. End-to-end Performance Isolation Through Virtual Datacenters. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. Broomfield, CO.

[4] AWS. 2024. Amazon EC2 M7i and M7i-flex instances. https://aws.amazon.com/ec2/instance-types/m7i/.

[5] AWS. 2024. Amazon Elastic Compute Cloud. Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot/.

[6] Azure. 2024. Azure Spot VM. https://azure.microsoft.com/en-us/services/virtual-machines/spot/.

[7] Ricardo Bianchini, Marcus Fontoura, Eli Cortez, Anand Bonde, Alexandre Muzio, Ana-Maria Constantin, Thomas Moscibroda, Gabriel Magalhaes, Girish Bablani, and Mark Russinovich. 2020. Toward ML-Centric Cloud Platforms. *Communication of ACM* 63, 2 (2020).

[8] Stephen J. Bigelow and John Moore. 2021. Software-Enabled Flash for Hyperscale Data Centers. https://searchstorage.techtarget.com/post/Software-Enabled-Flash-for-Hyperscale\-Data-Centers.

[9] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. Santa Clara, CA.

[10] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. Shanghai, China.

[11] GraphChi. 2021. https://github.com/GraphChi/graphchi-cpp.

[12] Hadoop. 2021. Hadoop Terasort Package Documentation. https://hadoop.apache.org/docs/r3.2.0/api/org/apache/hadoop/examples/terasort/package-summary.html.

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

[14] Timothy Harris. 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *Proceedings of the 15th International Symposium on Distributed Computing (DISC'01)*. Lisbon, Portugal.

[15] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17)* (Belgrade, Serbia).

Association for Computing Machinery.

[16] Hong Zhiguo. 2013. Blk-throttle: simplify logic by token bucket algorithm. https://lwn.net/Articles/570333/.

[17] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. 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.

[18] IBM. 2017. IBM Flash Storage and Software Defined Storage. *White Paper* (2017).

[19] Giorgos Kappes and Stergios V. Anastasiadis. 2020. Libservices: Dynamic Storage Provisioning for Multitenant I/O Isolation. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'20)*. Tsukuba, Japan.

[20] Jaeho Kim, Donghee Lee, and Sam H Noh. 2015. Towards SLO complying SSDs through OPS isolation. In *13th USENIX Conference on File and Storage Technologies (FAST'15)*. 183–189.

[21] Daniar H. Kurniawan, Levent Toksoz, Anirudh Badam, Tim Emami, Sandeep Madireddy, Robert B. Ross, Henry Hoffmann, and Haryadi S. Gunawi. 2021. IONet: Towards an Open Machine Learning Training Ground for I/O Performance Prediction. *Technical Report* (2021).

[22] Dongup Kwon, Junehyuk Boo, Dongryeong Kim, and Jangwoo Kim. 2020. FVM: FPGA-assisted Virtual Device Emulation for Fast, Scalable, and Flexible Storage Virtualization. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*.

[23] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. 2017. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference* (Haifa, Israel) *(SYSTOR '17)*. Association for Computing Machinery, New York, NY, USA, Article 13, 11 pages.

[24] Ning Li, Hong Jiang, Dan Feng, and Zhan Shi. 2016. Pslo: Enforcing the xth percentile latency and throughput slos for consolidated vm storage. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys'16)*. 1–14.

[25] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. 2018. RLlib: Abstractions for Distributed Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning (ICML'18)*. Stockholm, Sweden.

[26] Renping Liu, Xianzhang Chen, Yujuan Tan, Runyu Zhang, Liang Liang, and Duo Liu. 2020. SSDKeeper: Self-Adapting Channel Allocation to Improve the Performance of SSD Devices. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS'20)*.

[27] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. 450–462.

[28] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-Based Memory Allocation for C++ Server Workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. Lausanne, Switzerland.

[29] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM'19)*. Beijing, China.

[30] Microsoft. 2018. Project Denali to Define Flexible SSDs for Cloud-Scale Applications. https://www.opencompute.org/files/2018-03-OCP-Denali.pdf.

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

[32] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging AI applications. In *13th USENIX symposium on operating systems design and implementation (OSDI'18)*. 561–577.

[33] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. 2009. Migrating server storage to SSDs: analysis of tradeoffs. In *Proceedings of the 4th ACM European conference on Computer systems (EuroSys'09)*. 145–158.

[34] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 471–484.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[36] Stephen Pritchard. 2020. Cloud flash storage: SSD options from AWS, Azure, and GCP. https://www.computerweekly.com/feature/Cloud-flash-storage-SSD-options-from\-AWS-Azure-and-GCP.

[37] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*.

[38] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi Noghabi, and Jian Huang. 2022. BlockFlex: Enabling Storage Harvesting with Software-Defined Flash in Modern Cloud Platforms. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. Carlsbad, CA.

[39] 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 *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP'23)*. Koblenz, Germany.

[40] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[41] David Shue and Michael J Freedman. 2014. From application requests to Virtual IOPs: Provisioned key-value storage with Libra. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.

[42] David Shue, Michael J Freedman, and Anees Shaikh. 2012. Performance isolation and fairness for Multi-Tenant cloud storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI'12)*. 349–362.

[43] Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, et al. 2015. Schema-agnostic indexing with Azure DocumentDB. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1668–1679.

[44] Software-Defined Data Center. 2023. https://en.wikipedia.org/wiki/Software-defined_data_center.

[45] Software-Defined Storage. 2024. https://en.wikipedia.org/wiki/Software-defined_storage.

[46] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. 2013. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 182–196.

[47] Token Bucket Algorithm. 2024. https://en.wikipedia.org/wiki/token_bucket.

[48] Carl A Waldspurger and William E Weihl. 1995. Stride scheduling: deterministic proportional-share resource management. (1995).

[49] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems (EuroSys'21)*.

[50] Yawen Wang, Daniel Crankshaw, Neeraja J Yadwadkar, Daniel Berger, Christos Kozyrakis, and Ricardo Bianchini. 2022. SOL: Safe on-node learning in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 622–634.

[51] Yahoo! Cloud Serving Benchmark. 2021. https://github.com/brianfrankcooper/YCSB/wiki.

[52] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T Kaushik, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2015. Split-level I/O scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 474–489.

[53] Di Zhang, Dong Dai, Youbiao He, Forrest Sheng Bao, and Bing Xie. 2020. RLScheduler: An Automated HPC Batch Job Scheduler Using Reinforcement Learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20)*. Virtual Event.

[54] Ning Zhang, Junichi Tatemura, Jignesh Patel, and Hakan Hacigumus. 2014. Re-evaluating designs for multi-tenant OLTP workloads on SSD-basedI/O subsystems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. 1383–1394.

[55] Yunqi Zhang, George Prekas, Giovanni Matteo Fumarola, Marcus Fontoura, Inigo Goiri, and Ricardo Bianchini. 2016. History-Based Harvesting of Spare Cycles and Storage in Large-Scale Datacenters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA.

[56] Zhiheng Zhong, Minxian Xu, Maria Alejandra Rodriguez, Chengzhong Xu, and Rajkumar Buyya. 2021. Machine Learning-based Orchestration of Containers: A Taxonomy and Future Directions. *Computing Research Repository (CoRR'21)* abs/2106.12739 (2021).

[57] Giulio Zhou and Martin Maas. 2021. Learning on Distributed Traces for Data Center Storage Systems. In *Proceedings of the Machine Learning and Systems (MLSys'21)*. Austin, TX.