Towards User-Defined SLA in Cloud Flash Storage

Jinhao Fan, Ziyue Yang, Ran Shu, Peng Cheng, Yongqiang Xiong
{v-jinhaofan,ziyyang,rashu,pengc,yqx}@microsoft.com
Microsoft Research
China

ABSTRACT

NAND flash SSDs have gained increasing popularity in cloud storage services. However, there is a gap between what users need and what cloud SSDs provide. For users, storage applications often request asymmetric read and write bandwidth, with tail read latency guarantee. For cloud providers, typical cloud SSD offerings either provide read-write aggregate throughput guarantee or only specifies peak pure-read and write throughput. It is also hard for cloud NAND flash SSDs to provide tail latency guarantees because of their notorious read-write interference problem. As a result, users have to over-provision SSD resource to satisfy their service level agreement (SLA), leading to potential under-utilization.

We propose *Regulator* which enables users to define their SLA for NAND flash based cloud storage. With the user-defined SLA, users can get desired performance and cloud providers can improve their resource efficiency. Regulator first proposes a formalization of user-defined SLA as *SLA Curve*, which contains fine-grained throughput and latency requirements. Regulator then proposes an SLA-aware data placement algorithm to efficiently co-locate users according to their SLA. Finally, Regulator provides a runtime QoS module to enforce users' SLA guarantees. Evaluation shows that Regulator can increase cloud flash utilization by 15%~44% while satisfying user-defined SLAs.

CCS CONCEPTS

• **Information systems → Cloud based storage**.

KEYWORDS

NAND flash, service level agreement, cloud storage

APSys '21, August 24–25, 2021, Hong Kong, China

ACM ISBN 978-1-4503-8698-2/21/08…$15.00
https://doi.org/10.1145/3476886.3477509

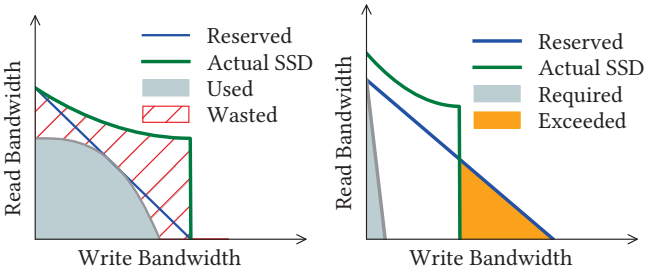
ACM Reference Format:
Jinhao Fan, Ziyue Yang, Ran Shu, Peng Cheng, Yongqiang Xiong. 2021. Towards User-Defined SLA in Cloud Flash Storage. In *12th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '21), August 24–25, 2021, Hong Kong, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3476886.3477509

1 INTRODUCTION

Recently, NAND flash SSD is receiving increasingly wider adoption in cloud data centers [16, 18, 20, 21], because they provide much higher throughput and much lower access latency compared with hard disks [6]. Specifically, NAND flash provides much better random access performance compared with hard disks, making it more suitable for serving multiple tenants at the same time. As a result, many cloud services including Amazon AWS [2] and Microsoft Azure [3] are providing SSD-based cloud storage services, e.g., EBS provisioned IOPS volumes, AWS I3 storage optimized instances, etc.

However, what cloud users need often does not align with what cloud SSDs could provide. For bandwidth, users require various kinds of performance guarantees. Different applications require quite different read/write bandwidth. However, cloud SSDs cannot provide such flexibility. For throughput, typically they only offer overall throughput guarantee that does not distinguish reads and writes [1, 4]. For latency, users expect their storage devices to have tail latency guarantee, e.g., less than 500us latency for 95% accesses. However, NAND flash suffers read-write interference, i.e. there is a severe regression on throughput and latency of read under mixed read/write traffic[22].

Figure 1 demonstrates this dilemma with two typical cloud storage workloads. In Figure 1a, the tenant runs a read-write mixed application (*Used*), e.g., the YCSB workload in Figure 3. It has to provision for the maximum aggregate throughput under all read-write-ratios (*Reserved*). Therefore, the reservation contains some situation that its application never reaches (*Wasted*). The reservation also fails to make full use of the available bandwidth of the physical SSD, whose maximum service capability is described by the green line (*Actual SSD*). For the read-intensive application in Figure 1b (e.g., index serving [20], photo tagging [30]), the SSD is not able to serve the tenant even though it only actually uses a small portion of its bandwidth capability. To make things even worse, if both types of applications require non-trivial



(a) Read-write mixed application wastes significant bandwidth.

(b) Read-intensive application gets rejected because its overall throughput reservation exceeded SSD's limits.

Figure 1: Example cloud storage applications and their bandwidth reservations under traditional cloud storage SLA.

read latency guarantee, their actual SSD throughput area would shrink dramatically, because only little write would be allowed under read-write interference. This makes SSD utilization even lower.

Our insight is that such flash under-utilization comes from the cloud operators' unawareness of user requirements. As resource disaggregation breaks server boundaries[16, 27], cloud operators are able to do more fine-grained resource allocation according to user requirements while achieving better utilization.

We propose *Regulator*, a system that enables users to define SLAs for their cloud-based flash SSDs. Regulator first formalizes user's SLA requirement as *SLA Curves*, This formalization is informative enough to express both read-write asymmetry and read latency sensitivity in applications and SSDs. To increase flash utilization, Regulator proposes a novel resource allocation algorithm that leverages SLA Curve information to efficiently place different kinds of tenants onto physical SSDs. Finally, Regulator provides a QoS module that schedules I/O requests sent to SSDs to enforce SLA.

We prototype Regulator using SPDK, a library for high performance storage processing. Simulations and experiments on synthetic workloads show that Regulator can increase flash utilization by 15%~44% while satisfying tenant SLAs.

Our work makes the following contributions:

- We propose a novel user-defined SLA for virtual SSDs. This SLA is informative enough to make cloud providers be aware of both read-write asymmetry and read latency sensitivity in applications.
- We carefully design and implement a system to increase flash utilization while satisfying the SLA formalized, consisting of an algorithm to partition and allocate SSD resource and virtual SSDs without read-write interference.

The rest of this paper is organized as the following. Section 2 introduces background information and related work. Section 4 and 5 presents the design and implementation of Regulator. Section 6 shows the evaluation results of Regulator, and Section 7 discusses the limitation of Regulator. Section 8 concludes the paper.

2 BACKGROUND AND RELATED WORK

2.1 User-Defined Storage

The current SSD-based cloud storage offerings does not align well with user needs. they still use an SLA that dates back to HDD-era which does not distinguish read and write accesses. Moreover, it only has vague latency guarantees. For example, AWS's most advanced io2 Block Express volumes claim to have an IOPS:GiB ratio of 1,000:1 with sub-millisecond average latency. As demonstrated in Figure 1, this forces users to reserve extra bandwidth that they will never use. The weak latency guarantee also pushes clients to turn to directly attached SSD instances, because modern OLTP services usually require strict tail latency guarantee [8]. However, even those directly attached SSD instances (e.g., AWS I3) only specifies pure-read and write throughput and has no information about what happens in between.

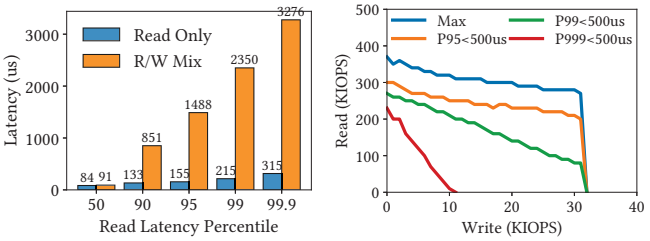
There have been research on user-defined hardware. User-Defined Cloud (UDC) [35], which is quite similar to our user-defined SLA methodology. Instead of letting cloud providers offer fixed hardware options, UDC proposes that users can customize virtual hardware according to their own requirement, which would increase overall resource utilization.

UDC assumes that it is in a hardware disaggregated system, where users have access to a seemingly infinite resource pool. There have been several SSD disaggregation systems [14, 17, 19, 23, 34]. However, they do not solve read-write asymmetry and read-write interference in NAND flash.

2.2 Tail Latency Problem with NAND Flash

Modern cloud applications require strict tail latency guarantees [8]. However, NAND flash based SSDs are notorious for their bad tail latency because of the *read-write interference problem*[6, 17, 25, 28]. When they serve mixed read and write workload, both bandwidth and read tail latency downgrade severely.

We measure how such tail latency problem exists on Samsung PM963 [26], which is a widely adopted NAND flash SSD in data center. The results are shown in Figure 2. When mixed with 4KB random write, 4KB random read tail latency increases up to 10x with write ratio only around 5%(Figure 2a). If we set an upper bound for its 95 (*P95*), 99 (*P99*) and 99.9 (*P999*) percentile latency (Figure 2b), the read bandwidth the SSD can provide also drops significantly as write throughput increases. We choose to focus on read tail latency here because in read-write interference, only read is influenced by write but not vice versa [25]. Both throughput



(a) Read Tail Latency with 200K IOPS Read and 10K IOPS Write

(b) Max R-W Throughput Under Different Read Tail Latency Requirements

Figure 2: Tail Latency Problem On Samsung PM963

and latency of write stays quite tuned no matter how read I/O pressure changes.

Such downgrade is believed to be caused by SSD background operations incurred by writes, including write buffer flush and garbage collection [6]. These operations block the execution of other requests on the same SSD, resulting in irregular latency of reads. Although applications can batch writes to minimize write interference on reads, it is not feasible to count on other co-located tenants to stop isssuing write requests when one tenant is doing reads. However, as shown in Figure 2b, by controlling read and write throughput, we can make SSD internal activities happen less frequently, thus improving tail latency, which is a statistical result.

There has been some research trying to resolve this downgrade. SmartIO [22] and FIOS [25] proposes to use an I/O scheduler suspend writes until there are no more ongoing reads. Such priority scheduling improves read latency, but they are not accurate enough for SLA enforcement. ReFlex [17] achieves both throughput and latency SLA according to an SSD performance model. It does so by assigning a weight to writes and using weighted IOPS for bandwidth allocation. However, its learning-based model is not accurate enough for all SSDs. Meanwhile, its bandwidth allocation scheme assumes users have a fixed read-write ratio, which is not true for all applications(§ 3). Flash on Rails [28] and Tiny-Tail Flash [33] try to solve the tail latency problem at the SSD level. They split SSD working time into different time slices, send only reads or writes in a single time slices and coordinate different SSDs to build a virtual SSD without read-write interference. SWAN [15] splits an all-flash array into front-end SSDs that performs log-structured writes and back-end SSDs that does GC in order to mitigate GC interference. Recently, LinnOS [12] uses a lightweight neural network to predict SSD's performance instability.

3 SLA CURVE AND ITS ARITHMETIC

In this paper, we propose **SLA Curve** to represent tenants' bandwidth requirements as well as storage devices' service capability. Specifically, the SLA Curve describes the maximum read throughput for each write throughput while achieving a certain read tail latency guarantee. It does not include write latency because writes are usually not on the critical path and write latency is also not easily impacted by multi-tenant interference(§ 2). Specifically, we define *SLA Curve* as a function of write throughput. As explained in § 2, by ensuring read and write throughput under the SLA Curve, we can keep the frequency of SSD's internal activities low enough, therefore guaranteeing tail latency. For example, for SLA Curve A, $A[w] = r$ means when the user's write throughput is w, if the user wants to guarantee its required tail latency, it can only send read requests with a rate up to r. This definition not only distinguishes read and write requirements, but also defines throughput requirements at *all* read-write ratios.

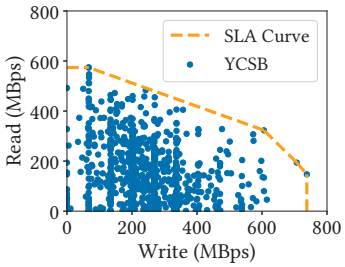


Figure 3: Read and Write throughput when running YCSB Workload A on RocksDB. An example SLA curve describes its bandwidth requirement.

The definition of SLA Curve naturally fits users' application needs. In Figure 3, we analyze the SNIA YCSB RocksDB SSD trace [32] collected by running YCSB Workload A [7]. The blue dots are throughput sampled at one-second interval during a twelve-minute period. As shown in the figure, the workload has asymmetric bandwidth requirements for reads and writes. It does not have an easily observable read-write ratio. We derive the SLA curve with a convex hull algorithm on the sampled points [5]. In practice, SLA Curves can be obtained by such sampling based approaches.

SLA Curves are also applicable for storage devices. Actually, the PM963 curve in Figure 2b can be viewed as the SLA Curve for PM963 under different tail latency requirements. We use 4KB random accesses to measure device curves because it is usually the worst case for all access patterns. As long as user requirements can be met for this worst case, it can be satisfied under other cases.

SLA Curve is informative enough to solve the problems in Figure 1. For the read-intensive workload, the SSD will accept the user after making sure the it can indeed fulfill the user's SLA Curve. For the read-write mixed workload, the maintainer can also introduce other tenants to share

the remaining space under the SSD's SLA Curve, or simply assign more bandwidth to this tenant.

Based on the definition of SLA Curves, we define the following *SLA Arithmetic*. These arithmetic operations form the basis of our SLA-aware SSD resource allocation design (§ 4.1).

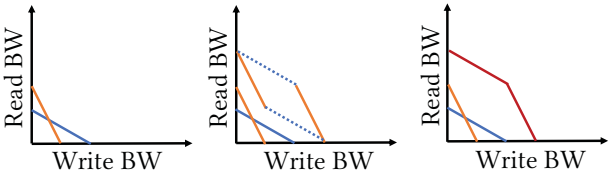


Figure 4: Sum of SLA curves

When multiple tenants are sharing the same storage device, we need to know their aggregate bandwidth requirements. Therefore, The first arithmetic operation defines the *addition* between SLA Curves. At any time, their aggregate bandwidth usage will be sum of their individual bandwidth. Because the SLA Curve represent the *lower bound* of their resource requirement, the summation $S[w]$ is defined as the *maximum* sum of read bandwidth of all points on the two curves whose write bandwidth sum as w, which is given in Definition 1 and demonstrated in Figure 4.

Definition 1. *The **sum** S of SLA Curves A and B is defined as:*

$$S[w] = \max_{0 \le i \le w} (A[i] + B[w - i]) \tag{1}$$

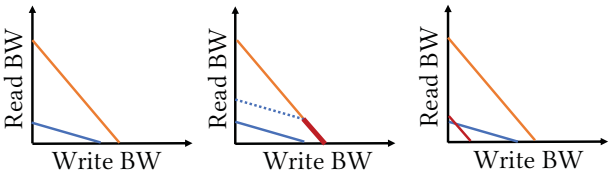


Figure 5: Subtraction between SLA curves

When one tenant is assigned to a storage device, We need to calculate the remaining service capability of the device. Therefore, the second operation is the *subtraction* between SLA Curves, which is only valid when one SLA Curve is contained in another. According to the definition of SLA Curves, the residual curve represents the *upper bound* of the SSD's remaining bandwidth. Therefore, the difference $D[w]$ is defined as the *minimum* difference between the read bandwidth of points on the two curves whose write bandwidth difference is w, which is given in Definition 2 and Figure 5.

Definition 2. *The **difference** D between SLA Curves A and B is defined as:*

$$D[w] = \min_{w \le i \le w_{max}(A)} (A[i] - B[i - w]) \tag{2}$$

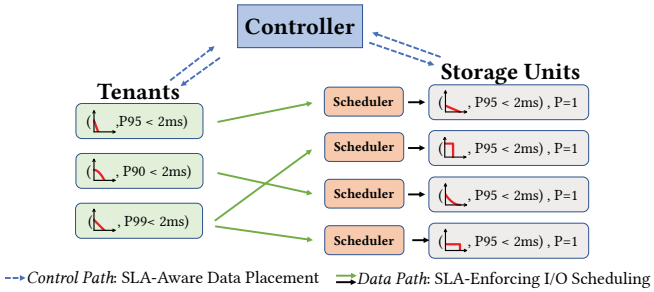


Figure 6: Regulator system overview

, where $w_{max}(A)$ means the maximum write bandwidth of A.

4 SLA CURVE BASED SYSTEM DESIGN

Based on *SLA Curves*, we design Regulator, a flash storage system that allows storage users to define their SLA by SLA Curves.

Figure 6 is an overview of Regulator's architecture. There are mainly four parts in our system: the tenants, the Regulator controller, the I/O scheduler and storage units. Both tenants and storage units describe their SLA with an *<SLA Curve, Read Tail latency requirement>* tuple. The difference is that each type of storage unit has a field P for its cost, which can be the price of price of the device, its power consumption, etc. On the control path, the Regulator controller is responsible for allocating tenants' storage spaces using a heuristic SLA-aware data placement algorithm (§ 4.1). On the data path, the Regulator I/O scheduler performs bandwidth allocation at runtime to ensure the tenants get their desired bandwidth and tail latency guarantee as defined by their SLA (§ 4.2).

4.1 SLA-Aware Data Placement

In a cloud flash storage system, multiple tenants typically share multiple physical storage units. When a new tenant is registered, the Regulator controller has to determine on which storage unit to place the tenant's data. This is particularly important for our SLA Curve based resource allocation scheme because failing to do efficient tenant co-location will severely downgrade the overall resource utilization. For example, in Figure 5, co-locating the red and blue curves will leave some of the SSD's possible bandwidth capability never used.

The multi-tenant resource placement problem is often modeled as a bin packing problem [31]. The traditional bin packing problem is described as packing multiple items of different sizes into fixed-sized bins to minimize the total number of used bins. Its optimal solution is known to be NP-hard, but there are a few effective approximate algorithms. One of the most widely used online bin packing algorithm

is *best-fit*. The algorithm executes in two phases when a new item comes in. First, it greedily selects the bin with the minimum capacity remaining that can fit the item. Then, if no available bin can fit the item, it starts a new bin.

Compared with the traditional bin packing problem, there are some differences that make our SLA Curve based storage allocation challenging. For the first phase, the "size" and "capacity" of tenants and storage units are described using SLA Curves instead of a scalar value. This makes finding the "best" remaining bin hard because there is no clear definition of what curve is "minimum". For the second phase, the storage units in cloud are highly heterogeneous with diverse SLA Curves and costs. This means we need to consider which type of storage unit to select when we need a new one.

Here we describe how we adapt the best fit algorithm to Regulator's SLA-aware data placement algorithm. In the first phase, we use metrics inspired by vector bin packing [11, 24] to determine the fitness between tenant and device SLA Curves. Specifically, we consider not only whether the device's curve fits tenant's demand, but also how well the two curves *align* with each other. The intuition is that if a tenant is assigned to a device with poor SLA Curve alignment, it is likely to cause more bandwidth wastes, which is already demonstrated in Figure 5. Three example metrics can be used for SLA Curve alignment: 1) Dot product which emphasizes the shape similarity between SLA Curves, 2) L2-norm which represents the distance between SLA Curves, 3) Ratio between areas of two SLA Curves. They perform similar in our preliminary experiments.

Next, if we need to start a new storage unit in the second phase, the Regulator data placement algorithm greedily chooses the storage unit that minimizes the cost for the current tenant. Specifically, it calculates the cost needed to fit the tenant's SLA requirement for each type of storage unit. If a single storage unit can not meet the need of the tenant, the algorithm strips the data across multiple storage units to avoid hotspots.

Some tenants may require tail latency guarantees so strict that it is hard to provide with plain NAND SSDs. To satisfy such tail requirements, we leverage [28] to construct redundant SSD arrays without read-write interference.

4.2 SLA-Enforcing I/O Scheduling

Although Regulator's data placement algorithm makes sure tenants' SLA can be effectively satisfied by the storage device, the SLA needs to be actually enforced at runtime. Regulator's I/O scheduler does so by controlling the read and write throughput of each tenant sharing a same storage unit. With the Regulator I/O scheduler, tenants can get their desired bandwidth specified by the SLA Curve. Also, the overall bandwidth usage is always controlled within the device's SLA Curve so the tail latency is guaranteed.

Algorithm 1: Regulator I/O Scheduling Algorithm

```
/* Schedule according to tenants' curve  */
disk.readBytes = disk.writeBytes = 0;
foreach tenant t do
    t.readBytes = t.writeBytes = 0;
    while t.readBytes <= t.curve[t.writeBytes] do
        if t.queue.empty() then break;
        else req = t.queue.pop();
        t.submitRequest(req);
        if req.isRead then t.readBytes += req.size;
        else t.writeBytes += req.size;
    end
    disk.readBytes += t.readBytes;
    disk.writeBytes += t.writeBytes;
end
/* Weighted-round-robin for spare disk BW */
while True do
    foreach tenant t do
        bytes = 0 ;
        while bytes < t.weight do
            if disk.readBytes >
              disk.curve[disk.writeBytes] then return;
            if t.queue.empty() then break;
            else req = t.queue.pop();
            t.submitRequest(req);
            if req.isRead then disk.readBytes +=
              req.size;
            else disk.writeBytes += req.size;
            bytes += req.size;
        end
    end
end
```

To control the bandwidth of both reads and writes, the Regulator I/O scheduler performs *curve-based rate limiting*. In each timeslice, it tracks the read and write bandwidth usage of each tenant. Upon the arrival of a tenant I/O request, it checks whether the current bandwidth usage has exceeded its SLA Curve. If not, the scheduler submits the request to the storage unit. Otherwise, it keeps the request in a waiting queue until a new timeslice begins.

We are also aware that it is hard to select an accurate SLA Curve. In fact, we do not recommend choosing an SLA Curve that covers all the bandwidth outliers because that will likely cause a waste of bandwidth, as shown in Figure 3. Therefore, the Regulator scheduler can optionally allow tenants to temporarily burst above their SLA Curves. The bursty bandwidth is considered best-effort and only available when the aggregate bandwidth does not exceed device's SLA Curve. To

avoid contention with non-bursty traffic, the bursty accesses are only scheduled when no non-bursty request is waiting to be dispatched. When multiple tenants burst above their curve, the Regulator scheduler uses weighted round robin to share the extra bandwidth. Requests larger than 64KB are split into ones less than 64KB to enforce fairness.

The I/O scheduling is done on the storage units, because they have full knowledge about all tenants that are accessing them. Moreover, only putting the scheduler at the storage server can make sure the storage device always runs under its desired SLA curve, because network-induced variable delay can make the scheduling results indeterministic.

5 IMPLEMENTATION

We implemented a prototype system for Regulator mainly based on Storage Performance Development Kit (SPDK) [34]. The preliminary prototype is implemented on a local storage pool. In the future, we plan to extend Regulator on a disaggregated storage cluster.

We implemented the Regulator controller in C++. It is responsible for assigning SSD resources to tenants, as well as managing storage units. When a new tenant is registered, the controller runs the data placement algorithm (§ 4.1) to determine a mapping between the tenant's logical storage space and SSD's storage space. Then, it sends the mapping to the new tenant. If the algorithm determines some new storage unit is needed, the controller informs the storage server through RPC.

We also provide a userspace Regulator library at the client side. The library is implemented as a *virtual block device* of SPDK. It stores the address mapping obtained on registration. When the tenant submits an I/O request, the Regulator library code translates the address from tenant's address space to the storage unit's address space. Then, it sends the request to the corresponding storage units on behalf of the tenant.

The Regulator I/O scheduler is implemented as an extension of SPDK's bdev layer at the storage server. The scheduler runs the SLA-enforcing algorithm (§ 4.2) at the storage server. We implemented the I/O scheduler with multi-core scalability in mind. Requests from all user threads are first sent to a dedicated scheduling thread through a lockless FIFO queue. All the scheduling work, including bookkeeping of tenant states, are then handled by the scheduling thread alone. This design minimizes the synchronization overhead between threads. The scheduling thread performs polling with high processing throughput. As SPDK can achieve 10MIOPS on a single thread [29], the scheduling thread will not be a performance bottleneck.

6 EVALUATION

In this section, we first run a simulation with hundreds of tenants to show the resource efficiency of Regulator's SLA-aware data placement algorithm. We then evaluate Regulator's I/O scheduler to show its ability of guaranteeing SLA. The evaluation is done on a server machine with 32 Intel Xeon CPU E5-2698 processor cores across two sockets running at 2.3GHZ. The server is equipped with 192GB DRAM. It runs Ubuntu LTS 18.04 with Linux kernel 4.17.12. We use Samsung PM963 SSDs in our experiment.

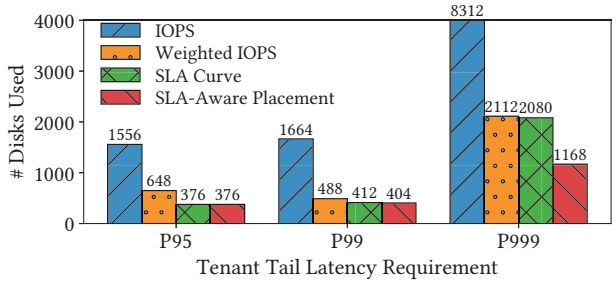


Figure 7: Resource Savings for SLA-Based SSD Allocation

6.1 Resource Utilization

In the first experiment, we run a simulation of multiple tenants with different SLA requirements registering on our Regulator system. We randomly generated 3 tenant groups, with P95, P99, P999 read tail latency requirements respectively. Each group has 300 tenants, with 100 read-intensive ones, 100 write-intensive ones, and 100 read-write mixed ones. We only used Samsung PM963 for the P95, P99 tests. As it is hard for PM963 to provide P999 guarantees, we used the method proposed in [28] to construct redundant SSD arrays with strict tail latency guarantee.

We compare Regulator's SLA-aware data placement with three resource allocation methods. "IOPS" uses the traditional SLA offering scheme in cloud that uses aggregate throughput of both read and writes, similar to Figure 1. "Weighted IOPS" assigns a weight to write requests, which is similar to ReFlex's method [17]. The write weight is set to 3, 6 and 20 for P95, P99 and P999 cases respectively. "SLA Curve" only uses SLA Curve to represent resource requirements, but does not use our bin-packing based data placement algorithm. The above three methods use a brute-force data placement algorithm that placement tenant randomly into available storage units. Because the exact solution is NP-hard, we do not compare with the ideal case.

To compare the resource utilization, we count the number of used disks for each method. As shown in Figure 7, the IOPS approach is too inefficient for SSDs with asymmetric read-write bandwidth, resulting in around 4x wastes compared with SLA Curve based approaches. In the P95 and P99 cases, simply utilizing our fine-grained SLA Curve to represent bandwidth requirements can improve utilization by 41% and 15% respectively. The advantage looks smaller for the

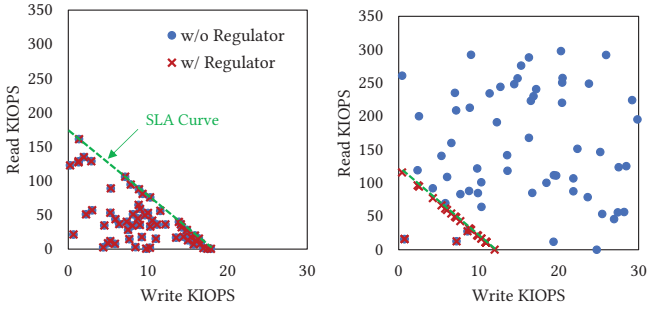
P999 case, because the device's SLA Curve is very close to linear(Figure 2b, which is suitable for using weighted IOPS. However, the benefit of SLA-aware data placement becomes crucial here because more storage unit types are introduced and resource selection and allocation becomes more important. It improves the overall utilization by 44% for the P999 case.

6.2 Performance SLA

The second experiment evaluates the Regulator I/O scheduler using two synthetic tenants. Tenant A runs a latency-sensitive application (e.g., online key-value store). Tenant A follows a linear SLA curve with max 180 KIOPS read and max 18 KIOPS write. Tenant B is a "noisy neighbor" that shares the same storage device with tenant A. It issues read and write randomly. When limited by Regulator, it has a linear SLA curve with max 120 KIOPS read and 12 KIOPS write. Together, these two tenants forms a linear SLA curve with max 300 KIOPS read and 30 KIOPS write, which provides 500us read 95th latency guarantee.

We run the two tenants on Samsung PM963. As shown in Figure 8, tenant A's bandwidth share is impacted by B's bursty accesses. We randomly chose a sixty-second period to measure tenant A's tail latency. As shown in Figure 9, tenant A violates its tail latency requirements because of interference from tenant B.

Therefore, we use Regulator's I/O scheduler to limit tenant B's bandwidth usage according to its SLA Curves. We set the tenant B's SLA Curve slightly larger than its normal throughput to throttle the bursty accesses. We turned off the scheduler's bursty feature to demonstrate its ability for controlling bandwidth. As demonstrated in Figure 8, both tenants are kept running under their SLA Curve. Tenant A's read tail latency is also precisely controlled within its latency requirement because the interference from tenant B is reduced. We also run the experiment on Intel DC P3700 SSD and get similar results, which are not presented here due to space limitation.



(a) Latency-Sensitive Tenant A **(b) "Noisy-Neighbor" Tenant B**

Figure 8: SLA Curve Enforcement

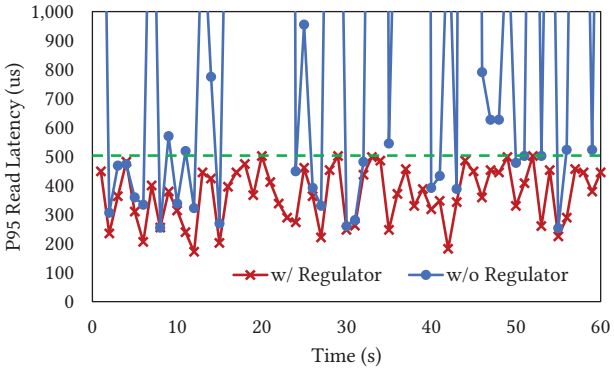


Figure 9: Tenant A's P95 Read Latency

7 DISCUSSION

Obtaining SLA Curves. In § 3, we use a simple convex hull algorithm on sampled throughput points to get the SLA Curve. In the future, more advanced methods can be derived to accurate predict an application's SLA Curve. We envision a future where the cloud providers can help users understand their workloads. For example, through profiling and collaborative filtering [9], the cloud provider can recommend a feasible SLA Curve to the user.

Hardware acceleration. Currently, Regulator uses software SPDK for I/O scheduling and NVMe accesses. This causes some host CPU and memory overhead. In the future, we plan to offload these functionality to SmartNICs for hardware acceleration. There are already works that offload QoS modules to accelerator hardware. For example, Microsoft Azure Networking uses SmartNIC for flow tracking and rate limiting [10].

Generalize to other types of storage. We mainly focus on NAND flash SSDs because is asymmetric read and write bandwidth and high tail latency causes a lot of headache for application developers. In fact, the concept of SLA Curves and user-defined storage can generalize to other devices in the storage hierarchy like HDD, DRAM and new types of NVM like Intel Optane SSDs [13].

8 CONCLUDING REMARKS

This paper proposes Regulator, a system enabling users to define SLA for NAND flash based virtual SSDs in cloud storage. Regulator formalizes virtual SSD SLAs as read-write curves with tail latency upper bounds. Regulator provides a novel resource allocation algorithm to place different kinds of tenants onto physical SSDs according to SLA Curves. We implement Regulator in SPDK, and evaluation shows Regulator's ability to increase flash utilization while keeping tenants' SLAs.

REFERENCES

[1] Amazon. 2021. Amazon EBS features. Retrived from https://aws.amazon.com/ebs/features/. Accessed: 2021-05-17.

[2] Amazon. 2021. Amazon elastic block store. Retrieved from https://aws.amazon.com/ebs/. Accessed: 2021-05-17.

[3] Microsoft Azure. 2021. Azure disk storage – cloud managed disks. Retrieved from https://azure.microsoft.com/en-us/services/storage/disks/. Accessed: 2021-05-17.

[4] Microsoft Azure. 2021. Pricing - Managed Disks. Retrieved from https://azure.microsoft.com/en-us/pricing/details/managed-disks/. Accessed: 2021-05-17.

[5] C Bradford Barber, David P Dobkin, and Hannu Huhdanpaa. 1996. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)* 22, 4 (1996), 469–483.

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

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

[8] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[9] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices* 49, 4 (2014), 127–144.

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

[11] Michaël Gabay and Sofia Zaourar. 2016. Vector bin packing with heterogeneous bins: application to the machine reassignment problem. *Annals of Operations Research* 242, 1 (2016), 161–194.

[12] 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)*. 173–190.

[13] Intel. 2021. Intel® Optane™ SSD 9 Series. Retrieved from https://www.intel.com/content/www/us/en/products/details/memory-storage/consumer-ssds/optane-ssd-9-series.html.

[14] Intel. 2021. *SPDK NVMe over Fabrics Target.* http://www.spdk.io/doc/nvmf.html

[15] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H Noh. 2019. Alleviating garbage collection interference through spatial separation in all flash arrays. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 799–812.

[16] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash storage disaggregation. In *Proceedingsof the Eleventh European Conference on Computer Systems (EuroSys)*. 1–15.

[17] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash= Local Flash. In *Proceedingsof the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 345–359.

[18] Ilias Marinos, Robert N.M. Watson, Mark Handley, and Randall R. Stewart. 2017. Disk|Crypt|Net: rethinking the stack for high-performance video streaming. In *Proceedingsof the 2017 ACM conference on SIGCOMM (SIGCOMM)*. ACM, 211–224.

[19] Mellanox. 2017. How To Configure NVMe over Fabrics (NVMe-oF) Target Offload. Retrieved from https://community.mellanox.com/docs/DOC-2918.

[20] Microsoft. 2021. GitHub - microsoft/SPTAG: A distributed approximate nearest neighborhood search (ANN) library. Retrieved from https://github.com/Microsoft/SPTAG.

[21] Netflix. 2021. Evolution of Application Data Caching : From RAM to SSD. Retrieved from https://netflixtechblog.com/evolution-of-application-data-caching-from\-ram-to-ssd-a33d6fa7a690.

[22] David T Nguyen, Gang Zhou, Guoliang Xing, Xin Qi, Zijiang Hao, Ge Peng, and Qing Yang. 2015. Reducing smartphone application delay through read/write isolation. In *Proceedingsof the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 287–300.

[23] Inc NVM Express. 2016. NVM express over fabrics, revision 1.0. Retrieved from https://nvmexpress.org/wp-content/uploads/NVMe_over_Fabrics_1_0_Gold_20160605.pdf.

[24] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. 2011. Heuristics for vector bin packing. *research. microsoft. com* (2011).

[25] Stan Park and Kai Shen. 2012. FIOS: A Fair, Efficient Flash I/O Scheduler. In *Proceedingsof the 10th USENIX Conference on File and Storage Technologies (FAST)* (San Jose, CA) *(FAST'12)*. USENIX Association, USA, 13–27.

[26] Samsung. 2021. Samsung SSD PM963. Retrieved from https://www.compuram.de/documents/datasheet/Samsung_PM963-1.pdf.

[27] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *Proceedingsof the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 69–87.

[28] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. 2014. Flash on rails: Consistent flash performance through redundancy. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*. 463–474.

[29] SPDK. 2019. 10.39M Storage I/O Per Second From One Thread. Retrieved from https://spdk.io/news/2019/05/06/nvme/.

[30] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. 2015. RIPQ: Advanced Photo Caching on Flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*. 373–386.

[31] Wikipedia. 2020. Bin packing problem — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/Bin_packing_problem

[32] Gala Yadgar, MOSHE Gabel, Shehbaz Jaffer, and Bianca Schroeder. 2021. SSD-Based Workload Characteristics and Their Performance Implications. *ACM Trans. Storage* 17, 1, Article 8 (Jan. 2021), 26 pages. https://doi.org/10.1145/3423137

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

[34] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. 2017. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 154–161.

[35] Yiying Zhang, Ardalan Amiri Sani, and Guoqing Harry Xu. 2021. User-defined cloud. In *Proceedingsof the Workshop on Hot Topics in Operating Systems (HotOS)*. 33–40.