ReFlex: Remote Flash ≈ Local Flash

Ana Klimovic * Heiner Litz* Christos Kozyrakis
Stanford University
{anakli, hlitz, kozyraki}@stanford.edu

Abstract

Remote access to NVMe Flash enables flexible scaling and high utilization of Flash capacity and IOPS within a datacenter. However, existing systems for remote Flash access either introduce significant performance overheads or fail to isolate the multiple remote clients sharing each Flash device. We present ReFlex, a software-based system for remote Flash access, that provides nearly identical performance to accessing local Flash. ReFlex uses a dataplane kernel to closely integrate networking and storage processing to achieve low latency and high throughput at low resource requirements. Specifically, ReFlex can serve up to 850K IOPS per core over TCP/IP networking, while adding 21μs over direct access to local Flash. ReFlex uses a QoS scheduler that can enforce tail latency and throughput service-level objectives (SLOs) for thousands of remote clients. We show that ReFlex allows applications to use remote Flash while maintaining their original performance with local Flash.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management

Keywords Flash; I/O scheduling; network storage; QoS

1. Introduction

NVMe Flash devices deliver up to 1 million I/O operations per second (IOPS) at sub 100μs latencies, making them the preferred storage medium for many data-intensive, online services. However, the Flash devices deployed in datacenters are often underutilized in terms of capacity and throughput due to the imbalanced requirements across applications and over time [37, 50]. In general, it is difficult to design machines with the perfect balance between CPU, memory, and Flash resources for all workloads, which leads to overprovisioning and higher total cost of ownership (TCO). Similar to sharing disks within a datacenter, remote access to Flash over the network can greatly improve utilization by allowing access to Flash on either any machine that has spare capacity and bandwidth or on servers dedicated to serving a large number of NVMe devices.

There are significant challenges in implementing remote access to Flash. Achieving *low latency* requires minimal processing overheads at the network and storage layers in both the server and client machines. In addition to low latency, each server must achieve *high throughput at minimum cost*, saturating one or more NVMe Flash devices with a small number of CPU cores. Moreover, managing interference between multiple tenants sharing a Flash device and the uneven read/write behavior of Flash devices [52, 61] requires *isolation mechanisms* that can guarantee predictable performance for all tenants. Finally, it is useful to have *flexibility* in the degree of sharing, the deployment scale, and the network protocol used for remote connections. Existing, software-only options for remote Flash access, like iSCSI [57] or event-based servers, cannot meet performance expectations. Recently proposed, hardware-accelerated options, like NVMe over RDMA fabrics [18], lack performance isolation and provide limited deployment flexibility.

We present ReFlex, a software-based Flash storage server that implements remote Flash access at a performance comparable with local Flash accesses. ReFlex achieves high performance with limited compute requirements using a novel dataplane kernel that tightly integrates networking and storage. The dataplane design avoids the overhead of interrupts and data copying, optimizes for locality, and strikes a balance between high throughput (IOPS) and low tail latency. ReFlex includes a QoS scheduler that implements priorities and rate limiting in order to enforce service level objectives (SLOs) for latency and throughput for multiple tenants sharing a device. ReFlex provides both a user-level library and a remote block device driver to support client applications.

The ReFlex server achieves 850K IOPS per core over commodity 10GbE networking with TCP/IP. Hence, it can serve several NVMe devices and meet networking line rates at low cost. Its unloaded latency is only 21μs higher than direct access to local Flash through NVMe queues. The ReFlex server can support thousands of remote tenants. Its QoS

* The first two authors contributed equally to this work.

ASPLOS '17, April 08 - 12, 2017, Xi'an, China

ISBN 978-1-4503-4465-4/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3037697.3037732

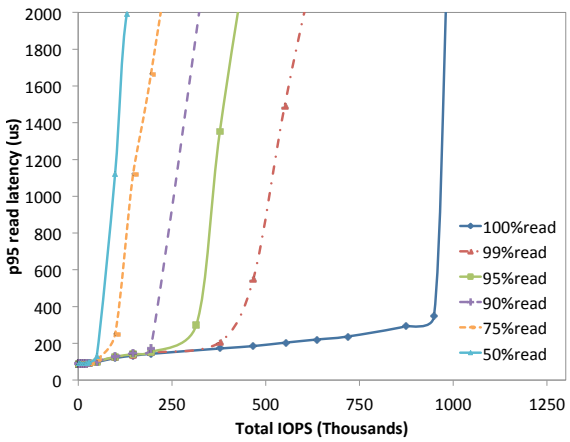


Figure 1: The impact of interference on Flash performance. Tail read latency depends on total IOPS and read/write ratio.

scheduler can enforce the tail latency and throughput requirements of tenants with SLOs, while allowing best-effort tenants to consume all remaining throughput of the NVMe device. Finally, using legacy applications, we show that even with heavy-weight clients, ReFlex allows for performance levels nearly identical to those with local Flash.

ReFlex is open-source software. The code is available at `https://github.com/stanford-mast/reflex`.

2. Background and Motivation

Remote access provides flexibility to use Flash regardless of its physical location in a datacenter, increasing utilization and reducing the total cost of ownership [37, 38].

Remote access to hard disks is already common in datacenters [8, 23], since their high latency and low throughput easily hide network overheads [3, 9]. A variety of software systems can make remote disks available as block devices (e.g., iSCSI [57]), network file systems (e.g., NFS [56]), distributed file systems (e.g., Google File System [23]), or distributed data stores (e.g., Bigtable [17]). There are also proposals for hardware-accelerated, remote access to Flash using Remote Direct Memory Access (RDMA) (e.g., NVMe over Fabrics [18]) or PCIe interconnects [6, 29, 67]. Existing approaches for remote Flash access face two main challenges: achieving high performance at low cost, and providing predictable performance in the presence of interference.

2.1 Performance Goals

Low latency: The unloaded read latency of NVMe Flash is 20-100μs [19]. To satisfy the requirements of latency-sensitive applications, remote access to Flash should have low latency overhead. Conventional remote server access over 10GbE and the TCP/IP network protocol adds at least 50μs to unloaded latency before any software-based storage protocol even begins [11]. More important, network processing in Linux introduces performance unpredictability due to interrupt management and core scheduling, increasing the tail latency of remote accesses [36, 40, 41]. Network storage systems like iSCSI introduce additional latency for protocol processing and copying data between kernel and user buffers [35, 37]. On the other hand, distributed file systems like GFS and HDFS are optimized for multi-megabyte data transfers to remote disks, introducing overhead for kilobyte-sized data accesses to remote Flash [23, 63].

High throughput at low cost: Modern Flash devices have throughput capabilities on the order of a million IOPS [55]. A remote Flash server must serve high I/O rates with low processing overhead to reduce the cost and increase the flexibility of sharing Flash over the network. Datacenter machines with spare Flash capacity and IOPS may not have many cores available to serve remote requests. Existing software-based approaches require significant compute resources to saturate Flash throughput. For example, the iSCSI protocol achieves 70K IOPS per CPU core [37], thus requiring 14 cores to serve the 1M IOPS of a high-end NVMe device. Similarly, in §5, we show that a light-weight server for remote Flash access based on Linux `libevent` and `libaio` achieves only 75K IOPS per core.

2.2 Interference Management

Remote Flash is useful if it provides *predictable performance* even when multiple tenants share a device. Predictable performance is a challenge for NVMe Flash devices because of the impact of read/write interference. Figure 1 plots the tail read latency (95th percentile) on Flash as a function of throughput (IOPS) for workloads with various read/write ratios. Tail read latency depends on throughput (load) and the read/write ratio. This behavior is typical for all NVMe Flash devices we have tested because write operations are slower and trigger activities for wear leveling and garbage collection that cannot always be hidden. Read/write interference can be managed when a single application uses a local Flash device, but becomes a big challenge with remote Flash and multiple tenants that share the same device and are unaware of each other.

Hardware acceleration is not enough: Interference mitigation is a major omission of hardware-accelerated schemes like NVMe over Fabrics [18], QuickSAN [15] or iSCSI extensions for RDMA (iSER) [16]. The current isolation features of NVMe devices, namely hardware queues and namespaces, are not sufficient to reduce interference between multiple remote clients without additional software. The number of queues is limited (e.g., 64 queues in high-end devices) and request arbitration is simplistic (round-robin). Namespaces are host-side logical partitions of the device, so requests issued to different namespaces will still interfere. Existing NVMe over Fabrics solutions do not perform I/O scheduling to mitigate interference between multiple remote clients. The existing solutions do not offer a way for clients to specify quality of service objectives and are thus unable to manage Flash devices in a QoS-aware manner [48].

Hardware-accelerated approaches have other disadvantages. RDMA-based schemes require network fabrics with RDMA capabilities, which may not be readily available in legacy datacenters. Approaches for sharing Flash over a PCIe backplane limit sharing to a single rack and also lack support for performance isolation [6, 29, 67].

3. ReFlex Design

ReFlex provides low latency and high throughput access to remote Flash using a dataplane architecture that tightly integrates the networking and storage layers. It serves remote read/write requests for logical blocks of any size over general networking protocols like TCP and UDP. While predominately a software system, ReFlex leverages hardware virtualization capabilities in NICs and NVMe Flash devices to operate directly on hardware queues and efficiently forward requests and data between NICs and Flash devices without copying. Its polling-based execution model (§3.1) allows requests to be processed without interruptions, improving locality and reducing unpredictability. ReFlex uses a novel I/O scheduler (§3.2) to guarantee latency and throughput SLOs for tenants with varying ratios of read/write requests. ReFlex can serve thousands of tenants and network connections, using as many cores as needed to saturate Flash device IOPS.

3.1 Dataplane Execution Model

Each ReFlex server thread uses a dedicated core with direct and exclusive access to a network queue pair for packet reception/transmission and an NVMe queue pair for Flash command submission/completion.

Figure 2 reviews the execution model for a ReFlex server thread processing an incoming Flash read (or write) request. First, the NIC receives a network packet and delivers it via DMA to a pre-allocated memory buffer provided by the networking stack ①. The ReFlex thread polls the receive descriptor ring and processes the packet through the Ethernet driver and networking stack (e.g., TCP/IP), generating event conditions indicating the availability of a new message ②. The same thread uses `libix` [11], a library similar to Linux `libevent` [54], to process the event. This involves switching to the server code that parses the message, extracts the I/O request, performs access control checks and any other storage protocol processing required before submitting a Flash read (write) system call ③. The thread then switches to system call processing and performs I/O scheduling to enforce SLOs across all tenants sharing the ReFlex server (§3.2). Once scheduled, the request is submitted to the Flash device through an NVMe submission queue ④. The Flash device performs the read (write) I/O and delivers (retrieves) the data via DMA to (from) a pre-allocated user-space buffer ⑦. The thread polls the completion queue ⑤ and delivers a completion event ⑥. The event callback executes through `libix` and emits a send system call ⑦. Finally, the thread processes

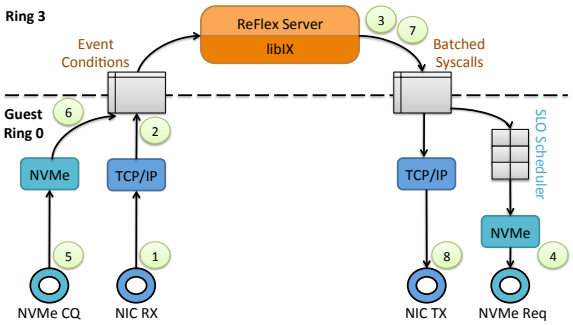
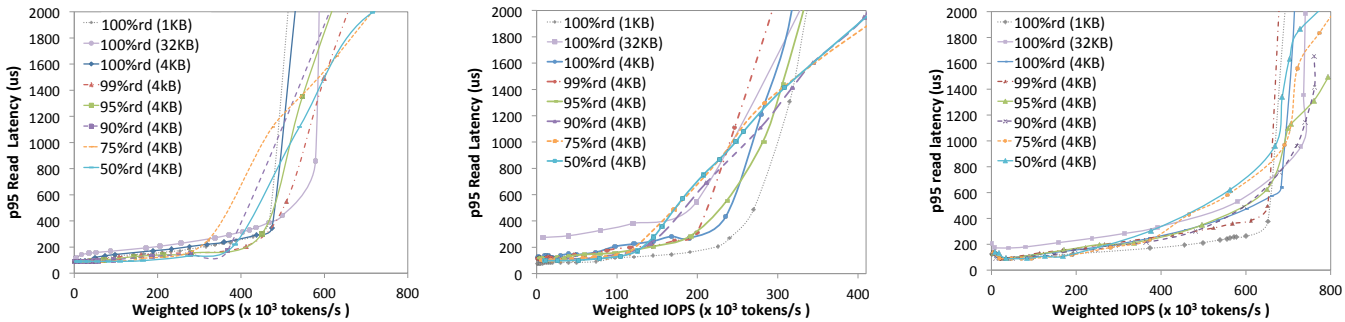


Figure 2: The execution model for a each ReFlex thread.

the send system call to deliver the requested data back to the originator through the network stack ⑧. The execution model supports multiple I/O requests per network message and large I/Os that span across multiple network messages.

This execution model achieves low latency for remote Flash requests. It runs to completion the two steps of request processing, the first between network packet reception and Flash command submission (① - ④) and the second between Flash completion and network packet transmission (⑤ - ⑧), without any additional interruptions or thread scheduling. Running to completion eliminates latency variability and improves data cache locality for request processing. ReFlex's two-step run to completion model avoids blocking on Flash requests. ReFlex avoids interrupt overhead by polling for network packet arrivals and Flash completions. Moreover, ReFlex implements zero-copy by passing pointers to the buffers used to DMA data from the NIC or Flash device.

In addition to the benefit from lower latency, ReFlex improves the throughput of remote Flash requests using two methods. First, it uses asynchronous I/O to overlap Flash device latency (50μs or more) with network processing for other requests. Once a command is submitted to the Flash device ④, the thread polls the NMVe completion queue for previously issued requests that require outgoing network processing and polls the NIC receive queue for incoming packets that require incoming network processing. As long as there is work to do, the thread does not idle. Second, ReFlex employs adaptive batching of requests in order to amortize overheads and improve prefetching and instruction cache efficiency [11]. Under low load, incoming packets or completed NVMe commands are processed immediately without any delay. As load rises, the NIC receive and NVMe completion queues fill up and provide the opportunity to process multiple incoming packets or multiple completed accesses in a batch. The batch size increases with load but it is capped to 64 to avoid excessive latencies. Unlike conventional batching, which trades off latency for bandwidth, adaptive batching achieves a good balance between high throughput and low latency [11].



(a) Device A Request Cost Model (b) Device B Request Cost Model (c) Device C Request Cost Model

Figure 3: Request cost models for various workloads on 3 different NVMe Flash devices.

ReFlex scales to multiple threads, each using a dedicated core and separate hardware queue pairs. Threads need to coordinate only when issuing accesses to the NVMe command queue so that they collectively respect the tail latency and throughput SLOs of the various tenants that share the ReFlex server (see §3.2). A local control plane periodically monitors the load of all ReFlex threads and their ability to achieve the requested SLOs in order to increase or decrease the number of ReFlex threads. When the number of threads changes, remote tenants and network connections are rebalanced across threads as explained in [53]. Rebalancing takes a few milliseconds and does not lead to packet dropping or reordering.

3.2 QoS Scheduling and Isolation

The QoS scheduler allows ReFlex to provide performance guarantees for tenants sharing the Flash device(s) in a server. A *tenant* is a logical abstraction for accounting for and enforcing service-level objectives (SLOs). An SLO specifies a tail read latency limit at a certain throughput and read/write ratio. For example, a tenant may register an SLO of 50K IOPS with 200μs read tail latency (95$^{\text{th}}$ percentile) at an 80% read ratio. In addition to such *latency-critical (LC) tenants* which have guaranteed allocations in terms of tail latency and throughput, ReFlex also serves *best-effort (BE) tenants*, which can opportunistically use any unallocated or unused Flash bandwidth and tolerate higher latency. A tenant definition can be shared by thousands of network connections, originating from different client machines running any number of applications. An application can use multiple tenants to request separate SLOs for different data streams.

As shown in Figure 1, enforcing an SLO on Flash device accesses is complicated by two factors. First, the maximum bandwidth (IOPS) the device can support depends on the overall read/write ratio of requests it sees across all tenants. Second, the tail latency for read requests depends on both the overall read/write ratio and the current bandwidth load. Hence, the QoS scheduler requires global visibility and control over the total load on Flash and the type of outstanding I/O operations. We use a request cost model to account for each Flash I/O's impact on read tail latency (§3.2.1) and a novel scheduling algorithm that guarantees SLOs across all tenants and all dataplane threads (§3.2.2). ReFlex does not assume a priori knowledge of workloads.

3.2.1 Request Cost Model

The model estimates read tail latency as a function of *weighted* IOPS, where the cost (weight) of a request depends on the I/O size, type (read vs. write), and the current read/write request ratio r on the device:

$$\text{I/O cost} = \left\lceil \frac{\text{I/O size}}{4\text{KB}} \right\rceil \times C(\text{I/O type}, r)$$

Cost is a function of r because some Flash devices provide substantially higher IOPS for read-only loads (r=100%) compared to 99% or lower read loads, as seen in Figure 1. Hence, the model adjusts the cost of read requests when the device load is read-only. Costs are expressed in multiples of tokens, where one token represents the cost of a 4KB random read request. In all Flash devices we have used, cost scales linearly with request size for sizes larger than 4KB (e.g., a 32KB request costs as much as 8 back-to-back 4KB requests). Cost is constant for requests 4KB and smaller, as these Flash devices seem to operate at 4KB granularity.

We calibrate the cost model for each type of Flash device deployed in the ReFlex server. First, we measure tail latency versus throughput with local Flash for workloads with various read/write ratios and request sizes (see 4KB example in Figure 1). Since the cost of write requests depends on the frequency of garbage collection and page erasure events, we conservatively use random write patterns to trigger the worst case. Next, we use curve fitting to derive $C(\text{I/O type}, r)$. The model can be re-calibrated after deployment to account for performance degradation due to Flash wear-out [13].

Figure 3 shows latency versus weighted IOPS (tokens) plots for three different NVMe devices from multiple vendors and representing multiple generations and capacities of Flash devices. Device A is the one characterized in Figure 1. The value of $C(\text{write}, r < 100\%)$ is 10, 20, and 16 tokens for devices A, B, and C respectively. This means that write operations are 10 to 20 times more expensive than read operations, depending on the device. For device A, when the load

from all tenants is read-only, the cost of a 4KB read request is half a token (i.e., $C(\text{read}, r = 100\%) = \frac{1}{2}$ token). For all three devices, our linear cost model leads to similar behavior for tail latency versus load across all read/write load distributions and request sizes. This uniform behavior allows the ReFlex scheduler to manage SLOs when serving multiple tenants with different throughput requirements and read/write request ratios. Although non-linear curve-fitting models can achieve better fit, the accuracy of the linear model has been sufficient for our scheduler and we prefer it because of simplicity.

3.2.2 Scheduling Mechanism

The QoS scheduler builds upon the cost model to maintain tail latency and throughput SLOs for LC tenants, while allowing BE tenants to utilize any spare throughput in a fair manner.

Token management: The ReFlex scheduler generates tokens at a rate equal to the maximum weighted IOPS the Flash device can support at a given tail latency SLO. ReFlex enforces the strictest (lowest) latency SLO among all LC tenants that share a Flash device. For example, to serve two tenants with tail read latency SLOs of 500μs and 1ms, respectively, on a Flash device with the cost model shown in Figure 3a, the scheduler generates 420K tokens/sec to enforce the 500μs SLO. LC tenants are guaranteed a token supply that satisfies their IOPS SLO, weighted by the read/write ratio indicated in their SLO. For example, assuming 4KB requests and a write cost of 10 tokens, a tenant registering an SLO of 100K IOPS with an 80% read ratio is guaranteed to receive tokens at a rate of 0.8(100K IOPS)(1 $\frac{\text{token}}{\text{I/O}}$) + 0.2(100K IOPS)(10 $\frac{\text{tokens}}{\text{I/O}}$) = 280K tokens/sec. Tokens generated by the scheduler but not allocated to LC tenants are distributed fairly among BE tenants. The scheduler spends a tenant's tokens based on per-request costs as it submits the tenant's requests to the Flash device.

Scheduling Algorithm: Each ReFlex thread enqueues Flash requests in per-tenant, software queues. When the thread reaches the QoS scheduling step in the dataplane execution model (§3.1), the thread uses Algorithm 1 to calculate the weighted cost of enqueued requests and submit all admissible requests to the Flash device, gradually spending each tenant's tokens. Depending on the thread load and the batching factor, the execution model enters a scheduling round every 0.5μs to 100μs. The control plane and the batch size limit ensure that the time between scheduler invocations does not exceed 5% of the strictest SLO. Frequent scheduling is necessary to avoid excessive queuing delays and to maintain high utilization of the NVMe device.

Latency-critical tenants: The scheduling algorithm starts by serving LC tenants. First, the scheduler generates tokens for each LC tenant based on their IOPS SLO and the elapsed time since the previous scheduler invocation. Since the control plane has determined each LC tenant's weighted IOPS reservation is admissible, the scheduler can typically submit all queued requests of LC tenants to the NVMe device. However, since traffic is seldom uniform and tenants may issue more or less IOPS than the average IOPS reserved in their SLO, the scheduler keeps track of each tenant's token usage. We allow LC tenants to temporarily burst above their token allocation to avoid short term queueing. However, we limit the burst size by rate-limiting LC tenants once they reach the token deficit limit (`NEG_LIMIT`). This parameter is empirically set to -50 tokens to limit the number of expensive write requests in a burst. We also notify the control plane when this limit is reached to detect tenants with incorrect SLOs that need renegotiation.

LC tenants that consume less than their available tokens are allowed to accumulate tokens for future bursts. Accumulation is limited by the `POS_LIMIT` parameter. When reached, the scheduler donates a big fraction of accumulated tokens (empirically 90%) to the global token bucket for use by BE tenants. `POS_LIMIT` is empirically set to the number of tokens the LC tenant received in the last three scheduling rounds to accommodate short bursts without going into deficit.

Best-effort tenants: The scheduler generates tokens for BE tenants by giving each BE tenant a fair share of unallocated throughput on the device. Unallocated device throughput corresponds to the token rate the device can support while enforcing the strictest LC latency SLO minus the sum of LC tenant token rates (based on LC tenant IOPS SLOs). Assuming N BE tenants, every scheduling round, each BE tenant receives $\frac{1}{N}$th of the unallocated token rate times the time elapsed since the previous scheduling round. If a BE tenant does not have enough tokens to submit all of its en-

Algorithm 1 QoS Scheduling Algorithm

```
1:  procedure SCHEDULE
2:      time_delta = current_time() − prev_sched_time
3:      prev_sched_time = current_time()
4:      for each latency-critical tenant t do
5:          t.tokens += generate_tokens_LC(t.SLO, time_delta)
6:          if t.tokens < NEG_LIMIT then
7:              notify_control_plane()
8:          while t.demand > 0 and t.tokens > NEG_LIMIT do
9:              t.tokens −= submit_next_req(t.queue)
10:         if t.tokens > POS_LIMIT then
11:             atomic_incr_global_bucket(t.tokens × FRACTION)
12:             t.tokens −= t.tokens × FRACTION
13:     for each best-effort tenant t in round-robin order do
14:         t.tokens += generate_tokens_BE(time_delta)
15:         d = t.demand − t.tokens
16:         if d > 0 then
17:             t.tokens += atomic_decr_global_bucket(d)
18:         t.tokens−=submit_admissible_reqs(t.queue, t.tokens)
19:         if t.tokens > 0 and t.demand == 0 then
20:             atomic_incr_global_bucket(t.tokens)
21:             t.tokens = 0
22:     if all_threads_scheduled() then
23:         atomic_reset_global_bucket()
```

queued requests, the tenant can claim tokens from the global token bucket, which are supplied by LC tenants with spare tokens (if any). BE tenants are scheduled in a round robin order across scheduling rounds to provide fair access to the global token bucket. The scheduler *conditionally* submits a BE request only if the tenant has sufficient tokens for the request. Rate limiting BE traffic is essential for achieving LC SLOs. Since scheduling rounds occur at high frequency, a typical round may generate only a fraction of a token. BE tenants accumulate tokens over multiple scheduling rounds when their request queues are not empty. When a BE tenant's software queue is empty, we disallow token accumulation to prevent bursting after idle periods. This aspect of the scheduler is inspired by Deficit Round Robin (DRR) scheduling [60]. Tokens left unused by a BE tenant are donated to the global token bucket for use by other BE tenants. To avoid large accumulation allowing BE tenants to issue uncontrolled bursts, we periodically reset the bucket.

If a ReFlex server manages more than one NVMe device, we run an independent instance of the scheduling algorithm for each device with separate token counts and limits. We assume the server machine has sufficient PCIe bandwidth, a condition easily met by PCIe Gen3 systems.

4. ReFlex Implementation

ReFlex consists of three components: the server, clients, and control plane. Their implementation leverages open-source code bases.

4.1 ReFlex Server

The remote Flash server is the main component of ReFlex. We implemented it as an extension to the open-source, IX dataplane operating system [1, 11]. IX uses hardware support for processor virtualization (through the Dune module [10]) and multi-queue support in NICs (through the Intel DPDK driver [30]) to gain direct and exclusive access to multiple cores and network queues in a Linux system. These resources are used to run a dataplane kernel and any applications on top of it. The original IX dataplane was developed for network-intensive workloads like in-memory, key-value stores. IX uses run to completion of incoming requests and bounded, adaptive batching to optimize both tail latency and throughput. IX also splits connections between threads, using separate cores and queues to scale without requiring synchronization or significant coherence traffic. These optimizations make IX a great starting point for the ReFlex server.

ReFlex extends IX in the following ways. First, we developed an NVMe driver leveraging Intel's Storage Performance Development Kit (SPDK) [31] to interface to Flash devices and gain exclusive access to NVMe queue pairs. Second, we implemented the dataplane model shown in Figure 2. In IX, the run to completion model includes all work for a key-value store request, from packet reception to reply transmission. Directly applying this monolithic run to completion model in ReFlex would require blocking for every Flash access. Instead, we introduce a two step model that retains the efficiency of run to completion but allows for asynchronous access to Flash. The first run to completion step is from packet reception to Flash command submission and the second is from Flash command completion to reply transmission. We maintained adaptive batching with a maximum batch size of 64. Third, we implemented the QoS scheduler as part of the first run to completion step. Fourth, we introduced the system calls and events needed for remote Flash accesses shown in Table 1. The original IX defines system calls and events for opening and closing connections, receiving and sending network messages, and managing network errors. We introduced system calls and events to register and unregister tenants, submit and complete NVMe read and write commands, and manage NVMe errors. Finally, we developed the ReFlex user-level server code that consumes events delivered by the dataplane and issues system calls back to it. The cookie parameter allows the user-space server code to track requests and retrieve their context upon an event notification. Note that all event and system calls are communicated over shared memory arrays without the need for blocking, interrupts, or thread scheduling. This also enables batching of systems calls under high load in order to reduce overheads. The dataplane implements zero-copy; buffers for read and write data are initialized in the ReFlex user-space code and provided as a parameter for read and write system calls. They are released after the user-space code is notified of a send completion.

The ReFlex server is written in `C` and consists of the following source lines of code (SLOC): 490 SLOC for the user-level server, 954 SLOC for the IX NVMe driver and 628 SLOC for the dataplane including the QoS scheduler.

System Calls (batched)		
Type	**Parameters**	**Description**
register	id, latency, IOPS, rw_ratio, cookie	Registers a tenant with SLO
unregister	handle	Unregisters a tenant
read	handle, buf, addr, len, cookie	Read data from Flash into user buf
write	handle, buf, addr, len, cookie	Write data from user buf into flash

Event Conditions		
Type	**Parameters**	**Description**
registered	handle, cookie, status	Registered tenant, or out of resources error
unregistered	handle	Unregistered tenant
response	cookie, status	NVMe read completed
written	cookie, status	NVMe write completed

Table 1: The systems calls and event conditions that the ReFlex dataplane adds to the IX baseline.

We leverage code from Intel's DPDK and SPDK and the IX dataplane, including the lwIP TCP stack [21].

Multi-threading operation: ReFlex scales to multiple threads, each using a separate core and separate network and NVMe queues. We parallelize the load by dividing tenants across threads. All operations across threads are independent and can occur without synchronization, excluding some QoS scheduling actions described in §3.2.2. Specifically, threads need to occasionally synchronize in order to exchange any spare tokens from their LC tenants so that any BE tenant on any thread can benefit from unused Flash bandwidth. Threads use atomic read-modify-write operations to access the global token bucket. The bucket is reset periodically by having each thread asynchronously mark that it has completed at least one scheduling round. The last thread resets the global bucket. This approach avoids locking overheads and decouples QoS scheduling across threads. In particular, it allows threads to perform scheduling at different frequencies, while still maintaining fairness and guaranteeing system-wide SLOs.

Security model: The ReFlex server enforces access control list (ACL) policies at the granularity of tenants and network connections. It checks if a client has the right to open a connection to a specific tenant and if a tenant has read or write permission for an NVMe namespace (range of logical blocks). These checks can be extended to use certificate mechanisms.

Following IX, ReFlex runs its dataplane in protected kernel mode (guest ring 0), while the high-level server code runs in user space (ring 3) as shown in Figure 2. Any exploitable bug in parsing remote requests or other high-level server functions cannot lead to loss of hardware control and cannot affect the operation of the dataplane or any ordinary Linux application running on the same machine. This approach allows ReFlex to share Flash devices with other Linux applications. Access to Flash by Linux workloads (kernel or user) is mediated through the protected part of ReFlex that includes the QoS scheduler. From a QoS perspective, Linux requests are treated as latency-critical with specific throughput and latency guarantees. We achieve virtually indistinguishable performance compared to running ReFlex all in user mode. The inherit cost of the kernel to user-mode transition is similar to that of a main memory access [11] and it is compensated for by the improved locality that ReFlex achieves using run to completion and zero-copy.

Limitations: The current server implementation has some non-fundamental limitations that we will remove in future versions. First, we limit each tenant to using a single ReFlex thread. Since ReFlex can serve up to 850K remote IOPS per thread (§5.3) and an application can use multiple tenants to access the same data, this is not a significant bottleneck for any application. In the future, we will load balance connections for individual tenants across threads if their overall demands exceed a single thread's throughput. Second, we have implemented a single networking protocol in the ReFlex dataplane, the ubiquitous TCP/IP [21]. Since TCP/IP is the most heavy-weight protocol used in datacenters, this is a conservative choice that defines a lower bound on ReFlex performance. Both tail latency and throughput will improve when we implement UDP or other, lighter-weight transport protocols. Finally, ReFlex currently serves remote read and write requests without any ordering guarantees, beyond ordering forced by the networking protocol (e.g., order within a TCP connection). In the future, we will support barrier operations that can be used to force ordering and build high-level abstractions like atomic transactions.

4.2 ReFlex Clients

Applications can access ReFlex servers over the network using a variety of clients. We have implemented two alternatives that represent extreme points in terms of performance.

The first implementation is a user-level library (536 SLOC), similar to the client library for the binary protocol of the memcached key-value store [46]. The library allows applications to open TCP connections to ReFlex and transmit read and write requests to logical blocks. This client approach avoids the performance overheads of the file-system and block layers of the operating system in the client machine. Nevertheless, the client is still subject to any latency or throughput inefficiencies of the networking layer in its operating system (see §5).

To support legacy client applications, we also implemented a remote block device driver that exposes a ReFlex server as a Linux block device (845 SLOC). The driver translates conventional Linux block I/O (`bio`) requests to ReFlex accesses issued with the user-level library discussed above. The driver implements the multi-queue (`blk-mq`) kernel API [12] and supports one hardware context per core to enable linear scaling with cores. For each hardware context, the driver opens a socket to the ReFlex server and spawns a kernel thread for receiving and completing incoming responses. To minimize latency, the driver directly issues each block to the server without coalescing as the overhead of ReFlex requests is small (38 bytes per 4KB request) and the bandwidth of NVMe devices does not change significantly if we use requests larger than 4KB. At 4KB, the Linux TCP stack supports up to 70K messages per thread and hence the driver needs to execute at least 4 threads (or 6 for improved latency) to fully utilize a 10GbE interface, as we will show in §5.6.

4.3 ReFlex Control Plane

The ReFlex control plane consists of two components, a local component that runs on every ReFlex server and a global one. We have currently implemented the former.

The local control plane is responsible for the following actions. First, when a new LC tenant is registered, the control plane determines if the tenant is admissible and which server thread it should be bound to. It uses the strictest latency SLO

from all LC tenants and the throughput-latency characteristics of each device to check if the new tenant's SLO can be met without violating SLOs of existing tenants. When a tenant registers or terminates, the control plane re-calculates the rate of token generation for LC and BE tenants. The control plane intervenes if an LC tenant consistently bursts above its SLO allocation by notifying the tenant to renegotiate its SLO. Second, the local control plane monitors the request latency and the thread load. If latency and load are high, it allocates resources for additional threads and rebalances tenants. If load is low, it deallocates threads and their resources, returning them to Linux for general use. This last function is a derivative of the IX control plane that can dynamically rightsize the number and clock frequency of threads used by the IX dataplane without packet loss or reordering [53]. Finally, the control plane periodically calibrates the request cost model and determines the throughput-latency characteristics of each Flash device (see §3.2).

In future work, we will develop a global control plane that manages remote Flash resources across a datacenter cluster and optimizes the allocation of Flash capacity and IOPS. For example, the global control plane should try to co-locate tenants with similar tail latency requirements such that strict requirements of one tenant do not limit the IOPS available to other tenants. The global control plane should also maintain global latency and throughput SLOs for applications that span across multiple ReFlex servers [4, 28, 65, 66].

5. Evaluation

5.1 Experimental Methodology

Hardware setup: Our experimental setup consists of identical server and client machines with Intel Xeon CPU E5-2630 processors (Sandy Bridge EP) with 12 physical cores across two sockets running at 2.3 GHz and 64GB DRAM. The machines use Intel 82599ES 10GbE NICs connected via an Arista 7050S-64 switch. They run Ubuntu LTS 16.04 with a 4.4 Linux kernel. Server machines house PCIe-attached Flash devices, preconditioned with sequential writes to the whole address space followed by a series of random writes to reach steady state performance. We tested ReFlex with three different Flash devices whose request cost models are shown in Figure 3. We show results for ReFlex using device A as it achieves the highest raw IOPS, up to 1M IOPS for read-only workloads (see Figure 1). For all experiments, we disable power management and operate CPU cores at their maximum frequency to ensure result fidelity. NICs are configured with jumbo frames enabled and large receive offload (LRO) and generic receive offload (GRO) disabled. LRO and GRO distort unloaded latency as received packets are sometimes buffered instead of being directly delivered to the kernel. We enable interrupt coalescing with a 20μs interval.

Clients: We use Linux-based clients in most experiments. We extend the `mutilate` load generator [40] to use our user-level client library and issue read/write requests to ReFlex. `mutilate` coordinates a large number of client threads across multiple machines to generate a desired throughput while a separate, unloaded client measures latency by issuing one request at a time. To reduce client-side performance overheads, we also evaluate unloaded latency and peak IOPS per core for ReFlex with clients running a similar load generator on top of the IX dataplane, which achieves significantly lower latency and higher throughput than the Linux networking stack.

	Reads (μs)		**Writes (μs)**	
	Avg	p95	Avg	p95
Local (SPDK)	78	90	11	17
iSCSI	211	251	155	215
Libaio (Linux Client)	183	205	180	205
Libaio (IX Client)	121	139	117	144
ReFlex (Linux Client)	117	135	58	64
ReFlex (IX Client)	99	113	31	34

Table 2: Unloaded Flash latency for 4KB random I/Os, including round-trip network latency for client and server.

I/O size: We issue 4KB read and write requests in most experiments. Since we use a 10GbE network infrastructure, clients issuing 4KB IOPS can saturate the NIC of the ReFlex server before they saturate the NVMe Flash device (1M IOPS peak). Hence, we use 1KB requests in some experiments to stress IOPS of the ReFlex server. Modern datacenters include 40GbE networking infrastructure and future datacenters will likely deploy 100GbE. Both technologies will remove this bottleneck.

Baseline: We compare the performance of remote accesses over ReFlex to that of issuing local accesses to the Flash device using SPDK [31]. SPDK offers the best local performance we can expect as it gives software direct access to NVMe queues without the need to go through the Linux filesystem or block device layers. We also compare ReFlex to two software-based schemes for remote Flash access: 1) the Linux iSCSI system [49] and 2) a lightweight remote storage server that maximizes performance on Linux by efficiently handling multiple connections per thread using `libevent` and overlapping communication and computation using `libaio`. We do not have access to a hardware-accelerated remote Flash environment, but we compare to results quoted in a public presentation on NVMe over RDMA Fabrics [45]. In §5.4, we evaluate the performance problems that arise when NVMe devices are shared without a software-based QoS scheduler like the one in ReFlex.

5.2 Unloaded latency

We first measure unloaded latency for Flash accesses. Table 2 shows the average and 95$^{\text{th}}$ percentile latency of 4KB random read and write requests issued with queue depth 1. Remote accesses include the round-trip networking overheads in both client and server. Remote access over iSCSI

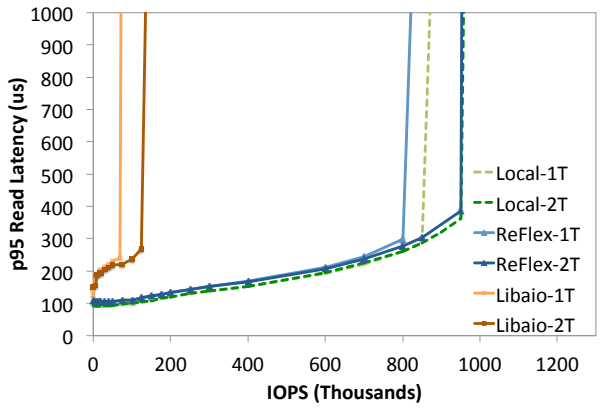
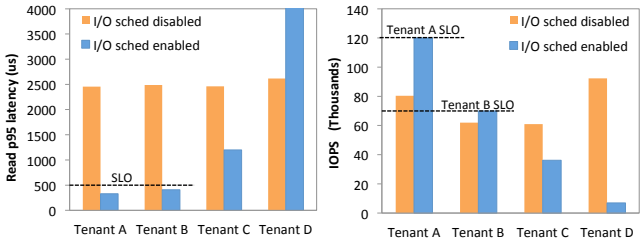
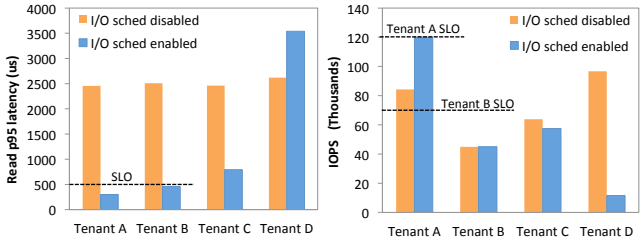


Figure 4: Tail latency vs. throughput for 1KB, read-only requests.



(a) Scenario 1: p95 latency (b) Scenario 1: IOPS



(c) Scenario 2: p95 latency (d) Scenario 2: IOPS

Figure 5: Tail latency and IOPS for 4 tenants sharing a ReFlex server. Tenants A and B are LC, while tenants C and D are BE. Tenants issue 4kB I/Os with read ratios of 100%, 80%, 95%, and 25%, respectively. In Scenario 1, tenants A and B attempt to use all the IOPS in their SLO. In Scenario 2, tenant B uses less than its reservation.

increases latency by 2.8× for read requests due to heavy-weight protocol processing on both the client and server side, involving data copying between socket, SCSI and application buffers. The `libaio-libevent` remote Flash server is significantly faster than iSCSI, but the Linux network and storage stacks still add over 100μs to average and tail latency. The dataplane execution model of ReFlex adds 21μs to local Flash latency (IX client). At 113μs of tail read latency, a ReFlex server is close to the performance of many (local) NVMe devices. Unloaded write latency is lower than read latency due to DRAM buffering on the Flash device. Thus, the overhead of iSCSI and `libaio-libevent` is even more significant for write I/Os. ReFlex outperforms both iSCSI and `libaio-libevent`, adding 20μs to local write latency (IX client). Comparing ReFlex latency results with Linux and IX clients shows that for low latency remote Flash access, it is also important to optimize the client.

NVMe over Fabrics provides marginally lower latency overhead (8μs), measured with a higher throughput 40GbE Chelsio NIC (lower transmit latency for 4KB) and a 3.6GHz Haswell CPU (versus a 2.3GHz Sandy Bridge CPU) [45]. Remote Flash latency with ReFlex includes a full TCP/IP stack and a QoS scheduler that allows multiple clients to connect to the Flash server. ReFlex would likely benefit from TCP offloading in Chelsio NICs.

5.3 Throughput and CPU Resource Cost

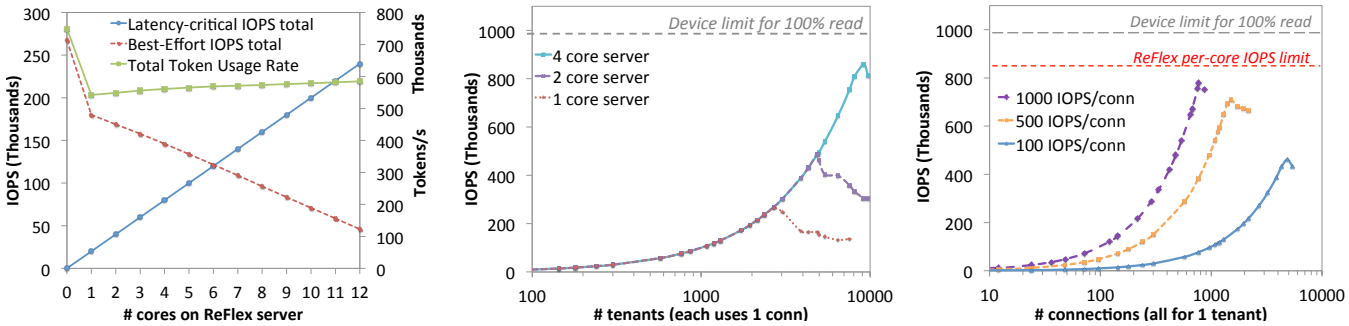
Figure 4 plots tail latency (95th percentile) as a function of throughput (IOPS) for 1KB read-only requests. Even for local accesses to Flash with SPDK, it takes two cores to saturate the 1M IOPS of the Flash device. A single core can support up to 870K IOPS on local Flash. ReFlex achieves up to 850K IOPS with a single core for network and storage processing. With two cores, ReFlex saturates 1M IOPS on Flash, introducing negligible latency overhead compared to local access. In contrast, the `libaio-libevent` server achieves only 75K IOPS/core and at higher latency due to higher compute intensity for request processing. This server requires over 10× more CPU cores to achieve the throughput of ReFlex. The hardware-accelerated NVMe over Fabrics can reportedly achieve 460K IOPS at 20% utilization of a 3.6GHz Haswell core [45].

At high load, a ReFlex thread spends about 20% of execution time on TCP/IP processing. Hence, coupled with a lighter network protocol, ReFlex can deliver even higher throughput. The time spent on QoS scheduling varies between 2% and 8%, depending on the number of tenants served.

ReFlex's ability to serve millions of IOPS with a small number of cores without impacting tail latency is important for making remote Flash practical and cost effective in datacenters. To put IOPS per core into perspective, assume we deploy ReFlex on the latest Broadwell or Skylake class CPUs by Intel. The improved core performance will likely allow ReFlex to reach 1M IOPS/core. Assuming 20 cores per CPU socket, ReFlex will be able to share a 1M IOPS Flash device using 2.5% of the compute capacity of a 2-socket server. Alternatively, using 4 Flash devices, ReFlex will need 8% of the server's compute capacity to saturate a 100GbE link with 4KB I/Os.

5.4 Performance QoS and Isolation

We now evaluate the QoS scheduler using multiple tenants with different SLOs. The following experiments use a single ReFlex thread. We evaluate multi-core scalability in §5.5.



(a) Multi-core scaling (b) Tenant scaling (c) Connection scaling

Figure 6: Scalability experiments. In Fig 6a, ReFlex scales to 12 cores, enforcing a 2ms latency SLO for 90% read (LC) and 80% read (BE) tenants while maintaining high Flash utilization (570K tokens/s). Fig 6b shows a single ReFlex core can support up to 2,500 tenants. Fig 6c shows a single ReFlex core can serve thousands of TCP connections.

We first consider Scenario 1 (Figure 5a - 5b), where two latency-critical (A, B) and two best-effort (C, D) tenants share a ReFlex server. Both A and B require 95th percentile read latency of 500μs. Tenant A requires 120K IOPS at 100% read, while B requires 70K IOPS at 80% read. C and D are best effort tenants with 95% and 25% read loads, respectively. To guarantee read tail latency below the 500μs SLO, our Flash device can support up to 420K weighted IOPS. Thus, the QoS scheduler generates 420K tokens/sec. LC tenant A receives 120K tokens/sec while tenant B receives 196K tokens/sec = 0.8(70K IOPS)(1 $\frac{\text{token}}{\text{I/O}}$) + 0.2(70K IOPS)(10 $\frac{\text{tokens}}{\text{I/O}}$). Thus, the two LC tenants collectively reserve 75% of the device throughput, leaving 25% of tokens for BE tenants. Figure 5 shows the tail latency and IOPS for each tenant with the QoS scheduler disabled and enabled. Without QoS scheduling, tail read latency is above 2ms for all tenants due to read/write interference. Tenant B also operates below its SLO throughput. With QoS scheduling enabled, latency and throughput SLOs are met for both LC tenants (Figure 5a) at the expense of BE throughput (Figure 5b). BE tenants C and D receive a fair share of unallocated tokens (52K tokens/sec each), but D achieves lower IOPS than C due to its higher percentage of write I/Os (writes cost 10 times more tokens than reads).

Scenario 2 uses the same tenants as Scenario 1 with identical SLOs. However, latency-critical tenant B issues only 45K IOPS instead of the 70K reserved in its SLO. BE tenants can now reach higher throughput (Figure 5c - 5d), as they acquire the unused tokens of tenant B, in addition to tokens not allocated to LC tenants. The round-robin serving of BE tenants ensures fair access to unused tokens.

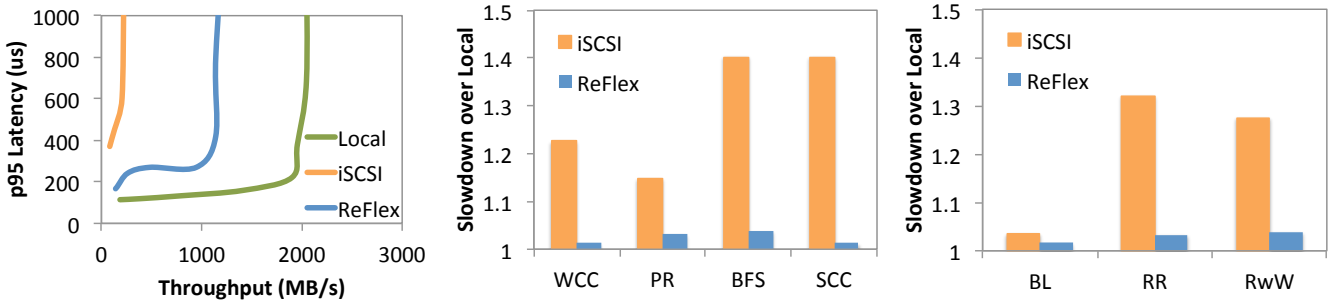
While these scenarios involve just 4 tenants, they are sufficient to show the need for QoS scheduling for remote Flash accesses, beyond what hardware provides. Our QoS scheduler can guarantee SLOs while being work-conserving and fair for best-effort tenants.

5.5 Scalability

We now evaluate how ReFlex scales in the dimensions of cores, tenants, and connections.

Cores: We run ReFlex with up to 12 cores (6 cores per socket) to test the scheduler's multi-core scalability. Each thread manages a single LC tenant with an SLO of 20K IOPS (90% read, 4KB requests) at up to 2ms tail read latency (95th percentile). The 2ms latency SLO allows our Flash device to serve up to 12 such tenants before the SLO is no longer admissible due to too much write interference. Two ReFlex threads also each serve a BE tenant (80% read, 4KB). Figure 6a shows a linear increase in the aggregate IOPS for LC tenants as we scale the number of cores (tenants) without any scaling bottleneck in the scheduler. Meanwhile, aggregate BE IOPS decrease due to rate-limiting with less spare bandwidth on the device. Although not shown in Figure 6, the tail read latency of all LC tenants stays below the 2ms SLO. The total token usage rate (green line with values marked on the secondary y-axis) is high when no LC tenants are registered since the two BE tenants are allowed to issue as many requests as the device can handle. As soon as the first LC tenant registers, the scheduler caps the token rate to 570K tokens/s to enforce the 2ms SLO. Token usage remains at this level as we scale the number of cores, as ReFlex saturates the Flash device at all points without violating SLOs.

Tenants: We evaluate the number of tenants each ReFlex thread can serve before tenant management becomes a performance bottleneck. Each tenant uses a single connection to issue 100 1KB read IOPS. In this experiment, low IOPS per connection are necessary to avoid saturating the Flash device before reaching a tenant scaling limit. Figure 6b shows that a single ReFlex core can serve up to 2,500 tenants, while 2 ReFlex cores serve 5,000 tenants, and a 4-core ReFlex server comes close to supporting 10K tenants, at which point we approach the 1M read-only IOPS limit of the Flash device. As we scale the number of tenants per thread, tail latency may increase as the QoS scheduling frequency decreases.



(a) FIO p95 Access Latency (b) FlashX Performance (c) RocksDB

Figure 7: Performance evaluation with Linux block device driver for ReFlex.

This case is detected by the ReFlex control plane which allocates more cores and rebalances tenants as needed (see §4.3).

TCP Connections: A tenant may be used to track the QoS requirement of an application that uses multiple client machines and threads. Hence, it is important to know how many TCP connections the ReFlex server can handle. Figure 6c plots the throughput of a single ReFlex thread when scaling the number of TCP connections associated with a tenant. At 100 IOPS per connection, a single ReFlex thread can support up to 5K connections. Performance degrades beyond this point as the TCP connection state no longer fits in the last-level cache and TCP/IP processing slows down due to main memory accesses. This is similar to the connection scaling behavior of IX [11], which saturates at 10K connections in experiments with smaller messages (64B vs. 1KB) which trigger fewer misses. With 1K IOPS/connection, the ReFlex core approaches its peak bandwidth, achieving 780K IOPS with 850 connections. The peak bandwidth is lower than the 850K IOPS in §5.3 due to higher cache pressure.

5.6 Linux Application Performance

We now use the remote block device driver for ReFlex to evaluate performance with legacy Linux applications. We compare to performance with the local NVMe device driver and Linux iSCSI remote block I/O. We show results for the following applications: the flexible I/O tester (FIO) [32], the FlashX graph analytics framework [75], and Facebook's RocksDB key-value store [22].

FIO: Figure 7a shows the latency-throughput curves for FIO issuing 4KB random reads with queue depth up to 64. We need multiple FIO threads to reach maximum throughput: 5 threads with the local NVMe driver, 3 threads with iSCSI, and 6 with the ReFlex block driver. As expected, ReFlex stops scaling when it saturates the 10GbE network interfaces at both the client and the server. However, since FIO on top of ReFlex scales linearly up to 6 threads, we expect it will be able to match local throughput given higher bandwidth network links. The higher latency of ReFlex in this experiment is due to the client-side overheads of the Linux block and networking layers. Still, ReFlex provides 4× higher throughput than iSCSI and 2× lower tail and average latency. We evaluated ReFlex with optimized clients in §5.3.

FlashX: We use FlashX, a graph processing framework that uses the SAFS user-space filesystem to efficiently store and retrieve vertex and edge data from Flash. We execute four graph benchmarks including weakly connected components (WCC), pagerank (PR), breadth-first search (BFS) and strongly connected components (SCC) on the SOC-LiveJournal1 social network graph from SNAP [39]. The graph contains 4.8M vertices and 68.9M edges, which we store on a local block device or on a remote block device through iSCSI or ReFlex. Figure 7b shows the impact of accessing remote Flash on the end-to-end application execution time. Compared to performance on local Flash, ReFlex introduces only a small slowdown, between 1% for WCC and 3.8% for BFS. In contrast, iSCSI reduces performance by 15% for PR and up to 40% for BFS and SCC.

RocksDB: Finally, we use RocksDB to evaluate key-value store performance on Flash with ReFlex. We install an ext4 filesystem on the NVMe block device and mount it as either local or remote via ReFlex or iSCSI. We place both RocksDB's database and its write-ahead-log on Flash. We generate a workload using `db_bench`, a benchmarking tool provided with RocksDB. We use `cgroups` [43] to limit memory and reduce the effect of Linux's page cache, thus exercising Flash storage with a short experiment on a 43GB database. We first populate the database with the *bulkload* (BL) routine and then execute the *randomread* (RR) and *readwhilewriting* (RwW) benchmarks. Figure 7c shows the end-to-end execution time slowdown of RocksDB over iSCSI and ReFlex compared to local Flash. For the write-heavy BL test, performance is almost equal between local and remote as the Flash itself limits IOPS. For RR and RwW, iSCSI shows a slow down of 32% and 27%, respectively, while ReFlex slows down performance by less than 4%.

6. Discussion

There are two limitations of current Flash hardware that are particularly relevant to ReFlex.

Read/write interference: Write operations have a big impact on the tail latency of concurrent read requests. Our scheduler uses a request cost model to avoid pushing beyond the latency-throughput capabilities of the Flash device for the current read/write ratio. However, we are still limited to enforcing tail read latency SLOs at the 95th percentile. Stricter SLOs, such as 99th or 99.9th percentile are difficult to enforce on existing Flash devices without dramatically reducing IOPS as reads frequently stall behind writes, garbage collection, or wear leveling tasks. Future Flash devices should limit read/write interference, targeting tail behavior in addition to average. For example, the Flash Translation Layer (FTL) could always read out and buffer Flash pages before writing in them [2, 74].

Hardware support for request scheduling: Existing Flash devices schedule requests from different NVMe hardware queues using simplistic round-robin arbitration. To guarantee SLOs, ReFlex has to use a software scheduler that implements rate limiting and priorities. The NVMe specification defines a weighted round-robin arbiter [47], but it is not implemented by any Flash device we are aware of. This arbiter would allow ReFlex threads to submit requests to hardware queues with properly weighted priorities, thus eliminating the need to enforce priorities between tenants in software. ReFlex would still implement rate limiting in software as it must manage the device latency-throughput characteristics under varying read/write ratios, defend against SLO violations (long bursts by LC tenants), and support a number of tenants that may exceed the number of hardware queues.

7. Related Work

We discuss related work on storage QoS and high performance network stacks. Alternative approaches for remote access to Flash are discussed in §2.

Storage QoS: Prior work has extensively studied quality of service and fairness for shared storage [5, 24, 25, 27, 44, 62, 73]. Timeslice-based I/O schedulers like Argon, CFQ, and FIOS offer tenants exclusive device access for regulated time quanta to achieve fairness [7, 52, 69]. This approach can lead to poor responsiveness and timeslices may not always be fair on Flash as background tasks (i.e., garbage collection) impact device performance.

In contrast, fair-queuing-based solutions interleave requests from all tenants. The original weighted fair queuing schedulers [20, 51] have successfully been adapted from network to storage I/O with support for reordering, throttling, and/or batching requests to leverage device parallelism [14, 26, 34, 58, 59, 68]. Our I/O scheduler resembles Deficit Round Robin in that tenants accumulate tokens each round, so long as they have demand [60]. Zygaria and AQuA also apply a token bucket approach to serve real-time tenants while offering spare device bandwidth to best-effort traffic, but they only provide throughput guarantees while ReFlex also enforces tail latency SLOs [71, 72].

Tail latency SLOs: PriorityMeister provides tail latency guarantees even at the 99.99th percentile by mediating access to shared network and storage resources using a token-bucket mechanism similar to ReFlex [76]. Unlike ReFlex, the scheduler profiles workloads to assign different priorities to latency-critical tenants. Cake uses a feedback controller to enforce tail latency SLOs, but only supports a single latency-critical tenant [70]. Avatar relies on feedback instead of device-specific performance models to control tail latency on disk [73].

Flash-specific challenges: Many I/O schedulers specifically designed for Flash use a request cost model to account for read/write interference. FIOS was one of the first schedulers to address Flash write interference and provide fairness using timeslices [52]. FlashFQ, a virtual-time based scheduler, improves fairness and responsiveness through throttled dispatch and I/O anticipation [58]. Libra is an I/O scheduling framework that allocates per-tenant throughput reservations and uses a virtual IOPS metric to capture the non-linearity between raw IOPS and bandwidth [61]. While FIOS, FlashFQ and Libra all assign I/O costs, their cost models do not necessarily capture a request's impact on the *tail latency* of concurrent I/Os, since these schedulers are designed for fairness and throughput guarantees rather than latency QoS.

High Performance Networking: ReFlex leverages the IX dataplane for high performance networking [11]. Alternative network stacks, such as mTCP [33], Sandstorm [42] and OpenOnload [64], apply similar techniques in user space to achieve high throughput and/or low latency networking.

8. Conclusion

We described ReFlex, a software system for remote Flash access over commodity networking. ReFlex uses a dataplane design to closely integrate and reduce the overheads of networking and storage processing. This allows the system to serve up to 850K IOPS per core while adding only 21μs over direct access to local Flash. The QoS scheduler in ReFlex enforces latency and throughput SLOs across thousands of tenants sharing a device. ReFlex allows applications to flexibly allocate Flash across any machine in the datacenter and still achieve nearly identical performance to using local Flash.

Acknowledgments

We thank the anonymous reviewers and Christina Delimitrou for their valuable feedback. This work is supported by the Stanford Platform Lab, Samsung Semiconductor and NSF grant CNS-1422088. Ana Klimovic is supported by a Stanford Graduate Fellowship and a Microsoft Research PhD Fellowship.

References

[1] IX-project: protected dataplane for low latency and high performance. `https://github.com/ix-project`, 2016.

[2] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *USENIX Annual Technical Conference*, pages 57–70, 2008.

[3] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proc. of USENIX Hot Topics in Operating Systems*, HotOS'13, pages 12–12, 2011.

[4] Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch. Hippodrome: Running circles around storage administration. In *Proc. of the 1st USENIX Conference on File and Storage Technologies*, FAST '02. USENIX Association, 2002.

[5] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. End-to-end performance isolation through virtual datacenters. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'14, pages 233–248, October 2014.

[6] Avago Technologies. Storage and PCI Express – A Natural Combination. `http://www.avagotech.com/applications/datacenters/enterprise-storage`, 2016.

[7] Jens Axboe. Linux block IOpresent and future. In *Ottawa Linux Symp*, pages 51–61, 2004.

[8] Microsoft Azure. Storage. `https://azure.microsoft.com/en-us/services/storage/`, 2016.

[9] Luiz Andr Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 2009.

[10] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'12, pages 335–348, 2012.

[11] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'14, pages 49–65, October 2014.

[12] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proc. of International Systems and Storage Conference*, page 22. ACM, 2013.

[13] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: Measurements and analysis. In *Proc. of USENIX Conference on File and Storage Technologies*, FAST'10, pages 9–9. USENIX Association, 2010.

[14] John Bruno, Jose Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. In *Proc. of the IEEE International Conference on Multimedia Computing and Systems - Volume 2*, ICMCS '99, pages 400–405. IEEE Computer Society, 1999.

[15] Adrian M. Caulfield and Steven Swanson. QuickSAN: A storage area network for fast, distributed, solid state disks. In *Proc. of International Symposium on Computer Architecture*, ISCA '13, pages 464–474. ACM, 2013.

[16] Mallikarjun Chadalapaka, Hemal Shah, Uri Elzur, Patricia Thaler, and Michael Ko. A study of iSCSI extensions for RDMA (iSER). In *Proc. of ACM SIGCOMM Workshop on Network-I/O Convergence: Experience, Lessons, Implications*, NICELI '03, pages 209–219. ACM, 2003.

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 205–218. USENIX Association, 2006.

[18] Chelsio Communications. NVM Express over Fabrics. `http://www.chelsio.com/wp-content/uploads/resources/NVM_Express_Over_Fabrics.pdf`, 2014.

[19] Franois Alexandre Colombani. HDD, SSHD, SSD or PCIe SSD. Storage Newsletter, `http://www.storagenewsletter.com/rubriques/market-reportsresearch/hdd-sshd-ssd-or-pcie-ssd/`, 2015.

[20] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Symposium Proceedings on Communications Architectures &Amp; Protocols*, SIGCOMM '89, pages 1–12. ACM, 1989.

[21] Adam Dunkels. Design and implementation of the lwip, 2001.

[22] Facebook Inc. RocksDB: A persistent key-value store for fast storage environments. `http://rocksdb.org`, 2015.

[23] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proc. of ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43. ACM, 2003.

[24] Ajay Gulati, Irfan Ahmad, and Carl A. Waldspurger. PARDA: proportional allocation of resources for distributed storage access. In *Proc. of USENIX File and Storage Technologies*, FAST '09, pages 85–98, 2009.

[25] Ajay Gulati, Arif Merchant, Mustafa Uysal, Pradeep Padala, and Peter Varman. Efficient and adaptive proportional share I/O scheduling. *SIGMETRICS Perform. Eval. Rev.*, 37(2):79–80, October 2009.

[26] Ajay Gulati, Arif Merchant, and Peter J. Varman. pclock: An arrival curve based approach for qos guarantees in shared storage systems. In *Proc. of ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 13–24. ACM, 2007.

[27] Ajay Gulati, Arif Merchant, and Peter J. Varman. mClock: handling throughput variability for hypervisor io scheduling. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'10, pages 437–450, 2010.

[28] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl Waldspurger, and Mustafa Uysal. Pesto: Online storage performance management in virtualized datacenters. In *Proc. of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 19:1–19:14. ACM, 2011.

[29] Intel Corp. Intel Rack Scale Architecture Platform. http://www.intel.com/content/dam/www/public/us/en/documents/guides/rack-scale-hardware-guide.pdf, 2015.

[30] Intel Corp. Dataplane Performance Development Kit. https://dpdk.org, 2016.

[31] Intel Corp. Storage Performance Development Kit. https://01.org/spdk, 2016.

[32] Jens Axboe. FIO: Flexible I/O Tester. https://github.com/axboe/fio, 2015.

[33] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: A highly scalable user-level TCP stack for multicore systems. In *Proc. of USENIX Networked Systems Design and Implementation*, NSDI'14, pages 489–502, 2014.

[34] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. In *Proc. of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '04/Performance '04, pages 37–48. ACM, 2004.

[35] Abhijeet Joglekar, Michael E. Kounavis, and Frank L. Berry. A scalable and high performance software iSCSI implementation. In *Proc. of USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 20–20. USENIX Association, 2005.

[36] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proc. of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 9:1–9:14, New York, NY, USA, 2012. ACM.

[37] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proc. of European Conference on Computer Systems*, EuroSys '16, pages 29:1–29:15, 2016.

[38] Yossi Kuperman, Eyal Moscovici, Joel Nider, Razya Ladelsky, Abel Gordon, and Dan Tsafrir. Paravirtual remote I/O. In *Proc. of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 49–65. ACM, 2016.

[39] Jure Leskovec and Andrej Krevl. SNAP datasets: Stanford large network dataset collection. 2015.

[40] Jacob Leverich. Mutilate: High-Performance Memcached Load Generator. https://github.com/leverich/mutilate, 2014.

[41] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proc. of the ACM Symposium on Cloud Computing*, SOCC '14, pages 9:1–9:14. ACM, 2014.

[42] Ilias Marinos, Robert N.M. Watson, and Mark Handley. Network stack specialization for performance. In *Proc. of ACM SIGCOMM*, SIGCOMM'14, pages 175–186, 2014.

[43] Menage, Paul. cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt, 2004.

[44] Arif Merchant, Mustafa Uysal, Pradeep Padala, Xiaoyun Zhu, Sharad Singhal, and Kang G. Shin. Maestro: quality-of-service in large disk arrays. In *Proc. of International Conference on Autonomic Computing*, ICAC'11, pages 245–254, 2011.

[45] J. Metz, Amber Huffman, Steve Sardella, and Dave Mintrun. The performance impact of NVM Express and NVM Express over Fabrics. http://www.nvmexpress.org/wp-content/uploads/NVMe-Webcast-Slides-20141111-Final.pdf, 2015.

[46] Trond Norbye. Memcached Binary Protocol. https://https://github.com/memcached/memcached/blob/master/protocol_binary.h, 2008.

[47] NVM Express Inc. NVM Express: the optimized PCI Express SSD interface. http://www.nvmexpress.org, 2015.

[48] NVM Express Inc. NVM Express over Fabrics Revision 1.0 . http://www.nvmexpress.org/wp-content/uploads/NVMe_over_Fabrics_1_0_Gold_20160605.pdf, 2016.

[49] Open-iSCSi project. iSCSI tools for Linux. https://github.com/open-iscsi/open-iscsi, 2016.

[50] Jian Ouyang, Shiding Lin, Jiang Song, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: software-defined flash for web-scale internet storage systems. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 471–484, 2014.

[51] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Trans. Netw.*, 1(3):344–357, June 1993.

[52] Stan Park and Kai Shen. FIOS: a fair, efficient flash I/O scheduler. In *Proc. of USENIX File and Storage Technologies*, FAST'12, page 13, 2012.

[53] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proc. of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 342–355. ACM, 2015.

[54] Niels Provos and Nick Mathewson. libeventan event notification library. http://libevent.org, 2016.

[55] Samsung Electronics Co. Samsung PM1725 NVMe PCIe SSD. http://www.samsung.com/semiconductor/global/file/insight/2015/11/pm1725-ProdOverview-2015-0.pdf, 2015.

[56] R. Sandberg. Design and implementation of the Sun network filesystem. In *In Proc. of USENIX Summer Conference.*, pages 119–130. 1985.

[57] Satran, et al. Internet Small Computer Systems Interface (iSCSI). https://www.ietf.org/rfc/rfc3720.txt, 2004.

[58] Kai Shen and Stan Park. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *Proc. of USENIX Annual Technical Conference*, ATC'13, pages 67–78. USENIX, 2013.

[59] Prashant J. Shenoy and Harrick M. Vin. Cello: A disk scheduling framework for next generation operating systems. Technical report, Austin, TX, USA, 1998.

[60] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *Proc. of the Conference on Applications, Technologies, Architectures, and Protocols for*

Computer Communication, SIGCOMM '95, pages 231–242. ACM, 1995.

[61] David Shue and Michael J. Freedman. From application requests to virtual IOPs: provisioned key-value storage with Libra. In *Proc. of European Conference on Computer Systems*, EuroSys'14, pages 17:1–17:14, 2014.

[62] David Shue, Michael J. Freedman, and Anees Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proc. of USENIX Operating Systems Design and Implementation*, OSDI'12, pages 349–362, 2012.

[63] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *Proc. of IEEE Mass Storage Systems and Technologies*, MSST '10, pages 1–10. IEEE Computer Society, 2010.

[64] Solarflare Communications Inc. OpenOnload. `http://www.openonload.org/`, 2013.

[65] Ioan Stefanovici, Bianca Schroeder, Greg O'Shea, and Eno Thereska. sRoute: Treating the storage stack like a network. In *Proc. of USENIX Conference on File and Storage Technologies*, FAST '16, pages 197–212, Santa Clara, CA, 2016.

[66] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. IOFlow: A software-defined storage architecture. In *Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196. ACM, 2013.

[67] Cheng-Chun Tu, Chao-tang Lee, and Tzi-cker Chiueh. Secure I/O device sharing among virtual machines on multiple hosts. In *Proc. of International Symposium on Computer Architecture*, ISCA '13, pages 108–119. ACM, 2013.

[68] Paolo Valente and Fabio Checconi. High throughput disk scheduling with fair bandwidth distribution. *IEEE Trans. Computers*, 59:1172–1186, 2010.

[69] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance insulation for shared storage servers. In *Proc. of USENIX File and Storage Technologies*, FAST '07, pages 5–5, 2007.

[70] Andrew Wang, Shivaram Venkataraman, Sara Alspaugh, Randy Katz, and Ion Stoica. Cake: Enabling high-level SLOs on shared storage systems. In *Proc. of ACM Symposium on Cloud Computing*, SoCC '12, pages 14:1–14:14. ACM, 2012.

[71] Theodore M. Wong, Richard A. Golding, Caixue Lin, and Ralph A. Becker-Szendy. Zygaria: Storage performance as a managed resource. In *Proc. of IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '06, pages 125–134. IEEE Computer Society, 2006.

[72] Joel Wu and Scott A. Brandt. The design and implementation of aqua: an adaptive quality of service aware object-based storage device. In *Proc. of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 209–218, May 2006.

[73] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. Storage performance virtualization via throughput and latency control. *Trans. Storage*, 2(3):283–308, August 2006.

[74] Yiying Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *FAST*, page 1, 2012.

[75] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity SSDs. In *Proc of USENIX Conference on File and Storage Technologies*, FAST '15, pages 45–58, 2015.

[76] Timothy Zhu, Alexey Tumanov, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. Prioritymeister: Tail latency QoS for shared networked storage. In *Proc. of ACM Symposium on Cloud Computing*, SOCC '14, pages 29:1–29:14. ACM, 2014.