ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Rearchitecting Linux Storage Stack for μs Latency and High Throughput

Jaehyun Hwang
Cornell University

Midhul Vuppalapati
Cornell University

Simon Peter
UT Austin

Rachit Agarwal
Cornell University

Abstract

This paper demonstrates that it is possible to achieve μs-scale latency using Linux kernel storage stack, even when tens of latency-sensitive applications compete for host resources with throughput-bound applications that perform read/write operations at throughput close to hardware capacity. Furthermore, such performance can be achieved without any modification in applications, network hardware, kernel CPU schedulers and/or kernel network stack.

We demonstrate the above using design, implementation and evaluation of `blk-switch`, a new Linux kernel storage stack architecture. The key insight in `blk-switch` is that Linux's multi-queue storage design, along with multi-queue network and storage hardware, makes the storage stack conceptually similar to a network switch. `blk-switch` uses this insight to adapt techniques from the computer networking literature (*e.g.*, multiple egress queues, prioritized processing of individual requests, load balancing, and switch scheduling) to the Linux kernel storage stack.

`blk-switch` evaluation over a variety of scenarios shows that it consistently achieves μs-scale average and tail latency (at both 99th and 99.9th percentiles), while allowing applications to near-perfectly utilize the hardware capacity.

1 Introduction

There is a widespread belief in the community that it is not possible to achieve μs-scale tail latency when using the Linux kernel stack. A frequently cited argument is that, due to its high CPU overheads, Linux is struggling to keep up with recent $10-100\times$ performance improvements in storage and network hardware [17, 38]; the largely stagnant server CPU capacity further adds to this argument. In addition, many in the community argue that the resource multiplexing principle is so firmly entrenched in Linux that its performance stumbles in the common case of multi-tenant deployments [22, 38, 42]—when latency-sensitive `L`-apps compete for host compute and network resources with throughput-bound `T`-apps, Linux fails to provide μs-scale tail latencies. These arguments reflect a broad belief that, despite Linux's great success, it has emerged as the core bottleneck for modern applications and hardware.

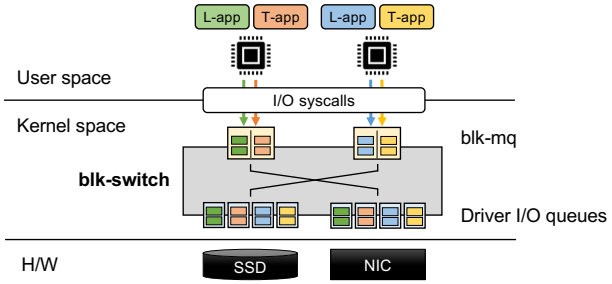


Figure 1: **The key insight in `blk-switch` design: Linux's per-core block layer design, along with modern multi-queue storage and network hardware, makes the storage stack conceptually similar to a network switch.**

This paper focuses on storage stacks used by applications to access data on local and/or remote servers. We show that it is possible to achieve μs-scale tail latency using Linux, even when applications perform read/write operations at throughput close to hardware capacity. Furthermore, low latency and high throughput can be simultaneously maintained even when tens of `L`-apps and `T`-apps compete for host resources at each of the compute, storage and network layers of the kernel stack. Finally, such performance can be achieved without any modifications in applications, network and storage hardware, kernel CPU schedulers and/or kernel network stack; all that is needed is to rearchitect the Linux storage stack.

`blk-switch` is a new Linux storage stack architecture for μs-scale applications. The key insight in `blk-switch` is that Linux's per-core block layer queues [19, 27], combined with modern multi-queue storage and network hardware [8], makes the storage stack conceptually similar to network switches (Figure 1). Building upon this insight, `blk-switch` adapts classical techniques from the computer networking literature (*e.g.*, multiple egress queues, prioritized processing of individual requests, load balancing along multiple network connections, and switch scheduling) to the Linux storage stack.

To realize the above insight, blk-switch introduces a per-core, multi-egress queue block layer architecture for the Linux storage stack (Figure 1). Applications use standard Linux APIs to specify their performance goals, and to submit read/write requests (§3). Underneath, for each application class, blk-switch creates an "egress" queue on a per-core basis that is mapped to a unique queue of the underlying device driver (that is, storage driver for local storage access, or remote storage driver for remote storage access). Such a multi-egress queue design allows blk-switch to decouple ingress (application-side block layer) queues from egress (device-side driver) queues since requests submitted at an ingress queue on a core can now be processed at an egress queue at any of the cores. blk-switch merely acts like a "switch"—at each individual core, blk-switch steers requests submitted at the ingress queue of that core to one of the egress queues, based on application performance goals and load across cores.

blk-switch integrates three ideas within such a switched architecture to simultaneously enable μs-scale tail latency for L-apps and high throughput for T-apps. First, blk-switch maps requests from L-apps to the egress queue on that core, and processes the outstanding requests in a prioritized order; that is, at each individual core, requests in L-app egress queues are processed before requests in T-app egress queues. This ensures that L-apps observe minimal latency inflation due to head-of-line blocking from T-app requests. However, strict prioritization at each core can lead to starvation of T-apps due to transient load (bursts of requests from an L-app on the same core) or due to persistent load (multiple contending L-apps on the same core). To avoid starvation during transient loads, blk-switch exploits the insight that decoupling the application-side queues from device-side queues, and interconnecting them via a switched architecture enables efficient realization of different load balancing strategies, even at the granularity on individual application requests. blk-switch thus uses request steering to load balance requests from T-apps across corresponding egress queues at all available cores. To avoid starvation due to persistent loads, blk-switch uses application steering, that steers application threads across available core at coarse-grained timescales with the goal of minimizing persistent contention between L-apps and T-apps. At its very core, the two steering mechanisms in blk-switch highlight the conceptual idea that load balancing within the Linux storage stack can be applied at two levels of abstraction: individual requests and individual threads; and, both of these are necessary to simultaneously achieve μs-scale latency for L-apps and high throughput for T-apps—the former enables efficient handling of transient loads, and the latter enables efficient handling of persistent loads on individual cores.

We have implemented blk-switch within the Linux storage stack. Our implementation is available at: https://github.com/resource-disaggregation/blk-switch. We evaluate blk-switch over a wide variety of settings and workloads, including in-memory and on-disk storage, single-threaded and multi-threaded applications, varying load induced by L-apps and T-apps, varying read/write ratios, varying number of cores, and with RocksDB [9], a widely-deployed storage system. Across all evaluated scenarios (except for sensitivity analysis against number of cores and T-app load), we find that blk-switch achieves μs-scale average and tail latency (at both 99th and 99.9th percentiles, or P99 and P99.9, respectively), while allowing applications to nearly saturate the 100Gbps link capacity, even when tens of applications contend for host resources. In comparison to Linux, blk-switch improves the average and the P99 latency by as much as 130× and 24×, respectively, while maintaining $84-100\%$ of Linux's throughput. We also compare blk-switch to SPDK, a widely-deployed state-of-the-art userspace storage stack. We find that SPDK achieves good tail latency and high throughput when each application runs on a dedicated core; in the more realistic scenario of applications sharing server cores, in comparison to SPDK, blk-switch improves the average and P99 tail latency by as much as 12× and 18×, respectively, while achieving comparable or higher throughput; as we will discuss, this is because polling-based userspace stacks like SPDK do not interpolate very well with Linux kernel CPU schedulers. When compared to both Linux and SPDK, blk-switch achieves similar or higher improvements for P99.9 tail latency. All these benefits of blk-switch can be achieved without any modifications in the applications, Linux CPU scheduler (blk-switch uses the default CFS scheduler), Linux network stack (blk-switch uses Linux kernel TCP/IP stack), and/or network hardware.

2 Understanding Existing Storage Stacks

This section presents a deep dive into the performance of two state-of-the-art storage stacks—Linux (including remote storage stack [29]) and SPDK (a widely-deployed userspace stack). We first describe our setup (§2.1), and then discuss several results and insights (§2.2). Our key findings are:

- Despite significant efforts in improvement of Linux storage stack performance (per-core queues [19], per-core storage and network processing [29], etc.), existing Linux-based solutions suffer from high tail latencies due to head-of-line blocking, especially in increasingly common multi-tenant deployments [31, 52], that is, when L-apps compete for host resources with T-apps that perform high-throughput reads/writes to remote storage servers. Intuitively, such scenarios result in a complex interference at three layers of the stack—compute, storage, and network—requiring careful orchestration of host resources to achieve μs-scale tail latency, while sustaining throughput close to hardware capacity. Existing Linux-based solutions fail to efficiently achieve such orchestration. For instance, even with one L-app competing with one T-app, we observe tail latency inflation of as much as 7× (when compared to isolated case, where the L-app runs on a dedicated server).

Table 1: **The storage stack, network stack and CPU scheduler used in the evaluated systems.**

System	Storage stack	Network stack	CPU scheduler
Linux	kernel, i10 [29]	kernel TCP	kernel CFS
SPDK	userspace	kernel TCP	kernel CFS

- Polling-based storage stacks (*e.g.*, SPDK) can achieve low latency and high throughput when each application is given a dedicated core. However, when multiple applications share a core, polling-based stacks that use kernel CPU schedulers suffer from undesirable interactions between the storage stack and the kernel CPU scheduler. Even when one L-app shares a core with one T-app, we observe 5× tail latency inflation *and* 2.4× throughput degradation, when compared to the respective isolated cases; both of these get worse with increasing number of applications sharing a core (108× tail latency inflation and 6.2× throughput degradation for the case of four L-apps sharing a core with one T-app). Prioritizing L-apps does not help—while tail latency inflation can be avoided, throughput for T-apps drops to near-zero with just one L-app.

2.1 Measurement Setup

The storage stack, the network stack and the CPU schedulers used in evaluated systems are summarized in Table 1. Linux uses block multi-queue design with per-core software queues mapped to underlying device driver queues (NVMe driver for local storage access, and i10 [29] queues for remote storage access). SPDK is a polling-based system, where applications poll on their I/O queues (for local storage access) and/or on their respective TCP sockets (for remote storage access); underneath, SPDK uses its own driver for accessing remote storage devices over TCP.

In §5, we evaluate these systems over different storage devices, workloads, and experimental setups. This section focuses on a specific setting: a single-core setup where one T-app contends with an increasing number of L-apps to execute read requests on remote in-memory storage connected via a 100Gbps link. This setting allows us to both hide high NVMe SSD access latencies, and dive deeper into factors contributing to individual system performance. Latency-sensitive L-apps generate 4KB requests and throughput-bound T-apps generate large requests; to ensure a fair comparison, for each individual system, we set the "ideal" load and request size for T-apps using the knee point on the latency-throughput curve for that system (see discussion in §5.1 for more details, including information about network and storage hardware).

We measure average and P99 tail latency for L-apps and throughput-per-core for T-apps for both isolated (where each application has host resources to itself) and shared scenarios (where all applications share host resources). An ideal system would maintain the isolated-latency for L-apps, with minimal impact on isolated throughput for T-apps.

2.2 Existing Storage Stacks: Low latency *or* high throughput, but not both

We start by discussing the isolated performance for each of the systems (shown in Figure 2 in the leftmost bars). Here, Linux achieves P99 tail latency of 118μs and throughput-per-core of 26Gbps; when compared to Linux, SPDK achieves 5× lower latency, and 1.5× higher throughput. While these results are not surprising in comparison, some interesting numbers stand out in an absolute sense. In particular, the absolute numbers for Linux—118μs P99 tail latency (comparable to our NVMe SSD access latency) and $>$ 25Gbps throughput-per-core—may be surprising. We attribute these to several relatively recent optimizations in the Linux storage stack (*e.g.*, blk-mq [19] and CPU-efficient remote storage stacks [29]).

High tail latencies due to lack of prioritization: head-of-line (HoL) blocking. In early incarnations of Linux storage stacks, requests submitted at all cores were processed at a single queue, resulting in contention across cores as well as HoL blocking due to requests submitted across cores. Today's Linux alleviates these issues using per-core block layer queues [19]; however, we find that HoL blocking can still happen at the block layer queues (rare) or at the device driver queues (more prominent). This is because the Linux storage stack [19, 29] uses a single per-core non-preemptive thread to process all requests submitted on that core. When multiple applications contend on a core, this results in high tail latencies for L-apps due to HoL blocking caused by large requests from T-apps; we observe as much as 7× higher latencies in Figure 2. Figure 3(a) shows that, as expected, the impact of HoL blocking increases linearly with the request size of T-apps.

High tail latencies due to lack of prioritization: fair CPU scheduling. We find that polling-based designs do not interplay well with the default kernel CPU scheduler—Completely Fair Scheduler (CFS)—that allocates CPU resources equally across applications sharing the core (albeit, at coarse-grained millisecond timescales, referred to as "timeslices"). Polling completely utilizes the core; thus, the scheduler deallocates the core from an application only when the application has used its share of the core. As a result, requests from L-apps initiated at the boundary of L-app timeslices are the ones whose latency is impacted the worst since these would not be processed until the application's next timeslice. As a result, even when one L-app contends with one T-app, SPDK suffers from 5× inflation in tail latency when compared to the isolated case; the latency inflation increases to 108× and higher when multiple L-apps share the core with a T-app. Since CPU is fairly shared across contending applications, such polling-based systems not only suffer from latency inflation but also from degraded throughput for T-apps proportional to the number of applications contending at the core.

The impact on tail latency depends on two factors: (1) length of individual timeslices; and (2) the time gap between successive timeslices. The former determines the number of

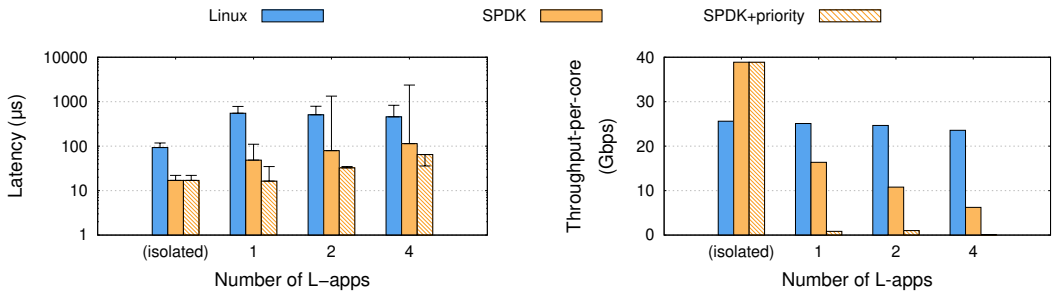


Figure 2: **When each application runs in isolation (isolated case, with no other applications on the server), existing storage stacks can achieve low latency *and* high throughput. However, when multiple applications compete for host resources, performance of existing storage stacks stumbles**—they are either unable to maintail μs-scale latency (Linux and SPDK), or are unable to maintain high throughput (SPDK+priority). With increasing number of L-apps contending with the T-app, performance degrades further. See §2.2 for discussion.

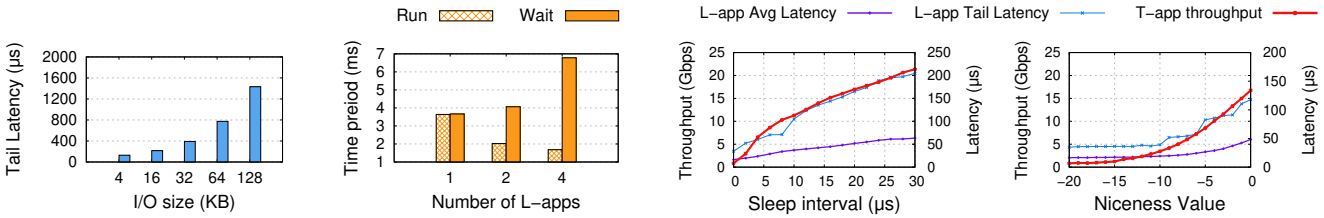


Figure 3: **Root cause for the trends in Figure 2. (left–right) (a): Linux** suffers from high tail latency due to HoL blocking. **(b): SPDK** suffers from high tail latency and low throughput since CPU scheduler performs fair scheduling of CPU resources, resulting in increasingly higher waiting times and increasingly lower runtimes for each application. **(c, d): SPDK+priority** suffers from complete starvation of T-app; increasing the sleep interval and/or niceness value of L-apps leads to an increase in T-app throughput, at the cost of increased average and tail latency for L-apps. Detailed discussion in §2.2.

requests that can be processed within a single timeslice (these requests will achieve near-optimal latency), and the latter determines the amount of "waiting time" for requests that could not be processed within the timeslice in which they were submitted. We measure these two factors in our experiments by examining CFS scheduler traces. In Figure 3(b), the "Run" bar shows the average length of the timeslice given to each L-app, and the "Wait" bar shows the average time gap between consecutive timeslices of each L-app. We observe that as the number of L-apps increases, the length of individual L-app timeslices decreases, and the wait time increases. This leads to (1) a larger latency impact for requests at the boundary of timeslices, hence inflation in tail latency; and, (2) a larger fraction of requests being impacted by the gap between consecutive timeslices, hence inflation in average latency.

Near-zero throughput due to strict prioritization: starvation in polling-based designs. Linux CPU scheduler allows prioritization of L-apps. Unfortunately, polling-based designs do not interplay well with prioritization either. We rerun SPDK results above but with L-apps having higher priority (niceness value -20) than T-app. The corresponding results, referred to as "SPDK+priority" in figures, show that such prioritization results in two undesirable effects: (1) complete starvation of T-apps—since L-apps have higher priority and are always active due to their polling-based design, the scheduler does not preempt these applications; and (2) if more than a single L-app contend on a core, CPU resources are shared fairly across these applications, resulting in increased average latency. We note that tail latency is not impacted much when the number of L-apps is increased. This is because, when given higher priority, L-apps get longer timeslices, and are able to process more requests in each timeslice, leaving only a small fraction of requests to be impacted by the gap between consecutive timeslices. Hence, while the waiting time between timeslices increases, the effect is not visible at P99 (higher percentiles see significant inflation). This is also the reason for the case of four L-apps in Fig. 2: the tail latency is worse than the average (since the latency distribution is extremely skewed towards higher percentiles).

In Figure 3(c), we re-run the single L-app and T-app case, this time making the L-app sleep for a certain interval after submitting requests, and vary this interval. When the L-app sleeps, it yields, allowing the T-app to get scheduled. As can be seen, increasing the sleep interval leads to an increase in T-app throughput. However, it comes at the cost of increasing tail latency for L-apps. In Figure 3(d), we repeat the single L-app and single T-app experiment, but with varying the L-app priority by adapting the niceness value (lower niceness implies higher priority): T-app's niceness value is set to 0, and we vary L-app niceness value from -20 (highest priority) to 0. CFS allocates timeslices to processes based on the niceness value. Hence, with increasing niceness values, the L-app gets a smaller share of CPU cycles, leading to an increase in the T-app's share. As a result, T-app's throughput increases but only at the cost of inflated latency for the L-app.

3 blk-switch Design

As mentioned earlier, blk-switch builds upon the insight that Linux's per-core block layer queues [19, 27], combined with modern multi-queue storage and network hardware [8], makes the storage stack conceptually similar to network switches. To realize the above insight, blk-switch introduces a "switched" architecture for the Linux storage stack that allows requests submitted by an application to be steered to and processed at any core in the system. In §3.1, we describe this switched architecture, and how it enables the key technique in blk-switch to achieve low latency for L-apps—prioritized processing of individual requests. In §3.2 and in §3.3, we describe how decoupling the application-side queues from device-side queues, and interconnecting them via blk-switch's switched architecture enables efficient realization of different load balancing strategies to achieve high throughput for T-apps.

Before diving deeper into blk-switch design details, we make two important notes. First, we describe blk-switch design using a single target device (local and/or remote storage server) since, similar to Linux, blk-switch treats each target device completely independently. Second, blk-switch does not require modifications in applications and/or system interface—applications submit I/O requests to the kernel via standard APIs such as io_submit(). Similar to any other system that provides differential service, blk-switch must identify application goals. Being within the Linux kernel makes this task easy for blk-switch: it uses the standard Linux ionice interface [6] that allows setting a "scheduling class" for individual applications/processes (without any changes in applications and/or kernel request submission interface). In the current implementation (§4), blk-switch uses two of the ionice classes to differentiate L-apps from T-apps. It is easy to extend blk-switch to support additional application requirements—for instance, applications that require both low latency and high throughput can use an additional application class (using ionice) to specify their performance goal, and blk-switch can be extended in a manner that each core not only appropriately prioritizes but also performs load balancing for requests for such applications. In addition, the ionice interface also allows applications to dynamically change their class, if performance goals change over time (*e.g.*, from latency-sensitive to throughput-sensitive requests). Note that ionice is only for the storage stack interpretation, and is different from CPU scheduling priority classes.

3.1 Block Layer is the New Switch

Linux storage stack architecture, in particular the block layer, has evolved over time. In early incarnations of Linux storage stacks, requests submitted at all cores were processed at a single queue. In today's Linux, block layer uses a per-core queue (blk-mq [19]) where requests submitted by all applications running on that core are processed. We refer to these per-core block layer queues as *ingress* queues. Today, these ingress queues are directly mapped to the driver queues (storage device driver for local storage access, or remote storage driver [21, 29] for remote storage access)[1]. Introduction of per-core ingress queues in Linux storage stack resolved contention across cores; however, since all requests submitted to an ingress queue are processed at the same core, it can lead to high tail latency due to head-of-line blocking at the driver queues when L-apps and T-apps submit requests to the same ingress queue (Figure 2). blk-switch's architecture avoids this using a multi-egress queue design, that we describe next.

Multiple egress queues. blk-switch introduces a per-core, multi-egress queue block layer architecture for the Linux storage stack. For each class of application running on the server, blk-switch creates an "egress" queue on a per-core basis. Each of these egress queues is mapped to a unique queue of the underlying device driver—storage driver for local storage access, and remote storage driver [29] with a dedicated network connection for remote storage access. blk-switch assigns a dedicated kernel thread for processing each individual egress queue and assigns priorities to these threads based on application performance goals. For instance, in the case of L-apps and T-apps, blk-switch assigns highest priority to the thread processing L-app requests (both in transmit and receive queues); thus, at each individual core, the kernel CPU scheduler will prioritize the processing of L-app requests over T-app requests, immediately preempting the T-app request processing thread. As a result, the latency inflation observed by L-app requests over the isolated case is minimal: in addition to the necessary overhead of a context switch, the only source of latency is other L-app requests on that core.

Decoupling request processing from application cores. Existing block layer multi-queue design tightly couples request processing to the core where the application submits the request. While efficient when cores are underutilized, such a design could result in suboptimal core utilization: if a core C0 is overloaded and another core C1 is underloaded, current storage stacks do not utilize C1 cycles to process requests submitted at C0.

blk-switch exploits its multi-egress queue design to enable a switched architecture that alleviates this limitation (Figure 4): it allows requests submitted at a core to be steered from the ingress queue of that core to any of the other cores' egress queues (for that application class), be processed on that core, and responses returned on that core to be rerouted back to the appropriate application core. Decoupling request processing from application cores has some overheads (both in terms of latency and CPU), but allows blk-switch to ef-

[1] Modern storage devices have multiple hardware queues and corresponding drivers allow creating a large number of queues (*e.g.*, NVMe standard allows creating as many as 64k queues); in case of multiple hardware devices, each device has its own set of queues. Similarly, modern remote storage stacks [29] also create one driver queue per-core for each remote server.

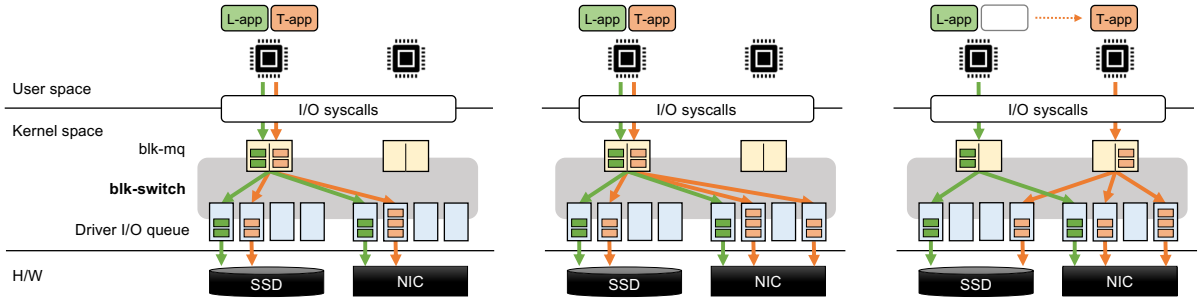


Figure 4: **An illustration of `blk-switch`'s design. (left)** multi-egress queue architecture: the first two and the last two queues on each device are for the left and the right core, respectively (one for each application class); **(center)** request steering mechanism: upon transient congestion on left core NIC queue, requests are steered on the queue corresponding to the right core ; **(right)** application steering mechanism: upon persistent congestion on the left core NIC queue, T-app is steered to the right core. See §3 for discussion.

ficiently utilize all cores in the system. For instance, in the case of L-apps and T-apps, `blk-switch` can steer requests to and process them at lightly-loaded cores, improving throughput for T-apps. Moreover, among requests processed on each core, `blk-switch` continues to provide isolation: prioritized processing of requests in L-app egress queues ensures that requests from L-apps are always prioritized over other requests.

We will discuss in §4 how existing block layer infrastructure (*e.g.*, `bio` and `request` data structures) enable efficient implementation of such a switched architecture with minimal modifications. The rest of the section describes `blk-switch` mechanisms to efficiently exploit this switched architecture to achieve high throughput for T-apps.

3.2 Request Steering

Decoupling processing of individual requests from application cores via `blk-switch`'s switched architecture enables efficient realization of different load balancing strategies. In this subsection, we describe one such strategy used in `blk-switch` for efficiently handling transient loads on individual cores—request steering.

Transient loads can result in temporarily starving T-app requests, *e.g.*, when a burst of (high-priority) L-app requests end up temporarily consuming all CPU cycles on a core, or when multiple L-apps on a core end up generating requests at the same time, or when large requests from one T-app block requests from other T-apps on that core to be processed, etc. Under such transient loads, `blk-switch` uses request steering to load balance the load on the system across the available cores—it steers T-app requests at ingress queues of transiently overloaded cores to the corresponding egress queues on other cores at the granularity of individual T-app requests. Importantly, `blk-switch` performs request steering only for throughput-bound applications. Request steering incurs some overheads (*e.g.*, latency due to reduced data locality, and CPU overheads due to request steering processing and due to contention among cores for accessing the same egress queue), but it is a good tradeoff to make for T-apps: during transient loads, `blk-switch` is able to efficiently utilize available CPU cycles at other cores to maintain T-app throughput. Figure 4(center) shows an example.

Algorithm 1 : `blk-switch` request steering framework.

```
request processing on local core (for destination T):
1: if load on local core < threshold then
2:     Move the request to local core's egress queue
3: else
4:     candidates ← cores with egress queue to T
5:     for each core ∈ candidates do
6:         if load on the core > threshold then
7:             remove core from candidates
8:     Randomly pick two cores in candidates
9:     Move the request to the core with smaller load
```

Making request steering decisions requires an estimate of instantaneous load on individual cores in the system. For T-apps where I/O is the main bottleneck, `blk-switch`'s multi-egress queue design enables an efficient approach—using the instantaneous sum of bytes of outstanding requests for the T-app egress queue to determine instantaneous per-core load and to steer requests to lightly-loaded cores. For such applications, instantaneous sum of bytes of outstanding requests is a good indicator of the presence of congestion in the end-to-end datapath, as congestion at any point will eventually build up the amount of bytes of outstanding requests in T-app egress queues. In our implementation for T-app requests that perform data access to remote storage servers, we use a default threshold of 16×64KB based on the latency-throughput curve for T-apps [30]. However, without any additional mechanisms, such an approach could lead to imperfect request steering decisions since it does not take into account the many other important factors (*e.g.*, queueing delay, request type being read/write, compute-I/O ratios, etc.); there is a large body of research on estimating load on the cores [15, 22, 46], and any of these mechanisms can be incorporated within `blk-switch` decision making.

Algorithm 1 shows a general framework for `blk-switch`'s request steering mechanism. `blk-switch` performs request

steering at the granularity of individual requests. Upon a request submission, `blk-switch` first checks if the local core is available: if the load on the local core is less than `threshold`, the local core is considered available and the request is enqueued in its egress queue. This is to ensure that `blk-switch` only incurs the overhead of request steering when necessary. If the local core is overloaded, `blk-switch` uses a mechanism based on power-of-two choices [41] to select a core to steer the request to. Among egress queues to the same destination (as described in §3.1), it randomly chooses two of these cores, and steers the request to the core with the lower load. The power-of-two choices is efficient as (1) at most two egress ports need to be examined when the local core is overloaded; and (2) it reduces contention between cores on queues since two cores are unlikely to write requests to the same core at the same time.

We provide details about `blk-switch`'s request steering implementation in §4. `blk-switch` does not implement request steering at the remote storage server side; if there is transient congestion at the remote storage server, then corresponding egress queues at the application side will build up. In that case, our application-side request steering algorithm will not pick this egress queue, and will forward the requests to queues at other cores. Thus, application-side request steering alone is enough to deal with transient congestion at both the application and the remote storage server.

3.3 Application Steering

The benefits of request steering at a per-request granularity can be overshadowed if each request submitted at a core has to be steered to other cores, *e.g.*, due to persistent load on a core due to multiple contending `L`-apps submitting requests at that core. For instance, if `L`-apps generate requests at low but consistent loads, frequent context switching between `L`-app and `T`-app request processing threads leads to reduced throughput. Similarly, if two high-load `T`-apps are contending on a core, it is better to move one of them to a less utilized core, avoiding long-term overheads of request steering.

To handle such persistent loads, `blk-switch` observes that load balancing within the Linux storage stack can be done at two levels of abstraction: individual requests and individual threads—while the former enables efficient handling of transient loads, the latter enables efficient handling of persistent loads. Thus, under such persistent loads, `blk-switch` performs application steering, that is CPU allocation to individual application threads by steering threads from persistently overloaded cores to one of the underutilized cores. Figure 4 shows an example. `blk-switch` performs application steering at coarse-grained timescales (in our implementation, default is 10 milliseconds) since it is required only for handling persistent loads. Note that application steering is performed at the granularity of individual application threads. Unlike request steering, `blk-switch` implements a version of application steering at both the application side and at the remote storage server; for the latter, it steers threads that perform processing at `blk-switch`'s receive-side egress queues.

Algorithm 2 : `blk-switch` application steering framework.

$\hat{L}_c$: weighted average load induced by `L`-apps at core `c`.
$\hat{T}_c$: weighted average load induced by `T`-apps at core `c`.
$L^\star$: threshold on weighted average load induced by `L`-apps

`L`-apps:

1: `candidates` $\leftarrow$ all cores with $0 < \hat{L}_c < L^\star$
2: $c^\star \leftarrow$ core in `candidates` with minimum $\{\hat{L}_c + \hat{T}_c\}$
3: Move the application to $c^\star$

`T`-apps:

1: `candidates` $\leftarrow$ all cores with $\hat{L}_c$ less than local core
2: $\hat{c} \leftarrow$ core in `candidates` with minimum $\hat{T}_c$
3: Move the application to $\hat{c}$

For application steering, `blk-switch` uses a framework similar to request steering with minor modifications (Algorithm 2). Unlike the request steering framework, `blk-switch`'s application steering explicitly takes into account the weighted average load on the core induced by `L`-apps. This is due to two reasons. First, application steering is performed to reduce long-term contention between `L`-apps and `T`-apps; thus, we want `T`-apps to be steered to the core with low weighted average load induced by `L`-apps (with an additional constraint that the weighted average of `T`-app load on the new core is lower than the current core). Together, this ensures that steering the `T`-app does not increase the number of context switches (the new core has lower `L`-app load), and also that the new core's `T`-app load is lower than that of the current core, thus minimizing contention among `T`-apps. Second, we also want to potentially place multiple `L`-apps on the same core in order to further reduce interference between `L`-apps and `T`-apps—colocating `L`-apps on a core will not negatively impact their performance as long as `L`-apps generate low weighted average load on the core. The second modification is for the case of applications performing data access on remote storage servers: we now use a default threshold of $L^\star = 100$KB to ensure that only a small number of `L`-apps are aggregated on the same core.

4 `blk-switch` Implementation Details

We have implemented `blk-switch` in Linux kernel 5.4. Throughout the implementation, our focus was to reuse existing kernel storage stack infrastructure as much as possible. To that end, our current implementation adds just 928 lines of code—530 in blk-mq layer, 118 at device driver layer, and 280 for target-specific functions at remote storage layer. In this section, we summarize the core components of Linux kernel implementation that `blk-switch` uses, along with some of the interesting `blk-switch` implementation details.

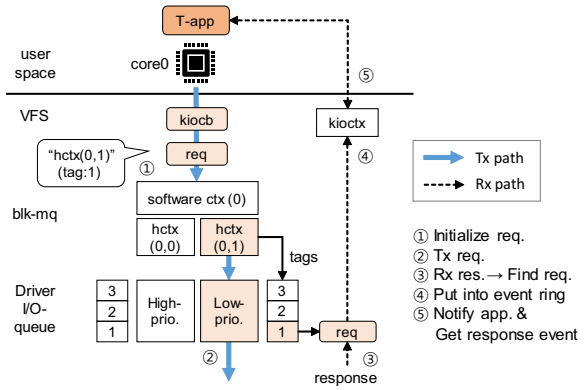
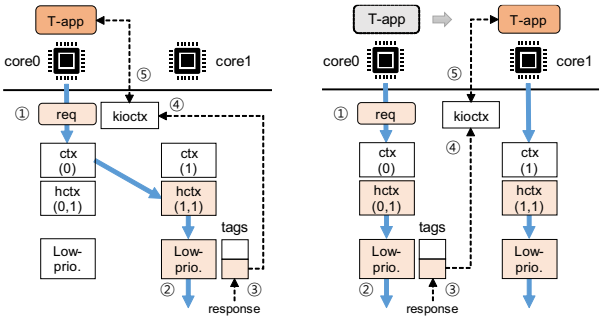


Figure 5: **Request datapath in `blk-switch` for `T-app`.** A request from `T-app` is forwarded to `T-app` egress queue obtaining a tag from that I/O queue. Linux maintains several data structures to enable forwarding back the response to the right application. `blk-switch` uses the same infrastructure.



(a) Request steering (b) Application steering

Figure 6: **Request datapath in `blk-switch`.** (a) **(w/ request steering):** request is steered to the queue on core1 via `hctx`(1,1) acquiring a tag from the steered queue. The response comes back to the steered queue on core1. (b) **(w/ application steering):** When an application is moved from core0 to core1, the *in-flight* request, sent before application steering, comes back on core0. `blk-switch` finds the corresponding `kioctx` via the tag and wakes up the application.

Linux block layer overview. We describe how the Linux storage stack works with the asynchronous I/O interface [4] (see Figure 5, but ignore prioritization). Before creating I/O requests, application needs to setup an I/O context using `io_setup()`, which creates a `kioctx` structure at VFS layer. This kioctx includes (1) a ring buffer where request completion events are enqueued (so that the application pulls them up later asynchronously); and, (2) application process information to wake up the application whenever a new completion event is ready. Each `kioctx` is associated with a context identifier. When application submits a request with the context identifier, the VFS layer creates `kiocb` that represents the I/O request and finds the corresponding `kioctx` using the identifier. `kiocb` has a pointer for the `kioctx`. The block layer creates a `bio` instance, based on `kiocb`, and encapsulates it in a `request` instance: this includes a hardware context (`hctx`) that is associated with one of the device-driver I/O queues. Before forwarding the `request` to the device-driver queue, the block layer needs to get a tag number. `tags` is an array of request pointers, and its size is the same as the queue depth of the driver queue.The block layer maintains a bitmap to keep track of the occupancy of the tags. When all tags are occupied (i.e., the driver queue is full), then the block layer needs to wait for a tag to be available. After getting the tag, the request is sent to the driver queue associated with hctx.

After I/O processing at the device, the response is returned to the kernel with the same tag number. The kernel finds the corresponding `request` instance from the `tags` array using the tag number. The tag number is released, and `kiocb` from the bio instance is extracted to find the `kioctx`. Finally, the completion event is enqueued into the ring buffer of `kioctx` and a notification is sent to the application.

`blk-switch` request steering implementation. Since each `hctx` is regarded as an egress queue, the main goal of the request steering algorithm is to select a non-congested `hctx` across cores if the local one is congested. `blk-switch` maintains the per-core load required for request steering (updating on a per-request basis). After that, the `request` will obtain a tag from the steered `hctx`. Once the `request` is enqueued into the corresponding driver queue, the following driver-level and block-layer receive processing will be done on the core that is associated with the steered `hctx`. When the response comes back to the kernel from the device, we are able to find the steered request instance from the `tags`; thus, going back to the original `kioctx` is straightforward as the `kioctx` can be extracted from the `request` instance (Figure 6(a)). The kernel sends a wake-up signal to the application running on the core associated with the ingress port via the `kioctx`.

`blk-switch` application steering implementation. Upon application steering deciding to move the application to a new core, `blk-switch` invokes the `sched_setaffinity` kernel function to execute the move. Once this is done, requests generated by the steered application will be submitted to the ingress queue on the new core. `blk-switch` maintains the weighted average per-core load required for application steering (updating on a per-request basis). It is easy to maintain application semantics even when there are "in-flight" requests during application moving from one core to another. `blk-switch` forwards the "in-flight" requests to the right application by exploiting the `tags` (Figure 6(b)); similar to the request steering, `blk-switch` is able to find the original `kioctx` that keeps track of the application's location and thus can wake up the associated application. Therefore, the responses can be delivered to the right application.

5 Evaluation

We now evaluate `blk-switch` performance, with the goal of understanding the envelope of workloads where `blk-switch` is able to provide μs-scale average and tail latency, while maintaining high throughput for `T-apps`. To do so, we evaluate

blk-switch across a variety of scenarios and workloads with varying amount of load induced by L-apps and T-apps, number of cores, read/write sizes, read/write ratios and storage settings (in-memory and SSD). In each evaluated scenario, a number of latency-sensitive applications (#L-apps) compete for host resources with a number of throughput-bound applications (#T-apps) that perform large read/write requests on remote storage servers. We describe the individual settings inline.

We describe our evaluation setup in §5.1, followed by a detailed discussion of our results in §5.2 and §5.3. Finally, in §5.4, we provide a detailed breakdown of how each design aspect of blk-switch contributes to its overall performance.

5.1 Evaluation Setup

blk-switch focuses on rearchitecting the storage stack for μs-scale latency and high throughput. Thus, our evaluation setup focuses on scenarios where performance bottlenecks are pushed to the storage stack—that is, where systems are bottlenecked by processing of storage requests, and not by network bandwidth.

Evaluated Systems. We compare blk-switch performance with Linux and widely-deployed userspace storage stack (SPDK) [51] (the CPU scheduler, storage stack and TCP/IP stack used for Linux and SPDK are shown in Table 1). For accessing data in remote servers, we make one modification in Linux: rather than using its native NVMe-over-TCP driver, we use i10 [29], a state-of-the-art Linux-based remote storage stack since it provides much higher throughput (using its default parameters, at the cost of introducing $\sim 50-100\mu$s latency at low loads); for accessing data on remote servers with SPDK, we use its native support for NVMe-over-TCP [13]. We apply core affinity to applications in Linux since that provides best performance. SPDK pins threads to cores by default since it makes use of DPDK's Environment Abstraction Layer (EAL). For both Linux and SPDK, we evenly distribute the applications across cores to the extent possible. For blk-switch, we use its default parameters (§3).

Hardware setup. All our experiments are run on a testbed with two servers directly connected via a 100Gbps link. The servers have a 4-socket NUMA-enabled Intel Xeon Gold 6234 3.3GHz CPU with 8 cores per socket, 384GB RAM and a 1.6TB Samsung PM1735 NVMe SSD. Both servers run Ubuntu 20.04 (kernel 5.4.43). To achieve CPU-efficient network processing for all evaluated systems (since all of them use Linux kernel network stack), we enable TCP Segmentation Offload (TSO), Generic Receive Offload (GRO), packet coalescing using Jumbo frames (9000B), and accelerated Receive Flow Steering (aRFS). To minimize experimental noise, we disable irqbalance and dynamic interrupt moderation (DIM) [10]. Finally, we disable hyper-threading since doing so maximizes performance for all evaluated systems.

We present results for both in-memory storage (RAM block device) and on-disk storage (NVMe SSD). Except for SSD and RocksDB experiments, we use the former due to three reasons. First, unlike on-disk storage, in-memory storage allows us to evaluate scenarios where T-apps generate load close to our network hardware capacity (100Gbps). Second, a single NVMe SSD can be saturated using two cores [29]; in-memory storage, on the other hand, allows us to evaluate scalability of blk-switch (and other systems) with larger number of cores. Finally, our NVMe SSDs have an access latency of $\sim 80\mu$s, which hides a lot of latency benefits of userspace stacks; we find it a fairer comparison to use in-memory storage to hide such high latencies.

Performance metrics. We evaluate system performance in terms of average and tail latency for L-apps, total throughput of all applications, and throughput-per-core calculated as "total throughput / core utilization" (we take the maximum of the application-side and the storage server-side core utilization when computing core utilization). Unless mentioned otherwise, we present results for average latency (shown by bars) and P99 tail latency (shown by top whiskers) since, as we will show, SPDK has significantly worse P99.9 tail latency.

Default workload. To generate loads for L-apps and T-apps, we use the standard methodology, where applications submit storage requests to the underlying system in a closed-loop (that is, the I/O depth of the application specifies a maximum number of outstanding requests). For Linux, we use FIO [16] that uses the lightweight libaio interface. For SPDK, we use its default benchmark application, perf (while FIO has been ported to SPDK, it has higher overheads compared to the lightweight perf application). These benchmarking applications are used to evaluate system performance to again push the bottlenecks to the underlying system (since real-world storage-based applications can have high overheads); nevertheless, we also evaluate blk-switch with RocksDB [9], a prominently used storage system.

L-apps generate 4KB read/write requests with an I/O depth of 1. To ensure that each system is running at its "knee-point" in its latency-throughput curve, we use the optimal T-app operating point for each system—for RAM block device, the optimal (request size, I/O depth) for T-apps is as follows: Linux (64KB, 32), SPDK (128KB, 8), and blk-switch (64KB, 16). While our default setup uses the above request sizes and I/O depths, we also present sensitivity analysis against varying I/O depths and request sizes. Finally, we use the random read workload in our default setup, and also present results for varying read/write ratios.

Unless stated otherwise, we give each system 6 cores on a single NUMA node. We use six cores for each system because we observed that, when given more than 6 cores, Linux ends up being bottlenecked by network bandwidth (that is, it can saturate the 100Gbps link in our testbed) in several of our experimental scenarios. Nevertheless, we also show performance with varying number of cores.

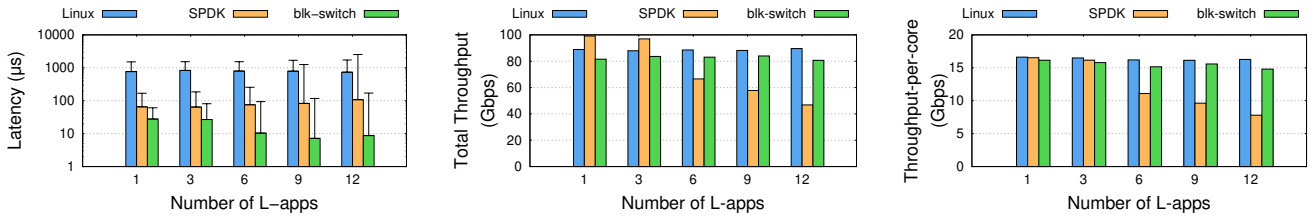


Figure 7: **`blk-switch` achieves μs-scale average and tail latency for `L`-apps and high throughput for `T`-apps even with tens of `L`-apps competing for host compute and network resources with `T`-apps.** As we increase the number of `L`-apps, both Linux and SPDK fail to simultaneously achieve low latency and high throughput for reasons discussed in §2. Linux achieves high throughput, but at the cost of high average and tail latencies; SPDK, on the other hand, suffers from both high tail latency and low throughput. Detailed discussion in §5.2.

5.2 Goal: Low-Latency *and* High-Throughput

Recall that an ideal system would ensure that both `L`-apps and `T`-apps observe performance close to the respective isolated performance (that is, when the application has all the host and storage server resources to itself).

Impact of increasing number of `L`-apps competing for host resources with `T`-apps (Figure 7 and Figure 8). For this experiment, each system is given six cores, and executes requests from six `T`-apps and varying number of `L`-apps.

Linux and SPDK performance trends are similar to Figure 2 in §2. Linux suffers from high average and tail latencies, but maintains high throughput even with increasing number of `L`-apps. SPDK achieves high throughput when number of `L`-apps is less than the number of cores; however, it suffers from inflated latency and degraded throughput with increasing number of `L`-apps (significantly degraded performance with just six `L`-apps). We already discussed the root cause for this behavior for each system in §2; however, for both Linux and SPDK, we observe slightly worse latency and throughput-per-core relative to that observed in Figure 2. Digging deeper, we found that both of these are due to increased L3 cache miss rates. Specifically, since the cores used by the systems are on the same NUMA node, they share a common L3 cache; the resulting increased contention for L3 cache leads to higher cache miss rate—for $x = 1$ in Figure 2, cache miss rates for Linux and SPDK are 1.12% and 3.68%, respectively, but for $x = 6$ in Figure 7, cache miss rates increase to 34% and 63%. Higher cache miss rates lead to an increase in the per-byte CPU overhead for kernel TCP processing (mainly due to data copy), resulting in lower throughput-per-core. Interestingly, for Linux, this also leads to higher latency inflation for `L`-apps (when comparing $x = 6$ in Figure 7 to $x = 1$ in Figure 2), as each `T`-app request takes a larger number of CPU cycles to process, hence exacerbating the effect of HoL blocking. Figure 8 single-threaded case shows the P99.9 tail latency for all systems for the $x = 6$ data point in Figure 7. Both Linux and SPDK exhibit high P99.9 tail latency, but SPDK in particular observes significantly worse P99.9 tail latency ($33\times$ higher than the P99). As discussed in §2, this is because `L`-app requests processed at the boundary of time slices are impacted, and this effect is prominently visible in higher percentiles.

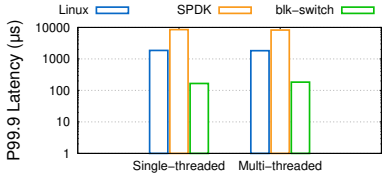


Figure 8: **The P99.9 tail latency corresponding to $x = 6$ in Figure 7 and Figure 9.**

`blk-switch` consistently achieves μs-scale latency for `L`-apps, even with 12 `L`-apps competing for host resources with 6 `T`-apps. In comparison to Linux, `blk-switch` achieves $28-110\times$ better average latency, $10-25\times$ better P99 tail latency and $6-15\times$ better P99.9 tail latency; in comparison to SPDK, `blk-switch` achieves $2-12\times$ better average latency, $2-15\times$ better P99 tail latency and $33-101\times$ better P99.9 tail latency. `blk-switch` achieves all these latency benefits while sacrificing $5-10\%$ throughput relative to Linux. `blk-switch` achieves such performance benefits using a combination of its techniques: it first performs application steering to isolate `L`-apps to a subset of cores, and to distribute `T`-apps over the remaining cores. This results in slightly increased tail latency for `L`-apps compared to a single `L`-app case, but significantly reduces context switching overheads when compared to `L`-apps and `T`-apps sharing individual cores. Further, `blk-switch` performs request steering to utilize unused `L`-app cores for processing `T`-app requests opportunistically. Finally, separation of I/O queues along with prioritization enables maintaining low latency for `L`-apps even when `T`-app requests are steered to the `L`-app cores. Note that prioritization of I/O queue processing also leads to `blk-switch` having slightly better average and tail latencies when compared to the isolated Linux latency in Figure 2; however, this is not fundamental.

We observe a somewhat surprising and counter-intuitive benefit of `blk-switch`'s application steering mechanism that steers `L`-apps onto a small number of cores—for example, in Figure 7, `blk-switch`'s average latency reduces with increasing number of `L`-apps. This is because of better packet aggregation opportunities through TSO/GRO and Jumbo frames: as more `L`-apps are steered on the same core, they begin to

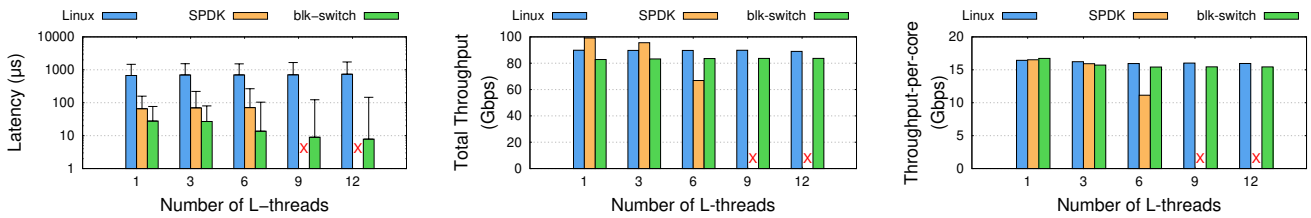


Figure 9: **`blk-switch` achieves μs-scale average and tail latency for `L`-apps and high throughput for `T`-apps even when tens of `L`-app threads compete for host compute and network resources with `T`-apps.** We observe the same trend as in Figure 7 for each system; the only difference is that SPDK does not support more application threads than the number of cores.

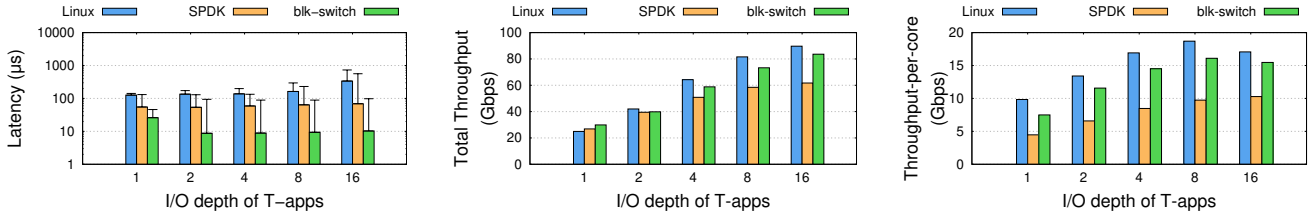


Figure 10: **As the load induced by `T`-apps increases, `blk-switch` continues to achieve low latency and high throughput.** For reasons discussed in §2, Linux and SPDK fail to simultaneously achieve low latency and high throughput: Linux suffers from high latency due to HoL blocking; SPDK experiences increasingly higher latency and lower throughput as the load induced by `T`-apps increase.

share the same egress queue and hence the same underlying TCP connection (recall that `blk-switch` maintains a single per-core egress queue for each application class); as a result, more `L`-app requests can be aggregated, resulting in lower per-request processing overheads, and improved average latency.

Impact of increasing number of `L`-app threads competing for host resources with `T`-app threads (Figure 9 and Figure 8). We now evaluate the performance of existing storage stacks for multi-threaded applications. To do this, we slightly modify the evaluation setup from Figure 7 experiment: we now use one `T`-app with six `T`-threads and one `L`-app with varying number of `L`-threads (varying from 1 to 12). Note that, while the recent SPDK NVMe-oF target implementation supports user-level threads [13], SPDK's `perf` benchmark application running on the host-side does not support user-level threads; as a result, it does not support creating more threads than the number of cores in the system (for each individual application). As one would expect, Figure 9 and Figure 8 results show exactly the same trend as single-threaded applications.

Impact of increasing the load induced by `T`-apps (Figure 10). We now evaluate the performance of each system with varying load induced by `T`-apps. There are two ways to vary the load induced by `T`-apps—by varying I/O depth, and by varying request sizes. Since our setup uses TSO/GRO, these two mechanisms to vary the load induced by `T`-apps lead to essentially the same set of results. We present and discuss results for the former here; the latter can be found in [30]. For this experiment, we fix the number of `L`-apps and `T`-apps to 6 each, and increase the I/O depth for `T`-apps. The request size for `T`-apps is now fixed to 64KB for all systems.

Linux and SPDK show trends similar to previous results. Average and tail latencies for `L`-apps increase with increased contention for host resources (in these results, increased contention is due to higher load induced by `T`-apps). As one would expect, for both of these systems, total throughput and throughput-per-core for `T`-apps increases with an increase in load induced by `T`-apps. `blk-switch` handles contention differently from both of these systems: by prioritizing `L`-app requests, and using request and application steering to efficiently load balance `T`-app requests across unused cores. Thus, `blk-switch` continues to maintain μs-scale latency with increase in `T`-app load—in comparison to Linux, `blk-switch` achieves $5-33\times$ lower average and $2-8\times$ lower tail latency; in comparison to SPDK, `blk-switch` achieves $2-7\times$ lower average and $1.3-6\times$ lower tail latency. `blk-switch`'s mechanisms for handling contention results in a slightly different tradeoff in terms of `T`-app performance. When the load induced by `T`-apps is small, `blk-switch` reduces Linux latency without any degradation in throughput (since it does not pay the overheads of request steering at low loads); at higher loads, `blk-switch` continues to achieve low latency, but observes 10% lower throughput than Linux due to the overheads of request steering.

We note that `blk-switch` average latency improves with load induced by `T`-apps. For smaller loads, `blk-switch`'s application steering does not steer `L`-apps on to a subset of cores (as in previous experiments), leaving `L`-apps evenly distributed across available cores. As a result, `blk-switch` does not get to exploit the benefits of reduced per-request processing overheads (due to TSO/GRO and jumbo frames) associated with aggregating multiple `L`-apps on the same core.

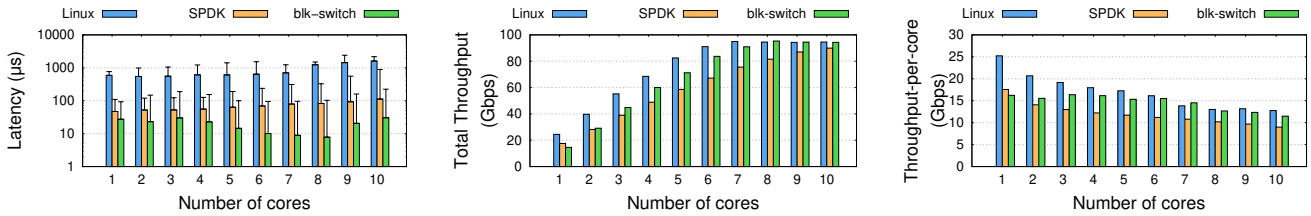


Figure 11: `blk-switch` **maintains its μs-scale average and tail latency for** `L`**-apps with varying number of cores, even when scheduling across NUMA nodes.** For small number of cores, compared to Linux, `blk-switch` trades off improvements in latency with slightly reduced throughput (due to request prioritization, and fewer opportunities for application and request steering). For smaller number of cores, SPDK achieves low latency; as the number of cores are increased, SPDK starts suffering from inflated tail latency and degraded throughput.

Impact of number of cores (Figure 11). We now evaluate the performance of all systems with varying number of cores, including the case when the cores belong to different NUMA nodes. The challenge with doing this evaluation is that, if `T`-apps were to not interfere with `L`-apps, $\sim$4 cores would be sufficient to saturate the network bandwidth (as can be inferred from the isolated case in Figure 2); thus, to understand the performance with increasing number of cores, we have to ensure that `L`-apps and `T`-apps continue to contend at host storage and network processing resources rather than competing for network bandwidth. Thus, we use the following evaluation strategy. Our servers have eight cores on each NUMA node; for each data point up to x = 8 on the x-axis (x = number of cores used for that data point), we use the cores on the same NUMA node and for the last two data points, we use two additional cores from one of the other NUMA nodes. For each data point, we run a total of x `L`-apps and x `T`-apps to ensure that the system is neither lightly-loaded nor overloaded. With this setup, we are able to evaluate for larger number of cores—Linux, `blk-switch` and SPDK now become network bandwidth bottlenecked at $7, 8$ and 10 cores, respectively.

Linux and SPDK performance can be explained using our prior insights. As the number of cores increase, Linux experiences increasingly higher latency but is able to achieve high throughput; SPDK, on the other hand, suffers from increasingly higher latency, and relatively lower throughput.

For the single core case, `blk-switch` improves Linux's latency, but at the cost of 40% lower throughput (similar to SPDK); this is due to lack of request steering and application steering opportunities, and due to prioritization being the dominant mechanism for isolation. As the number of cores increase, `blk-switch` starts exploiting the benefits of request and application steering—it achieves μs-scale latency as in earlier experiments, while getting throughput increasingly closer to Linux (with 7 cores, it is only 4.2% worse than Linux; for 8 or more cores, `blk-switch`'s throughput matches Linux as the network link is saturated). For number of cores between 3 and 8, we see a reduction in `blk-switch`'s average latency due to higher opportunities to exploit the benefits of TSO, GRO and jumbo frames (due to application steering aggregating increasingly more `L`-apps on same subset of cores). Beyond 8 cores, we see slight increase in average and tail latency for `blk-switch` because of NUMA effects.

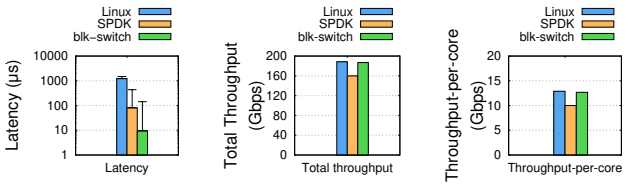


Figure 12: `blk-switch` **is able to maintain low average and tail latencies even when applications operate at throughput close to** 200**Gbps**. The experiment uses 16 `L`-apps and 16 `T`-apps running across 16 cores from two NUMA nodes.

Besides latency results, there are several other interesting observations to be made in Figure 11(center). First, `blk-switch` is able to completely saturate a 100Gbps link using 8 cores, at which point it is bottlenecked by network bandwidth. Since the server has many more cores, we expect that these cores will allow `blk-switch` to maintain its performance with future NICs that have larger bandwidths (we show this for 200Gbps network bandwidth setup below). Second, while the total throughput of `blk-switch` scales well with the number of cores, it has slightly lower total throughput compared to Linux for smaller number of cores. This is due to application steering resulting in `T`-apps being steered away from `L`-apps, and the `L`-apps cores observing transient underutilization when request steering decisions are imperfect. Under such imperfect decisions, fewer number of cores are available for `T`-app request processing. However, as the number of cores increase, the benefits of reduced context switching (due to lower contention between `L`-app and `T`-app requests after application steering) start to offset core underutilization resulting in similar or even higher throughput when compared to other systems. Finally, Figure 11(right) demonstrates that all systems experience reduced throughput-per-core with increasing number of cores. We found that this is due to an increased number of L3 cache misses with increase in total throughput as the number of cores is increased.

Performance beyond 100**Gbps (Figure 12).** We now evaluate the performance of all systems in the Terabit Ethernet

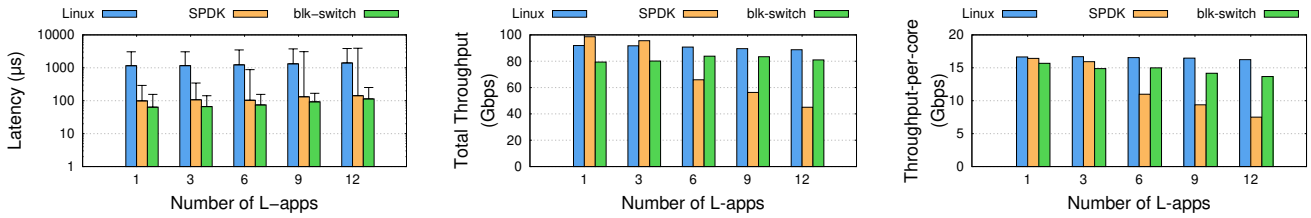


Figure 13: **For experiments with SSDs (corresponding to Figure 7), `blk-switch` latency is largely overshadowed by SSD access latency.** Rest of the trends are similar to those in Figure 7.

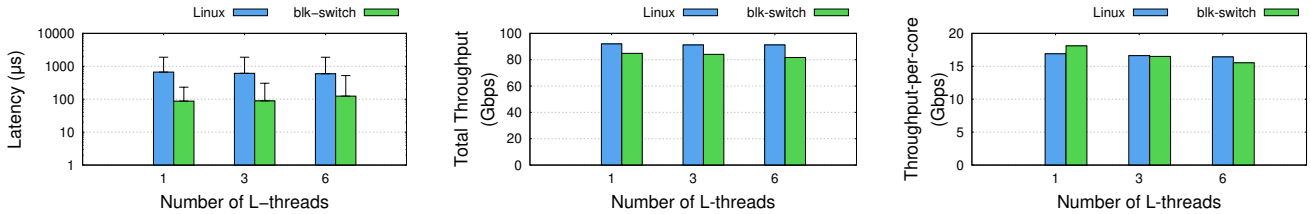


Figure 14: **Evaluation results with RocksDB: `blk-switch` performance benefits over Linux are similar to previous results.**

regime (above 100Gbps). For this we installed an additional NIC on each of the two servers in our setup, and connected these NICs with an additional 100Gbps link, enabling a total of 200Gbps network bandwidth between the servers. The two NICs on each server are attached to separate NUMA nodes. We use all of the cores on both of these NUMA nodes (total of 16), while running 16 L-apps and 16 T-apps. The performance trends remain identical to previous results — `blk-switch` is able to maintain μs-scale average and tail latency (10μs average, 143μs P99 , and 296μs P99.9), while nearly saturating the 200Gbps network bandwidth (within 1% of Linux).

Performance with different storage access latency. We repeat the experiment shown in Figure 7, but with L-app requests being executed on an NVMe SSD (T-app requests are still executed in-memory). The access latency of our SSD ($\sim$80μs) causes increase in average latencies for all systems, but the performance trends among the evaluated systems remain identical to earlier results. Importantly, `blk-switch`'s latency is largely overshadowed by SSD access latency.

Additional results. We present several additional results in [30], including performance with varying request sizes for T-apps, varying read/write ratios, applications that access data distributed between local and remote storage servers, and bursty application workloads.

5.3 RocksDB with `blk-switch`

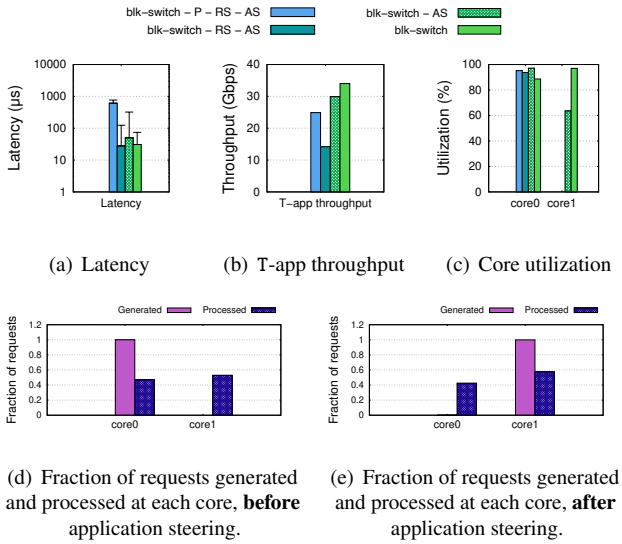
We now evaluate `blk-switch` with RocksDB [9], a widely-deployed storage system, as the L-app. We mount a remote SSD block device at the host-side with XFS file system (only Linux and `blk-switch` support mounting a file system). We setup RocksDB to use the mounted XFS file system backed by remote SSD device and enable direct I/O. To generate workload for RocksDB, we use the `db_bench` benchmarking tool with ReadRandom workload and 4KB request sizes, with an I/O depth of 1 for each thread. We colocate a T-app that accesses remote RAM block device using FIO [16], as before. We run this benchmark on 6 cores, with 6 T-app threads and varying number of L-app threads.

Figure 14 shows that both Linux and `blk-switch` achieve slightly higher latency compared to previous results due to RocksDB's higher application-layer overheads. However, in comparison, `blk-switch` achieves over an order of magnitude latency reduction when compared to Linux, while sacrificing throughput by at most 10%. Furthermore, `blk-switch` maintains these benefits even with increasing number of L-app threads competing for host resources with T-app threads.

5.4 Understanding `blk-switch` Performance

We now quantify the contribution of each of `blk-switch`'s mechanisms to its overall performance. To do so, we run a simple microbenchmark: we start the experiment with one L-app and one T-app on `core0`, and set the I/O depth of T-app to be 32. We then add `blk-switch` mechanisms (prioritization, request steering and application steering) incrementally.

Figure 15 shows that each of `blk-switch`'s mechanism contributes to its overall performance. Enabling prioritization only reduces tail latency by an order of magnitude (Figure 15(a)), but at the cost of lower T-app throughput on `core0` (Figure 15(b)); since request and application steering are disabled, strictly prioritizing processing of L-app requests results in reduced throughput due to larger number of context switches. As shown in Figure 15(c) and Figure 15(d), enabling request steering with prioritization allows the T-app to achieve high T-app throughput by utilizing spare capacity on less loaded cores (by steering T-app requests from heavily loaded `core0` and processing these requests at `core1`); however, this comes at the cost of slight increase in latency for



(a) Latency (b) T-app throughput (c) Core utilization

(d) Fraction of requests generated and processed at each core, **before** application steering.

(e) Fraction of requests generated and processed at each core, **after** application steering.

Figure 15: **Contribution of different techniques in `blk-switch` to its overall performance.** (`blk-switch`-P-RS-AS) is `blk-switch` with all mechanisms disabled; we then cumulatively enable prioritization (`blk-switch`-RS-AS), request steering (`blk-switch`-AS), and application steering (`blk-switch`). See discussion in §5.4.

L-apps (albeit, still μs-scale)—due to non-trivial CPU overheads of request steering and non-real-time prioritization in Linux kernel CPU schedulers, some of the L-app requests get blocked by the thread doing request steering. This problem is alleviated by `blk-switch`'s application steering algorithm (Figure 15(e))—it steers the T-app away from the L-app, allowing `blk-switch` to simultaneously achieve low latency and high throughput.

6 Related Work

We have already compared `blk-switch` with state-of-the-art Linux-based and widely-deployed userspace storage stacks. We now compare and contrast `blk-switch` with other closely-related systems.

Existing storage stacks. There is a large and active body of research on designing storage stacks that target various goals, including fairness [1, 2, 7, 26], deadlines [5, 7], prioritization [3], and even policy-based storage provisioning and management [24, 39, 47, 49]. However, none of these stacks target μs-scale latency. Furthermore, many of them can have high CPU overheads (for high-performance storage devices, the standard recommendation in Linux is to use no scheduler [26]), especially for applications that perform operations on remote storage servers [14, 23, 25, 50]. Recent work on storage stacks for remote data access [12, 29] achieves high CPU efficiency and throughput; however, as we have shown in our evaluation, they fail to achieve low latency in multi-tenant deployments when latency-sensitive and throughput-bound applications compete for host resources.

User-space stacks. We have already performed evaluation against SPDK, a widely-deployed state-of-the-art user-space storage stack. Our evaluation focuses on using SPDK with Linux kernel CPU scheduler and network stack, and highlights the poor interplay with SPDK's polling-based architecture and Linux CPU scheduler. It is possible to overcome some of these limitations by integrating SPDK with high-performance user-space or RDMA-based network stacks [13, 18, 32, 35–37, 40], user-space CPU schedulers [34], or both [22, 42–45]. However, with the exception of [22, 42], these user-space network stacks and CPU schedulers either do not provide μs-scale isolation in multi-tenant deployments, or require dedicated cores for each individual L-app resulting in potentially high core underutilization. The state-of-the-art among these user-space stacks [22, 42] demonstrate that by carefully orchestrating compute resources across L-apps and T-apps, it is possible to simultaneously achieve μs-scale latency and high throughput. However, they currently provide fewer features than Linux and require modifications in applications. `blk-switch` shows that it is possible to simultaneously achieve μs-scale latency and high throughput without any modifications in applications, Linux kernel CPU scheduler and/or network stack.

Hardware-level isolation. There has also been work on achieving performance isolation by exploiting hardware-level mechanisms in NVMe SSDs [20, 28, 48], including mechanism specification in the NVMe standard [11, 33]. These are complementary to `blk-switch`'s goals that focuses on software bottlenecks.

7 Conclusion

Using design, implementation and evaluation of `blk-switch`, this paper demonstrates that it is possible to achieve μs-scale tail latency using Linux, even when tens of latency-sensitive applications compete for host resources with throughput-bound applications that access data at throughput close to hardware capacity. The key insight in `blk-switch` is that Linux's multi-queue storage design, along with multi-queue network and storage hardware, makes the storage stack conceptually similar to a network switch. `blk-switch` uses this connection to adapt techniques from the computer networking literature (*e.g.*, prioritized processing of individual requests, load balancing, and switch scheduling) to the Linux kernel storage stack. `blk-switch` is implemented entirely within the Linux kernel storage stack, and requires no modification in applications, network and storage hardware, kernel CPU schedulers and/or kernel network stack.

Acknowledgments

We would like to thank our shepherd, Adam Belay, and the OSDI reviewers for their insightful feedback. This work was supported in part by NSF 1704742, NSF 1900457, a Google faculty research scholar award, a Sloan fellowship, the Texas Systems Research Consortium, and a grant from Samsung.

References

[1] BFQ (Budget Fair Queueing) — The Linux Kernel documentation. https://www.kernel.org/doc/html/latest/block/bfq-iosched.html.

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

[3] Kyber multiqueue I/O scheduler. https://lwn.net/Articles/720071/.

[4] Linux Asynchronous I/O. https://oxnz.github.io/2016/10/13/linux-aio/.

[5] Linux blk-mq scheduling framework. https://lwn.net/Articles/708465/.

[6] Linux ionice. https://linux.die.net/man/1/ionice.

[7] Linux Kernel/Reference/IOSchedulers - Ubuntu Wiki. https://wiki.ubuntu.com/Kernel/Reference/IOSchedulers.

[8] David S. Miller, Linux Multiqueue Networking. http://vger.kernel.org/~davem/davem_nyc09.pdf, 2009.

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

[10] Mellanox Technologies: Dynamically-Tuned Interrupt Moderation (DIM). https://community.mellanox.com/s/article/dynamically-tuned-interrupt-moderation--dim-x, 2019.

[11] NVM Express 1.4. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf, 2019.

[12] NVM Express over Fabrics 1.1. https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf, 2019.

[13] SPDK User Guides for NVMe over Fabrics. https://spdk.io/doc/nvmf.html, 2020.

[14] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novaković, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei. Remote regions: a simple abstraction for remote memory. In *USENIX ATC*, 2018.

[15] R. Apte, L. Hu, K. Schwan, and A. Ghosh. Look Who's Talking: Discovering Dependencies between Virtual Machines Using CPU Utilization. In *USENIX HotCloud*, 2010.

[16] J. Axboe. Flexible IO Tester (FIO) ver 3.13. https://github.com/axboe/fio, 2019.

[17] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. Attack of the killer microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[18] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX OSDI*, 2014.

[19] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *ACM SYSTOR*, 2013.

[20] M. Bjørling, J. Gonzalez, and P. Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *USENIX FAST*, 2017.

[21] N. Express. NVM Express over Fabrics 1.0 Ratified TPs. https://nvmexpress.org/, 2018.

[22] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *USENIX OSDI*, 2020.

[23] P. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. Network Requirements for Resource Disaggregation. In *USENIX OSDI*, 2016.

[24] R. Gracia-Tinedo, J. Sampé, E. Zamora, M. Sánchez-Artigas, P. García-López, Y. Moatti, and E. Rom. Crystal: Software-Defined Storage for Multi-tenant Object Stores. In *USENIX FAST*, 2017.

[25] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient Memory Disaggregation with Infiniswap. In *USENIX NSDI*, 2017.

[26] M. Hedayati, K. Shen, M. L. Scott, and M. Marty. Multi-Queue Fair Queuing. In *USENIX ATC*, 2019.

[27] C. Hellwig. High Performance Storage with blk-mq and scsi-mq. https://events.static.linuxfound.org/sites/events/files/slides/scsi.pdf.

[28] J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *USENIX FAST*, 2017.

[29] J. Hwang, Q. Cai, A. Tang, and R. Agarwal. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10. In *USENIX NSDI*, 2020.

[30] J. Hwang, M. Vuppalapati, S. Peter, and R. Agarwal. Rearchitecting Linux Storage Stack for μs Latency and High Throughput. https://github.com/resource-disaggregation/blk-switch/techreport, 2021.

[31] C. Iorgulescu, R. Azimi, Y. Kwon, S. Elnikety, M. Syamala, V. N. H. Herodotou, P. Tomita, A. Chen, J. Zhang, and J. Wang. PerfIso: Performance Isolation for Commercial Latency-Sensitive Services. In *USENIX ATC*, 2018.

[32] E. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX NSDI*, 2014.

[33] K. Joshi, K. Yadav, and P. Choudhary. Enabling NVMe WRR support in Linux Block Layer. In *USENIX HotStorage*, 2017.

[34] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μsecond-scale tail latency. In *USENIX NSDI*, 2019.

[35] A. Kalia, M. Kaminsky, and D. Andersen. Datacenter RPCs can be general and fast. In *USENIX NSDI*, 2019.

[36] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. TAS: TCP Acceleration as an OS Service. In *ACM Eurosys*, 2019.

[37] A. Klimovic, H. Litz, and C. Kozyrakis. ReFlex: Remote Flash $\approx$ Local Flash. In *ACM ASPLOS*, 2017.

[38] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *ACM SoCC*, 2014.

[39] J. Mace, P. Bodik, R. Fonseca, and M. Musuvathi. Retro: Targeted Resource Management in Multi-tenant Distributed Systems. In *USENIX NSDI*, 2015.

[40] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kokonov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: a Microkernel Approach to Host Networking. In *ACM SOSP*, 2019.

[41] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst.*, 12(10):1094–1104, Oct. 2001.

[42] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX NSDI*, 2019.

[43] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, 33(4), Nov. 2015.

[44] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *ACM SOSP*, 2017.

[45] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout. Arachne: Core-Aware Thread Management. In *USENIX OSDI*, 2018.

[46] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. CloudScale: Elastic Resource Scaling for Multi-Tenant Cloud Systems. In *ACM SoCC*, 2011.

[47] I. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska. sRoute: Treating the Storage Stack Like a Network. In *USENIX FAST*, 2016.

[48] A. Tavakkol, M. Sadrosadati, S. Ghose, J. Kim, Y. Luo, Y. Wang, N. Mansouri Ghiasi, L. Orosa, J. Gómez-Luna, and O. Mutlu. Flin: Enabling fairness and enhancing performance in modern nvme solid state drives. In *ACM ISCA*, 2018.

[49] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. IOFlow: A Software-Defined Storage Architecture. In *ACM SOSP*, 2013.

[50] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes. Building An Elastic Query Engine on Disaggregated Storage. In *USENIX NSDI*, 2020.

[51] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. SPDK: A development kit to build high performance storage applications. In *IEEE CloudCom*, 2017.

[52] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *ACM Eurosys*, 2013.