Endhost Remote Storage and Network Stack Performance in Contention

Sadman Sakib
University of Wisconsin-Madison

Abstract

Remote storage applications and network applications have different characteristics in terms of resource consumption. This paper studies the performance of storage applications and network applications when they run together. It is observed that scheduling overheads and shared L3 cache misses are the key reasons for performance degradation of both network and storage applications when these run on same or different cores. In addition, head-of-line blocking increases the average latency of latency-sensitive applications when these compete with network applications.

1 Introduction

Disaggregated storage is getting popular in datacenters because of cost efficiency and resource elasticity [6]. It is common in datacenter to have multiple remote storage and network applications running concurrently sharing endhost and network resources. As the network and storage performance have increased manyfold in recent years, corresponding software stacks still have many inefficiencies. Therefore, it is important to identify the performance bottlenecks in different scenarios such as when multiple applications compete for endhost resources, called contention.

NVMe-over-Fabrics allows accessing remote NVMe SSD devices over network fabrics. Currently three types of network transports are supported for carrying NVMe commands and responses- TCP/IP, RDMA and fibre channel. NVMe-over-TCP in Linux uses TCP/IP stack in kernel.

Contention between two network flows can happen when those run on the same core or on different cores. NVMe-over-TCP can be viewed as a special network application with additional storage stack processing. Recently, there have been studies in literature on contention between multiple network apps and contention between multiple storage apps [3] [7]. However, there has not been studies on contention between network and remote storage flows. So, in this paper I ask how the performance is affected when remote storage and network apps run together.

In this study, I find several interesting observations. The effect of storage and network apps on each other is asymmetric. A storage app have much degraded performance when it competes with multiple network apps which does not happen when a network app competes with multiple storage apps. The degradation of the throughput of storage and networking apps is due to scheduling overhead and L3 cache misses. When the throughput of storage apps do not degrade, it incurs higher CPU utilization. Storage applications requires more processing at host compared to target. Latency-sensitive applications have increased latency when competining with network apps because of default CFS scheduling in Linux and head-of-line blocking. When multiple latency-sensitive applications run on a core, a network app on the same core has degraded performance because of getting less CPU cycles.

In the following, section 2 introduces background concepts, section 3 describes measurement setup, section 4 describes the experiments and observations, section 5 describes the course contents used and finally section 6 provides a conclusion.

2 Background

The Linux network stack has different paths for sender-side and receiver-side data transmission. The sender-side initiates a write system call, buffers data in the kernel, processes it via the TCP/IP layer, segments the data, and queues it in the Network Interface Card (NIC) for transmission. Most NICs support hardware offload of packet segmentation (TCP segmentation offload - TSO), which is processed by the driver. Almost all sender-side processing is performed on the same core as the application in modern Linux networks. The receiver-side uses NIC's Rx queues and a per-Rx queue page-pool to allocate memory for Direct Memory Access (DMA). Upon receiving new data, the NIC generates an interrupt request (IRQ), which is processed by a CPU core selected using hardware steering mechanisms. The driver performs polling from then on to reduce the number of IRQs. The driver allocates a socket buffer (skb) for each frame. The network subsystem merges skbs, schedules TCP/IP processing, and appends in-order skbs to the socket's receive queue. [3].

Non-Volatile Memory Express (NVMe) is a protocol designed to leverage the full potential of high-speed storage media like solid-state drives (SSDs), utilizing parallelism in contemporary processors and high-speed connections [1]. It works by directly connecting the storage device to the system's CPU via a PCIe interface, which reduces latency and increases speed by allowing the SSD to operate closer to its peak throughput. NVMe over Fabrics (NVMe-oF) extends the benefits of NVMe across network fabrics. NVMe-over-TCP is a specific transport binding of NVMe-oF that operates over standard TCP/IP networks. This allows NVMe storage devices to be accessed over traditional network infrastructures, broadening the NVMe technology's reach. NVMe/TCP works by encapsulating NVMe commands within TCP packets, which are then sent over the network.

In NVMe, the controller uses pairs of I/O queues: a submission queue and a completion queue. The submission queue is where the host places commands to be processed by the NVMe controller. The completion queue, on the other hand, is where the NVMe controller places responses after processing the commands. In NVMe-over-TCP, a TCP connection is formed from each host core to a socket in the target during the connection setup stage for transferring IO requests and responses later.

In next section, we provide the measurement setup for the experiments.

3 Measurement Setup

All the experiements run on two servers directly connected by a 100 Gbps link. Each server has a 2-socket NUMA-enabled AMD EPYC 3.00 GHz CPU with 16-core per socket, 128GB memory and a Dell 1.6 TB NVMe SSD.

Storage workloads are generated by an standard benchmarking application, FIO [2] and network workloads are generated with iperf [4]. System performance, such as CPU utilization, is measured with sysstat [5]. All remote storage workloads are performed with asynchronous library over TCP transport. The host of the storage workloads is the client or sender of network apps and the storage target is the server or receiver of network apps. Latency-sensitive storage workloads (L-apps) are sequential read I/O with 4KB block size and queue depth 1. Throughtput-sensitive storage workloads (T-apps) are sequential read I/O with 128 KB block size and queue depth 4. CPU affinity was used to pin applications to a core. To get better network performance, TCP segmentation offload (TSO), generic receive offload (GRO), Jumbo frames with 9000B and accelerated Receive Flow Steering (aRFS) are enabled, and irqbalance is disabled [3]. Hyperthreading is disabled to get better CPU performance. All network and remote storage applications run on the NIC-local NUMA node to eliminate inter-node communication overhead.

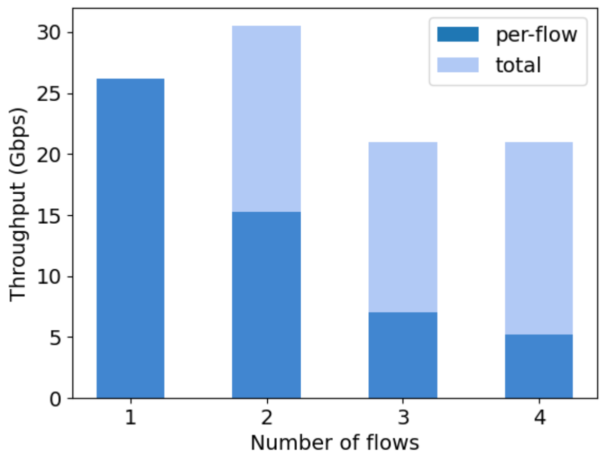

Figure 1: Multiple network applications in contention

4 Contention Experiments

Multiple network apps. First I measured the performance of multiple network applications running concurrently. I linearly scaled number of network flows from 1 to 4. When there is only one network application, the network flow is pinned to a core on both sender and receiver side. In multiple applications, each application is pinned to a separate core. This can be called one-to-one core mapping which is also done for storage applications in later experiments. Since aRFS is enabled, on receiver both TCP/IP processing and interrupt processing are done on the same core. The throughput achieved by the network flows is shown in Figure 1. When a single flow runs, the throughput of the application is 26Gbps and when 4 applications run together, the throughput drops to only 5.2 Gbps per-application. This happens because of higher number of L3 cache misses as the L3 cache is shared between the cores in the same NUMA node. Moreover, when number of flows increase, additional scheduling and memory management processing are required.

Multiple storage T-apps. Next I measured the performance of storage workloads. The default NVMe target module, nvmet, in Linux uses per-CPU worker threads. Each connection to a different namespace is handled by a different CPU core. We scaled number of storage T-apps from 1 to 8 in one-to-one core mapping. Throughput is divided equally between the storage applications as shown in Figure 2. When 8 applications run together, they saturate the SSD. The host CPU has higher utilization than target side. The standard NVMe-TCP implementation has higher CPU processing and context-switching overhead in host. In the implementation, each time an IO request is formed on the host, a doorbell is signalled to remote SSD. However, because of this, most of packets sent from host are only 72 bytes, a single NVMe PDU size, that incurs more CPU processing per byte. Also, NVMe-TCP host has 3 threads - one for blk-mq processing, one for

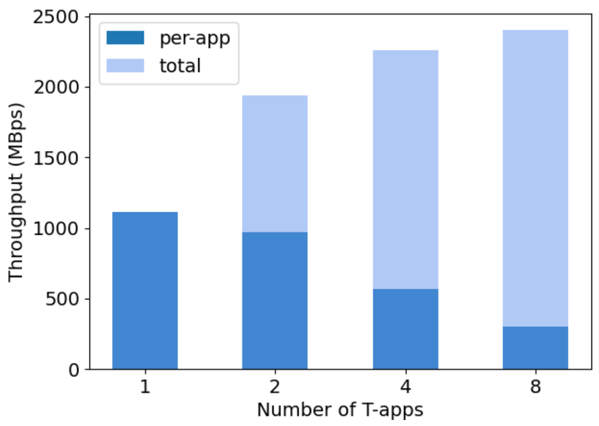

Figure 2: Throughput of concurrent storage applications

TX processing and one for interrupt and RX processing. The doorbells require context switching between these threads that limits the throughput of the storage application [6].

4.1 Contention between storage T-apps and network apps

Now we investigate the performance of storage and network workloads when the two types of applications run together. In this section, I look into contention between storage T-apps and network apps.

Multiple storage T-apps and a network app. First, I run multiple storage T-apps with a single network application. The storage apps run in a one-to-one core mapping and the network application share core with a single storage app. Figure 3 shows the performance of the storage applications including the one that run on the same core as the network app and of the T-apps on other cores. The performance of the T-apps on different cores are unaffected by the network application. However, the performance of the T-app on the same core drops by 33% when it alone competes with the network app and by 12% when 8 T-apps compete with the network app. This suggests that when the storage application has higher bandwidth, it degrades more when competing with a network app on the same core. The CPU in contention in both sender and receiver has over 90% utilization and the CPU cycles are shared between the storage T-app and the network app.

As the network app does not affect the throughput of storage apps in other cores, I looked at CPU utilization of those cores with and without the network app (Figure 4). The result shows that the throughput of the storage apps are maintained at the cost of higher CPU utilization in target cores. This is likely because of higher number of L3 cache misses during contention with a network app.

The throughput of the network app is reduced by 30% when it is in-contention with a storage application on the

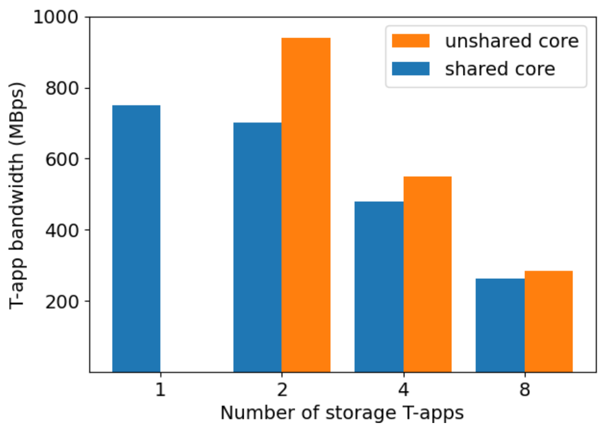

Figure 3: Throughput of multiple storage T-app running with a network app

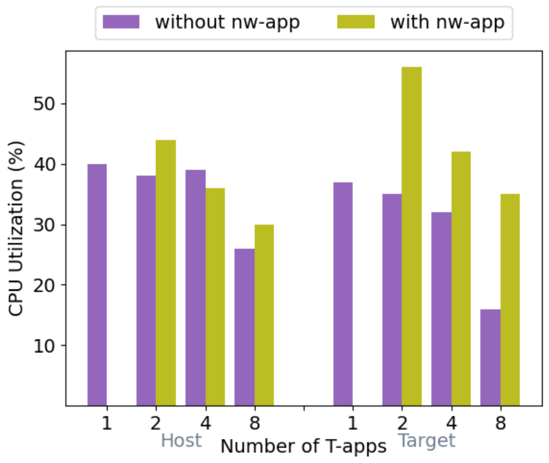

Figure 4: CPU utilization of cores with storage T-app with and without a network app on a different core

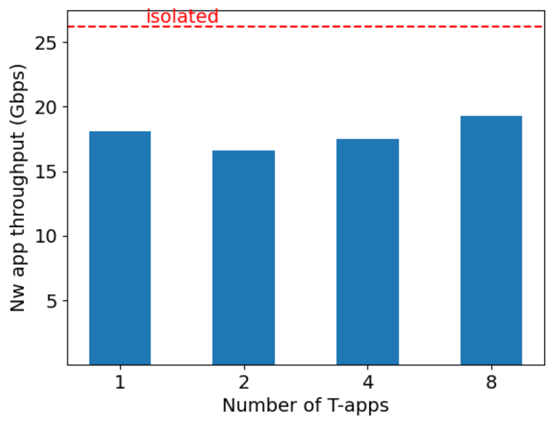

Figure 5: Throughput of a network app running together with multiple storage T-apps

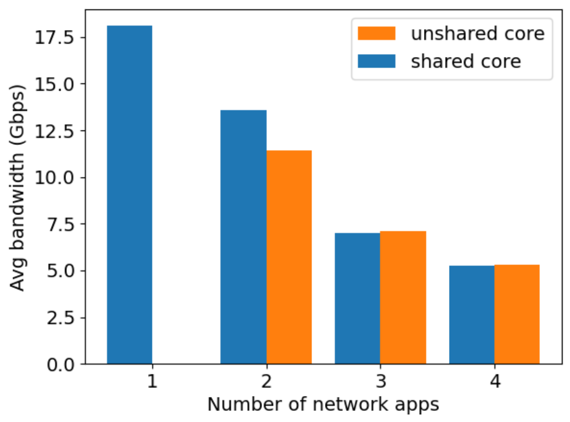

Figure 6: Throughput of network apps running together with a storage T-app

same core. This is because the network application gets less CPU cycles and has to compete with the storage app for cache space. The throughput decreases further when an storage application is added to a different core (Figure 5). However, the throughput then increases as more storage applications are added. Actually, when 8 storage applications run together with the network app, the throughput of the network application is higher than its throughtput when running with single storage T-app. This is because per-core throughput of the storage T-apps decreases as they scale. So, the network app can avail more CPU cycles.

A storage T-app and multiple network apps. Next, I measure the effect of a storage app in contention with multiple network applications. I scaled network applications from 1 to 4 in a one-to-one core mapping and a storage app shares a core with one of the network apps. The throughput of the network apps are unaffected by the storage T-app 6. In contrary, when the storage T-app competes with one network application in the same core, its throughput decreases by 33% (Figure 7). The performance degradation is more drastic as number of network applications increase. When the storage T-app run together with 4 network apps, its throughput reduces by 85%. Since the total throughput of the network apps remain same as the number of network apps are increased, NIC Rx/Tx queues should have similar length in all scenarios. However, scheduling the network applications on different cores introduces processing overhead.

We also looked at the core utilization in sender and receiver in this scenario (Figure 8). Interestingly, the unshared cores have higher CPU utilization in receiver due to higher network processing on receiver side [3]. However, the additional processing overhead for storage-app at sender(host) exceeds the additional overhead of the network-app at receiver(target) and therefore the shared core has higher CPU utilization at sender.

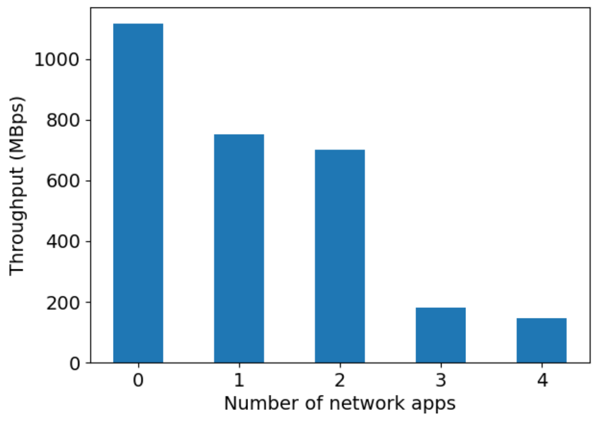

Figure 7: Throughput of a storage T-app running with and without network apps

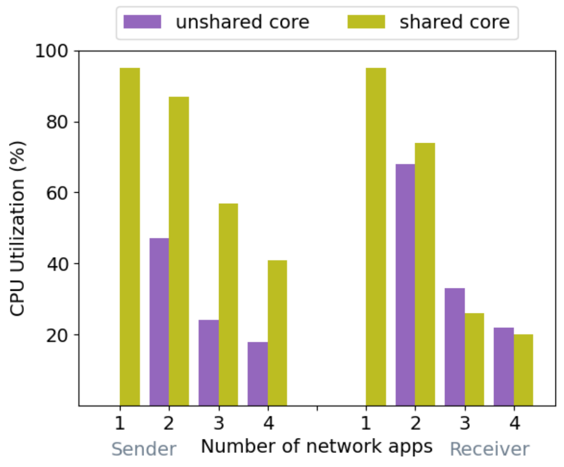

Figure 8: CPU utilization in sender and receiver when a storage T-app is in contention with multiple network apps

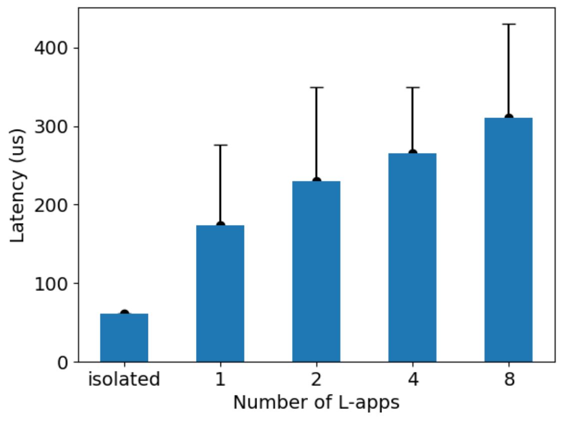

Figure 9: Latency of L-apps running on the same core as a network app

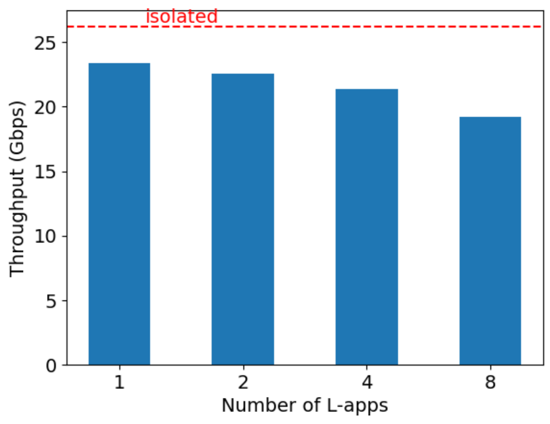

Figure 10: Throughput of a network app running on the same core as multiple L-apps

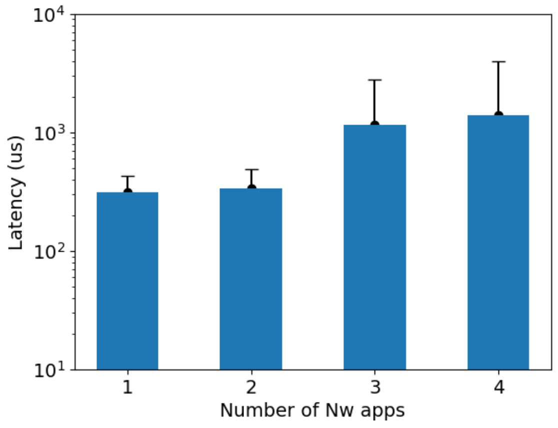

Figure 11: Latency of L-apps running with multiple network apps

4.2 Contention between storage L-apps and network apps

In this section, I look into the performance of storage L-apps and network apps running together. Since, the throughput of L-apps are very small (about 20MBps), I ran L-apps on the same core.

Multiple storage L-apps and a network app. I pinned a network app on a core and then scaled number of L-apps from 1 to 8 in the same core. As seen in Figure 9. The average latency of an L-app became 3x when it shared the core with a network app. This is because of two reasons- 1. the default Complete Fair Scheduling (CFS) algorithm in Linux that divides CPU time equally to applications on the same core, 2. Head of line (HoL) blocking of the L-app packets by the high-throughput network app packets in the NIC Tx/Rx queue. In the case of 8 L-apps, the average latency of the L-apps was 5x compared to isolated latency and the P99 tail latency became 8x. When number of L-apps increases beyond 1, the CPU utilization is 100% on host , and about 90% on target side suggesting that host side processing is the bottleneck for running multiple L-apps. The CPU cycles at both host and target are divided among the L-apps that introduces delays. The throughput of the network app decreased by 10% when it shared a core with an L-app (Figure 10). This is one-third of the reduction compared to sharing the core with a T-app. As L-app has less throughput, less processing is required compared to a T-app. The throughput of the network app decreases by 30% when the app shares core with 8 L-apps as it gets less CPU cycles.

Multiple storage L-apps and multiple network app. Finally, I run multiple network apps in one-to-one core mapping and have 8 L-apps share a core with a network app. The number of network apps are scaled from 1 to 4. The average latency of the L-apps becomes 5x when number of network apps increase from 1 to 4 (Figure 11. This shows that scheduling applications across cores and lack of prioritization can have severe impact on L-apps [7]. The effect of the L-apps on throughput of the network apps decrease as number of network apps increase. (Figure 12). This is because the L-apps have decreased performance as network apps scale and do not impact network apps in other cores.

Table 1 summarizes observations from the experiments-

5 Course Topics

For this project, I mainly used the knowledge of endhost networking stack taught in the course. From the paper "Understanding host network stack overheads", I learned about linux network stack datapath. From this paper, I have used optimization techniques to get better network performance, systematic techniques to experiment with network flows in contention and insights about processing overheads in contention. I could also apply this knowledge for remote storage access.

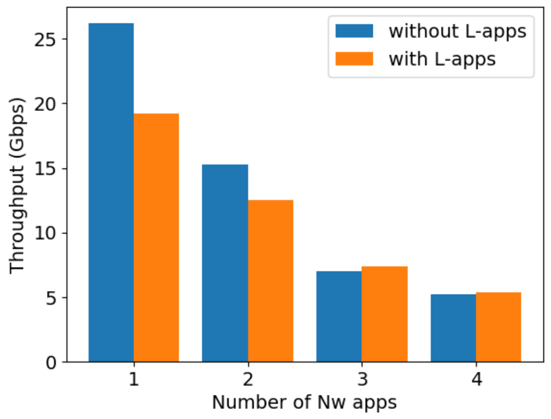

Figure 12: Throughput of network apps running with an L-app

T-apps	L-apps	Nw-apps	Performance	CPU utilization
multi		1	Less T-app throughput at shared core	Higher CPU utilization at target
1		multi	Less Nw-app bandwidth at shared core	Shared core has higher CPU util at sender
	multi	1	High average latency of L-apps	Sender shared core becomes bottleneck
	multi	multi	Effect of L-apps reduces as network apps scale	Sender shared core becomes bottleneck

Table 1: Observations from the experiments

6 Conclusion

The study looked into running multiple network and storage applications concurrently. It is found that when run together, storage and network apps can affect each other. When throughput-sensitive apps run together, throughput degrades because of higher L3 cache misses. Networking apps have higher CPU utilization at receiver while storage apps have higher CPU utilization at sender. This makes better use of the host and target CPU when network apps and storage apps are in contention. When latency-sensitive applications and network applications run together, average latency of L-apps can increase upto 10x because of default CFS scheduling and HoL blocking. Also, in this case, sender side CPU becomes the bottleneck that can degrade performance of network applications on the shared core. This project highlights the importance of understanding and managing core allocations and application characteristics in multi-tenant scenarios to optimize performance in systems running concurrent network and storage applications.

References

[1] Nvm express base specification 2.0b. Technical report, January 2022. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0b-2021.12.18-Ratified.pdf.

[2] Jens Axboe. Flexible io tester (fio) ver 3.33. Technical report, 2019.

[3] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, pages 65–77, New York, NY, USA, 2021. Association for Computing Machinery.

[4] Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. iperf - the ultimate speed test tool for tcp, udp and sctp. Technical report, May 2021.

[5] Sebastien Godard. Performance monitoring tools for linux. Technical report, May 2021.

[6] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP ≈ RDMA: CPU-efficient remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, Santa Clara, CA, February 2020. USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/hwang.

[7] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting linux storage stack for

µs latency and high throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 113–128. USENIX Association, July 2021. https://www.usenix.org/conference/osdi21/presentation/hwang.