Performance of real-world I/O workloads in QEMU/KVM through SPDK and `libvfio-user` on NVMe devices

Juan Sebastián Rolón Lancheros

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

School of Computer Sience

McGill University
Montréal, Québec, Canada

July 2023

Abstract

It is a critical concern for cloud computing providers to identify and adopt optimal virtual machine (VM) I/O storage paths. In Linux, these involve NVMe hardware, PCIe passthrough with VFIO, and userspace drivers (e.g. SPDK). `Vfio-user` is a library and protocol for inter-process communication, making use of sockets to perform I/O. This library aims to tie together all the above-mentioned host-side improvements. QEMU/KVM VMs using `vfio-user` and SPDK are theoretically expected to attain low latencies due to mostly userspace processing, but this VM datapath configuration is still experimental and performance data is lacking in the literature.

This thesis presents the setup and results of benchmarks comparing the I/O performance of a VM using `vfio-user` and SPDK as its storage backend against different datapath configurations. Additionally, we present `vfio_user_snoop`, a `bpftrace`-based tool to analyze `vfio-user` packets in transit; plus a layer-by-layer breakdown of I/O latencies using this setup. We observe negligible differences for baremetal, `vfio-user`, and passthrough in `fio` benchmarks, at $26.66 \pm 0.18\mu$s mean latency in random reads and $27.77 \pm 0.77\mu$s in random writes. For RocksDB, on random reads, `vfio-user` attains 19.75% fewer mean ops/sec vs. bare-metal and 1.66% higher mean ops/sec vs. passthrough; on 50/50 random reads and writes, we see 5.59% and 2.39% fewer mean ops/sec vs. bare-metal and passthrough, respectively.

Abrégé

Il est essentiel pour les fournisseurs de *cloud computing* d'identifier et d'adopter des *datapath* E/S optimaux pour les machines virtuelles (VM). Dans Linux, ces chemins impliquent le matériel NVMe, le *passthrough* PCIe avec VFIO et les pilotes au espace utilisateur comme SPDK. `Vfio-user` est une bibliothèque et un protocole pour la communication inter-processus utilisant des sockets Unix pour effectuer des E/S. Cette bibliothèque regroupe tous les aspects susmentionnés. Les machines virtuelles QEMU/KVM utilisant `vfio-user` et SPDK devraient théoriquement atteindre de faibles latences en raison du traitement essentiellement en espace utilisateur, mais cette configuration du *datapath* pour des VM est encore expérimentale et les données relatives aux performances manquent dans la littérature.

Cette thèse présente la configuration et les résultats de tests comparant les performances d'E/S d'une VM utilisant `vfio-user` et SPDK comme *backend* de stockage contre différentes configurations des *datapaths*. Aditionellement, nous présentons `vfio_user_snoop`, un outil basé sur `bpftrace` pour analyser les paquets `vfio-user` en transit, ainsi qu'une décomposition couche par couche des latences d'E/S en utilisant cette configuration. Nous observons des différences négligeables pour les benchmarks baremetal, `vfio-user` et passthrough avec `fio`, avec une latence moyenne de $26,66 \pm 0,18\mu$s en lecture aléatoire et de $27,77 \pm 0,77\mu$s en écriture aléatoire. Pour RocksDB, en lecture aléatoire, `vfio-user` obtient 19,75 % d'ops/sec en moins par rapport à bare-metal et 1,66 % d'ops/sec en plus par rapport à passthrough; sur les lectures et écritures aléatoires 50/50, nous constatons 5,59 % et 2,39 % d'ops/sec en moins par rapport à bare-metal et passthrough.

Related Publication

Most of Chapters 3, 4, and 5 have been published in:

Sebastian Rolon and Oana Balmau. 2023. Is Bare-metal I/O Performance with User-defined Storage Drives Inside VMs Possible?: Benchmarking `libvfio-user` vs. Common Storage Virtualization Configurations. In *3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '23)*, May 8, 2023, Rome, Italy. DOI: 10.1145/3578353.3589544

Acknowledgements

I would like to thank my supervisors, Professors Oana Balmau and Xue Liu, for their guidance, support, editing, feedback, and insight during the writing of this thesis and the publication it is based on.

I am also grateful to the Nutanix Acropolis hypervisor team, specifically Felipe Franciosi and Ben Levon, for their patience and guidance with regards to `libvfio-user` and advanced topics in Linux virtualization and performance.

I would additionally like to dedicate this work to my parents, whose support never wavered, and who truly believed that I would succeed at McGill.

To Taylor, with my deepest gratitude. You were a beacon that lit up my world after years of darkness. I sincerely thank you for sharing this period of our lives in Montreal. We were both there when it started, and we walked this path together until it wasn't possible to continue.

To my friends from Los Andes: Alejandro, Cristian, Juan Pablo, and Zahyra. I am happy that we've stayed together over more than 10 years, despite the distance.

To Gabriel and Ayman, who I can talk to about anything. To Jazlyn and Avinash, for the kind words and support, the fun and work at CSGS, and the writing sessions.

To Isabelle and Lauren, for their invaluable help and listening.

To the City of Montreal. Living here has been an experience, to say the least. It has driven me to grow in ways I never expected nor intended.

Table of Contents

Abstract . . . ii

Abrégé . . . iii

Related Publication . . . iv

Acknowledgements . . . v

Table of Contents . . . vi

List of Tables . . . viii

List of Figures . . . ix

List of Programs . . . xi

1 Introduction . . . 1
- 1.1 Problem Statement . . . 2
- 1.2 Contribution Overview . . . 3
- 1.3 Problem Relevance . . . 3
- 1.4 Thesis Overview . . . 4

2 Background . . . 6
- 2.1 Virtualization and QEMU/KVM . . . 6
  - 2.1.1 System VMs and hardware virtualization . . . 7
  - 2.1.2 QEMU and KVM . . . 9
  - 2.1.3 `virtio` and `vhost` . . . 10
- 2.2 Hardware and drivers . . . 12
  - 2.2.1 Driver paradigms . . . 13
  - 2.2.2 Linux userspace drivers: UIO and VFIO . . . 16
  - 2.2.3 NVMe . . . 17
  - 2.2.4 SPDK . . . 19
  - 2.2.5 `libvfio-user` . . . 22

3 Approach & methodology . . . 25
- 3.1 Choice of end-to-end applications . . . 26
  - 3.1.1 `fio` . . . 26
  - 3.1.2 RocksDB . . . 26
- 3.2 Storage virtualization configurations . . . 27
  - 3.2.1 Bare metal configuration . . . 28

3.2.2 `Libaio` configuration . . . 29
3.2.3 Passthrough configuration . . . 30
3.2.4 `Vfio-user` configuration . . . 31
3.2.5 Additional configurations . . . 31
3.3 Layer-by-layer latency measurement with `vfio_user_snoop` . . . 32

4 Experimental setup . . . 34
4.1 Hardware and software environment . . . 34
4.1.1 Hardware . . . 34
4.1.2 Software . . . 36
4.2 Provisioning and automation . . . 38

5 Results . . . 40
5.1 `Fio` microbenchmark results . . . 40
5.1.1 Reads . . . 41
5.1.2 Writes . . . 42
5.2 RocksDB end-to-end application results . . . 43
5.3 Latency breakdown . . . 48

6 Related Work . . . 50
6.1 `io_uring` . . . 50
6.2 SPDK-vhost-NVMe . . . 51
6.3 Other approaches and notable work . . . 52
6.3.1 Kernel-level datapath optimization . . . 52
6.3.2 SR-IOV . . . 53
6.3.3 Latency source modeling and formalization . . . 53
6.3.4 Measurement pitfalls . . . 54

7 Conclusion . . . 55
7.1 Future Work . . . 57

Bibliography . . . 59

Appendices

List of Tables

2.1 I/O latencies and cost per Gigabyte for different storage technologies as of 2021, forming a clear hierarchy. We are interested in Low-latency SSDs in this thesis, which sit in the middle of this latency-cost spectrum. Despite recent advances, SSDs still lag behind memory-bus storage technologies by ∼2 orders of magnitude in the best case. Data from Wu, et al. [90] 18

4.1 Hardware specifications of the shared laboratory server (`discslab-server1`) used to perform all benchmark runs. NVMe ratings are from the manufacturer's specifications [33]. 35

List of Figures

2.1 Virtual machine taxonomy, from Smith and Nair [75]. QEMU/KVM, the Virtual Machine Manager (VMM) used on all the work in this thesis, is classified as a **Same Instruction Set Architecture (ISA)** system virtual machine. 7
2.2 Basic QEMU/KVM execution loop, from Bugnion et al [19]. Notice how I/O handling breaks out of Kernel mode into userspace, incurring a context switch. 9
2.3 QEMU/KVM I/O handling with `virtio` and `vhost`, from Yang et al [93]. 12
2.4 Different driver models in (a) process-structured operating systems (microkernel) and (b) monolithic kernel operating systems, from Tanenbaum [80]. 14
2.5 Frameworks such as `vfio-user` make it possible to implement PCI devices in userspace, with close to no overhead. Note that the "large" SPDK on the right side is a host-level userspace process, while the "small" SPDK on the left is a different SPDK instance running in the userspace of the guest. 23

3.1 Bare metal configuration. Applications run inside the host OS and generate I/Os directly to the NVMe drive, through SPDK. 28
3.2 `Libaio` configuration. Higher latency is expected in this configuration compared to bare metal, passthrough, and `vfio-user`, as the I/O calls incur context switches. . . 29
3.3 Passthrough configuration. Applications run in the guest OS, but they do not perform system calls to access the NVMe hardware. 30
3.4 Layer-by-layer latency measurement for three layers: 1) application (`fio` here) to the guest NVMe driver (green); 2) guest NVMe driver to SPDK `bdev` (orange); and 3) SPDK `bdev` to NVMe (blue). 33

5.1 Mean IOPS per number of threads and queue depth results for `fio` random reads, 4KB block size. 41
5.2 Mean IOPS per number of threads and queue depth results for `fio` sequential reads, 4KB block size. 42
5.3 Mean IOPS per number of threads and queue depth results for `fio` random writes, 4KB block size. 42
5.4 Mean IOPS per number of threads and queue depth results for `fio` sequential writes, 4KB block size. 43
5.5 RocksDB sequential fill operations per second over time, using 1 thread 44
5.6 RocksDB random read operations per second over time, using 1 thread 45
5.7 RocksDB random write operations per second over time, using 1 thread 46
5.8 RocksDB random read and write operations per second over time, using 1 thread . . 47
5.9 RocksDB sequential read operations per second over time, using 1 thread 48
5.10 Layer-by-layer latency for reads and writes. 49

7.1 A possible "multiplexed" or multi-VM configuration using SPDK, to be tested in future work. Note that the arrows representing I/O between the memory mappings, present in Figures 2.5 and 3.4, are omitted here for simplicity. Each QEMU memory mapping has a corresponding memory mapping in SPDK's process space. 58

List of Programs

3.1 Example `fio` configuration file used for benchmarking. 26

4.1 Excerpt from the `libvirt` XML virtual machine schema file for the passthrough configuration, described in Section 3.2.3. 37

1

Introduction

The advent of cloud computing and datacenter-scale applications has brought on the need for ubiquitous usage of virtualized workloads and ever-more-performant storage devices at the bottom of the storage hierarchy, such as NVMe drives and Intel Optane persistent memory [46]. With fleets of millions of machines, application administrators need to ensure maximum usage of the available hardware.

In pursuit of this goal, on the software side, datacenter administrators and cloud providers rely on Virtual Machines (VMs) to provide task isolation, resource disaggregation [11], security [19], and flexibility for clients using their platforms, and also for themselves [60]. VMs allow cloud service providers to host more clients than they would be able with physical hardware alone; they allow for features such as dynamic creation and allocation of new machines for burst workloads; and they perform their tasks with the theoretical security that tenants will not be able to influence or affect each others' tasks in malicious ways (disregarding bugs and breaches) [75]. For example,

VMs are the fundamental technology that makes it possible for AWS, Azure, GCP, and others, to offer different operating system hosts with variable hardware characteristics for thousands of users. It would be a harder task if they only had access to bare-metal machines with the corresponding configurations.

On the hardware side, Non-Volatile Memory Express (NVMe) devices are as of 2023 one of the fastest available storage options, possibly only behind persistent memory (PMEM), in the set of available I/O devices that are now being referred as the storage jungle [46]. For a long time, application design was dictated by the implicit assumption that I/O would be the biggest bottleneck in terms of latency. New hardware breaks these preconceptions, due to support for a large amount of queues and long queue sizes whcih allow for massively parallel I/Os.

1.1 Problem Statement

It is commonly perceived that workloads running inside virtual machines have to pay a performance overhead [19, 47]. This has changed over the years, first with the implementation of direct virtualized code execution on the CPU facilitated by security rings (Intel and AMD's virtualization extensions) [19], and more recently through virtual machines being able to directly access bare-metal hardware through passthrough on Linux [81]. Most recently, there have been advances in developing userspace drivers which allow for processes to interact with hardware directly without having to go through the Kernel and triggering context switches, which in practice so far have shown great reduction in I/O latency overhead.

A prominent implementation of this paradigm is the SPDK framework [85, 92] created by Intel, Nutanix, et al., which is a series of Linux libraries and applications that allow processes direct access to NVMe devices by connecting with the Kernel's passthrough facilities (such as UIO and VFIO) and reimplementing Kernel storage stack features with the intention of them being used directly by applications. SPDK also follows the polling model for NVMe I/O, which as also shown great success lately with the `io_uring` Linux storage driver, as opposed to the widespread interrupt model. Recently, Nutanix has developed `libvfio-user`, an SPDK library that allows for userspace processes to "speak" the Kernel VFIO protocol and therefore allow for another layer of the I/O stack belonging to the Kernel to be bypassed.

Research Questions.

- In comparison with currently-existing and production-ready storage I/O for virtual machine configurations, where does `libvfio-user` sit in terms of latency and IOPS?
- Is `libvfio-user` performant enough to be used in production by cloud service providers?
- Given that it requires a complex and layered configuration, where does the CPU spend the most time when I/O requests are executed through `libvfio-user`?

1.2 Contribution Overview

In order to answer the questions above, this thesis exposes results of data collection on experiments executing Fio and RocksDB processes inside a QEMU/KVM virtual machine while using SPDK inside the VM, and using SPDK from the QEMU process towards the hardware while going through `libvfio-user`.

In order to get deeper insight into the contributions of each one of the steps involved in the I/O pipeline from process I/O submission to response, we developed `vfio_user_snoop`, a tool for measuring latency in specific layers of the stack, and its results were collected by generating traffic using Fio.

Descriptions of the tools, code, and automations developed to set up the scenarios is also included. In addition, results from synthetic benchmarks using fio, comparing bare-metal, default "naive" QEMU/KVM configuration, QEMU/KVM using `io_uring` at all levels, and QEMU/KVM using SPDK+`libvfio-user` are included to contribute to the body of data in the literature.

1.3 Problem Relevance

Datacenters are complex pieces of infrastructure that incur costs in many different dimensions, including cooling, energy usage, hardware, noise, among others [11, 60]. These costs can also be understood as directly economic costs for datacenter administrators and cloud service providers.

Improvements in the efficiency of usage of clock cycles can, at scale, have direct impacts on these costs. Using userspace drivers would involve fewer clock cycles, from I/O request from the

process all the way to I/O submission to the hardware, due to the fact that some Kernel facilities are not involved and there are no kernel to userspace context switches. In effect, less code needs to be executed to attain the same tasks.

If the above holds true in practice, it could enable cloud service providers different paths for server hardware usage, e.g.:

1. maintain datacenter size and application density to obtain slight increases in energy efficiency, due to the fact that fewer clock cycles are spent, or

2. increase application density and datacenter size for very slight degradation in application performance (since the provider can afford to run e.g. 11 instead of 10 VMs in a single machine due to the efficiency gains).

In a more practical sense, warehouse-scale computing makes it so that task optimization, even at small levels, can have measurable and noticeable effects on the environmental impact of datacenters and surrounding infrastructure. For the cloud service providers themselves, it can also represent cost savings.

The existence of `libvfio-user` was brought to our attention by Nutanix themselves, one of the principal companies involved in its development. As a virtualization technology developer and cloud service provider, Nutanix is directly invested in using `libvfio-user` in its own development cycle, and possibly incorporating it in its own products with the aforementioned goals of performance improvement and cost reduction. We collaborated with their Acropolis Hypervisor team in the formulation of the research questions, and throughout the work reported in this thesis.

1.4 Thesis Overview

This thesis is organized as follows:

- Chapter 2: Background. We go over the theoretical framework of virtualization in general and in Linux specifically using QEMU/KVM; plus the motivation of userspace drivers and a short overview of their implementation over time.

- Chapter 3: Study approach. We go over the general idea of the experiments that we ran, why they are useful to answer our research question, and their specifications.

- Chapter 4: Experimental setup and implementation. We cover the details on how the experiments were set up, tools and automation, pitfalls, and challenges.
- Chapter 5: Results. We present the complete results of our tests and examine them.
- Chapter 6: Related Work. We discuss how our work fits in the wider scope of research in operating systems, storage, and cloud computing.
- Chapter 7: Conclusion. We describe our vision for future work in this area, and we summarize our findings.

2

Background

We will cover the necessary concepts to get an understanding of why `libvfio-user` is a novel configuration option for providing storage I/O for virtual machines and why we want to measure its performance, placing it in the context of other currently available storage configurations.

2.1 Virtualization and QEMU/KVM

A virtual machine, in its simplest form, is a software implementation of the hardware parts that make a modern computer [19, 75]. This means that the CPU, the main memory, and peripherals such as network or storage cards are provided by the software. This is in contrast with bare-metal computing, where the operating system manages the real hardware resources of the machine.

A simple virtual machine is therefore a program that provides an *illusion* to other programs, including operating systems, of them interacting with real hardware. Programs subject to this

illusion are commonly referred to as running *inside* the virtual machine. Programs inside the virtual machine, particularly operating systems, are commonly referred to as *guests*; the machine with access to the actual hardware (the one providing the "illusion") is usually called the *host*.

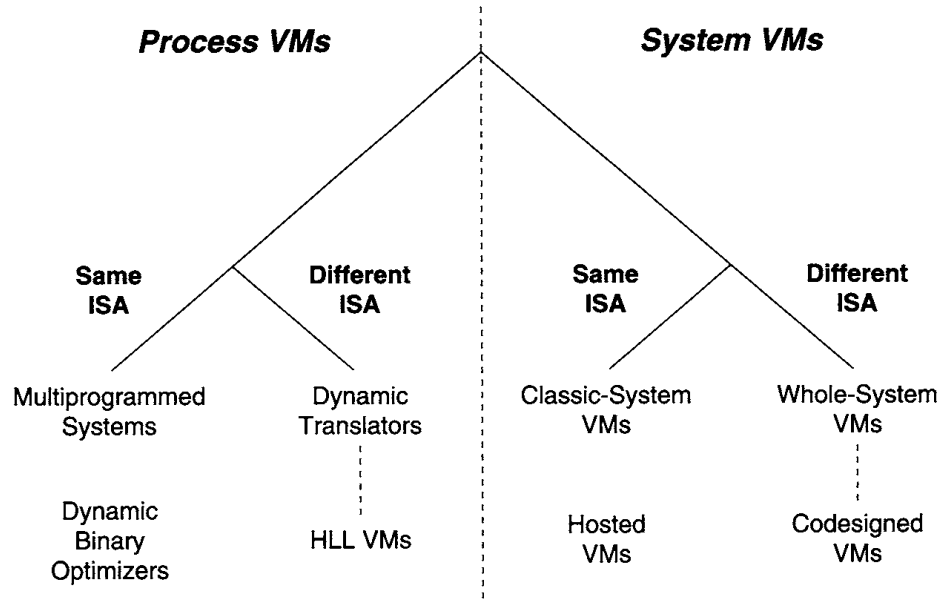


Figure 2.1: Virtual machine taxonomy, from Smith and Nair [75]. QEMU/KVM, the Virtual Machine Manager (VMM) used on all the work in this thesis, is classified as a **Same Instruction Set Architecture (ISA)** system virtual machine.

Simple virtual machines, namely machines that run entirely in software, are usually called *process virtual machines* [75]. Process VMs provide full abstractions of the hardware environment for the processes that execute inside of them, allowing applications to be run in a consistent environment regardless of the real hardware configuration. Process VMs can emulate existing hardware architectures, allowing console games to be played on personal computers, for example; but they can also describe architectures without any physical equivalents, such as the Java Virtual Machine (JVM) [57]. Crucially, however, process VMs are unable to run a full operating system, due to OS's requirements to run certain instructions in privileged modes in the CPU [73].

2.1.1 System VMs and hardware virtualization

Modern operating systems make use of certain CPU features to provide enhanced security and protection from possible failures from bugs in userland code. We refer to these as **protection rings** or **privilege modes**. Specific instructions in the CPU instruction set architecture (ISA) can make

it so that memory-access instructions, for example, trigger a hardware trap if attempting to reach an address outside their allowed range. Other instructions can also be guarded behind a protection ring, so that e.g. virtualization instructions (more on these later) can only be executed when the processor is running on the lowest (most privileged) ring. These features allow for more complete system emulation and a different type of virtual machines: *system* (or *full*) *virtual machines*.

According to Smith and Nair [75], system virtual machines originated in the mainframe-computing environment of the 60s and the 70s. The initial intention was to run multiple operating systems on a single machine. System VMs allow for total virtualization of a complete system environment, which makes it possible for full operating systems to be installed on top of them. In the *process VM* paradigm, the host operating system has the capability to execute and manage process virtual machines. The analogous piece of software that takes care of this task in the *system VM* paradigm is the *hypervisor*, or Virtual Machine Manager.

There are two types of hypervisors. Type-1 hypervisors have direct access to the real hardware and have no need for an operating system on the bare-metal hardware to operate. Type-2 hypervisors, also called hosted hypervisors, rely on the host operating system to provide access to certain services, such as hardware access through host drivers [19, 75]. Notice that both of these types are present under the **Same ISA** category on Figure 2.1.

In addition to protection rings, a crucial technology that makes modern system VMs possible is hardware-supported virtualization, implemented in the mid-2000s by Intel and AMD with the VT-x and AMD-v extensions for their CPU instructions sets. It was possible to have full system VMs before the introduction of VT-x, as evidenced by the existence of Xen, a Type-1 hypervisor [19]. The issue, however, was the requirement for guest kernels to be modified so that they could run in the virtualized environment. This is called *paravirtualization*: the virtualized task is aware of it being run in a non-bare-metal environment, and it has to undergo modifications that break the principle of **equivalence**: the virtualized task is not identical to how it would run on bare metal [19, 75].

Type-1 VMs have access to lower, distinct CPU protection rings in order to execute code with higher privileges [4]. This is a way to provide the guest operating systems a better illusion of them having full access to the hardware; under a Type-1 hypervisor configuration, guest operating systems will be able to execute their code on the real CPU, but some of the instructions that they

will attempt to execute (such as things related to paging and memory management) will actually cause traps that must be handled by the hypervisor.

2.1.2 QEMU and KVM

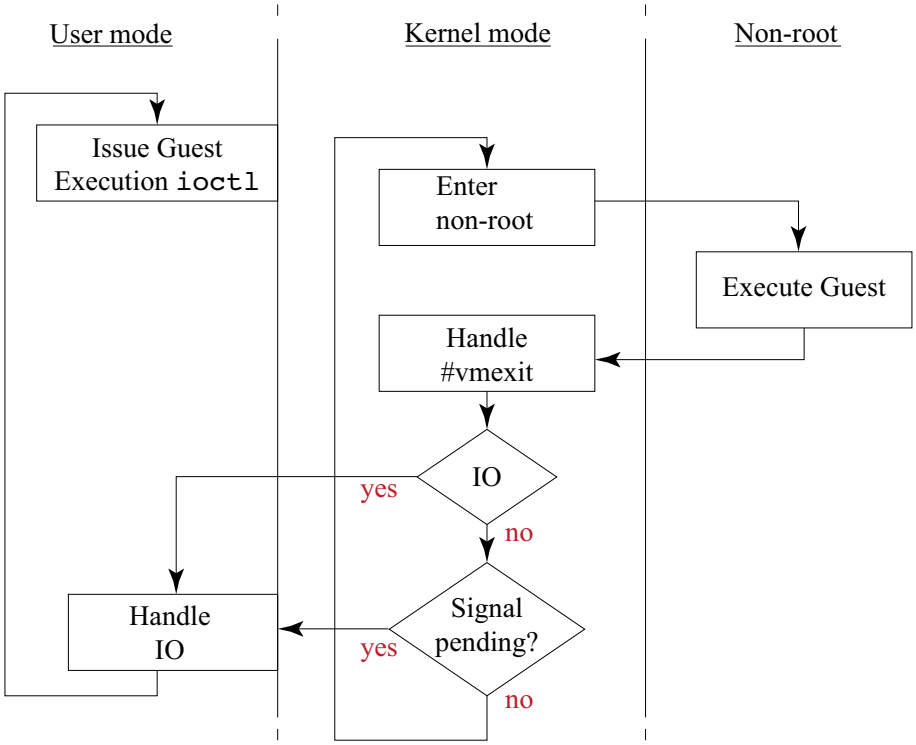


Figure 2.2: Basic QEMU/KVM execution loop, from Bugnion et al [19]. Notice how I/O handling breaks out of Kernel mode into userspace, incurring a context switch.

QEMU was released by Fabrice Bellard in early 2003, originally with the intention of creating an emulator (**Q**uick **Emu**lator) that would run Linux x86 binaries on a diverse set of CPU architectures [12]. This was (and still is) supported by a dynamic instruction translator, which QEMU calls TCG, the Tiny Code Generator [36]. Originally, QEMU was only a process VM, emulating the CPU exclusively, but by the middle of 2003 a full system emulation mode was added, which included peripheral and external hardware emulation [13].

By 2005, when the Intel VT-x technology was released [39, 63], Avi Kivity from Qumranet started working on KVM, which he would release in 2006. KVM stands for Kernel(-based) Virtual Machine [25], a Linux kernel module that allows QEMU to request certain of its operations to be executed on the real hardware, supported by the CPU virtualization extensions added by Intel and

AMD to the x86 and x86-64 instruction sets (some other CPU architectures also support virtualization extensions). According to Bugnion et al, QEMU/KVM is squarely a Type 2 hypervisor [6].

Essentially, KVM acts as a driver for the VT-x and VMX CPU features, such as the `VMXON`, `VMLAUNCH`, `VMCALL`, etc. instructions, that enable the CPU to conceptually work in a lower protection ring, colloquially called ring -1 [35]. When in this mode, guest kernel instructions are executed directly on the real CPU, except for certain high-privilege instructions which would trigger a `VM_EXIT` signal and return control flow to KVM in the host's kernel. KVM exposes an API to userspace through the `/dev/kvm` character device, on which processes (most commonly QEMU) can issue `ioctl()`s to create or destroy hardware-backed virtual machines, abstracted by a virtual CPU and MMU (vCPU) [6]. This way of exposing kernel features and access to hardware through the Virtual Filesystem is now common in Linux, as we will see with VFIO in Section 2.2.2.

Both QEMU and KVM are open-source projects that have enjoyed significant momentum in their development since release. As of 2017, Amazon has based its AWS hypervisor on a modified version of KVM [74]; Google has done similarly for its GCP products [45]. The cloud and datacenter-scale computing would not be possible, at least at the current scale, without these advances in virtualization technology, both for virtual machines (as described in this section) and for other resources such as networking [60].

Modern cloud infrastructure sits on a very deep and complex "layer cake" of abstraction supported by both software and hardware: time sharing operating systems that multiplex resources such as CPU (through sophisticated scheduling) and memory (through collaboration with hardware MMUs); over which hardware-supported virtual machines are placed to cooperate as distributed systems through software-defined virtual networks; supporting containers that further make use of kernel features for security and isolation; culminating in further virtual name resolution services that collect hundreds or thousands of containers through orchestrators like Kubernetes. The hope is that all of this infrastructure aids software be faster, easier to maintain, more reliable, and more helpful for both the users and the developers that strive to create it.

2.1.3 `virtio` and `vhost`

The QEMU/KVM hypervisor can, in a simple configuration, handle I/O and peripherals by fully emulating hardware. One benefit is the virtualization principle of **equivalence** is held: the unmod-

ified guest operating system is transparently executed by the hypervisor. However, the tradeoff is a loss of performance. The guest kernel is running a full driver that might waste operations that could be optimized, if only the driver could be modified to better work in a virtualized environment.

A way to mitigate this wasted work is to use `virtio`. Rusty Russell introduced `virtio` in 2007, a time when multiple hypervisor solutions for Linux were still vying for prominence (a post on Russell's own blog mentions four: Xen, KVM, VMWare and lguest [69]). Each one of these had its own way of emulating or virtualizing hardware accesses for the guest. Russell's idea was to introduce a standard for virtual devices to follow in order to facilitate virtualization of hardware. In his 2008 paper [70], Russell introduces the goals of:

1. a common configuration ABI for virtualized devices, as opposed to "boutique transport mechanisms (...) [which are] particular not only to a given hypervisor and architecture, but often to each particular kind of device";

2. ensuring all work related to `virtio` was part of the Linux kernel, so it would be included in the operating system and there would be no need to further convince third-parties to buy-in to additional libraries;

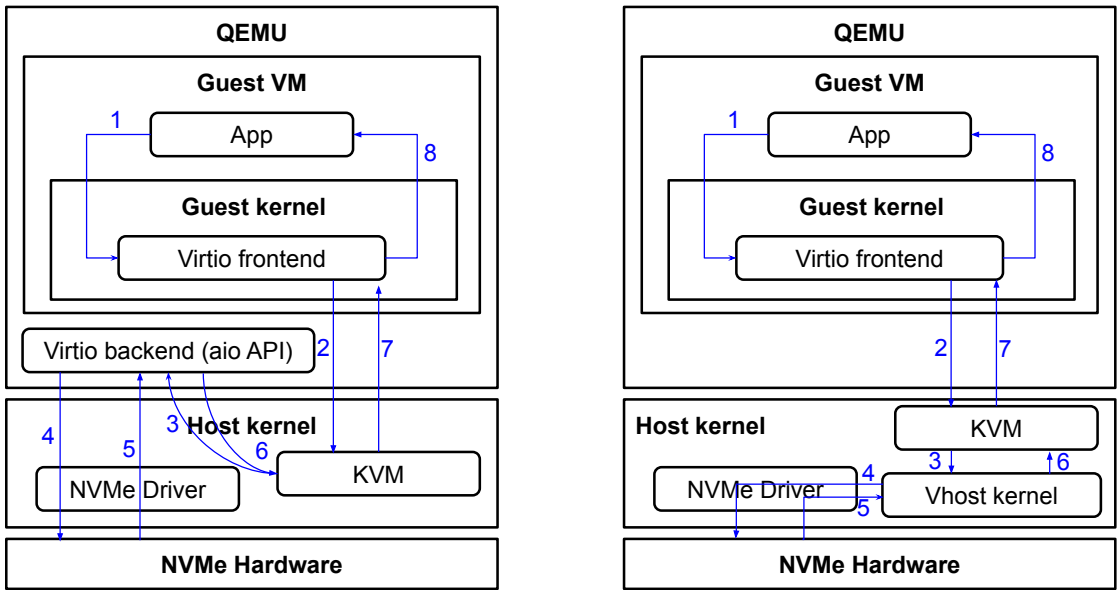
3. conceptually separating the concepts of drivers, transport, and configuration mechanisms for virtualized hardware;

4. providing a reference implementation that showed the model was feasible.

`virtio` is an example of paravirtualization, since the guest OS needs to include `virtio` frontend drivers in its kernel that are fully aware of the existence of a `virtio` backend in the hypervisor layer that handles I/O (this would usually be QEMU).

`virtio` can increase the performance of paravirtualized hardware devices, but it doesn't change the QEMU/KVM execution loop shown in Figure 2.2. Specifically, I/O handling is triggered from the guest, captured by the KVM host kernel module, switched back into the `virtio` backend in QEMU userspace, and then back to host kernel space through normal I/O system calls. Only the guest kernel layer is "slimmed down". An approach to reduce the number of context switches is to offload the backend emulation onto a kernel process.

This is the idea behind `vhost`. By 2009, users of the `virtio` network devices had been experiencing performance issues that were identified as being caused by very frequent system calls triggered by the QEMU `virtio` backend, in addition to packet data copying from kernel space into userspace [48]. As a solution, Michael Tsirkin created the first `vhost` device, `vhost-net`, which essentially moves the `virtio` backend code into userspace, allowing it to be configured and controlled via a character device exposed to userspace [82]. This approach is shown in Figure 2.3b. In time, additional `vhost` devices were created to reduce latency in block devices as well.



(a) Typical QEMU/KVM `virtio` handling

(b) `Virtio` handling with `vhost-kernel`

Figure 2.3: QEMU/KVM I/O handling with `virtio` and `vhost`, from Yang et al [93].

2.2 Hardware and drivers

Device drivers are programs that have knowledge of a piece of hardware in detail: whether they have clocks and/or timers (e.g sound cards), what the specifications of the busses for communicating with the devices are, how the device memory (if present) is laid out, where in this memory can configuration options be found and set, etc [7].

Operating systems attempt to provide higher-level APIs through layers of abstractions for hardware, with the intent of decoupling the system hardware from the OS implementation, encouraging code reuse in the kernel, and providing flexibility in the use of system resources for application

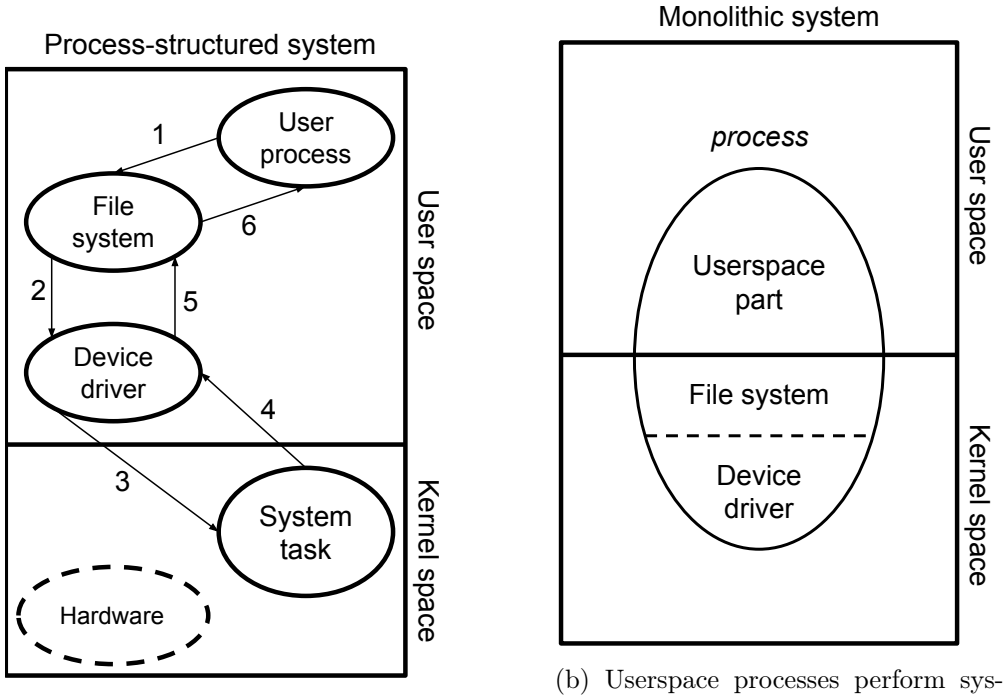
developers. Drivers sit directly on top of the hardware, and their tasks are usually the last step in the conversion of abstract calls (such as "write a line into a file") into the distinct operations that the device needs to execute to complete such a request (such as "move the read/write head onto the 4th cylinder and write 96 bytes" [7, 80].

2.2.1 Driver paradigms

Over the history of computing, software to hardware interaction has been handled in different ways. Some operating systems, such as the original MS-DOS, allowed for userspace programs to take direct control over hardware. In effect, programs themselves came bundled with drivers for peripherals and hardware, as exemplified by PC games that had specific settings for particular GPUs and sound cards [44, 76].

Traditional monolithic kernel operating systems usually bundle their drivers as part of the kernel distribution. One reason to do this is because drivers usually need to execute privileged code, such as the `out` x86 instruction for certain I/O devices, or accesses on PCI device memory-mapped regions. There may also be a performance requirement for the driver; userspace code can be scheduled away, or its memory can be paged, which constitutes an unpredictable environment that may affect real-time interaction with the hardware. However, allowing drivers to run in kernel space may come with its own sets of problems. It is a common thought that driver programmers have lower skills than "pure" kernel programmers [7], which may come from the fact that (at least up to the days of Windows XP), it was estimated that 85% of system crashes were caused by buggy drivers [79]. Despite whether this conception is true or not, if no mitigations are taken, badly programmed kernel drivers can e.g. overwrite memory regions reserved for other kernel tasks or userspace, among many other issues.

A different method for organizing drivers is the one used in microkernel operating systems, such as Minix. The microkernel is intended to be the minimum viable operating system, possibly managing only scheduling, memory management, and process isolation [80]. In this model, drivers are no different from other processes that execute in userspace. This solves the issue of badly written drivers being able to bring down the kernel due to bugs [7]. Certainly, microkernel operating systems must have mechanisms to allow userspace processes to access the privileged resources that would be behind kernel space in monolithic OSs. Minix, for example, allows some userspace processes access



(a) Interactions 1-6 are messages between four independent userspace processes.

(b) Userspace processes perform system calls to use kernel space functions, such as the file system, which in turn interacts with the device driver.

Figure 2.4: Different driver models in (a) process-structured operating systems (microkernel) and (b) monolithic kernel operating systems, from Tanenbaum [80].

to memory segments from other address spaces, it exposes API calls to access general purpose I/O, and it provides a way for processes to be notified of hardware interrupts through the `receive` mechanism [80].

The choice between monolithic and microkernels has been famously immortalized by the Torvalds-Tannenbaum debate of the early 1990s [1], which may have eventually led to Linux allowing for drivers to be partially separated from the kernel via modules, which can be dynamically loaded or unloaded as part of the running kernel image without having to reboot. Despite this decoupling, once loaded, module code runs at the same privilege level as the kernel — it can still crash the system if bugs are present. [3]

Finally, another approach that combines features from "traditional" Kernel-space drivers and microkernel-inspired userspace processes are **userspace drivers**. These have been possible in Linux since at least 2001 [68], albeit with limitations, such as no access to interrupts and an incomplete API to deal with general purpose I/O ports. However, before kernel 2.6.26 (July 2008), it was

possible for `sudo`ed processes to `mmap()` the `/dev/mem` file, a full image of the system memory [30]. It is still possible in modern systems, if the kernel is compiled with the `CONFIG_STRICT_DEVMEM` flag disabled, which is not the default configuration. Allowing userspace processes to access all of the system's memory is a security and stability risk. This tension between allowing and restricting access to system resources has been a constant underlying concern for driver and kernel developers in Linux, as will be covered in Section 2.2.2, much of the progress made in allowing secure access to the hardware from userspace is what makes `libvfio-user` possible.

An orthogonal concern for the architectural decisions on how drivers communicate with the hardware are the mechanisms or algorithms that are used. There are broadly 2 paradigms: programmed I/O (also commonly called *polling*) and interrupt-based I/O. Programmed I/O typically requires the CPU to spend cycles querying and writing to the device's control and data registers for its status and the availability of data. In cases where the device is busy, such as a Hard Disk Drive performing a write (which could require waiting for the read/write head to physically move), CPU cycles are also spent waiting for the device to become available. On time-sharing systems, such cycles could possibly be used for serving other userspace processes [2].

Interrupt-based I/O allows for the CPU to perform other tasks while the device controller takes care of the operations. Essentially, the CPU issues a transfer request onto the device, which could be the block identifier and the number of bytes expected for a block read request, for example. Once the device registers contain the requested data, the device raises a hardware interrupt that forces the CPU to step into kernel code which can handle the request [7]. This saves the CPU from wasting cycles checking for the device status, but copying data to and from the device still takes cycles that could be used for other tasks.

An optimization which uses some of the ideas of interrupt-based I/O is **Direct Memory Acces** (DMA). DMA requires hardware support as an additional controller on the system motherboard. It is essentially an offloading of programmed I/O onto this DMA controller. In the simplest possible cases, the CPU issues transfer requests to the DMA controller, which will then take care of managing requests towards the device controller. Typically, a system where DMA is available will allow device controllers themselves to read and write from specific buffers situated in the main system memory, which are allocated by the drivers. The DMA controller interrupts the CPU once the data is

available in the system memory buffers [80]. This subject is relevant for this thesis, since NVMe devices use DMA [87].

2.2.2 Linux userspace drivers: UIO and VFIO

As mentioned in Section 2.2.1, userspace drivers were partially possible to implement in Linux by 2001. An important feature that still remained was interrupt handling from userspace. Attempts to make this work happened since at least 2003, one of them being the Gelato project from the University of New South Wales [21], with one of the main motivations being to mitigate the introduction of new bugs into the kernel. Gelato introduced kernel patches that would expose interrupt queue (IRQ) files to userspace through the VFS at `/proc/irq/irqX/irq`, where `irqX` is an identifier corresponding to a specific interrupt cause. Calling `read()` on these files would cause the userspace process to wait until an interrupt of the appropriate type was triggered by the hardware. Interestingly, the Gelato project also added support for DMA, a feature that would return in `vfio`.

At around the same time, custom kernels were being compiled more or less ad-hoc to support userspace drivers for industrial PCI control cards. These customizations were usually not shared with the kernel development community at large until 2006, when Greg Kroah-Hartman published a rough draft of `iio`, the Industrial I/O driver [53]. The very first drivers implemented using `iio` used far fewer lines of code and complexity when compared with equivalent kernel module drivers for the same hardware [28]. This improvement in driver simplicity, plus the fact that proprietary drivers for industrial applications would be possible while also reducing the number of bugs, gave momentum to the generic userspace driver infrastructure in the Linux kernel, which would be renamed UIO. It was merged for general availability in kernel 2.6.23, from October 2007 [24]. UIO did not allow for DMA from userspace, it simply provided ways for userspace processes to be notified of interrupts and `mmap` the devices' memory through character devices under `/dev/uioX`.

Starting around 2008, Intel and AMD added support for I/O Memory Management Units (IOMMU) in their server and consumer platforms [14]. In a similar way to standard MMUs, IOMMUs are special hardware that sits between the device controllers and the system communication bus, with the task of translating between real addresses and bus addresses. This can enable devices to access memory at very high addresses (e.g. if the device can only perform 32-bit addressing), but it also adds the possibility of securing DMA: the CPU is able to program the IOMMU so that

devices are limited to specific main memory regions [5]. To take advantage of IOMMUs and their features, Alex Williamson proposed VFIO in 2011, with the intention of replacing UIO as the main kernel framework for implementing drivers in userspace. From the beginning, VFIO was designed with the intention of allowing virtual machines to access host hardware directly, supported by features such as full interrupt notification support, non-root access to device memory, and DMA (the latter two now safe due to IOMMU protection) [88]. By and large, the VFIO API is very similar to UIO: character devices are presented under `/dev/vfio/vfioX`, where `vfioX` corresponds to the PCI group the device belongs to.

In short, userspace applications communicate with the `vfio` character device using `ioctl()`s to obtain PCIe-level information about the devices, such as the contents of their PCI registers or their address ranges. It is also possible for applications to memory map the PCI device memory onto their own process space, due to the safety and address routing provided by the IOMMU. Once the device memory has been `mmap()`ed into a process, the kernel has no need to intervene in process-device communication. This allows for safe and feature-complete userspace drivers.

It is important to note that `vfio` works at the IOMMU device group granularity. This means that whenever `vfio` is activated for a specific device, total control of it is given to `vfio`: there is no possibility for any other system component to communicate with the device, except through the `vfio ioctl()` API. This makes it impossible for Kernel and userspace datapaths to coexist as actors on a particular NVMe, it has to be one or the other. In effect, the device becomes "invisible" to the Kernel drivers [88].

2.2.3 NVMe

Non-Volatile Memory Express (NVMe) is a storage communication protocol that uses PCI Express (PCIe) as its transport bus. It is designed for solid state storage devices (SSDs), as opposed to previous I/O busses such as Serial ATA (SATA), protocols such as the Advanced Host Controller Interface (AHCI), and abstractions such as block-based I/O that were designed on the assumption that the underlying hardware was a metal spinning disk drive [54].

The limits to improving legacy I/O interfaces and the need to create an SSD-specific protocol started being noticeable in the mid-2000s [54]. For example, starting with PCIe revision 1.0 and later, the bandwith available on the bus (at 1GB/s when using 4 PCI lanes [62]) was already higher

Storage Technology	Device	Latency	Cost ($/GB)
DRAM	SK Hynix DDR4 DRAM	80ns	∼7
NVDIMM	Intel Optane DCPM	300ns	∼5
Low-latency SSD	Optane SSD 905P	10μs	1
NVMe Flash SSD	Samsung 970 Pro	80μs	0.3
SATA Flash SSD	Intel 520 SSD	180μs	0.15

Table 2.1: I/O latencies and cost per Gigabyte for different storage technologies as of 2021, forming a clear hierarchy. We are interested in Low-latency SSDs in this thesis, which sit in the middle of this latency-cost spectrum. Despite recent advances, SSDs still lag behind memory-bus storage technologies by ∼2 orders of magnitude in the best case. Data from Wu, et al. [90]

than the state-of-the-art SATA interface released in 2008, revision 3.0, which supported up to 6Gb/s. Furthermore, the non-volatile solid-state hardware could support parallel operations much more easily than spinning disk drives due to the physical characteristics of the devices. Whereas SSDs can be conceptualized as bit arrays where the cost of accessing any address is always the same, HDDs have the constraints of seek time and its variation over the surface of the disk. The speed at which data could be addressed led the NVMe working group to include support for 64K I/O queues and 64K commands per I/O queue in the protocol specification [89]. It is theoretically possible to have 4 billion I/O commands in-flight when using an NVMe device, though normally the actual devices support much fewer queues than allowed by the spec; e.g. the hardware used for the work of this thesis only supports up to 31 data queues and 1 control queue [33]. Presumably, as of 2022, manufacturers have not yet found it cost effective to fabricate NVMe controllers with support for over 256 queues and queue depths of 2048 [71].

Due to the dominance of spinning disk I/O devices over decades, block device driver implementations were usually tuned to their behaviour, with the assumption that communication between the CPU and the disk would be a significant bottleneck in all applications. As described in Section 2.2.1, there were attempts at work optimization such as hardware interrupts and DMA, which free the CPU to work on actual processing tasks rather than waiting for the hardware. With the advent of NVMes, however, many of these assumptions and software implementations became obsolete. For example, the block interface's usefulness for Flash media and SSDs has been called into question over the past few years. The block abstraction defines devices as randomly addressable single-dimensional arrays that can only be accessed in chunks of memory known as blocks. Block

sizes are kernel- or filesystem-defined, and usually a multiple of the underlying spinning hard disk (HDD) physical sector size. Over time, they have "standardized" to a value of 4096 bytes [68].

The block interface was created and adopted at a time when the gap between primary and secondary storage media was much larger than what it is today [78]. The block abstraction allowed lower latency accesses to HDDs through I/O request buffering, which allows for asynchronous request fulfillment; it batches I/Os to reduce disk seek time and overhead; and it opens the door for abstract block scheduling algorithms that feed blocks into the drivers programmatically to minimize latency or maximize throughput. These very features, which justifiably improved the OS/HDD interface, have been considered a "tax" to be paid by flash media, as explained by Bjørling et al.: the block abstraction leads to capacity over-provisioning (excess SSD storage dedicated to garbage collection overhead), excess SSD device DRAM usage for flash page mapping tables, and host software complexity [16]. This is due to the physical architecture of flash media and the operation granularity it supports. Additionally, there is some evidence that NVMes have better latency characteristics for smaller block sizes and higher throughput for larger block sizes, decoupled from the classic HDD sector size of 512 bytes and its integer-multiple-sized blocks [15]. In short, a prominent reason why block interfaces are still used with NVMes is due to historical baggage and ease of NVMe onboarding for already-existing software that expects the block API.

Another standard HDD latency optimization technique that has been judged under a new light after the advent of NVMes is interrupt-based I/O. CPU polling for I/O completion, also called *busy waiting*, which would be considered a waste of cycles under the "slow-I/O" model, became a feasible approach for fast I/O, considering that hardware latency for I/O request completion started approaching single-digit microsecond values.

2.2.4 SPDK

The I/O polling approach was picked up by the Storage Performance Development Kit (SPDK), which describes its goals as the following [85]:

1. Moving all of the necessary drivers into userspace, which avoids syscalls and enables zero-copy access from the application.

2. Polling hardware for completions instead of relying on interrupts, which lowers both total latency and latency variance.

3. Avoiding all locks in the I/O path, instead relying on message passing.

Whereas normal applications would use the operating system's I/O path, which incurs I/O system calls (such as `write()`) and usually requires the OS to copy data from the NVMe's registers onto main memory, the goal of SPDK is to provide libraries that can be linked by high-performance and I/O intensive applications to have direct access to the devices' capabilities directly from their process space. SPDK also provides some programs that can support I/O intensive applications. One of these programs is `nvmf_tgt`, the NVMe-over-Fabrics target. NVMe-over-Fabrics (NVMe-oF) is a section of the NVMe specification that allows NVMe commands to travel over network fabrics (such as Infiniband) or over different communication channels locally, i.e. not PCIe. The `nvmf_tgt` is a process that acts as an NVMe-oF commands server, meaning that it can receive NVMe commands over (typically) a Unix domain socket. This is the program that `libvfio-user` leverages to implement userspace virtualized NVMe devices.

The benefits of polling for I/O instead of relying on interrupts, specifically for I/O intensive systems where latency is paramount, were suggested for NVMe and storage devices by researchers starting around 2012. Yang et al. [91] explicitly showed that wasting cycles on busy waiting could provide better performance than interrupt-based I/O as long as the underlying storage medium could attend requests synchronously at sub-5μs latencies. Their suggestion was to have two paths for I/O: a polling-based path that was to be used for small and frequent transfers; and a traditional interrupt-based path for large-size infrequent transfers. Notice that this recommendation is only valid for systems that attend I/O requests that match such a pattern. It has long been accepted that the trade-off for using polling-based I/O is the wasting of cycles that could be used for compute tasks; this does not prevent it from being the ideal solution when minimal latency is desired [77, 92]. There has been some effort recently in reducing this trade-off through the use of hybrid polling, where I/O wait is first scheduled as an interrupt after which polling is started. Choosing how long to wait before starting polling is the subject of active research, as performing heavy computations for heuristics can introduce unwanted latency and wipe out pure polling's performance advantage [56].

The SPDK components that perform the poll-based waiting are called *pollers*, lightweight thread abstractions that live inside application context environments called *reactors*. By definition, reactors exist at a one-to-one correspondence with CPU cores, and they can be understood as poller schedulers [84]. In the simplest configuration, pollers simply busy-wait on specific system memory areas where the NVMe device posts completed requests through DMA. To mitigate wasted cycles, pollers can be set to run on a schedule instead of all the time. Alternatively, the SPDK Scheduler framework can be used to automatically consolidate the lightweight threads onto a single core, thus allowing for other cores to go into an idle state. Further scheduling algorithms are pluggable into the framework [94].

As a comprehensive suite of libraries and tools, SPDK bundles implementations of software components that are readily available in the Linux kernel storage stack. Briefly, in a normal Linux system, the path of an I/O request from userspace goes through [38]:

1. the Virtual File System (VFS),
2. the block-based filesystem (ext4, btrfs),
3. the block I/O scheduler (`blk-mq`),
4. the NVMe driver.

SPDK has implementations of levels 2-4. Level 2 is a stub filesystem called BlobFS, as explained in Section 3.2.1. Level 3 is a component called `bdev`, an implementation of a block device interface. Level 4 is SPDK's asynchronous and lockless NVMe driver, which can also be linked directly into userspace applications, factors that differentiate it from Linux's implementation.

SPDK is an abstraction layer on top of NVMe hardware, whose memory has been made directly available to userspace through `vfio`. As such, it is ideally placed to provide multiplexing (virtualization) and caching to applications wishing to perform storage I/O. When hardware methods such as SR-IOV (covered in depth in Section 6.3.2) are unavailable for a particular device, SPDK can take over this task by having a single `nvmf_tgt` process in userspace that attends to multiple I/O generating processes. In this way, a single instance of each of the layers described above can be shared, minimizing resource consumption. The way this works is explained in Section 2.2.5. SPDK can also provide caching through one of `bdev`'s implementations, the Open CAS Framework Virtual

bdev [86]. In this configuration, the bdev component of an nvmf_tgt process is providing a caching layer between the hardware device and the I/O generating processes.

2.2.5 libvfio-user

In a standard QEMU/KVM configuration, QEMU takes care of the emulation of I/O and peripheral devices, whose actual operations are then offloaded to the host Linux kernel using standard system calls. For example, an I/O storage device seen by the guest is actually code being run by QEMU, which then transfers the read/write requests down to the host by calling the appropriate read() or write() system calls. Finally, the KVM host kernel module is responsible for the management of the "virtual" CPU, as explained before, and emulating the hardware MMU.

Figure 2.3a shows how this standard I/O loop works with a QEMU/KVM virtual machine using virtio. The guest userspace application makes an I/O system call to the guest kernel (1), which has a paravirtualized virtio driver that attends to the request. During the guest kernel control flow, a VM_EXIT event is triggered, which causes a context switch into the KVM code in host kernel space (2). Another context switch, this time back into userspace, is necessary for the I/O request to be attended to by the QEMU virtio backend (3). Finally, QEMU, which is simply a process in the userspace of the host, performs an I/O system call on the host kernel, causing yet another context switch (4).

It is possible and desirable to reduce the number of context switches from userspace to kernel space, since they incur a latency overhead on each I/O request. One approach to do this is shown in Figure 2.3b. With this configuration, it is possible to save 2 context switches from kernel to userspace, namely the return to the QEMU virtio backend and the system call from QEMU onto the host kernel's I/O stack. In their place, there are simple in-kernel-space function calls from KVM onto the vhost backend, and from vhost onto the kernel's block device stack and the NVMe driver further down the line. It is possible, however, to get the same reduction in system calls and kernel-to-userspace context switches with a different approach. Among these is SPDK-vhost-NVMe [93] (to be covered in Section 6.2); and the focus of this thesis, libvfio-user.

Libvfio-user is a library, currently used by the SPDK nvmf_tgt and a specific QEMU fork [61, 64], that allows for userspace inter-process communication (IPC) of vfio-style messages accross a channel such as a Unix-domain socket. From Section 2.2.2, the purpose of the vfio character device

was to allow userspace processes access to the PCIe devices directly through an `ioctl()` interface that also allowed for device memory mapping. `Vfio-user` builds upon this concept, by reusing most of the `vfio` API definitions, so that any userspace process can offer an emulated or virtualized PCIe device of any kind over an IPC channel.

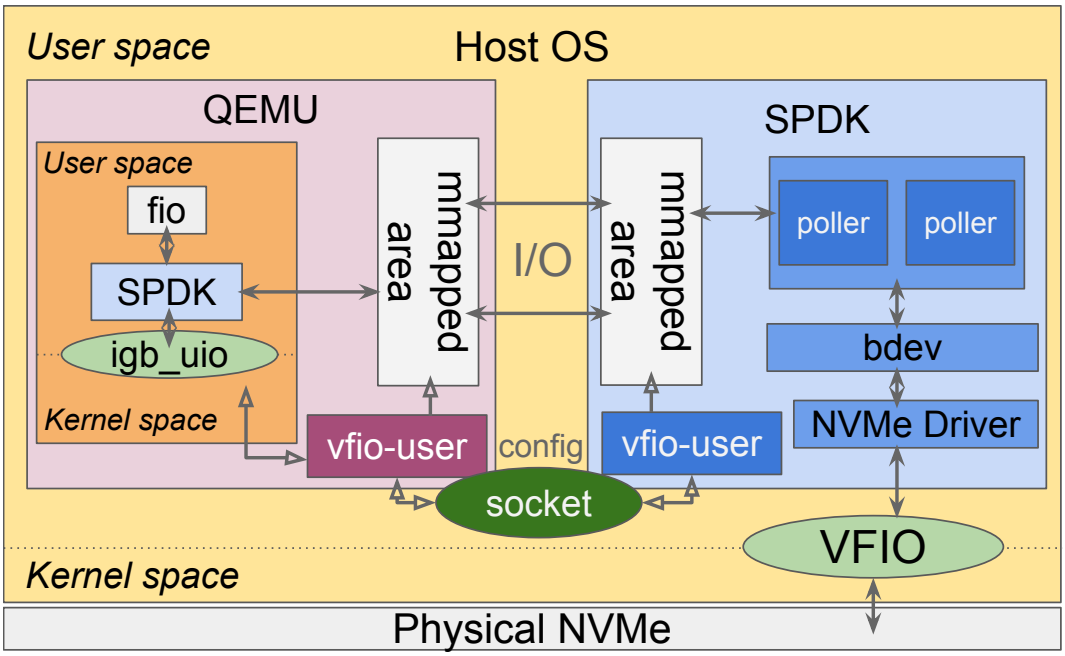


Figure 2.5: Frameworks such as `vfio-user` make it possible to implement PCI devices in userspace, with close to no overhead. Note that the "large" SPDK on the right side is a host-level userspace process, while the "small" SPDK on the left is a different SPDK instance running in the userspace of the guest.

Concretely, SPDK's `nvmf_tgt` offers a `vfio-user` server that listens for `vfio-user` messages over a Unix domain socket. At first, the messages have the goal of obtaining and providing information about an arbitrary PCIe device. `Nvmf_tgt` has the capacity to present any arbitrary PCIe-compliant device when responding to the `vfio-user` messages, but we are interested in the case where it transparently offers the information that it itself gets from the real hardware NVMe. It gets this information through the host `vfio` character device, as explained in Section 2.2.2. QEMU's `vfio-user` client connects to the Unix domain socket established by `nvmf_tgt`. Multiple connections can be attended by a single `nvmf_tgt` process; in a multiplexing scenario, all of the SPDK components on the right side of Figure 2.5 would be shared.

Once enough identifying information for the virtualized device has been passed through, SPDK sends a file descriptor over the Unix socket, which will be a memory mapping between the `vfio-user`

server and client userspace processes. This memory mapping works the in the same way as the host memory to NVMe controller interface; since the guest is seeing a "real" NVMe device memory space in the memory mapped area. Message passing and flow control are governed by NVMe protocol specifications, namely the presence of Admin Submission/Completion and I/O Submission/Completion queues [89].

`Nvmf_tgt` is a program with multiple layers in itself. At the top, the reactors (or pollers) continuously poll on the shared memory mapping to retrieve new I/O requests. They then pass them down the stack until they reach the block device layer of SPDK (`bdev`). Finally, `bdev` leverages the SPDK NVMe driver to submit I/O to the actual hardware, through the DMA provided by the host's `vfio`.

In terms of resource utilization, the memory available for the `nvmf_tgt` process and the CPU allowances for the pollers are explictly defined during `nvmf_tgt` initialization. SPDK's memory is backed by Hugepages, which can be used to effectively bound memory usage via manual allocation of a maximum desired value, through the `/proc/sys/vm/nr_hugepages` file that specifies the number of Hugepages in the system. Regarding CPU usage, there is a one-to-one correspondence between pollers and processor cores. Each poller is pinned to a CPU core and makes use of it exclusively, meaning that at least one of the cores in the machine has to be budgeted to be allocated to SPDK.

Thus, this `vfio-user` setup provides the QEMU/KVM guest with minimal-latency access to a virtualized NVMe SSD, by moving as much of the I/O stack into userspace as possible. This setup is shown in Figure 2.5.

3

Approach & methodology

Our objective in this thesis is to compare the performance characteristics of different configurations for providing NVMe-backed storage to a QEMU/KVM virtual machine. Our guiding principle is the fact that datacenter-scale tasks would greatly benefit, in terms of improved performance at scale, if we could reduce I/O latency at the bottom of the stack.

To answer our research question we implement a two-pronged approach. Firstly, we perform end-to-end comparisons of the latency captured by userspace processes, both inside the virtual machine, and on the bare-metal (so that we can obtain a baseline for the expected IOPS and latencies). The userspace applications we have chosen for this purpose are `fio` and RocksDB.

Secondly, we provide a detailed analysis of the latency between the layers of the `vfio-user` configuration (Figure 2.5). Briefly, we set up the simplest viable configuration that uses `vfio-user` to provide storage for a QEMU/KVM virtual machine, and we trace single I/O requests at different points of the stack to determine the average latecy in each layer (or groups of layers).

3.1 Choice of end-to-end applications

3.1.1 `fio`

the block I/O schedulerFio is an application designed for generating I/O requests as flexibly as possible, using an extensive set of configuration options that allow for synchronous and asynchronous I/O, using different backing storage APIs (such as Linux's POSIX-compliant `sync` engine, or the standard asynchronous `libaio`), different access patterns, and many more features [9]. It also has the convenient feature of reporting statistical summaries of latency and IOPS calculated from the generated workloads. We use the `fio`-reported values as-is for our results.

We use `fio` as a way to generate synthetic benchmarks. In order to facilitate the testing of different access patterns, number of threads, and queue size combinations, we make use of the `bench_fio` tool included in the `fio-plot` repository [58]. It generates `fio` configuration files dynamically based on a simple template, an example of which is on Listing 3.1. `Fio` is able to write directly to block devices without the need for a filesystem to be present. Unless otherwise noted, we make use of this feature for every benchmark and test.

```
[iotest]
rw=${MODE}
blocksize=${BLOCK_SIZE}
ioengine=${ENGINE}
iodepth=${IODEPTH}
numjobs=${NUMJOBS}
direct=${DIRECT}
group_reporting=1
invalidate=${INVALIDATE}
loops=${LOOPS}
write_bw_log=${OUTPUT}/${MODE}-iodepth-${IODEPTH}-numjobs-${NUMJOBS}
write_lat_log=${OUTPUT}/${MODE}-iodepth-${IODEPTH}-numjobs-${NUMJOBS}
write_iops_log=${OUTPUT}/${MODE}-iodepth-${IODEPTH}-numjobs-${NUMJOBS}
log_avg_msec=${LOGINTERVAL}
thread=1
```

Listing 3.1: Example `fio` configuration file used for benchmarking.

3.1.2 RocksDB

RocksDB is a popular NoSQL embedded key-value store based on the log-structured merge (LSM) tree data structure. It was created by Meta, originally based on Google's LevelDB, with the intent

of optimizing for fast flash storage and ramdisks, as well as multicore systems [18]. It is also used by Nutanix to handle system metadata.

Briefly, RocksDB attains fast write performance by initially performing all writes onto an in-memory data structure called the MemTable. Every write is treated as an append operation. Once the MemTable is full (at a configurable threshold), the data is flushed onto persistent storage (next level of the LSM tree) in Sorted String Table (SST) files, where key-value pairs are stored in order. Each level, just like the MemTable, has a configurable size. When the size is reached, background processes perform *compactions*, which remove deleted keys or outdated versions of valid keys [83].

The reason why we run RocksDB benchmarks as well, instead of relying on `fio` exclusively, is due to the fact that `fio` benchmarks are synthetic. The I/O patterns that `fio` generates can be very similar to the ones generated by any other arbitrary program, given a properly prepared `fio` configuration file; however, the overall resource consumption and environment of `fio` is much lighter and different to what RocksDB can effect on the system. For example, RocksDB runs compaction threads periodically, which can affect I/O performance in unpredictable ways. This is hard to simulate with `fio` alone.

3.2 Storage virtualization configurations

We study four popular configurations of I/O stacks for a QEMU/KVM virtual machine on Linux. These are the following:

1. **Bare metal configuration**, which is used as an ideal scenario to determine the upper bound for latency and throughput for the other configurations (Section 3.2.1).

2. `Libaio` **configuration**, where the application generates I/O calls from inside QEMU, through system calls, QEMU provides hardware emulation, and I/O is performed through system calls to the host kernel (Section 3.2.2).

3. **Passthrough configuration**, where the application generates I/O inside the QEMU guest through SPDK, and QEMU itself communicates with the NVMe hardware through `vfio`, (Section 3.2.3).

4. **`Vfio-user` configuration**, where the application generates I/O inside the QEMU guest through SPDK, and I/O is passed through a new layer emulating the NVMe device via `vfio-user` (Section 3.2.4).

3.2.1 Bare metal configuration

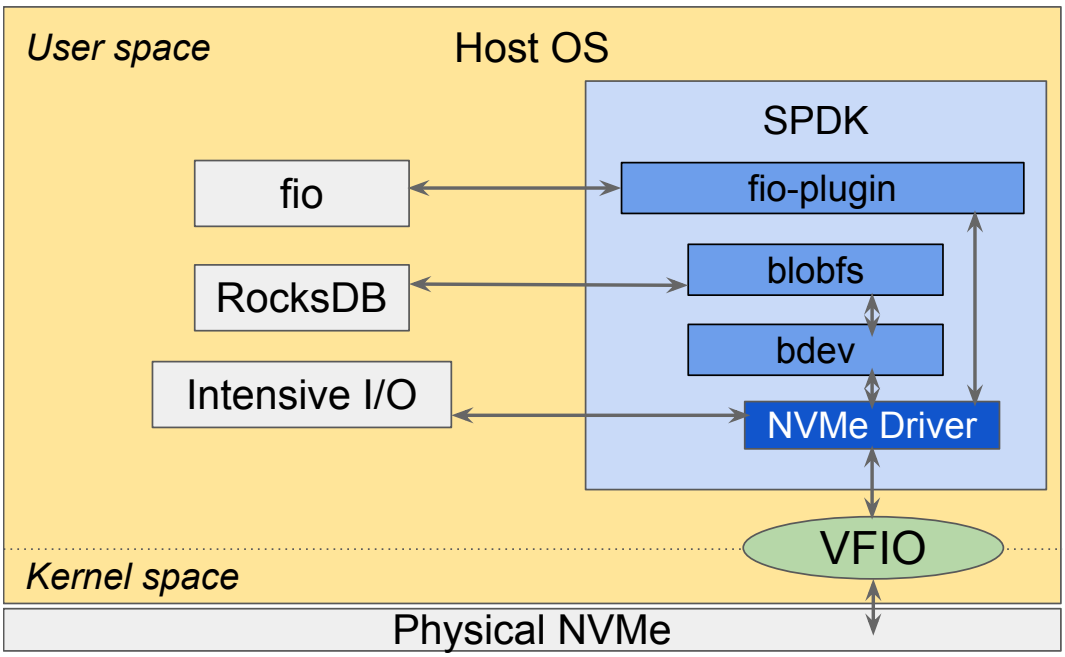


Figure 3.1: Bare metal configuration. Applications run inside the host OS and generate I/Os directly to the NVMe drive, through SPDK.

Figure 3.1 shows the Bare metal storage configuration. This configuration does not use virtualization. It is meant as a best-case scenario, over which components are added to construct the next configurations. We consider that the configurations involving virtualization are "good" if their IOPS and latency values are quantitatively close to those attained by this configuration.

The I/O generating applications (i.e., `fio`, RocksDB) run as userspace processes inside the host OS. The application uses SPDK to perform write and read requests. For `fio`, we use the `spdk-fio` plugin (as a statically loaded library), allowing `fio` to to call SPDK functions directly from its process space. For RocksDB, we use SPDK's fork of RocksDB[49] that integrates with BlobFS, an SPDK-provided filesystem meant to replace the OS-provided filesystem. BlobFS is built on top of blobstore, SPDK's power-fail safe block allocator. As of 2023, BlobFS only supports flat namespaces (i.e. directories are not supported), and writes to a file must always append to the end

of the file [92]. BlobFS (the underlying storage provider) is not to be confused with BlobDB, which is a variation of RocksDB that optimizes for large values and key/value separation (e.g., similar to the WiscKey [59] and Bourbon [32] key-value stores). SPDK is configured to communicate with the physical NVMe by using `vfio`, using the mechanisms described in Section 2.2.2.

3.2.2 `Libaio` configuration

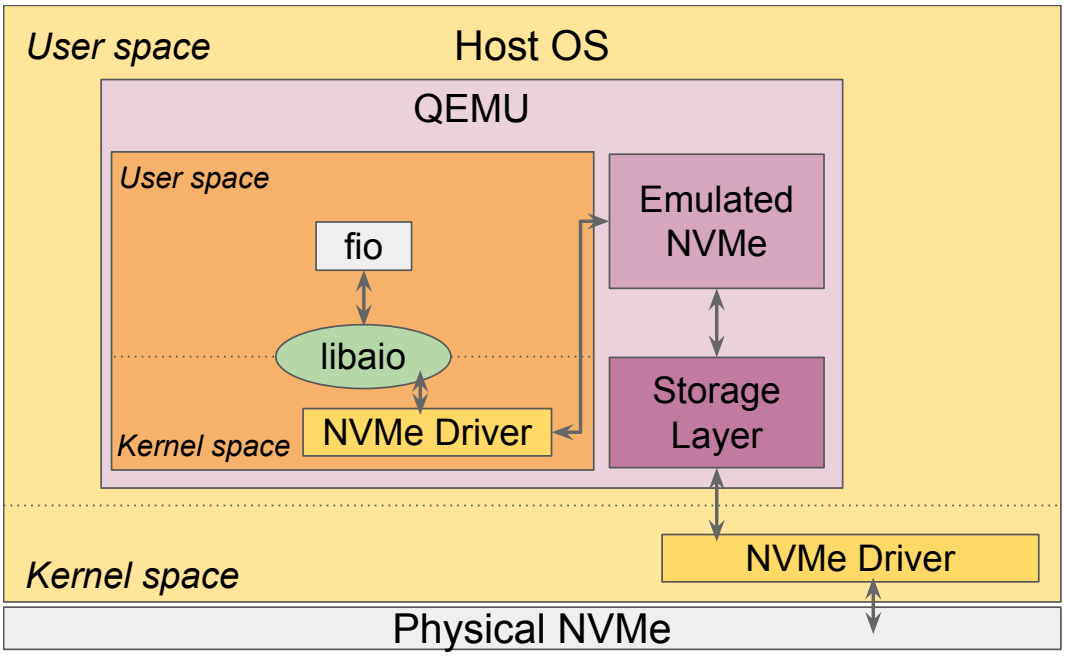


Figure 3.2: `Libaio` configuration. Higher latency is expected in this configuration compared to bare metal, passthrough, and `vfio-user`, as the I/O calls incur context switches.

Figure 3.2 presents the `libaio` configuration. This is a configuration where the NVMe device is accessed via system calls from QEMU to the host OS. The goal of this configuration is to evaluate the performance of a scenario where the NVMe drive is accessed without bypassing the kernel, in order to evaluate the overhead of system calls when the target application runs inside a VM, and obtain a lower bound on performance. This is an "out-of-the-box" configuration, with no optimization at all.

The application generates asynchronous I/O requests via `libaio` [34]. Note that the full QEMU I/O stack is traversed in this case: the guest is using an emulated NVMe device which in turn communicates with the QEMU I/O abstractions. QEMU makes use of a QCOW file which represents a virtual disk image backing the storage for the emulated NVMe device. This QCOW image is simply

a file in a file system created on the NVMe hardware as managed by the host I/O stack and kernel NVMe drivers. Accesses to this file incur further context switches.

3.2.3 Passthrough configuration

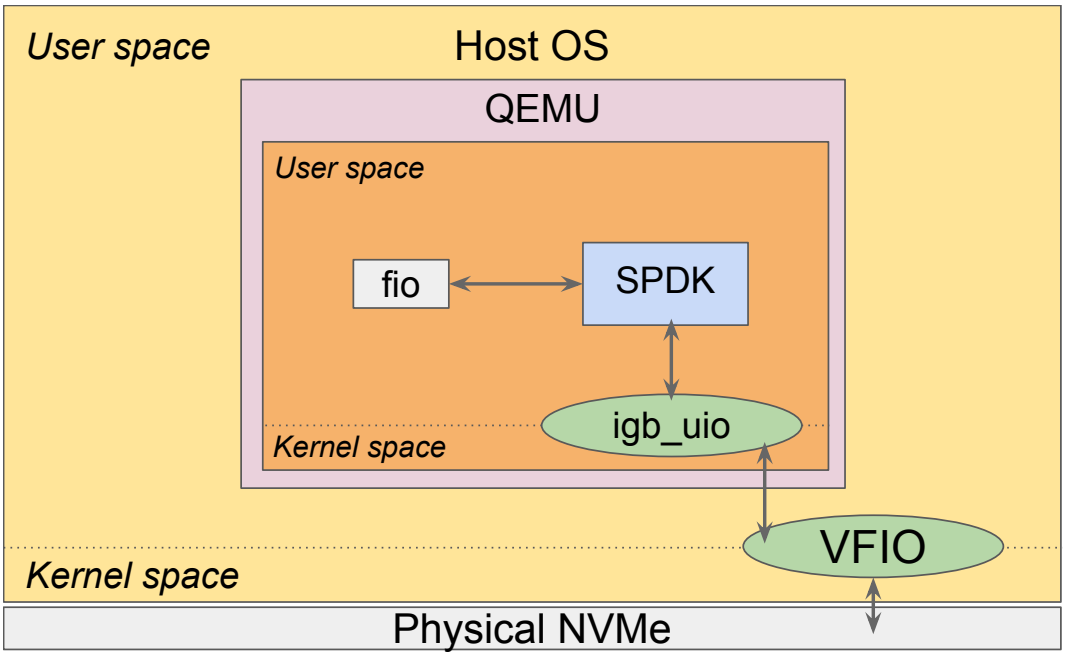


Figure 3.3: Passthrough configuration. Applications run in the guest OS, but they do not perform system calls to access the NVMe hardware.

Figure 3.3 presents the Passthrough configuration, which runs the I/O generating applications inside the VM, but uses a QEMU/KVM feature called *device assignment* to access the NVMe drive.

The application runs inside the guest OS. A configuration similar to Bare metal (i.e., Figure 3.1) is set up inside the guest, where SPDK is used to access the NVMe hardware. The only difference inside the guest is that the Linux facility providing DMA to the hardware is not `vfio`, but a custom kernel module developed by DPDK[22] called `uio_igb`. To perform device assignment, the QEMU process requests exclusive access to the NVMe device from the host kernel using `vfio`. Crucially, it is QEMU itself that takes care of the `ioctl` calls to the character device. QEMU then exposes the physical NVMe of the host directly to the guest.

It is worth mentioning that QEMU additionally supports a different mode that uses `vfio` as well, which is called simply "nvme" in the libvirt documentation[52]. This mode adds some of the QEMU I/O layer code on top of the `vfio` interaction with the hardware NVMe, which can support features

such as QCOW image layering, but we considered it unnecessary towards the goal of measuring performance.

3.2.4 `Vfio-user` configuration

Figure 2.5 shows the `vfio-user` configuration. As before, the application runs in userspace inside the QEMU guest. Note that this version of QEMU is different from the ones used for the previous configurations. We use a QEMU fork called multiprocess-QEMU,[64] where the Oracle team have implemented changes so that arbitrary userspace processes can provide services to the guests. We use the `uio_igb` kernel module inside the guest, analogous to `vfio` in the host.

In the passthrough configuration (Section 3.2.3), QEMU was able to speak directly to the NVMe due to the actual hardware being available through `vfio`. In the `vfio-user` configuration, however, there is no direct access by QEMU. A new layer is introduced, namely the `vfio-user` client on the QEMU side. As far as the guest is concerned, a real NVMe drive is providing I/O facilities. In effect, this NVMe is software that interfaces with SPDK through a memory-mapped region.

As described in Section 2.2.5, QEMU interacts with SPDK's `nvmf_tgt` through a control Unix socket before establishing a shared memory mapping for the actual passing of the data. SPDK then takes care of communicating with the real NVMe hardware through the host's `vfio` facility.

3.2.5 Additional configurations

It is to be noted that during the course of experimentation, some additional configurations were tested but their results were not collected and/or plotted to the same extent as the ones described above. The configurations chosen were considered enough to obtain meaningful comparisons and upper and lower bounds on performance. We briefly and roughly report the additional setups and their observed latencies at 1 thread and queue depth 1:

- Malloc bdev. Based on `vfio-user` but using a ramdisk, instead of the real NVMe hardware. **Avg random read:** $1.96 \pm 0.28\mu$s ; **Avg random write:** $2.88 \pm 0.37\mu$s
- QEMU SCSI. Based on `libaio`, but using the QEMU paravirtualized `virtio-scsi` drivers. **Avg random read:** $40.08 \pm 1.72\mu$s ; **Avg random write:** $46.78 \pm 4.45\mu$s

3.3 Layer-by-layer latency measurement with vfio_user_snoop

While end-to-end latency is a valuable metric, it is difficult to understand where the time is spent in the various layers described in Section 3.2 without a detailed breakdown of where time is spent. To this end, we develop `vfio_user_snoop`, a new open-source latency tool that provides detailed latency measurements at each level of the virtualized stack for the `vfio-user` configuration. `Vfio_user_snoop` works as follows.

For the kernel NVMe driver, we create a `bpftrace` [23] program, modelled after `nvmelatency` [41]. The program uses the `nvme_setup_cmd` and the `nvme_complete_rq` kernel tracepoints to output the duration of each NVMe *write* and *flush* I/O request on a new line of text over standard output. To accomplish this, each request ID is stored in a BPF map with a nanosecond timestamp which is then subtracted from the elapsed time once the same ID is observed in the `nvme_complete_rq` tracepoint handler. The complete text output is processed with an `awk` script that calculates the standard deviation, minimum, and maximum values on the dataset. We typically collect between 150K and 250K samples for each I/O request type (*write* and *flush*) for the statistical calculations. We observed an overhead of $\sim$6 to $\sim$10μs per I/O when the `bpftrace` probes are enabled.

In addition, we enable the SPDK tracing framework with the `bdev` and the NVMe driver tracepoints active. After this, the built-in `spdk_trace_record` tool was used to store binary trace events into a file that was then converted into parseable text with `spdk_trace`. This output is processed with the same `awk` script that calculates the statistical summary. We did not detect any statistical difference in latency when the SPDK tracing was enabled.

We create a new, specific QEMU I/O configuration for `vfio_user_snoop`, based on the `vfio-user` configuration. This new configuration is shown in Figure 3.4. It differs from the `vfio-user` configuration in Figure 2.5 by the absence of an SPDK instance inside the guest, meaning that the guest kernel NVMe driver is active. The motivation for this was to have `fio` use the synchronous POSIX I/O API to have the simplest possible traceable datapath on the guest side.

Figure 3.4 shows the three layers where latency is reported, highlighted in different colors (i.e., green, orange, and blue, for latency inside userspace, the guest NVMe driver, and SPDK respectively).

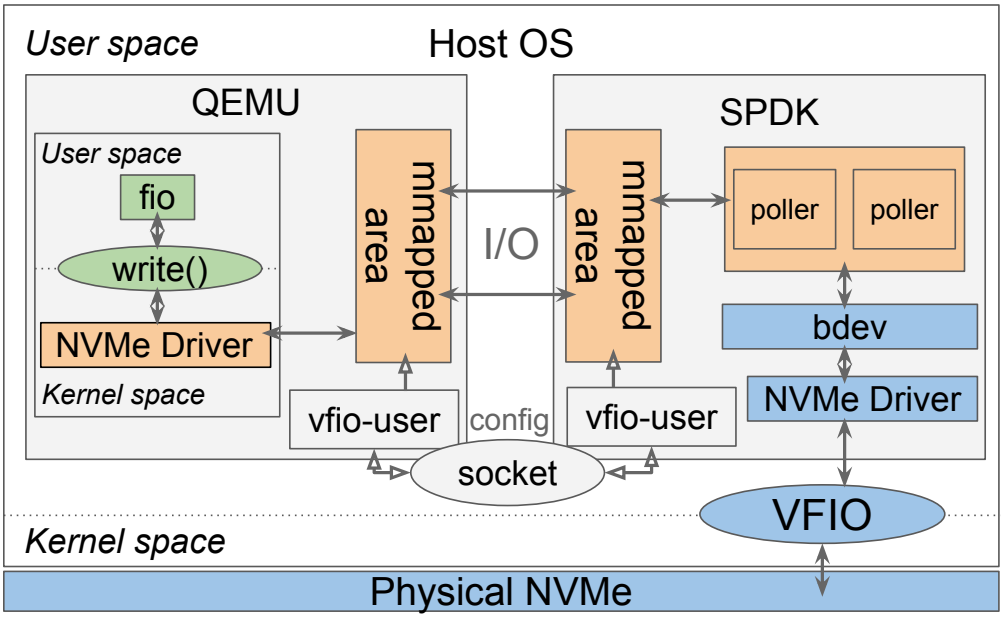


Figure 3.4: Layer-by-layer latency measurement for three layers: 1) application (`fio` here) to the guest NVMe driver (green); 2) guest NVMe driver to SPDK `bdev` (orange); and 3) SPDK `bdev` to NVMe (blue).

First, `vfio_user_snoop` measures end-to-end latency inside userspace (in green). Second, `vfio_user_snoop` measures the kernel NVMe driver latency. I/O requests sent from `fio` in guest userspace go through the guest kernel I/O stack, crossing VFS, `blk-mq`, and the NVMe driver. The driver sees a memory area that has been allocated by QEMU based on the settings transmitted through the `vfio-user` socket. This interface between the Linux NVMe driver and the `mmap()`ed region is the first spot at which the `vfio-user` configuration takes effect. There is no visibility on data going through this memory region, so the best locations to trace are at each end, namely right before memory accesses are performed at the NVMe driver, and right after requests are picked up by the SPDK pollers. After this point, the QEMU/KVM memory management emulation takes over to translate guest kernel physical memory addresses into host physical memory addresses.

Finally, `vfio_user_snoop` measures the SPDK `bdev` callback latency. The SPDK pollers are monitoring requests on the `mmap()`ed memory region through which they communicate with QEMU. Once they see something in the submission queues, they hand it over immediately to `bdev`. The latency measured here is the last step: the SPDK block device layer overhead, plus the SPDK NVMe driver, which speaks with the real NVMe hardware through `vfio`.

4

Experimental setup

4.1 Hardware and software environment

4.1.1 Hardware

All the experiments were executed on a DISCS Lab [10] machine, part of a 3-machine cluster available for use by DISCS Lab students. Table 4.1 displays the hardware specifications of the machine. This was a shared server, meaning that some unrelated and potentially disruptive programs were installed and had the potential to disrupt the benchmark runs. Some of these programs were impossible to get rid of due to IT administrative and security policies. Of note, the server was under control of IT Puppet scripts that would overwrite some `AppArmor` configurations every 30 minutes. To mitigate these factors, the following actions were taken before any benchmark execution:

`discslab-server1` **Hardware Specifications**	
CPU	Intel(R) Xeon(R) Gold 6240L CPU @ 2.60GHz
CPU Features	36 Cores, 2 NUMA Nodes, Hyperthreading disabled
RAM Amount	12×64GiB = 768GiB
RAM Specs	3200 MHz DDR4 DIMM
Target NVMe	375GiB Dell Express Flash NVMe P4800X
NVMe IOPS Ratings (4KB)	Random Reads: 550K, Random Writes: 550K

Table 4.1: Hardware specifications of the shared laboratory server (`discslab-server1`) used to perform all benchmark runs. NVMe ratings are from the manufacturer's specifications [33].

- All machine users, aside from the benchmark-running user, were logged out and their processes were `kill`ed
- SSH logins from any user, apart from the benchmark-running user, were disabled for the duration of the runs
- Any long-running services that were not explicitly required by administrative or security policies were stopped (e.g. `dockerd`, `containerd`).
- Source folders for SPDK, `libvfio-user`, `fio`, and DPDK were shared between host and guests using the `virtio-9p-device` virtual file system [72].

An important goal when setting up the server for the tests was ensuring result reproducibility. One place where variability can come from is the CPU frequency. Due to modern CPU power saving measures, C-states and P-states [37] can dynamically change the CPU frequency in ways that may appear random, which can affect results. This is because each CPU cycle ends up having a variable duration, which might affect latency measurements. To prevent this issue, C-states and P-states were disabled, more detail can be found in Section 4.2.

As explained in Section 2.2, a prerequisite for using `vfio` is for the host system to have an IOMMU, and for it to be enabled. This usually requires both a BIOS configuration setting and a kernel boot parameter [81]. In our case, since we had no physical access to the machine due to security policy, we requested such configuration changes from the IT team. A roadblock that we faced was regarding the `vfio` kernel facility inside the guest OS. In theory, QEMU supports a virtual IOMMU (vIOMMU) [65] which should allow for `vfio` to be used inside the guest. However, despite our best efforts, including following guides for nested virtual machine setups, we managed to enable

vIOMMU, but `vfio` was never enabled in the guest kernel. `Uio`, which is available in Linux kernels by default according to the documentation [53], also did not work despite explicitly enabling the kernel module using `modprobe`. This is the reason why DPDK's `uio_igb` module was necessary.

As final hardware note, we bring attention to the limitations of commercially available NVMe hardware with respect to the number of queues. As mentioned in Section 2.2.3, the NVMe spec allows for up to 64K queues. Our NVMe, however, only supported 31 data queues and an additional administrative queue. Despite this information being present in the device specifications [33], we did not discover this until we attempted to execute `fio` benchmarks with 32 and 36 threads, where `fio` was unable to run. In our settings, each thread makes exclusive use of a single queue. We had chosen 36 as the highest number of threads, because it would have used the total number of cores in the system.

4.1.2 Software

There are multiple ways to execute the QEMU userspace process. It is possible to simply run the QEMU binary corresponding to the system to emulate (e.g. `qemu-system-x86_64`), and passing command-line flags to enable desired features. These flags are comprehensive [26] and they grant the most freedom of choice, allowing easy access to experimental features (such as the multiprocess QEMU flags needed to run `vfio-user`). An issue with using the bare QEMU command line invocation is that it is difficult to maintain each virtual machine configuration under version control. Despite the fact that each virtual machine invocation can be wrapped in a specific script file and version controlled that way, Git's tooling is usually oriented towards changes on independent lines. This is doable with scripts, but we considered it suboptimal.

The alternative we chose is `libvirt` [17]. `Libvirt` is a suite of tools intended to support different kinds of virtualization technologies through a set of command-line applications and a standard XML-based language. `Libvirt` allows for XML schema documents to be full descriptions of containers or virtual machines for Docker, LXC [40], VirtualBox, etc. It evidently provides QEMU/KVM support. Many of the command-line-enabled features of QEMU are outright supported with specific XML tags. In rare cases, when QEMU features have not been yet added to the language (such as the `vfio-user` options for multiprocess-QEMU), `libvirt` XML schemas support passing command line arguments to the executable directly.

```
<domain type="kvm" xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
    <name>passthrough</name>
    <memory unit="GiB">16</memory>
    <vcpu cpuset="0-17" placement="static">18</vcpu>
    <devices>
        <emulator>/usr/bin/qemu-system-x86_64</emulator>
        <!-- ... -->
        <!-- disk under test -->
        <hostdev managed="yes" mode="subsystem" type="pci">
            <source>
                <address bus="0xbc" domain="0x0000" function="0x0" slot="0x00"/>
            </source>
            <driver name="vfio" />
        </hostdev>
        <!-- ... -->
    </devices>
    <!-- ... -->
</domain>
```

Listing 4.1: Excerpt from the `libvirt` XML virtual machine schema file for the passthrough configuration, described in Section 3.2.3.

One of the tools included with `libvirt` is `virsh`, a command-line program that manages startup and stopping of virtual machines based on their schema files. It also supports accessing the virtual machines' serial console through a virtual interface. With the `libvirt` tool suite, instead of running a long command with multiple complex flags passed to the QEMU binary, it is enough to run `virsh <vm_spec_file>.xml`, and the virtual machine schema defined in the file will be executed.

A notable pitfall when using `virsh` is that the documentation guides users by default to employ the *volume manager*. The volume manager is a feature that can create *storage pools*, or groups of automatically managed QCOW files, as backing storage for the virtual machines. There was a lot of time lost when attempting to have the pools use the real NVMe hardware. Storage pools do not support this, and it is not their intended purpose. Virtual machine XML schemas do not require storage pools to be used, and the specific storage configurations that this project needed were explicitly defined in each one of the XMLs for each machine.

It is important to note SPDK's requirement to use Linux HugePages. HugePages are a kernel feature available since 2.6, that allow Linux to allocate memory pages of sizes larger than the default 4KiB. The point of HugePages is reducing cache misses and TLB lookups [51]. SPDK specifically has a minimum memory allocation requirement in HugePages, and makes use of 2MiB and 1GiB HugePages if available.

4.2 Provisioning and automation

It was quickly evident that some form of automation of benchmark environment setup would be beneficial, due to the reduction of manual errors in e.g. data collection or configuration.

We make use of Ansible [43] to execute the preparation tasks for the server to get reproducible results in the benchmarks. Ansible is a robust open-source configuration management tool, where desired machine provisioning states can be specified in a declarative manner and executions are idempotent. In short, a collection of YAML-formatted lists of desired dependencies, files, or any other kind of system states can be specified, which are called *playbooks* and *roles*. We created a playbook and corresponding roles that deal with the following tasks:

- Disable processor C-states by keeping a file descriptor open on `/dev/cpu_dma_latency`.
- Disable CPU P-states by setting the kernel `pstate` drivers (`/sys/devices/system/cpu/intel_pstate/status`) to passive mode, selecting the `userspace` CPU frequency governor configuration, and specifying the nominal frequency of the processor (i.e. 2.60GHz, see Table 4.1) by using `cpupower`.
- Implement the mitigation actions specified in Section 4.1.1.

Every virtual machine configuration required manual work to initially set up and get to a working stable state. In order to get a faster feedback cycle, it was necessary to implement automation scripts that would clean up the environment created by a single virtual machine and get a clean slate to create another one. Many of the steps necessary to set up a specific virtual machine were repetitive terminal commands. These were collected into a series of scripts that can execute the following tasks with a single invocation:

1. create and destroy QCOW backing files for the guest OS,
2. drop kernel virtual memory caches,
3. `sync` or dump buffered data to disks,
4. allocate HugePage memory and relinquish control of the NVMe device from the kernel through `vfio`,

5. for the `libaio` configuration only, format and mount the NVMe device, since it will hold the QCOW storage file

A further step in aiding automation was the usage of Ubuntu cloud images [66] and `cloud-init` for guest configuration. `Cloud-init` is a set of tools that allow Ubuntu images, including virtual machine images, to be provisioned on boot. It is widely used by cloud service providers for OS configuration [42]. The cloud image of Ubuntu 20.04LTS was chosen as the base from which the operating system images for every other virtual machine were created. Using `cloud-init`, some variables were passed into the guest filesystem to aid in the automation of test results, and a script was created to run on guest boot to trigger tests automatically. Test results are stored in one of the shared guest-host directories, which makes it easy for us to collect, analyze, and plot test results. With the scripting infrastructure, a single script can set up every virtual machine in turn, while each virtual machine executes its benchmarks and outputs results to a shared folder, without any manual intervention.

These scripts and provisioning configuration files, in addition to the virtual machine XML specification files, are publicly accessible on the McGill Data-Intensive Storage & Computer Systems (DISCS) GitLab repositories [67].

5 Results

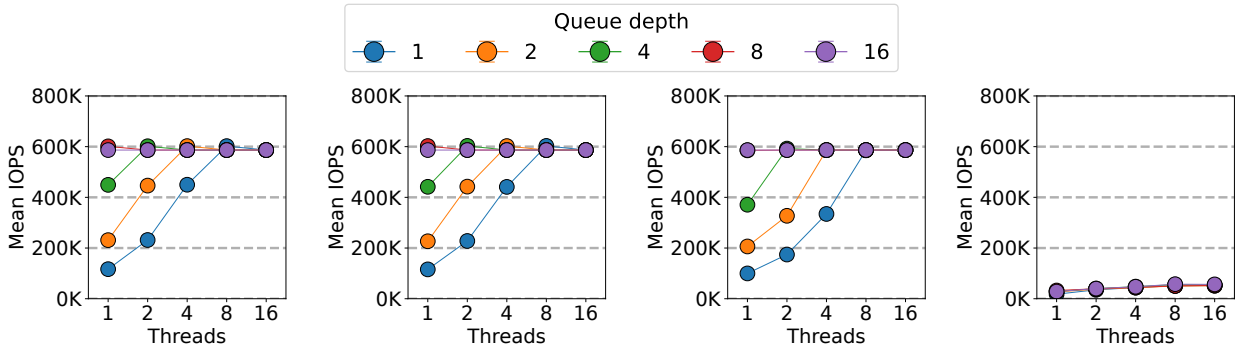
We compare the IOPS and latency of the four configurations described in Section 3.2. We measure the performance in microbenchmarks generated with `fio`, as well as end-to-end performance in RocksDB.

5.1 `Fio` microbenchmark results

We use `fio 3.30`. We perform random and sequential reads and writes directly on the NVMe block device, i.e. no filesystem is involved. We vary the I/O queue depth from 1 to 16 and vary the number of threads from 1 to 16. Each combination was benchmarked for a duration of 60 seconds.

5.1.1 Reads

Figure 5.1 shows the mean IOPS for the random read benchmark. Notice the hardware upper bound of ~600K IOPS for the baremetal configuration in Figure 5.1a. For any queue depth greater than 8, close to full NVMe saturation is reached even with only 1 thread submitting read requests. Conversely, for 1 thread, saturation is reached at queue depth size 8. Results are similar in Figure 5.1b and Figure 5.1c. For all configurations, the IOPS growth trend is as expected, with higher IOPS as threads increase. The rate of IOPS growth between baremetal and passthrough is very similar, but the rate of IOPS growth is slower for `vfio-user`. This is evident if we follow the mean IOPS values for queue depth 1. For passthrough at 2 threads we observe ~230K IOPS and ~190K for `vfio-user`. At 4 threads, passthrough performs ~440K iops while `vfio-user` manages ~330K. Due to the high cost of context switches, the `libaio` configuration provides peak performance that is on average 30x lower than the other three configurations.

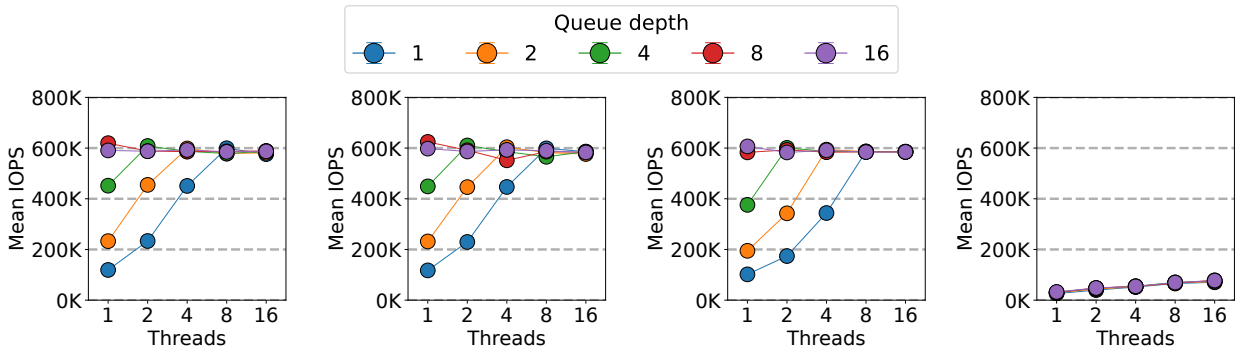


(a) baremetal (b) passthrough (c) `vfio-user` (d) `libaio`

Figure 5.1: Mean IOPS per number of threads and queue depth results for `fio` random reads, 4KB block size.

Figure 5.2 shows mean IOPS for sequential reads. The same behaviour as the one shown on Figure 5.1 is present here. In fact, the performance values are exceedingly similar; including the large differences in performance between `vfio-user` and baremetal/passthrough, where the mean IOPS at queue depth 1 and 4 threads differ by between ~80K and ~100K IOPS. An interesting anomaly observed in the sequential read results are the pronounced performance "peaks" at 1 thread and queue depth 8 for baremetal and passthrough, which project slightly over the 600K IOPS line. Since it is impossible for the hardware to support a higher number of IOPS than its rated spec, and it only happened in these two benchmarks, we consider that it must be an extrapolation or

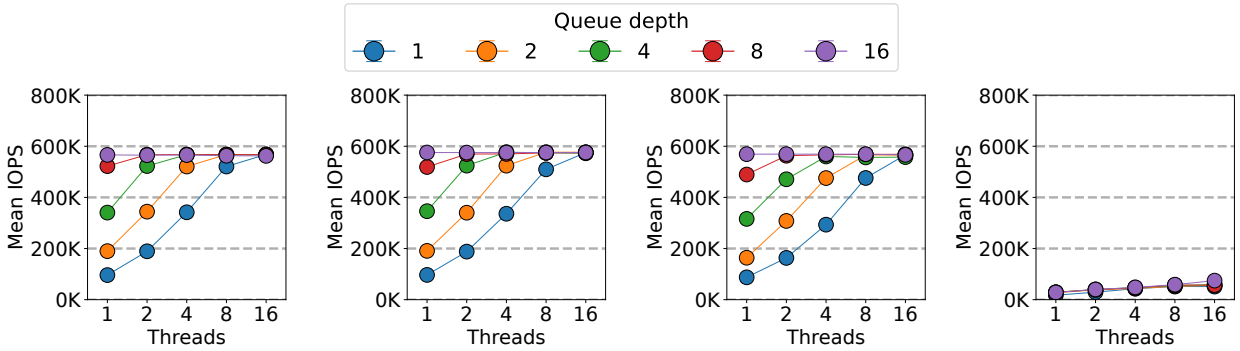
calculation error on the part of `fio`. Another anomaly to point out is the discrepancy between the passthrough IOPS value for 4 threads and queue depth 8; since that workload is fully able to saturate the hardware and attain peak IOPS. We consider that this must be an artifact of the testing environment, where possibly some background process triggered and affected the results of that benchmark.



(a) baremetal (b) passthrough (c) `vfio-user` (d) `libaio`

Figure 5.2: Mean IOPS per number of threads and queue depth results for `fio` sequential reads, 4KB block size.

5.1.2 Writes

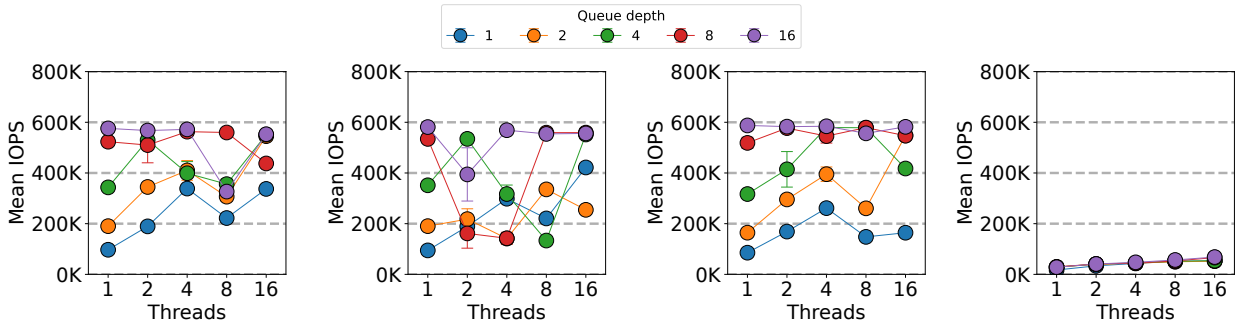


(a) baremetal (b) passthrough (c) `vfio-user` (d) `libaio`

Figure 5.3: Mean IOPS per number of threads and queue depth results for `fio` random writes, 4KB block size.

Figure 5.3 shows virtually identical trends as Figure 5.1. However, the differences in IOPS between different configurations, except for `libaio`, are slightly harder to discern. A possible reason why this is the case is due to the in-NVMe writing time taking up a larger part of the total request

duration, making the software layers above have a smaller contribution to the total latency. It is noticeable how `vfio-user`'s performance is scarcely different than baremetal and passthrough's. If we compare the mean IOPS values for the same queue depth and number of threads as we did for random reads, namely queue depth 1 and 2 threads, it is easy to see that all configurations (except for `libaio`) perform between ~190K and ~200K IOPS.



(a) baremetal (b) passthrough (c) `vfio-user` (d) `libaio`

Figure 5.4: Mean IOPS per number of threads and queue depth results for `fio` sequential writes, 4KB block size.

Figure 5.4 displays the `fio` results we obtained when running sequential write benchmarks. Despite multiple runs, at different dates and times of day, while ensuring result reproducibility and interference minimization strategies were put in place, we were not able to obtain data resembling the expected patterns presented in the previous figures. The only consistent result for this benchmark is seen in Figure 5.4d, for the `libaio` configuration, that keeps its pattern of providing much fewer mean IOPS than all other configurations. The baremetal and `vfio-user` configurations show data roughly suggesting improving average performance as queue depth increases, but the pattern doesn't hold in all cases, as can be seen for queue depth 16 at 8 threads in the baremetal configuration. Passthrough shows the most widely distributed results so far, where not even a performance hierarchy being clearly visible. We have no systemic explanation for this behaviour, suggesting a bug in `fio` 3.30's implementation of the sequential write benchmark.

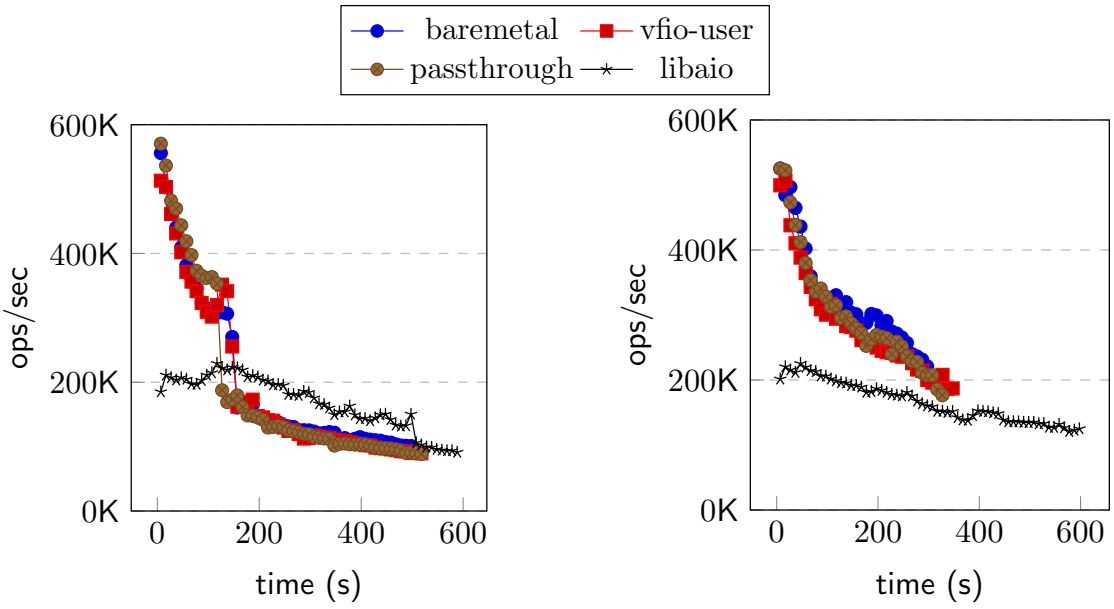
5.2 RocksDB end-to-end application results

To analyze the end-to-end effect of the four storage configurations, we set up RocksDB, a popular key-value store, with each of the four back-ends. RocksDB was run inside each VM, via the `db_bench`

tool [49]. We use the YCSB [27] benchmark, workloads YCSB-LOAD (sequential inserts), YCSB-A (1:1 read:write ratio), and YCSB-C (read-only). For YCSB-A and YCSB-C we use a uniform key distribution. Each workload performs read/write operations on 100 million key-value pairs (16B keys, 1KB values) and runs for a duration of 10 minutes. To vary the degree of parallelism of the workloads, we vary the number of compaction threads from 1 to 10. The number of worker threads is fixed to 1.

SPDK's blobfs is tested on a specific version of RocksDB, namely `6.15.fb`[49], which doesn't support time-based benchmarks for sequential reads (benchmark ends as soon as the total number of keys has been read). This required us adding a patch into the `db_bench_tool.cc` code from RocksDB.

A single database was reused for the first 4 benchmarks, `readrandomwriterandom` and random writes were performed on fresh databases. Results presented are in operations per second as reported by `db_bench`, no other monitoring tools were used at this point.



(a) YCSB-LOAD, 1 compaction thread

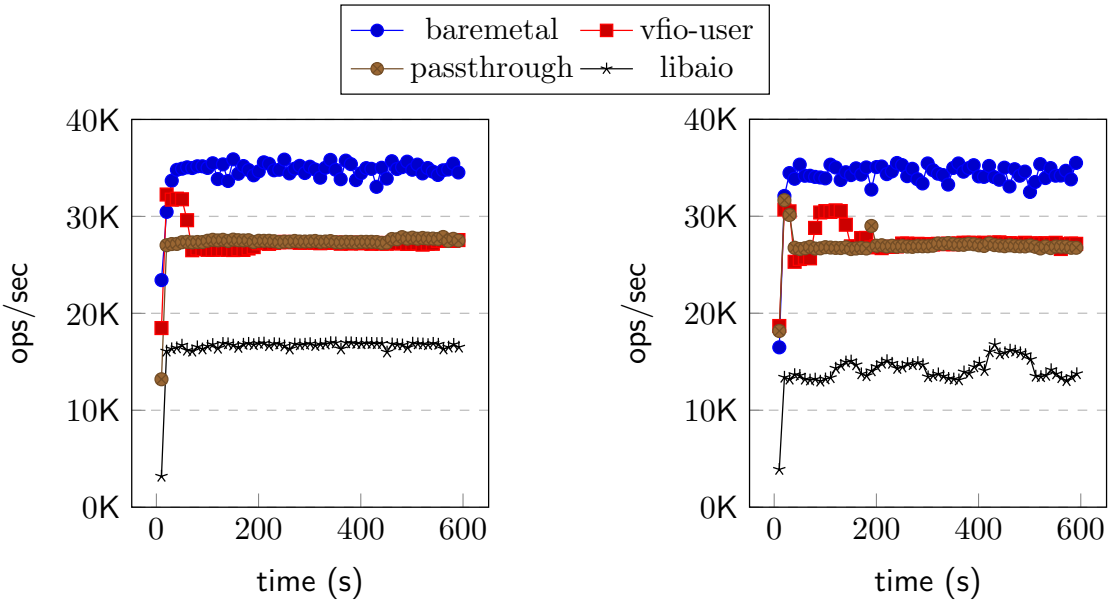
(b) YCSB-LOAD, 10 compaction threads

Figure 5.5: RocksDB sequential fill operations per second over time, using 1 thread

Figure 5.5 shows operations per second over time for sequential insertions as reported by `db_bench`, for a database running with 1 compaction thread (5.5a) and 10 compaction threads (5.5b). During the first minute, there are no apparent performance differences between baremetal,

passthrough, and `vfio-user`, while `libaio` is clearly capping at around 200K ops. The downward trend of both graphs is expected due to the design characteristics of RocksDB, as more keys are inserted, insertion time increases. At around 160 seconds, we see a precipitous decrease in ops/sec in 5.5a, caused by a compaction task triggering after a specific number of keys have been inserted. After this point, `libaio` reports higher ops/sec until it also triggers a compaction thread at around 500 seconds. The lower throughput of `libaio` is the reason for the compaction triggering later.

The precipitous decline does not happen when the database has 10 compaction threads. Figure 5.5b shows the expected progression of ops/second when performing sequential insertions in an LSM tree. Baremetal provided higher throughput than passthrough, which in turn provided higher throughput than `vfio-user`, during the period of around 100 to 300 seconds. This is consistent with the results seen in the `fio` benchmarks.



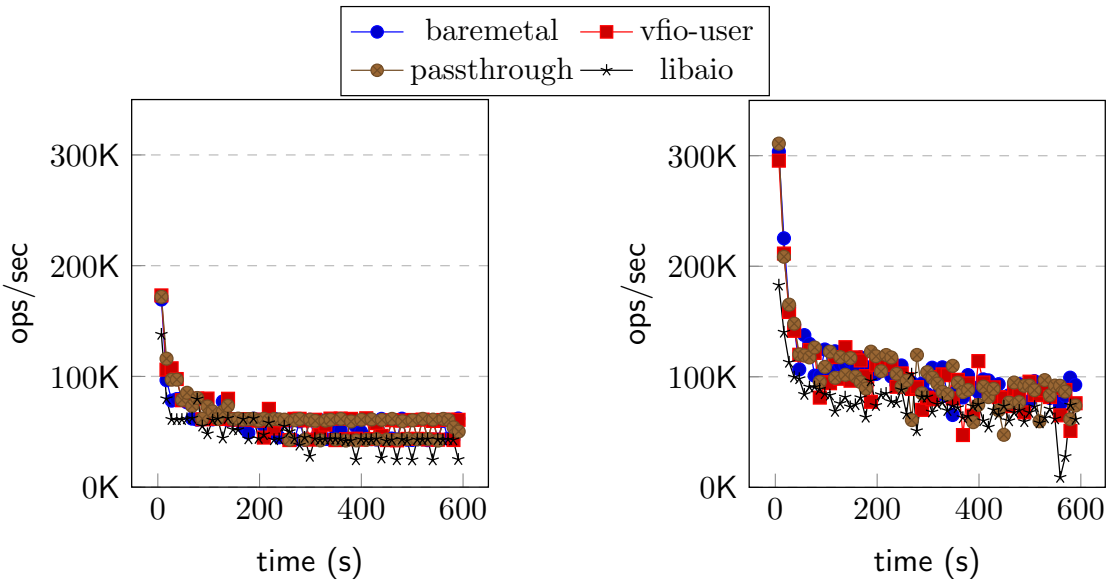
(a) YCSB-C, 1 compaction thread

(b) YCSB-C, 10 compaction threads

Figure 5.6: RocksDB random read operations per second over time, using 1 thread

Figure 5.6 shows the results of YCSB-C. Notice the clear hierarchy in ops/sec per configuration. Whereas `Fio` random reads saturate the hardware, achieving ~600K IOPS, `db_bench` random reads have to go through the RocksDB data structures, showing a smaller number of operations per second. Despite this, baremetal shows the highest throughput, followed by both `vfio-user` and

passthrough, which are essentially equivalent. The `Libaio` configuration achieves only around 60% of the throughput of the virtualized configurations.



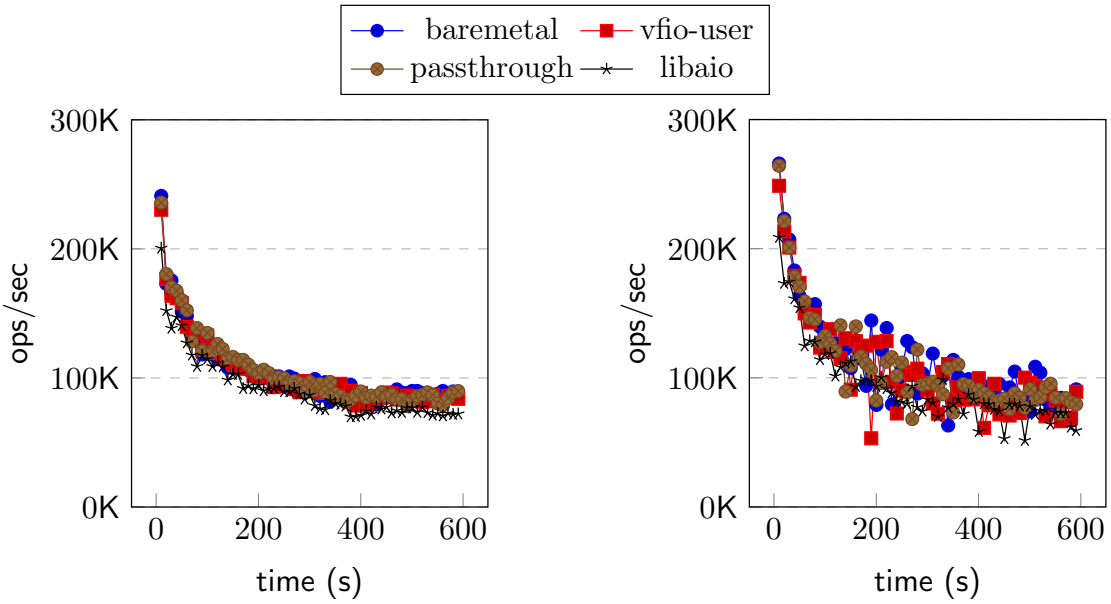
(a) YCSB-WRITE, 1 compaction thread

(b) YCSB-WRITE, 10 compaction threads

Figure 5.7: RocksDB random write operations per second over time, using 1 thread

Figure 5.7 shows the results for the YCSB-WRITE benchmark, which performs random write operations over an empty database. As with the YCSB-LOAD benchmark displayed on Figure 5.5, there is a clear descending throughput trend during insertion, though the throughput is much lower, with a peak of 300K operations per second vs. ~500K ops/sec in the YCSB-LOAD 10-compaction thread benchmark. Interestingly, there is a clear difference in ops/sec between 1 and 10 compaction threads in YCSB-WRITE, with the higher number of compaction threads attaining better throughput. Due to the nature of the log-structured merge tree data structure, which depends on keys being sorted on the SST files, it is possible that compaction tasks are being triggered earlier and before specific levels are full to keep the LSM tree structure, background tasks that were probably not necessary in the YCSB-LOAD test.

The YCSB-A benchmark results are shown in Figure 5.8, which displays how such a mixed workload appears to not be I/O bound. The data points are scattered over throughput that trends towards descent, just as it was seen with the sequential fills in figure 5.5. The pattern and the attained ops/sec values are much closer to the ones on the YCSB-WRITE benchmark, which may point to random writes being a bottleneck in this particular workload. There is a very clear difference



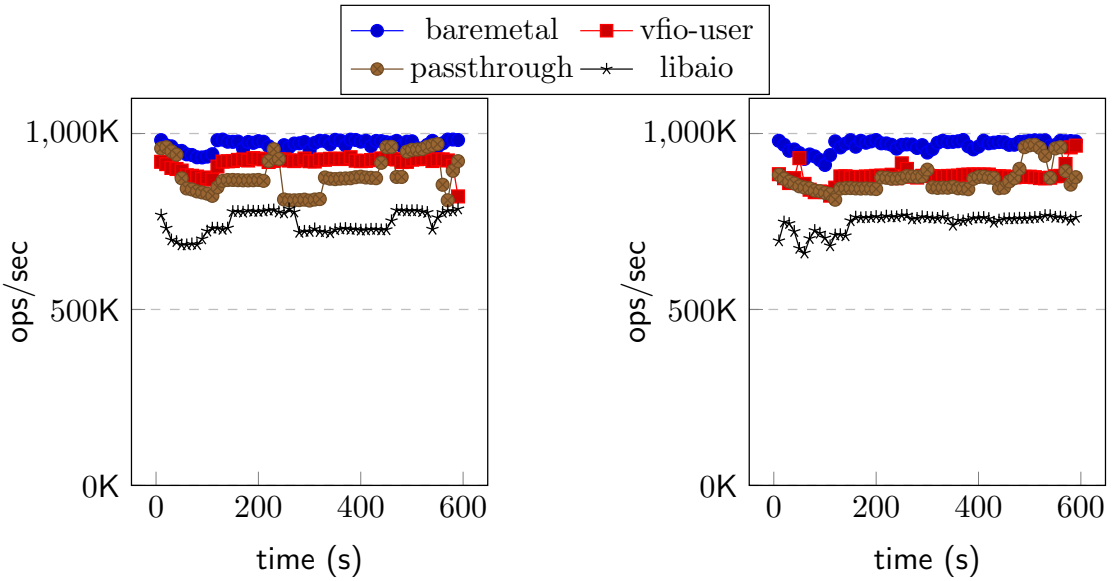
(a) YCSB-A, 1 compaction thread

(b) YCSB-A, 10 compaction threads

Figure 5.8: RocksDB random read and write operations per second over time, using 1 thread

in the ops/sec variance between Figures 5.8a and 5.8b, similar to the throughput difference shown in Figure 5.7. Again, considering that the only difference between these benchmarks is the number of compaction threads, we may hypothesize that their being in the background is slighly beneficial for throughput when running random write workloads.

Finally, Figure 5.9 shows the throughput for sequential reads. These results are very similar to the ones obtained for the YCSB-C benchmark in terms of the clear performance hierarchy, with baremetal at the top, and `vfio-user` and passthrough performing nearly identically. Interestingly, the `libaio` configuration has closer throughput to the other configurations in this particular workload. This may be to the strength of the LSM tree data structure and its use of caching. It is interesting to note how the number of compaction threads has no statistically visible effect on the sequential read throughput whatsoever. Furthermore, the peak operations per second attained in this benchmark exceed those of random reads by ∼400K, which is much higher than the IOPS supported by the hardware. Considering that one RocksDB operation is at least one or more I/O operations, this reinforces the fact that RocksDB must be making heavy use of main memory to attain these performance values.



(a) Sequential reads, 1 compaction thread

(b) Sequential reads, 10 compaction threads

Figure 5.9: RocksDB sequential read operations per second over time, using 1 thread

Overall, it is clear to see how there is practically no throughput or performance distinction between `vfio-user`, passthrough, or the baremetal configurations. Whereas the `fio` synthetic benchmarks showed some drastic mean IOPS differences for read workloads in particular, these appear to be able to be "smoothed out" by more realistic workloads that include some CPU work and the introduction of caching.

5.3 Latency breakdown

Figure 5.10 shows the mean time spent in each one of the layers corresponding to the colored regions in Figure 3.4, for read and write requests. Every `write()` call inside the guest is converted into 2 block operations by the guest Kernel's block layer: write and flush. These are passed to the NVMe driver. Flush is included because the tests use direct I/O, which waits until the NVMe hardware reports the write as successful before returning from the `write()` system call. The write latency values are therefore calculated from the sums of the average latencies of these 2 calls.

For reads, each layer has more or less an equal contribution to the total measured latency, considering the standard deviation of their distributions. It is important to note how the core

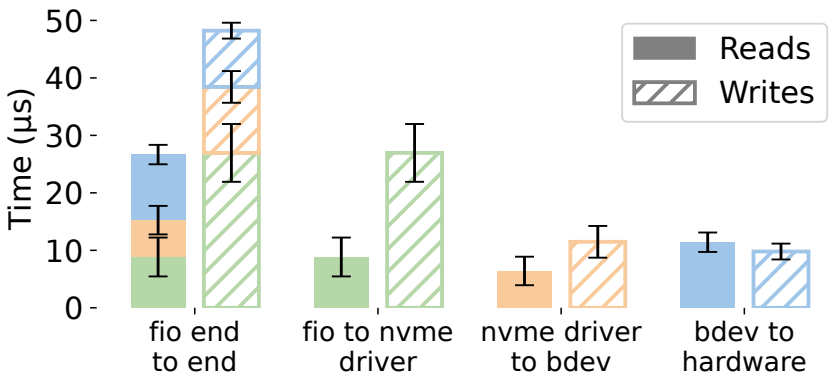


Figure 5.10: Layer-by-layer latency for reads and writes.

`vfio-user` component, namely the interface between QEMU and SPDK, does not have a larger impact on latency than the other components.

One of our expectations was that the memory management component of Qemu/KVM would take a disproportionate amount of time to pass on guest kernel memory accesses through to the physical memory managed by the host kernel, due to the possibility of `VM_EXIT` events being triggered forcing some switches into the KVM context. This does not seem to be the case. Observe that the NVMe hardware read completion time, which involves real peripheral interaction through the MMU, takes around the same time as the other 2 layers, which are mostly CPU-bound. the sub-10μs latency for I/O completion is consistent with the specifications of the hardware.

For writes, it is evident that the Fio to NVMe driver stage takes longer than the other 2 layers, which behave more or less equally as they do in reads. This is due to guest `blk-mq` involvement in I/O scheduling, and the overhead of creating the flush and write `bio` structures in the Kernel. Mean flush operation duration at the bdev level was measured at 0.25$\pm$0.15μs, most of the duration is due to write operations.

6

Related Work

6.1 `io_uring`

`Io_uring` is a novel Linux I/O API [29], proposed by `fio` creator Jens Axboe in 2019 [8]. One of its main design goals is to improve on Linux's `libaio` asynchronous block I/O API with regards to latency, though it is now flexible enough to be used for networking I/O as well [31]. The key insight brought by `io_uring` is that data copying between kernel space and userspace is expensive and wasteful. Recall from Section 2.2 that standard DMA I/O allows for device controllers to write data onto system memory; though without an IOMMU, the DMA controller can only access memory addresses reserved to the kernel. Therefore, when data is requested by a userspace process, it has to be copied from kernel space into userspace, usually when returning from I/O system calls. Data copying can be worked around by several means, including zero-copy system calls that are already available in Linux [31]. `Io_uring`'s approach is to have the kernel and processes share memory

through a ring buffer data structure that is kept alive during the whole process lifespan. Processes using `io_uring` perform a small set of system calls to set up submission and completion queues, over which polling can be performed. It is important to note that `io_uring` is a kernel-level API, meaning that QEMU would have to go through the host's I/O stack when using it.

An extensive collection of synthetic benchmark results comparing Linux I/O API performance is available in the literature. Since the first time SPDK was released, it has shown 6x to 10x single-core performance improvements (latency and IOPS) over `libaio` on the bare-metal as well as in virtualized tasks [92]. Results are echoed in Didona [34], where SPDK is shown to be the only API able to saturate NVMe device bandwidth against `libaio` and `io_uring`, which end up being CPU bound and 100% and 20% slower in terms of latency, respectively, in the best case.

Despite the results above, it is somewhat unclear whether SPDK definitively outperforms `io_uring` in every case. Considering that `io_uring` is also an asynchronous polling interface where I/O does not involve system calls, it is unknown whether the majority of the performance improvements shown by SPDK come mostly from its own re-implementation of the block stack and DMA facilities enabled by `vfio`. Even if `io_uring`'s performance eventually is shown to be on average better than SPDK's, the fact that SPDK+`vfio-user` allows for userspace IPC access and virtualization of arbitrary PCIe hardware can ensure its usefulness in developing and prototyping novel storage devices.

6.2 SPDK-vhost-NVMe

Yang et al [93] present an essentially equivalent approach in context switch reduction for QEMU/KVM as our `vfio-user` configuration, by moving as much of the I/O stack as possible into userspace. Recall, from Section 3.2.4, that `libvfio-user` allows for IPC virtualization of NVMe devices through a configuration channel (a Unix socket, traditionally) and a memory mapping meant for data transfer between the `vfio-user` server and client.

SPDK-vhost-NVMe follows the same architecture. Namely, the QEMU process has a client component that interacts with a server component on SPDK's `nvmf_tgt` process. The latter, in turn, accesses the NVMe hardware through `vfio` as in our case. The key difference between `vfio-user` and SPDK-vhost-NVMe is the protocol going through the userspace communication channels, and

the optional addition of an emulated NVMe device at the QEMU-SPDK interface. Their approach uses the `vhost-user` protocol, based on `vhost`, covered in Section 2.1.3. This allows virtualization of `virtio` devices, namely `virtio-scsi`, `virtio-blk`, and `virtio-net`. Furthermore, an NVMe emulation layer is installed between the guest operating system and the `virtio` SPDK server process, that allows the guest to use the non-paravirtualized Linux NVMe driver.

SPDK-vhost-NVMe's throughput performance is compared against a standard kernel `vhost-scsi` configuration, and QEMU's NVMe emulation. Due to lack of details on the latter, it is difficult for us to ascertain how similar their emulated NVMe configuration is to our `libaio` configuration (Section 3.2.2), except for the fact that the same QEMU emulated NVMe is present in the datapath. SPDK-vhost-NVMe attains ∼4x higher throughput than the `vhost-scsi` configuration, and ∼8x higher throughput than QEMU NVMe emulation. Random reads appear to saturate the hardware at ∼600K IOPS, though random writes have a hard ceiling of ∼80K IOPS regardless of the number of threads. We regard this asymmetry as unusual. Furthermore, the lack of an upper bound configuration makes it difficult to determine how close their configuration sits to the bare metal.

Regardless of performance characteristics, `vfio-user`'s strength is still the same as covered previously: it allows for universal PCIe communication virtualization. There is also no need for an emulation layer for NVMe devices to be introduced at any layer in the stack, also favoring the usage of the non-paravirtualized kernel NVMe driver. The lines between `vfio-user` and `vhost-user` may blur in the future, as manufacturers adopt hardware `virtio` implementations.

6.3 Other approaches and notable work

6.3.1 Kernel-level datapath optimization

Whereas we focus on solutions that move the I/O path into userspace, there have been efforts to reduce latency at the kernel level. One of these, as shown in Section 2.1.3, is `vhost`, which is commonly used by QEMU/KVM operators.

Lee et al [55] propose a lightweight block layer, specifically designed for NVMe devices, that eschews most of Linux's `blk-mq` by submitting I/Os asynchronously. Operations that are synchronously performed in the default Kernel I/O path, namely Page Allocations, Cache Insertions, and `bio` structure submission, are either performed *after* I/O submission or replaced with lighter

versions, such as a *Lightweight `bio` submission* that can be done ∼5x faster than a standard `bio` submission. They report up to 44% IOPS improvement on RocksDB tasks using their solution.

Zhang et al [95] note the significant overhead posed by current operating systems when I/O operations reach sub-10μs latencies, when individual kernel-level block device operations introduce hundreds of nanoseconds of overhead. Their solution, called the *Demikernel*, allows for multiple kernel-bypass solutions to operate simultaneously accross heterogenous devices. In a practical sense, libraries such as SPDK become a built-in part of the operating system, and applications are mostly expected to operate through their use instead of using system calls. Their results show at least a 15% reduction in I/O request latency when using a *Demikernel* vs. Linux or Windows.

6.3.2 SR-IOV

SR-IOV is a PCIe specification extension that allows for hardware-supported resource virtualization. With PC BIOS and operating system support, it's possible for a single PCIe device to offer multiple *Virtual Functions* (VFs) that are seen as independent devices by the operating system, with certain limitations. VFs are distinguished from Physical Funcions (PFs), which represent the actual workings of the hardware. Each device supporting SR-IOV has one Physical Function, and it is allowed to offer up to 256 Virtual Functions, which can be directly assigned to specific virtual machines, in the same way in which we set up our passthrough configuration (Section 3.2.3).

Devices supporting SR-IOV must have controllers that support hardware-level multiplexing, which incurs additional complexity and cost when compared with PCIe devices that don't comply with the specification. Furthermore, a similar functionality can be achieved using software-only solutions such as SPDK+`vfio-user`, which supports PCIe device multiplexing through userspace inter-process communication.

In any case, the NVMe hardware available for our testing (refer to Section 4.1.1) did not have SR-IOV capabilities, as reported by Linux's `lspci` utility.

6.3.3 Latency source modeling and formalization

Casini et al. [20] describe a series of mathematical models that allow for guarantees on worst- and best-case scenario latency bounds for three different I/O virtualization techniques. Under these models, we could consider `vfio-user` to be a kind of type C: I/O Para-Virtualization with I/O VM

and Shared Buffers. In future work, it would be possible to apply these models to the configurations presented in our work to understand whether Casini's generalizations hold.

6.3.4 Measurement pitfalls

Kogias et al. [50] bring attention to the difficulty of measuring latency accurately when working at microsecond scales. We attempt to follow their suggestions with regards to experiment duration (ensuring convergence in the YCSB results); measurement collection (latency collected independently for each request type in Section 5.1); statistical aggregation; and heavy-tailed distribution identification.

7

Conclusion

In this thesis we have covered the motivation for reducing I/O virtualization latency due to the high impact potential on computing tasks that depend on virtual machines at scale, in terms of power usage, operating cost reduction, and quality of service for end-users. As the popularity and relevance of warehouse-scale computing continues to increase, we reassert our belief that performance improvements at the bottom of the stack are crucial to support applications on the layers above. Additionally, we surveyed some of the theoretical and technical concepts needed to understand the current state of I/O virtualization configurations, and we covered the factors that make `libvfio-user` special and flexible as part of virtual machine storage datapaths.

We showed that the throughput performance of `libvfio-user` on synthetic `fio` benchmarks (Section 5.1) is similar to that of bare metal. Specifically, we observed at most a 29.41% difference in IOPS for random reads at queue depth 1 and 4 threads (the highest performance difference we observed throughout all the tests we performed); random writes showed at most a 15.23%

difference under the same queue depth and thread parameters. `Fio` read benchmarks showed a greater performance difference than writes, on average. Crucially, `libvfio-user` attained hardware IOPS saturation at the same queue depths and thread numbers as both bare metal and passthrough. Our RocksDB benchmarks (Section 5.2) showed much closer performance between bare metal, passthrough, and `libvfio-user`, the latter two being indistinguishable in every workload tested.

As part of our layer-by-layer latency analysis, we demonstrated the fact that it is possible to probe specific components of the `libvfio-user` configuration (Section 3.3), despite the fact that no standalone tool previously existed to accomplish this. The results from this analysis show that neither SPDK nor `libvfio-user` appear to have significant effects on latency overhead. We hope that our results, configurations, and analysis are a valuable contribution to the literature.

Research Questions Revisited.

- *In comparison with currently-existing and production-ready storage I/O for virtual machine configurations, where does `libvfio-user` sit in terms of latency and IOPS?*

  We have shown that `libvfio-user` consistently achieves throughput and latency performance close to bare-metal, and it is almost indistinguishable to tried-and-tested solutions such as passthrough device assignment.

- *Is `libvfio-user` performant enough to be used in production by cloud service providers?*

  Our tentative answer is affirmative, based on our single-machine synthetic and real-world application benchmarks, though we recommend further work with benchmarks on distributed systems to ascertain this claim.

- *Given that it requires a complex and layered configuration, where does the CPU spend the most time when I/O requests are executed through `libvfio-user`?*

  From our layer-by-layer latency analysis, we observed that most of the I/O request completion latency for the "simplified" `libvfio-user` configuration (Section 3.3) is spent on the software block layer at the kernel level. SPDK and `libvfio-user` appear to not impose significant latency overhead.

7.1 Future Work

We focused on virtual machine I/O configurations that offered virtual NVMe devices to QEMU/KVM guests. Our motivation behind this decision was to perform a like-for-like comparison for I/O intensive processes, since our throughput upper-bound configuration was ran on the bare metal, without any virtualization involved, with access to an NVMe device through SPDK (Section 3.2.1). Due to this restriction, we did not compare our configurations with other possible I/O virtualization configurations, such as SPDK-vhost-NVME and the `vhost-scsi` configuration tested by Yang et al [93]; a standard `virtio-blk` datapath with both the `virtio` frontend and backend in userspace; or any other configuration using `io_uring` in the guest or the host. Such experiments would be valuable contributions to the literature.

For our layer-by-layer latency analysis, we decided to eschew using our full `vfio-user` configuration, which involves SPDK both inside the guest and the host (with the `igb_uio` kernel module replacing `vfio` inside the guest). The reason why we did this, as explained in Section 3.3, was that we needed the simplest possible configuration that made use of `vfio-user`, which would also allow us to follow synchronous I/O requests across the stack for ease of identification. Given the exploratory work described in Section 3.3, we consider it feasible to trace layer-by-layer latency for the full `vfio-user` configuration, which would possibly give us more insight regarding SPDK's performance when inside the guest, and whether the `igb_uio` kernel module shows any overhead when compared to `vfio`. It is also worth noting that, as mentioned in Section 4.1.1, we were not able to set up vIOMMU with QEMU/KVM, which forced us to use `igb_uio`. Iterating further on this issue may allow for the `vfio` kernel module to be used inside the guest.

A natural next step in studying `libvfio-user` is to set up multiplexing benchmarks, i.e. having the SPDK `nvmf_tgt` process attend to multiple virtual machines on the same physical host while accessing a single NVMe device. As mentioned in Sections 2.2.4 and 2.2.5, a single `nvmf_tgt` process is able to attend to multiple virtual machines or userspace processes due to the `libvfio-user` client-server model. Executing these benchmarks standalone would provide useful information about SPDK's CPU and NVMe hardware usage when under load from multiple VMs, though it would be best to have a point of comparison for multiplexing, ideally using SR-IOV, since it is hardware-

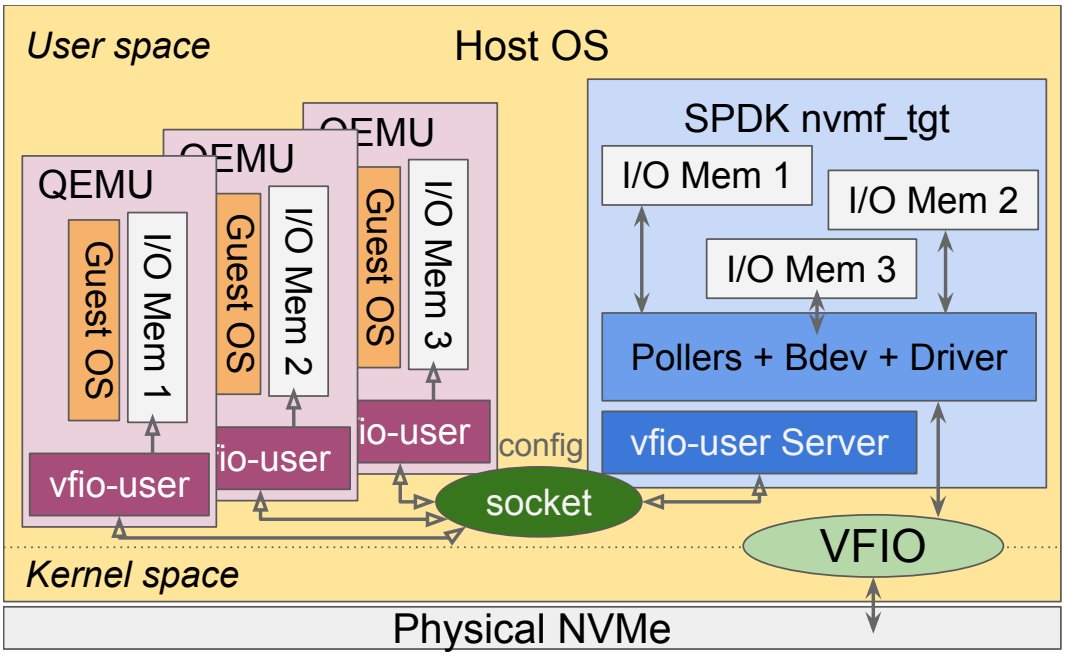


Figure 7.1: A possible "multiplexed" or multi-VM configuration using SPDK, to be tested in future work. Note that the arrows representing I/O between the memory mappings, present in Figures 2.5 and 3.4, are omitted here for simplicity. Each QEMU memory mapping has a corresponding memory mapping in SPDK's process space.

supported. This would require the sourcing of new NVMe hardware for the DISCS Laboratory, as explained in Section 6.3.2.

Though we performed real-world application benchmarks using RocksDB, we limited ourselves to one machine. We believe that setting up Kubernetes or EC2 Auto Scaling groups with nodes using `libvfio-user` as their storage backend for distributed persistence tasks (e.g. sharded databases) would provide valuable information about `libvfio-user`'s performance at a larger scale. We would expect to observe very similar throughput and latency as with current solutions, barring any emergent behaviours. Further confirmation of `libvfio-user`'s capacity to replace or complement current I/O virtualization solutions in cloud-scale computing tasks could be obtained by installing it in more realistic scenarios such as these.

Bibliography

[1] Appendix A. The Tanenbaum-Torvalds Debate. In Chris DiBona, Sam Ockman, and Mark Stone, editors, *Open sources: voices from the open source revolution*. O'Reilly, Beijing ; Sebastopol, CA, 1st ed edition, 1999. ISBN 978-1-56592-582-3. OCLC: ocm40889566.

[2] Chapter 15. Memory Mapping and DMA. In *Linux device drivers*. O'Reilly & Associates, Sebastopol, 2nd ed edition, 2001. ISBN 978-0-596-00008-0.

[3] Chapter 17. Devices and Modules. In *Linux kernel development*, Developer's library : essential references for programming professionals. Addison-Wesley, Upper Saddle River, NJ, 3rd ed edition, 2010. ISBN 978-0-672-32946-3. OCLC: ocn268788260.

[4] Chapter 4. x86-64: CPU Virtualization With VT-x. In *Hardware and software support for virtualization*, number # 38 in Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, San Rafael, CA, 2017. ISBN 978-1-62705-693-9.

[5] Chapter 6. x86-64: I/O Virtualization. In *Hardware and software support for virtualization*, number # 38 in Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, San Rafael, CA, 2017. ISBN 978-1-62705-693-9.

[6] 4.3 KVM—A HYPERVISOR FOR VT-X. In *Hardware and software support for virtualization*, number # 38 in Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, San Rafael, CA, 2017. ISBN 978-1-62705-693-9.

[7] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating systems: three easy pieces*. Arpaci-Dusseau Books, LLC, Erscheinungsort nicht ermittelbar, 2018. ISBN 978-1-985086-59-3.

[8] Jens Axboe. Efficient IO with io_uring. Technical report, 2019. URL `https://kernel.dk/io_uring.pdf`.

[9] Jens Axboe. 1. fio - Flexible I/O tester rev. 3.33 — fio 3.33 documentation, 2022. URL `https://fio.readthedocs.io/en/latest/fio_doc.html#i-o-engine`.

[10] Oana Balmau. McGill DISCS Lab, 2023. URL `https://sites.google.com/view/discslab`.

[11] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Springer International Publishing, Cham, 2019. ISBN 978-3-031-00633-3 978-3-031-01761-2. doi: 10.1007/978-3-031-01761-2. URL `https://link.springer.com/10.1007/978-3-031-01761-2`.

[12] Fabrice Bellard. [announce] QEMU x86 emulator version 0.1, March 2003. URL `https://www.winehq.org/pipermail/wine-devel/2003-March/015577.html`.

[13] Fabrice Bellard. QEMU CPU Emulator, August 2003. URL https://web.archive.org/web/20030801214438/http://fabrice.bellard.free.fr/qemu/.

[14] Muli Ben-Yehuda, Jimi Xenidis, Michal Ostrowski, Karl Rister, and Alexis Bruemmer. The Price of Safety: Evaluating IOMMU Performance. 2007.

[15] Timo Bingmann. NVMe "Disk" Bandwidth and Latency for Batched Block Requests - panthema.net, March 2019. URL https://panthema.net/2019/0322-nvme-batched-block-access-speed/.

[16] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. 2021.

[17] Roman Bolshakov and Andrea Bolognani. libvirt: The virtualization API, 2018. URL https://libvirt.org/.

[18] Dhruba Borthakur. The Story of RocksDB: Embedded Key-Value Store for Flash and RAM, 2013. URL https://raw.githubusercontent.com/facebook/rocksdb/gh-pages-old/intro.pdf.

[19] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. *Hardware and software support for virtualization*. Number # 38 in Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, San Rafael, CA, 2017. ISBN 978-1-62705-693-9.

[20] Daniel Casini, Alessandro Biondi, Giorgiomaria Cicero, and Giorgio Buttazzo. Latency Analysis of I/O Virtualization Techniques in Hypervisor-Based Real-Time Systems. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 306–319, May 2021. doi: 10.1109/RTAS52030.2021.00032. ISSN: 2642-7346.

[21] Peter Chubb. UserLevelDrivers - Gelato@UNSW WiKi, January 2004. URL https://web.archive.org/web/20040108042329/http://www.gelato.unsw.edu.au/IA64wiki/UserLevelDrivers.

[22] DPDK Project Contributors. 7. Linux Drivers — Data Plane Development Kit 22.11.0 documentation, 2017. URL https://doc.dpdk.org/guides/linux_gsg/linux_drivers.html#uio.

[23] IO Visor Project Contributors. bpftrace, February 2023. URL https://github.com/iovisor/bpftrace. original-date: 2018-08-31T04:34:44Z.

[24] Kernel Newbies Contributors. Linux_2_6_23 - Linux Kernel Newbies, 2017. URL https://kernelnewbies.org/Linux_2_6_23#UIO.

[25] Linux Kernel Contributors. KVM - Kernel Virtual Machine, 2016. URL https://www.linux-kvm.org/page/Main_Page.

[26] QEMU Contributors. Invocation — QEMU documentation. URL https://qemu-project.gitlab.io/qemu/system/invocation.html.

[27] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, June 2010. Association

for Computing Machinery. ISBN 978-1-4503-0036-0. doi: 10.1145/1807128.1807152. URL https://doi.org/10.1145/1807128.1807152.

[28] Jonathan Corbet. UIO: user-space drivers [LWN.net], May 2007. URL https://lwn.net/Articles/232575/.

[29] Jonathan Corbet. The rapid growth of io_uring [LWN.net], 2020. URL https://lwn.net/Articles/810414/.

[30] Jonathan Corbet. Killing off /dev/kmem [LWN.net], April 2021. URL https://lwn.net/Articles/851531/.

[31] Jonathan Corbet. Zero-copy network transmission with io_uring [LWN.net], 2021. URL https://lwn.net/Articles/879724/.

[32] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. 2020.

[33] Dell. Dell Express Flash P4800X Technical Specifications. 2019.

[34] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 120–127, Haifa Israel, June 2022. ACM. ISBN 978-1-4503-9380-5. doi: 10.1145/3534056.3534945. URL https://dl.acm.org/doi/10.1145/3534056.3534945.

[35] Christopher Domas. The Memory Sinkhole, July 2015.

[36] Stephane Duverger. A deep dive into QEMU: The Tiny Code Generator (TCG), part 1, 2021. URL https://airbus-seclab.github.io/qemu_blog/tcg_p1.html.

[37] Werner Fischer. Processor P-states and C-states - Thomas-Krenn-Wiki, 2019. URL https://www.thomas-krenn.com/en/wiki/Processor_P-states_and_C-states.

[38] Werner Fischer. Linux Storage Stack Diagram - Thomas-Krenn-Wiki-en, March 2023. URL https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram.

[39] Johan De Gelas. Hardware Virtualization: the Nuts and Bolts, March 2008. URL https://www.anandtech.com/show/2480.

[40] Stéphane Graber. Linux Containers, 2023. URL https://linuxcontainers.org/.

[41] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, November 2019. ISBN 978-0-13-662458-5. Google-Books-ID: ihTADwAAQBAJ.

[42] Canonical Group. cloud-init 23.1.1 documentation, 2023. URL https://cloudinit.readthedocs.io/en/latest/.

[43] Red Hat. Ansible, 2023. URL https://www.ansible.com.

[44] Greg Hewgill. Answer to "How did old MS-DOS games utilize various graphic cards?", June 2019. URL https://retrocomputing.stackexchange.com/a/11220.

[45] Andy Honig and Nelly Porter. 7 ways we harden our KVM hypervisor at Google Cloud: security in plaintext, January 2017. URL `https://cloud.google.com/blog/products/gcp/7-ways-we-harden-our-kvm-hypervisor-at-google-cloud-security-in-plaintext`.

[46] Kaisong Huang, Tianzheng Wang, Darien Imai, and Dong Xie. SSDs Striking Back:The Storage Jungle and Its Implications on Persistent Indexes. page 8, 2022.

[47] Khoa Huynh and Stefan Hajnoczi. KVM/QEMU Storage Stack Performance Discussion, 2010. URL `https://docs.huihoo.com/virtualization/2010/02-kvm-storage-stack-performance.pdf`.

[48] Intel. 23. Vhost Sample Application — DPDK documentation, 2014. URL `https://doc.dpdk.org/guides-1.8/sample_app_ug/vhost.html`.

[49] Intel and SPDK Contributors. spdk/rocksdb, September 2022. URL `https://github.com/spdk/rocksdb`. original-date: 2017-03-10T21:16:22Z.

[50] Marios Kogias, Christos Kozyrakis, and Edouard Bugnion, editors. *Measuring Latency: Am I doing it right?* 2017. Meeting Name: 14th USENIX Symposium on Networked Systems Design and Implementation.

[51] Mike Kravetz. Huge Page Concepts, 2022. URL `https://project.linuxfoundation.org/hubfs/Webinars/Webinar_Slides/Huge-Page-Concepts.pdf?hsLang=en`.

[52] Peter Krempa. libvirt: Domain XML format, 2023. URL `https://libvirt.org/formatdomain.html`.

[53] Greg Kroah-Hartman. [RFC] Simple userspace interface for PCI drivers, August 2006. URL `https://lore.kernel.org/all/20060830062338.GA10285@kroah.com/#r`.

[54] Dave Landsman and Don Walker. AHCI and NVMe as interfaces for SATA Express™ Devices. November 2013.

[55] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs. 2019.

[56] Gyusun Lee, Seokha Shin, and Jinkyu Jeong. Efficient hybrid polling for ultra-low latency storage devices. *Journal of Systems Architecture*, 122:102338, 2021. ISSN 1383-7621. doi: 10.1016/j.sysarc.2021.102338. URL `https://www.sciencedirect.com/science/article/pii/S1383762121002319`.

[57] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, and Daniel Smith. The Java® Virtual Machine Specification. February 2022.

[58] Lowrentius. louwrentius/fio-plot: Create charts from FIO storage benchmark tool output, 2023. URL `https://github.com/louwrentius/fio-plot`.

[59] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Transactions on Storage*, 13(1):5:1–5:28, March 2017. ISSN 1553-3077. doi: 10.1145/3033273. URL `https://doi.org/10.1145/3033273`.

[60] Dan C. Marinescu. *Cloud computing: theory and practice*. Morgan Kaufmann is an imprint of Elsevier, Cambridge, MA, third edition edition, 2023. ISBN 978-0-323-85277-7. OCLC: on1346432663.

[61] Philippe Mathieu-Daudé, Stefan Weil, Paolo Bonzini, and John Johnson. Multi-process QEMU — QEMU documentation, 2021. URL `https://www.qemu.org/docs/master/devel/multi-process.html`.

[62] Paul McLellan. The History of PCIe: Getting to Version 6 - Breakfast Bytes - Cadence Blogs - Cadence Community, March 2021. URL `https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/pcie-the-next-generation`.

[63] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel® Virtualization Technology: Hardware Support for Efficient Processor Virtualization. *Intel Technology Journal*, 10(03), August 2006. ISSN 1535864X.

[64] Oracle and QEMU Contributors. Out-of-process QEMU README, December 2022. URL `https://github.com/oracle/qemu`. original-date: 2019-03-07T20:38:04Z.

[65] Peterx and Dwmw2. Features/VT-d - QEMU, 2023. URL `https://wiki.qemu.org/Features/VT-d`.

[66] Joshua Powers. Ubuntu Server - Cloud images introduction, 2023. URL `https://ubuntu.com/server/docs/cloud-images/introduction`.

[67] Sebastian Rolon and Oana Balmau. DISCS Lab McGill / Vfio user measurements · GitLab, April 2023. URL `https://gitlab.cs.mcgill.ca/discs-lab/vfio-user-measurements`.

[68] Alessandro Rubini and Jonathan Corbet. *Linux device drivers*. O'Reilly & Associates, Sebastopol, 2nd ed edition, 2001. ISBN 978-0-596-00008-0.

[69] Rusty Russell. Rusty's Bleeding Edge Page, May 2007. URL `https://ozlabs.org/~rusty/index.cgi/tech/2007-05-21.html`.

[70] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, July 2008. ISSN 0163-5980. doi: 10.1145/1400097.1400108. URL `https://dl.acm.org/doi/10.1145/1400097.1400108`.

[71] James Salvadore. NVMe multiqueue interface | NVMe, NVMe/TCP, and Dell SmartFabric Storage Software Overview - IP SAN Solution Primer | Dell Technologies Info Hub, 2022. URL `https://infohub.delltechnologies.com/l/nvme-nvme-tcp-and-dell-smartfabric-storage-software-overview-ip-san-solution-primer-1/nvme-multiqueue-interface`.

[72] Schoenebeck and Paolo Bonzini. Documentation/9psetup - QEMU, 2023. URL `https://wiki.qemu.org/Documentation/9psetup`.

[73] Amit Shah. Ten years of KVM [LWN.net], November 2016. URL `https://lwn.net/Articles/705160/`.

[74] Simon Sharwood. AWS adopts home-brewed KVM as new hypervisor, November 2017. URL `https://www.theregister.com/2017/11/07/aws_writes_new_kvm_based_hypervisor_to_make_its_cloud_go_faster/`.

[75] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Elsevier, July 2005. ISBN 978-0-08-052540-2. Google-Books-ID: JPhQw41vD2MC.

[76] snips-n snails. Answer to "How did old MS-DOS games utilize various graphic cards?", June 2019. URL `https://retrocomputing.stackexchange.com/a/11221`.

[77] Jonathan Stern. Why SPDK?, April 2016. URL `https://spdk.io/update/2016/04/13/not-your-mothers-storage/`.

[78] Sriram Subramanian. *Beyond the Block-Based Interface for Flash-Based Storage*. PhD thesis, 2013.

[79] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 207–222, Bolton Landing NY USA, October 2003. ACM. ISBN 978-1-58113-757-6. doi: 10.1145/945445.945466. URL `https://dl.acm.org/doi/10.1145/945445.945466`.

[80] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating systems: design and implementation*. Pearson Education : Dorling Kindersley, Delhi., 3rd ed edition, 2006. URL `https://archive.org/details/operatingsystems0000tane`. OCLC: 1302148470.

[81] Red Hat Virtualization Documentation Team. Appendix G. Configuring a Host for PCI Passthrough Red Hat Virtualization 4.1 | Red Hat Customer Portal, 2017. URL `https://access.redhat.com/documentation/en-us/red_hat_virtualization/4.1/html/installation_guide/appe-configuring_a_hypervisor_host_for_pci_passthrough`.

[82] Michael Tsirkin. vhost_net: a kernel-level virtio server [LWN.net], 2009. URL `https://lwn.net/Articles/346267/`.

[83] Bosmat Tuvel. Speedb | How does RocksDB Memory Management work?, November 2022. URL `https://www.speedb.io/blog-posts/how-does-rocksdb-memory-management-work`.

[84] Daniel Verkamp. SPDK: Under the Hood, 2017. URL `https://s3.us-east-2.amazonaws.com/intel-builders/day_1_spdk_under_the_hood.pdf`.

[85] Daniel Verkamp and Benjamin Walker. SPDK: What is SPDK, 2022. URL `https://spdk.io/doc/about.html`.

[86] Benjamin Walker. SPDK: Block Device User Guide, 2022. URL `https://spdk.io/doc/bdev.html`.

[87] Benjamin Walker, Seth Howell, and Daniel Verkamp. SPDK: Direct Memory Access (DMA) From User Space, 2022. URL `https://spdk.io/doc/memory.html`.

[88] Alex Williamson. VFIO - "Virtual Function I/O" — The Linux Kernel documentation, 2011. URL `https://docs.kernel.org/driver-api/vfio.html`.

[89] NVM Express Workgroup. NVM Express revision 1.0e specification, 2013. URL `https://nvmexpress.org/wp-content/uploads/NVM-Express-1_0e.pdf`.

[90] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus. 2021.

[91] Jisoo Yang, Dave B Minturn, and Frank Hady. When Poll is Better than Interrupt. 2012.

[92] Ziye Yang, James R. Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E. Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, December 2017. doi: 10.1109/CloudCom.2017.14. ISSN: 2330-2186.

[93] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang Cao. SPDK Vhost-NVMe: Accelerating I/Os in Virtual Machines on NVMe SSDs via User Space Vhost Target. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 67–76, November 2018. doi: 10.1109/SC2.2018.00016.

[94] Tomek Zawadzki and Jim Harris. SPDK Schedulers: Realizing Power Savings in Polled Mode Applications, September 2021. URL `https://www.snia.org/sites/default/files/SDC/2021/pdfs/SNIA-SDC21-Zawadzki-Harris-SPDK-Schedulers.pdf`.

[95] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 195–211, New York, NY, USA, October 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483569. URL `https://doi.org/10.1145/3477132.3483569`.