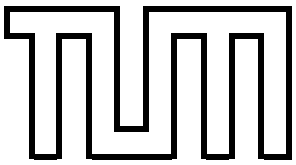


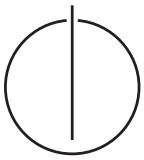
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

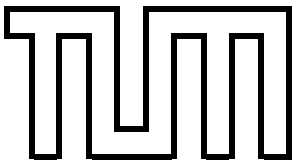
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Effects of Linux VFIO for User Space I/O

Adrian Simon Würth





SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

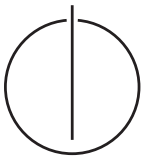
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Effects of Linux VFIO for User Space I/O

Effekt von Linux VFIO auf User Space E/A

Author:	Adrian Simon Würth
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Simon Ellmann, M.Sc.
Submission Date:	August 15, 2024



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, July 29, 2024 Adrian Simon Würth

Abstract

The research into direct memory access in the userspace has increased over the latest years especially in the field of virtualization. The main reason for this is the need for high performance and low latency in virtualized environments. The hardware that enables this is called the IOMMU. Through the IOMMU, the guest operating system can directly access the hardware without the need for the hypervisor to intervene. This technology is not bound to only virtualization, but can also be used for high-performance drivers. In the past, the IOMMU was only accessible through the kernel, but with the introduction of the *Intel VT-d* and *AMD-Vi* extensions, the IOMMU can now be accessed from the userspace. The VFIO framework and the IOMMUFD user API build the foundation for this. We aim to achieve the same performance and low latency as directly mapping the hardware into the guest operating system, while increasing the security of the system.

Contents

1 Introduction

During his speech "Null Reference: The Billion Dollar Mistake" in 2009, Tony Hoare, a renowned computer scientist, well known for the invention of Quick-sort, proposes the idea how null pointers are the reason for at least a billion dollars in damages [9]. This quote could not be anymore important than at this time. In the July of 2024, Microsoft devices faced what has been described as the "most spectacular IT meltdown the world has ever seen" [11], affecting 8.5 million Microsoft Windows devices and severly impacting institutions including critical infrastructure like hospitals and airports [4]. During the error analysis of Crowdstrike, the company which deployed the faulty code to Windows devices, it was soon clear that null pointer deferencing in C++ caused the systems to crash [5]. A memory-safe programming language like Rust helps to prevent such incidents. As even the Linux Kernel, which has stuck to C for more that 30 years not admitting any other languages like C++, not permits Rust code in its codebase, we can see Rusts impact on the systems development community. But it is also necessary to look at the limits of Rusts memory-safety. While using Rust for a driver improves the overall safety inside of the process while not compensating on performance, direct memory and I/O operations have to be implemented in an unsafe way. In order to enforce safety at the device level, we need to take use of the IOMMU, a safe way of doing direct memory accesses. The IOMMU acts as a layer of isolation between devices and the CPU. By using virtual addresses the IOMMU is able to provide a bigger virtual address space and enforce memory access rights [1]. This thesis lays its focus on analysing the effects and performance impact of using the IOMMU in the context of userspace I/O. We demonstrate this by implementing IOMMU support on vroom, a NVMe driver written in Rust [10], and comparing it to using physical addresses. To implement the IOMMU functionality we use the Linux framework VFIO, which has the advantageous side effect of the driver being able to run without root privileges.

2 Background

2.1 vroom

Vroom is a userspace NVMe driver written in Rust. While it is not production ready at the time of writing, it already provides good performance and the functionality needed for general I/O operations. A NVMe driver consists of submission and completion queues, implemented as ring buffers. The driver adds commands to the submission queue, which the NVMe controller reads and executes. The executed command gets placed on a corresponding completion queue. We unbind the kernel driver and bind it to pci-stub. Pci-stub does not do anything but occupy the pci-driver such that the kernel or another application can not bind to the device.

2.2 Memory Mapped I/O

vroom currently uses hugepages and locks them using mlock to prevent the Kernel from swapping them out. In order to 'pin' pages, preventing the Linux Kernel from switching them around, Hugepages have to be used. The kernel does not have the ability to move these pages like 4KiB pages. For accessing the devices memory as well as the device accessing the host memory, it is necessary to either use the physical addresses and compromise on safety and use root privileges or use the IOMMU for virtualization, which can introduce performance overhead.

2.3 Peripheral Component Interconnect Express

2.4 Direct Memory Access

Using Direct Memory Access we can bypass the CPU for I/O operations. Previously this was handled by a separate DMA-controller hardware (third-party DMA) but using PCI, we can directly access it through bus mastering (first-party DMA).

2.5 I/O Memory Management Unit

Memory Management Units (MMU) for the CPU have been in use since the 1980s. After their first integrated application featuring on Intels 80286 chip [8], they since have become the defacto standard for addressing memory on computers. By providing processes with a virtual address space instead of physical addresses, every process is isolated and can not access without having the privilege to. The MMU uses pages for the translation of addresses. Each address points to a region of memory called a page. These pages can have different sizes, with the default being 4Kib pages. The translation of these pages are stored in a page table structure. A page table structure consists of multiple tables that store parts of the physical address. Certain parts of an address are used as offset in these tables. When an address is translated, a page table walk has to performed. On a 4 level page table structure as the IOMMU uses for 4KiB pages, one address resolution uses 4 memory accesses. Thus, a page table walk is a performance costly operation. To circumvent this, there exists an Translation Lookaside Buffer (TLB). This TLB can store a certain amount of page translations, and is very performant to access. Frequent access to the same address can be done at a fraction of the time needed to perform a page table walk. If an address is not stored in the TLB, it is called a TLB miss, and the IOMMU has to perform a page table walk. The advantages and success of the CPUs MMU as well as the introduction of the PCIe bus specification have incentivised Hardware manufacturers to apply this concept on peripheral device busses. In 2006, Intel introduced their "Virtualization Technology for Directed I/O" (VT-d) and AMD their "AMD I/O Virtualization Technology" (AMD-Vi/IOMMU). In this thesis, the term IOMMU references both technologies. Using DMA Remapping (DMAR) the CPU is bypassed and the direct memory access translated by the IOMMU. The IOMMU paging structures are each 512 * 8 Bytes = 4kib in size. The IOMMU uses the upper portions to determine the location of the stored page tables, and the lower portion of the address as page offset. In the case of 4KiB pages its 12 bits as $2\hat{1}2$ = 4096 = 4Kib, for 2MiB its 21 as $2\hat{2}1$ = 2097152 = 2Mib.

2.6 Hugepages

As the demand for bigger memory mappings e.g. for big files increased, the amount of TLB cache misses rose proportionally. With the CPUs TLB having space for only 4096 4KiB pages, only an address space of 16MiB could be stored and accessed quickly. In order to increase the amount of virtual memory space, hardware producers reacted by providing bigger page sizes on their architectures than the default 4KiB. x86 architectures now normally provide 4K, 2M and 1G page sizes. Linux currently provides two

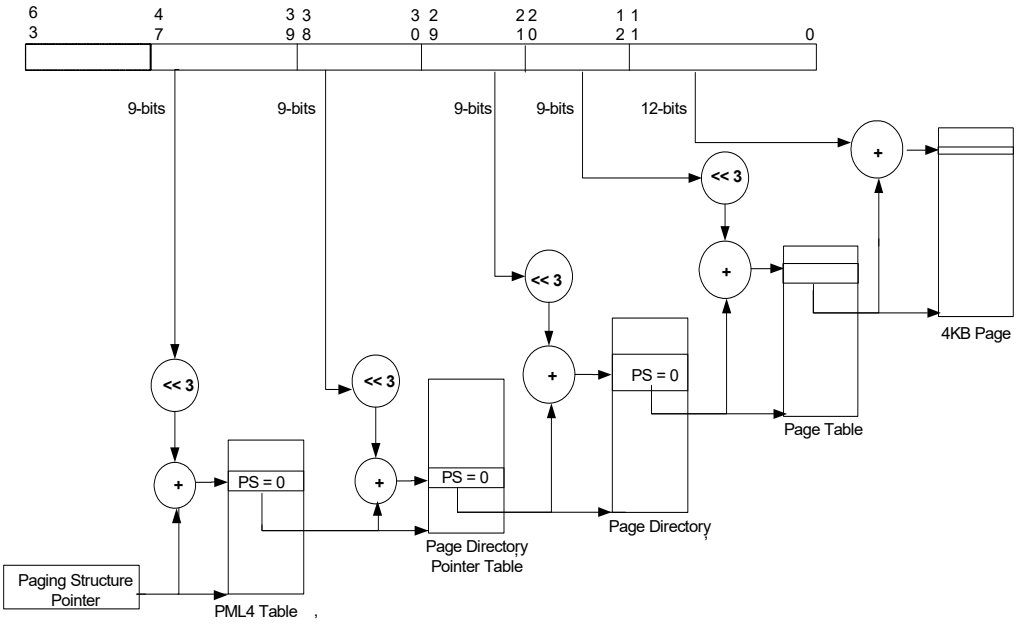


Figure 2.1: VT-d Paging structure for translating a 48-bit address to a 4-KByte page

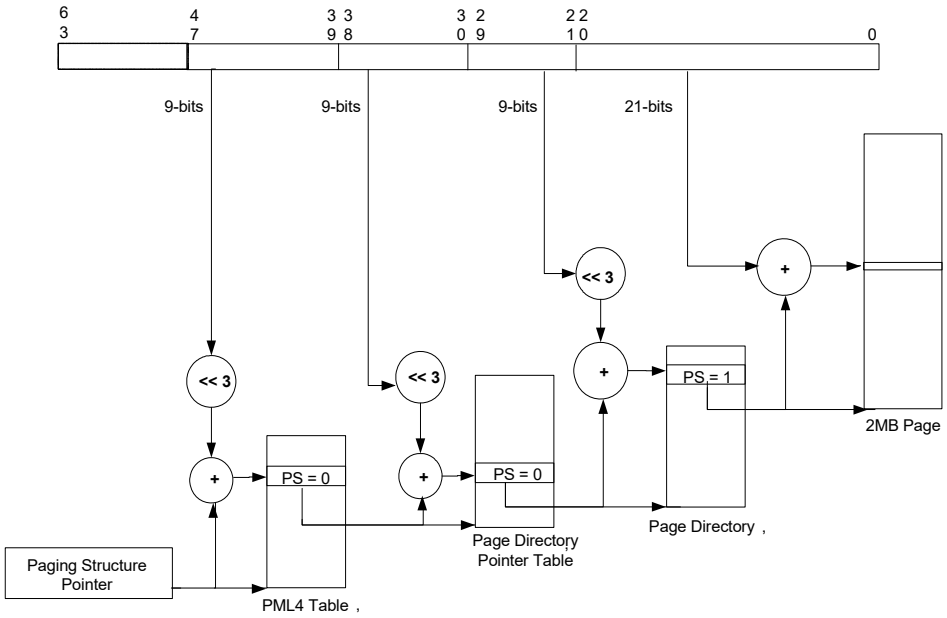


Figure 2.2: VT-d Paging structure for translating a 48-bit address to a 2-MByte page

ways of using Hugepages. Using a 2MiB or 1GiB page size should result in a 512 or 512*512=262144 times reduction in cache misses compared to 4KiB pages. Especially in high-performance computing this make a huge difference.

- **Persistent Hugepages**: Persistent Hugepages, are reserved in the kernel and cannot be swapped out or used for another purpose. To use these hugepages, they are mounted as a (pseudo) filesystem called hugeltbfs, which lays in the directory `/mnt/huge`. [7] The amount and size of the pages can be specified either during boot on the kernel commandline with e.g. `hugepagesz=1g hugepages=16` or dynamically using the Linux `sysfs` pseudofilesystem e.g. `echo 16 > /proc/sys/vm/nr_hugepages`
- **Transparent Hugepages**: Transparent Hugepages are a more recent addition to the kernel. Transparent hugepages are not fixed or reserved in the kernel and allow all unused memory to be used for other purposes. The `khugepaged` daemon scans memory and collapses sequences of basic pages into bigger pages [13]. THPs can either be enabled, disabled or only be used on `madvise(MADV_HUGEPAGE)` memory regions.

2.7 I/O Translation Lookaside Buffer

As page table walks are rather costly in performance, a cache on the IOMMU is used to store previously calculated addresses. This cache is called the Input/Output Translation Lookaside Buffer. The IOTLB possesses a limited capacity for entries which is not officially documented.

2.7.1 Character and Block Devices

Unix/Linux use two types of devices: Character and Block devices. Character devices are used for devices with small amounts of data and no frequent seek queries, like keyboard and mouse. Block devices on the other hand have a large data volumes, which are organized in blocks and where search is common, like harddrives and ram disks. Read and Write operations on character devices are done sequentially byte-by-byte, while on block devices, read/write is done at the data block level. These constraints also impact how the drivers for these devices work. CDev drivers directly communicate with the device drivers, while block device drivers work in conjunction with the kernel file management and block device subsystem. This allows efficient asynchronous read/write operations for large data amounts, but small byte sized data transfer achieves lower latency on character devices.

2.7.2 Rust

Rust as a programming language offers a lot of benefits, especially in the systems programming field. The memory safety enforced by the borrow checker and the focus on providing the most concise and exact syntax like default variable immutability while obmitting boilerplate make it an excellent choice for modern systems development.

3 Related Work

3.1 Storage Performance Development Kit

The Storage Performance Development Kit (SPDK) provides “a collection of tools and libraries for writing high performance, scalable, user-mode storage applications” [12]. It includes an user-space NVMe driver which is fast and production-ready. While this driver supports the use of the driver without the IOMMU, the SPDK Documentation recommends using the IOMMU as using VFIO and the IOMMU is the "future proof...long-term foundation" for SPDK [3].

3.2 ixy

Ixy is a user space network driver for network interface cards.

4 Implementation

4.1 Virtual Function I/O

Virtual Function I/O (VFIO) is an IOMMU agnostic framework for exposing devices to userspace. The VFIO driver binds to the PCI device and manages address translation through the IOMMU. This allows the driver to be safe and non-privileged in comparison to directly mapping the device memory to userspace. Using the VFIO works by using ioctl system calls. While there is Rust's extensive `libc` library providing the system calls `ioctl` and `mmap` and their flags, the Linux `vfio.h` constants and structs need to either be defined manually or with a crate like bindgen, which automates bindings for C and C++ libraries [2]. To keep the binary and dependency list as small as possible we chose the manual implementation.

Using `VFIO_IOMMU_GET_INFO` we can see the supported pagesizes. As our IOMMU supports 4K, 2M and 1G pagesizes the field iova_pgsizes has the value 0x40201000.

4.2 syscalls

A variety of syscalls are used in the process of allocation and mapping. These syscalls are mmap, ioctl, pread, pwrite (and mlock for the non IOMMU version). While there exist crates which implement the syscall functionality, to avoid inflating the dependency list and executable size we use the libc crate to implement them. As these require C-like syntax and an unsafe block in Rust, wrapper macros/inline functions are used to provide local, secure error handling, while also minimising the calling of unsafe code. In Listing 4.2 the macro for the mmap syscall can be seen. As part of our error handling, we introduce an error enum variant for each syscall.

```
#[macro_export]
macro_rules! mmap {
    ($addr:expr, $len:expr, $prot:expr, $flags:expr, $fd:expr, $offset:
        ↪ expr) => {{
        let ptr = unsafe { libc::mmap($addr, $len, $prot, $flags, $fd,
            ↪ $offset) };
        if ptr == libc::MAP_FAILED {
```

```
            Err(Error::Mmap {
                error: (format!("Mmap␣with␣len␣{}␣failed", $len)),
                io_error: (std::io::Error::last_os_error()),
            })
        } else {
            Ok(ptr)
        }
    }};
}
```

4.2.1 Initialising the IOMMU

To use the IOMMU for the driver, we first need to initialize the VFIO kernel module and bind the VFIO driver to the NVMe device. As this binds the driver to a device, it has to be run with root privileges. By changing the owner of the VFIO container to an unprivileged user, the driver can use the VFIO driver to interact with the device without root. In Listing 4.1 the initialization script for the Vfio driver is shown.

Listing 4.1: Initializing VFIO using bash

```
#!/bin/bash
modprobe vfio-pci
nvme_vd="$(cat␣/sys/bus/pci/devices/$nvme/vendor)␣$(cat␣/sys/bus/pci/
    ↪ devices/$nvme/device)"
echo $nvme > /sys/bus/pci/devices/$nvme/driver/unbind
echo $nvme_vd > /sys/bus/pci/drivers/vfio-pci/new_id
chown $user:$group /dev/vfio/*
```

1. Add VFIO kernel module using `modprobe`
2. Unbind kernel driver from NVMe
3. Use vendor and device id to bind VFIO to the device
4. Set VFIO group permissions to user/group using `chown`

4.2.2 Initialising the NVMe device

Using the Vfio, the initialization process of the NVMe SSD works as followed.

1. Map the NVMe device memory into host memory using VFIO resource info.

2. Allocate Admin SQ, CQ and I/O SQ, CQ
3. Create a mapping on the IOMMU using VFIO
4. Configure the NVMe device
5. Pass I/O Queue addresses to NVMe device using admin queues

4.2.3 Enabling DMA

To enable DMA we need to set a bit in the PCIe device registers.

4.2.4 Mapping DMA

In order to provide a section of memory on which the device can perform DMA operations, the user needs to allocate some memory in the processes address space. This can be achieved by using the Linux syscall `mmap`. Using `mmap`'s flags we can also define the page size used. The `MAP_HUGETLB` flag is used in conjunction with the `MAP_HUGE_2MB` and `MAP_HUGE_1GB` flags for 2MiB and 1 GiB pages respectively. By default `mmap` uses the default page size of 4KiB. The main IOMMU work is done by then creating the map struct `vfio_iommu_type1_dma_map`. We set the DMA mapping to read and write, and provide the same IOVA as the Virtual address. By then passing it to an ioctl call with the according VFIO operation `VFIO_IOMMU_MAP_DMA` we can create a mapping in the page tables of the IOMMU. This way we can give the IOVA to the NVMe controller, which it will use to access the memory through the address translation of the IOMMU. On Figure 4.1 an I/O operation timeline graph can be seen.

4.2.5 Unmapping DMA

Unmapping DMA happens when the process exits, yet for performance and application reasons there is the unmap_dma function which can be used to unmap a DMA. It is necessary to increase the allocated size to a multiple of the page size as otherwise the munmap operation will result in a failure.

4.2.6 Regions

Using regions, we can directly mmap device memory into host memory for easy access to the NVMe controller. VFIO provides structs for using mmap to directly map the NVMe device into memory. Using `VFIO_DEVICE_GET_REGION_INFO` we can attain the length and the offset needed for mmap.

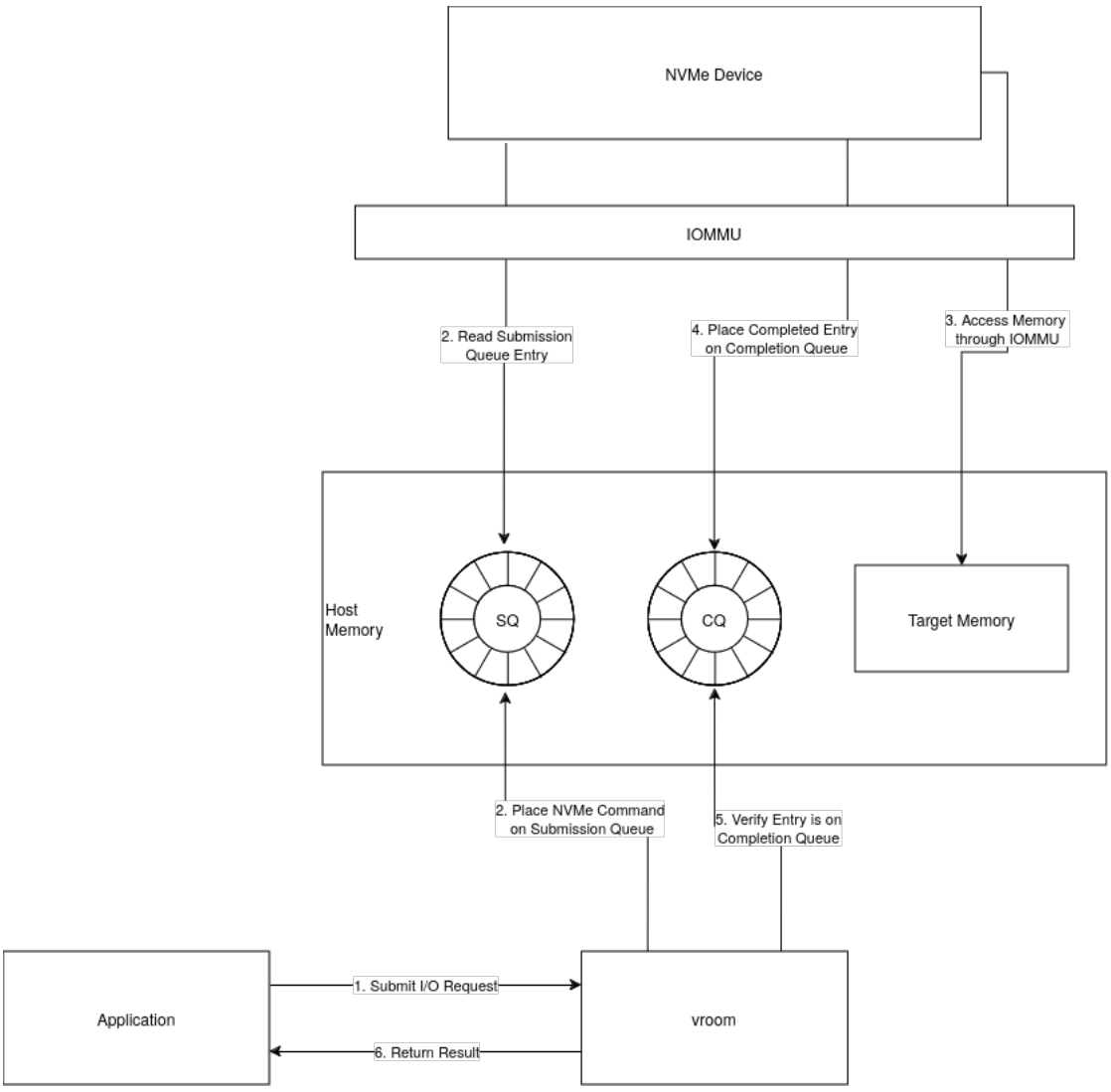


Figure 4.1: I/O operation vroom

4.2.7 Groups and Containers

VFIO uses group to distinguish between groups of devices which can be isolated from the host system. In the ideal case, every device would only be part of one group in order to increase security by providing single-device isolation. Groups are the smallest unit size on a system to ensure secure user access. To further reduce overhead from the IOMMU Containers are used in VFIO, which can hold multiple groups. These containers can be used to ease translation and reduce TLB page faults. In our implementation we use one group and container each for our NVMe device.

4.3 IOMMUFD

The IOMMU File Descriptor user API (IOMMUFD) offers a way of controlling the IOMMU subsystem using file descriptors in user-space [6]. It allows management of I/O address spaces (IOAS), enabling mapping user space memory on the IOMMU. IOMMUFD has only been recently added to the Linux Kernel in December of 2022. E.g. Debian 12 does not include it, Fedora 40 does, but it is not enabled in the kernel configuration. Considering that it is not widely available or enabled on many distributions, our driver offers both options of using the IOMMU. The performance tests are done using the 'legacy' VFIO-only way, but it can be assumed that the framework/user api does not have a noteworthy performance impact. The device file descriptor, which was previously attained with `VFIO_GROUP_GET_DEVICE_FD` can now be gotten through opening the character device /dev/vfio/devices/vfioX. By using this character device pointer we can claim the ownership over the VFIO device. That way VFIO does not rely on group/container/iommu drivers.

4.3.1 IOAS

5 Evaluation

In this chapter, we analyse the performance impact of the IOMMU, directly comparing it to the physical address approach. We will not be comparing the performance of memory allocation and mapping as in high throughput applications it should be negligable. The main focus lies on the IOMMU itself and how it performs with different page sizes.

5.1 Setup

All benchmarks are run on a system with an Intel Xeon E5-2660 with 256 GB of RAM running Ubuntu 23.10 with a 1 TB Samsung Evo 970 Plus NVMe SSD.

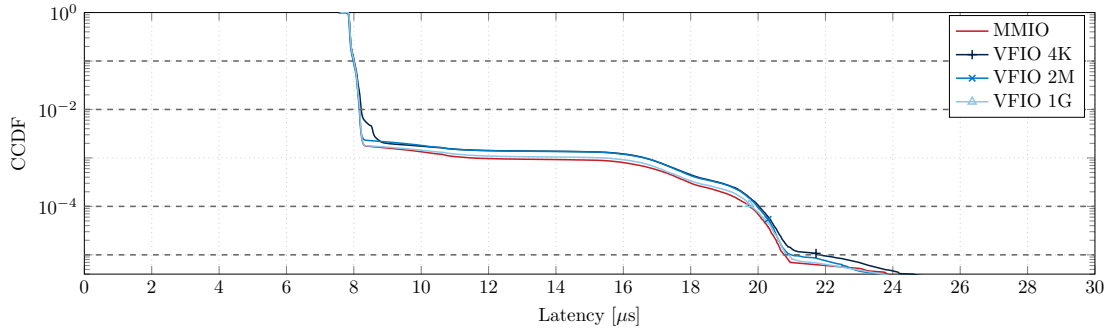
During our tests we will use 4KiB unit sizes for read and write accesses. As Linux as well as our IOMMU supports 4KiB, 2MiB and 1GiB page sizes we will test and analyse how it affects the overall performance.

5.2 Overall Latency and Throughput

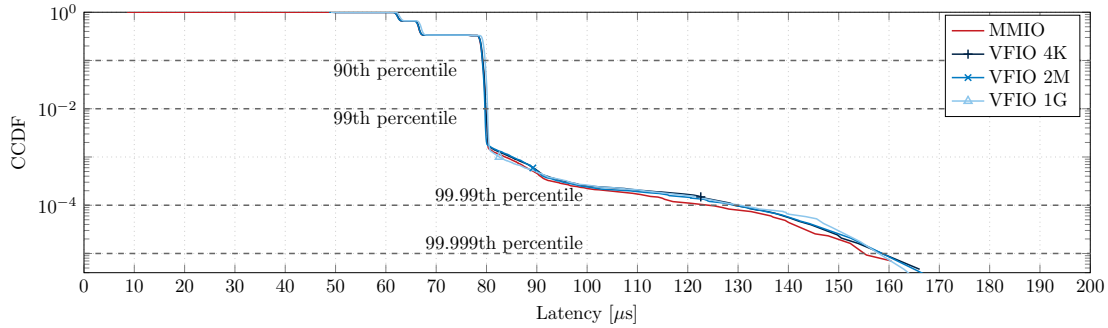
First, we will compare the VFIO implementation to the MMIO implementation using latency and throughput tests. In these tests, we can see that there is practically a negligable amount of overhead. As this test only uses a one page buffer at maximum, the buffer is constantly stored in the IOTLB. This test uses one buffer from which the NVMe driver reads/writes to. This buffer and the Queues can fit on the IOTLB. Fetching addresses from the IOTLB is very efficient and thus, no significant performance impact occurs.

5.3 IOTLB

As the size of the IOTLB is not stated in hardware and VT-d specifications, we use a latency test to see the behaviour of the IOMMU. We can assume that the IOTLB entry count must be a power of two. In order to isolate the effect of the IOMMU we use the median of random write latencies on the emptied NVMe. We use a test that repetitly writes one byte to the ssd from to 8, 16, 32, 64, ... pages. Taking the median



(a) Random write



(b) Random read

Figure 5.1: Tail latencies

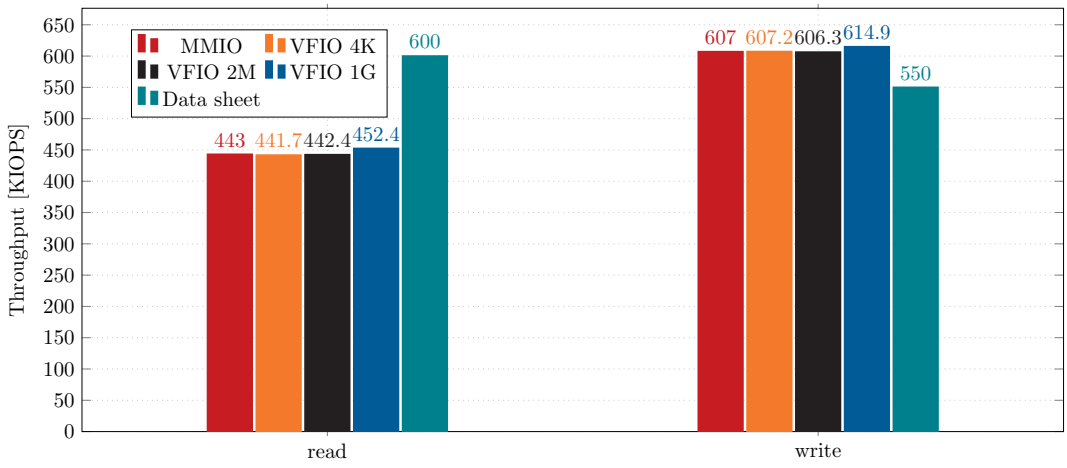


Figure 5.2: Write and Read Operations with queue depth 32 and 4 threads

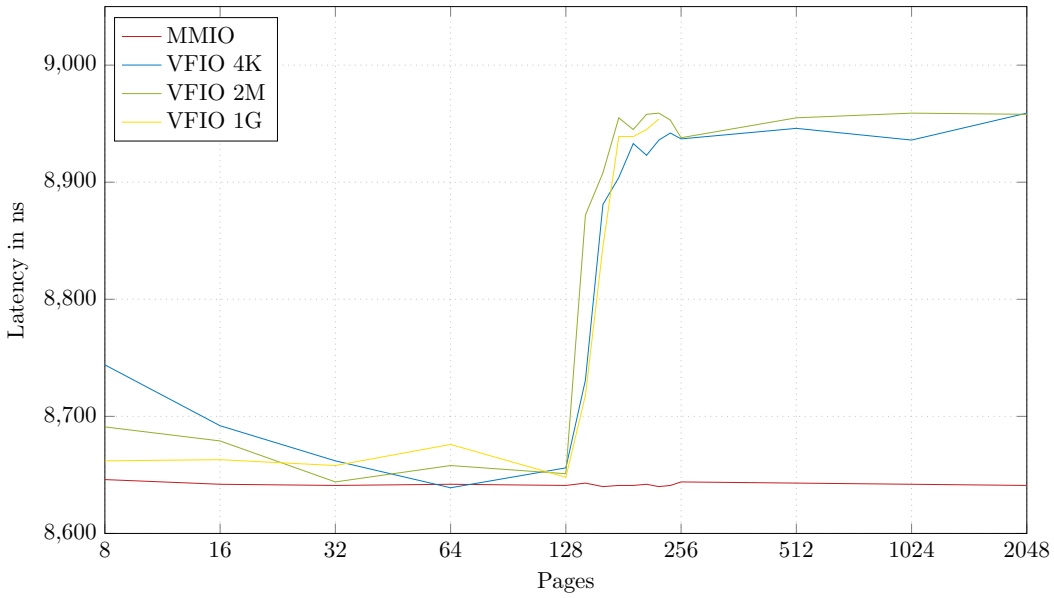


Figure 5.3: Pagesize Medians

and comparing them we can figure out where a latency spike occurs and can then derive the IOTLB size. We configure the queues, buffer and prp-list to each take up one page, resulting in 6 allocated pages before the actual workload. This test is done using memory-mapped I/O and the IOMMU with 4Kib and 2Mib pages. In order to test 1GiB pages, we first need to increase the ulimit settings, as memlock limits the amount of locked-in-memory address space. On the resulting graph Figure 5.3 we can observe a performance spike of around 200 nanoseconds for each write between 128 and 256 allocated pages.

5.4 IOMMU modes

6 Conclusion

In this thesis, we improved vroom's safety by implementing IOMMU support. We come to the same conclusion as SPDK. The advantages of using the VFIO such as variable page support by not using page pinning as well as the ability to run the driver without root privileges overweigh the small performance impact that can be registered in niche cases. Considering that IOMMU technology has seen a rise in popularity in the use of hardware passtrough for virtualization it is also likely that in the future the IOMMU performance and the IOTLB size will increase, further closing the gap.

List of Figures

List of Tables

Listings

Bibliography

[1] M. Ben-Yehuda, J. Xenidis, M. Ostrowski, K. Rister, A. Bruemmer, and L. van Doorn. "The price of safety: Evaluating IOMMU performance." In: *Ottawa Linux Symposium (OLS)* (Jan. 2007), p. 13.

[2] *Crate bindgen*. URL: https://docs.rs/bindgen/0.69.4/bindgen/ (visited on 07/22/2024).

[3] *Direct Memory Acccess (DMA) From User Space*. URL: https://spdk.io/doc/memory.html (visited on 07/22/2024).

[4] L. Doan and M. Day. "CrowdStrike Crash Affected 8.5 Million Microsoft Windows Devices." In: *Bloomberg* (July 20, 2024). URL: https://www.bloomberg.com/news/articles/2024-07-20/crowdstrike-crash-affected-8-5-million-microsoft-windows-devices (visited on 07/23/2024).

[5] R. Eikenberg, C. Kunz, and V. Zota. "CrowdStrike-Fiasko: Der Null Pointer ist Schuld." In: *heise online* (July 20, 2024). URL: https://www.heise.de/hintergrund/Fataler-Fehler-bei-CrowdStrike-Schuld-war-ein-Null-Pointer-9807896.html (visited on 07/23/2024).

[6] J. Gunthorpe and K. Tian. *IOMMUFD*. URL: https://docs.kernel.org/userspace-api/iommufd.html (visited on 07/08/2024).

[7] *HugeTLB Pages*. URL: https://docs.kernel.org/admin-guide/mm/hugetlbpage.html (visited on 07/25/2024).

[8] Intel. *80286 Microprocessor with memory management and protection*. Sept. 1993. URL: https://datasheets.chipdb.org/Intel/x86/286/datashts/210253-016.pdf (visited on 07/23/2024).

[9] *Null References: The Billion Dollar Mistake*. URL: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/ (visited on 07/23/2024).

[10] T. Pirhonen. "Writing an NVMe Driver in Rust." BA thesis. Technical University of Munich, 2024.

[11] D. Rovella. "Tech Meltdown Collapses Systems Worldwide." In: *Bloomberg* (July 20, 2024). URL: https://www.bloomberg.com/news/newsletters/2024-07-19/bloomberg-evening-briefing-tech-meltdown-collapses-systems-worldwide (visited on 07/23/2024).

[12] *Storage performance Development Kit*. URL: https://spdk.io/ (visited on 07/22/2024).

[13] *Transparent Hugepage Support*. URL: https://docs.kernel.org/admin-guide/mm/transhuge.html (visited on 07/23/2024).