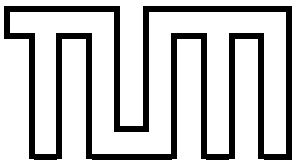


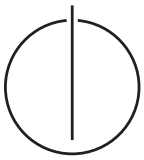
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

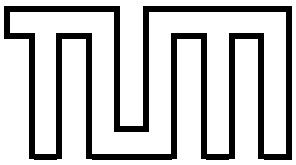
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Effects of Linux VFIO for User Space I/O

Adrian Simon Würth





SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

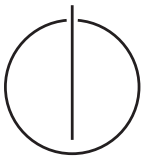
TECHNISCHE UNIVERSITÄT MÜNCHEN

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, August 9, 2024 Adrian Simon Würth

# Abstract

Userspace storage drivers like SPDK rely on Direct Memory Access to read or write to memory. DMA bypasses the CPU and directly performs the I/O operation on the memory. This can be a huge security risk as malicious firmware or faulty operations could lead to detrimental access to memory, potentially extracting data or corrupting the system. The workaround to this is the IOMMU, which maps physical to I/O virtual addresses, similar to the CPU's MMU. As address translation can be a memory and performance intensive operation, it is necessary to look at how impactful the IOMMU is on the whole driver performance. In this thesis we implement IOMMU support for a userspace NVMe driver written in Rust and probe the performance, directly comparing using DMA with physical addresses and with IOMMU I/O virtual addresses. We show that it is possible that with the use of 2 MiB pages it is possible to achieve practically equivalent performance, with increased system security and being able to run the driver with no root permissions. Additionally, we implement support for IOMMUFD, a modern replacement for the VFIO IOMMU Interface.

# 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, proposed the idea of how null pointers are the reason for at least a billion dollars in damages [14]. This quote could not be more important than at this time. In July 2024, Microsoft devices faced what has been described as the "most spectacular IT meltdown the world has ever seen" [19]. This meltdown affected 8.5 million Microsoft Windows devices and severely impacted public institutions, including critical infrastructure like hospitals and airports [5]. During the error analysis of Crowdstrike, the company that deployed the faulty code to the Windows kernel, it was soon clear that null pointer deferencing in C++ caused the systems to crash [6]. In the root cause analysis paper by Crowdstrike, it was revealed that proper compile time validation and missing runtime array bounds checks were a big part of the error [7].

Poor code analysis and faulty error handling are common occurrences in system-level code. The damage that can be done by a single ring 0 driver like Crowdstrike's Falcon software shows how critical it is to achieve memory safety. By using Rust, a memory-safe yet highly performant programming language with a restrictive compiler, we could drastically improve security and memory safety. We can witness Rust's influence on the systems development community since even the Linux kernel, which has been using C for almost 30 years without accepting any other languages like C++, now allows Rust code in its codebase [18].

However, it's also important to consider Rusts safety limits. While using Rust for a driver improves the overall safety of the process while not compensating on performance, direct memory and I/O operations have to be implemented in an unsafe way. A userspace driver using physical DMA addresses enables a device to practically have full access to the memory and potentially do detrimental I/O operations. Malicious firmware attacks are a rising threat. In order to enforce safety at the device level, we need to make 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 [2].

The primary goal of this thesis is to examine how the IOMMU impacts performance in the context of userspace I/O. We demonstrate this by implementing IOMMU support

on vroom, a NVMe driver written in Rust [17], and comparing it to using physical addresses. We use the Linux framework VFIO to implement the IOMMU functionality, which has the added benefit of enabling the driver to run without root privileges. Additionally, we will be looking at IOMMUFD, a modern replacement for VFIO's IOMMU API.

# 2 Background

## 2.1 vroom

Vroom is a userspace NVMe driver written in Rust. As of this writing, it offers good performance and the functionality required for general I/O operations, but it is not yet production ready. Unlike interrupt-driven drivers, vroom uses polling to determine the state of the I/O operations. Polling is often preferable in high-performance applications, as interrupts are relatively performance-intensive operations [22].

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. A deeper explanation of the steps will be done in chapter 4. As vroom does not have a kernel driver part, 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 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 [11], they have since become the de facto standard for addressing memory on computers. By providing processes with a virtual address space instead of physical addresses, every process is isolated and prevented access to memory regions without permission. 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 on modern x86-64 architectures.

The translations 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. 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 results in 4 memory accesses. Thus, a page table walk is a performance costly operation. To circumvent this, an Translation Lookaside Buffer (TLB) is used to cache translation.

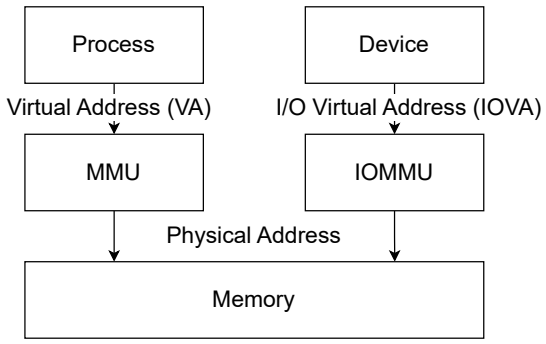



**Figure 2.1:** MMU and IOMMU relation to physical memory, adapted from [16]

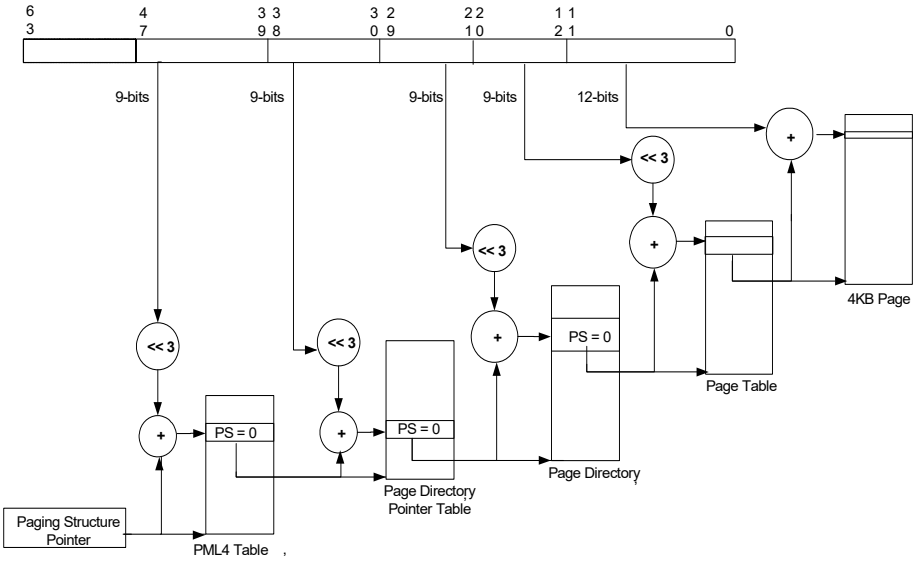
The TLB is very performant to access. Frequent access to the same address can be done at a fraction of the time needed to perform a page table walk. A TLB miss describes the scenario in which a physical address needs to translated but it has no entry in the TLB, resulting in a expensive page walk.

The advantages and success of the CPU's MMU as well as the introduction of the PCIe bus specification have incentivized hardware manufacturers to apply this concept on peripheral device buses. 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.

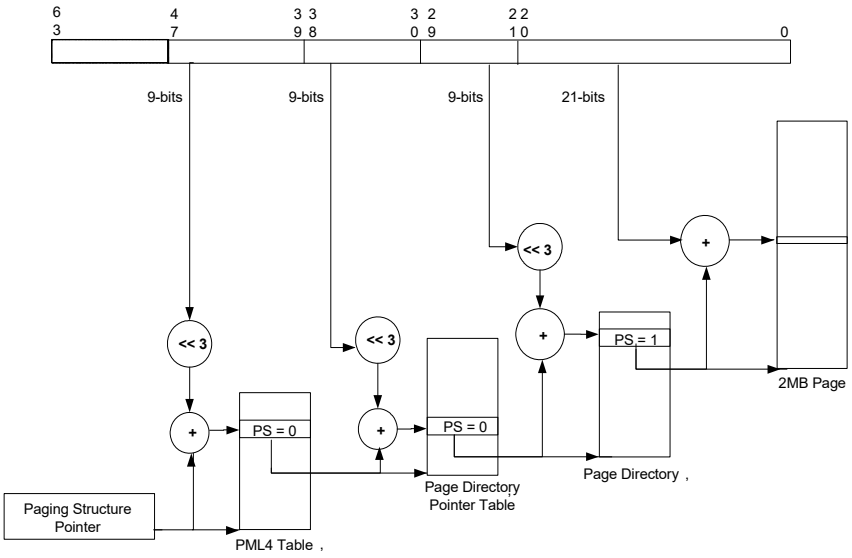
Using DMA Remapping (DMAR) the CPU is bypassed and the direct memory access translated by the IOMMU. The IOMMU paging structures consist of 4KiB page tables storing 512 8-byte entries. The IOMMU uses the upper portions to determine the location of the stored page tables, and the lower portion of the address as page offset. In the case of 4 KiB pages its 12 bits, for 2MiB its 21. Mappings can be allocated using the VFIO framework.

## 2.3 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 [9].




**(a)** Translation of a 4KiB page




**(b)** Translation of a 2MiB page

**Figure 2.2:** VT-d Paging structure for translating a 48-bit address to a 4 KiB and a 2 MiB page, grabbed from [12]

## 2.4 Memory Mapped I/O

Memory Mapped I/O, in contrast to Port-Mapped I/O, places Devices in the same address space as the main memory. Instead of using Assembly instructions like `in` or `out`, a device can be addressed just like the RAM can be. The addressing of this memory itself can currently be done using two ways. Either accessing it with the physical address, or using a hardware translation device like the I/O Memory Management Unit (IOMMU). Before our implementation, vroom currently only supports the former.

Using the Linux syscall `mmap(2)` we can create a mapping in the virtual address space of our process. These mapping use so-called Pages. Pages are units of memory with a predefined size, the pagesize. The supported pagesizes can differ depending on the cpu architecture. As we use x86-64 for our tests, we have the default pagesize of 4 KiB and the bigger pagesizes 2 MiB and 1 GiB. Pages allocated with these bigger pagesizes are called Hugepages. In addition to mapping memory into our virtual address space, calling `mmap` with no address can be used as an allocation method for big data amounts [13].

## 2.5 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). Using the IOMMU the request gets intercepted and translated to the physical address.

## 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 4 KiB pages, only an address space of 16 MiB 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 4 KiB. Linux currently provides two ways of using Hugepages. In the optimal case, using a 2 MiB or 1 GiB page size should result in a 512 or 262144 times reduction in cache misses compared to 4 KiB 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,

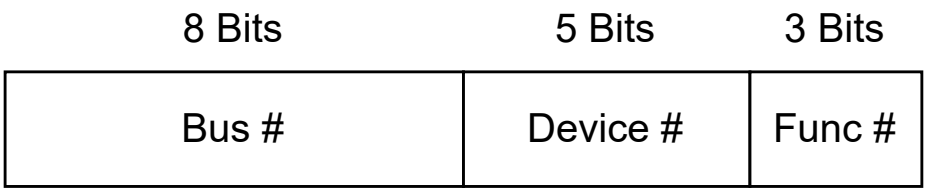



**Figure 2.3:** segmented PCI identifier

they are mounted as a (pseudo) filesystem called hugetlbfs, which lays in the directory `/mnt/huge`. 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`[10].

- **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. THPs provides a way of utilising the TLB effectively without reserving huge amounts of memory. The `khugepaged` daemon scans memory and collapses sequences of basic pages into bigger pages. THPs can either be enabled, disabled or only be used on `madvise(MADV_HUGEPAGE)` memory regions [23].

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.

## 2.7 Peripheral Component Interconnect Express

PCIe is a standard for peripheral device busses. Each device on the PCI bus has a unique PCI address, which is segmented into for parts, as seen in Figure 2.3.

## 2.8 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.

## 2.9 Non-Uniform Memory Access

Non-Uniform Memory Access is a type of system architecture for multi-CPU systems. Hardware resources are bundled into "nodes". These nodes can contain multiple CPUs, memory or I/O buses [20]. This can relieve stress on the memory bus and improve performance if the memory is part of the same node as the CPU. On the other hand, performance may decrease if the memory is situated in another node. Using `numactl` it is possible to specify the locality of memory and cpu and run a process on one node [15].

## 2.10 Rust

While userspace drivers can theoretically and practically be written in any language, as proven by the network driver Ixy section 3.3, Rust excels as it not only offers memory-safety but memory-safety without garbage collection. This is especially important, as garbage collected languages have overhead and latency spikes, which can lower performance. Another huge factor is that Rust, like C, does not use exceptions. By handling, or even better, being forced to handle errors assures that no rogue exception can take down critical code infrastructure. Additionally, Rust provides low-level access while also offering a high-level development experience through zero-cost abstractions.

# 3 Related Work

## 3.1 Data Plane Development Kit

The Data Plane Development Kit (DPDK) is a framework for developing user-space network card drivers. It allows for high performance network applications. It can either run using direct memory access with physical addresses or using VFIO [1]. DPDK offers polling drivers for a variety of network cards. It is one of the most successful projects in the world of userspace drivers and has influenced many advances in the IOMMU space.

## 3.2 Storage Performance Development Kit

The demand for high-speed userspace drivers in storage applications inspired the development of the Storage Performance Development Kit (SPDK). SPDK uses some shared libraries and architecture with DPDK. Especially through the wide adoption of the NVMe protocol and the standardization of said protocol, only one driver for all NVMe SSDs has to be developed. NVMe is a storage protocol which is widely used, modern and highly performant. Therefore it is a protocol for which many drivers have been written, including userspace drivers. The Storage Performance Development Kit (SPDK) provides “a collection of tools and libraries for writing high performance, scalable, user-mode storage applications” [21]. 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 [4]. Even though SPDK is the established userspace NVMe driver option, the drawbacks include its high complexity even for simple applications, as well as it being written in C.

## 3.3 ixy

Ixy is a user space network driver for network interface cards. In the worst case a packet is 64 Bytes long and two packets fit on one 4K Page. In their implementation of

IOMMU support a performance decrease of up to 75% [9].

# 4 Implementation

## 4.1 Virtual Function I/O

Virtual Function I/O (VFIO) is an IOMMU agnostic framework for exposing devices to userspace.

To use the IOMMU we need to initialize the IOMMU, VFIO, DMA and the NVMe device:

1. **Initialize the IOMMU:** First, we need to initialize the IOMMU. This is done using the VFIO IOMMU API, which is an interface for the IOMMU driver.
2. **Initialize the VFIO group:**
3. **Enable DMA in the PCI configuration space of the NVMe device**
4. **Use `mmap` to map NVMe base address register to memory**

VFIO consists of two parts, the `vfio-pci` driver and an IOMMU API (`vfio_iommu_type1`). The VFIO PCI driver can be bound to a PCI device. This allows using `mmap(2)` to map the PCI device registers into memory. The type1 VFIO IOMMU API is used for mapping and unmapping address translations in the IOMMU. Alternatively, the IOMMUFD API can be used instead of the container API, which currently is not feature complete, but will eventually replace the container based solution [24]. 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 [3]. 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 are used to provide locality of behaviour, secure error handling, while also minimising the calling of unsafe code. In Listing 4.1 the macro for the mmap syscall can be seen. As part of our error handling, we introduce an error enum variant for each syscall. The macro is marked as unsafe in its name in order to not hide any unsafe code.

**Listing 4.1:** Syscall `mmap` macro, with own error variant

```
#[macro_export]
macro_rules! mmap_unsafe {
    ($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 Binding NVMe to `vfio-pci`

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 4.2 the initialization script for the Vfio driver is shown.

**Listing 4.2:** 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 IOMMU

To initialise the IOMMU, we first need to get the container file descriptor. The container is accessible under the path `/dev/vfio/vfio`. Using the raw container file descriptor, we can use the following `ioctl` calls to initialise the IOMMU.

**Listing 4.3:** ioctl calls needed for IOMMU initialization

```
ioctl_unsafe!(container_fd, VFIO_GET_API_VERSION)
ioctl_unsafe!(container_fd, VFIO_CHECK_EXTENSION, VFIO_TYPE1_IOMMU)
ioctl_unsafe!(group_fd, VFIO_GROUP_GET_STATUS, &group_status)
ioctl_unsafe!(group_fd, VFIO_GROUP_SET_CONTAINER, &container_fd)
ioctl_unsafe!(container_fd, VFIO_SET_IOMMU, VFIO_TYPE1_IOMMU)
ioctl_unsafe!(group_fd, VFIO_GROUP_GET_DEVICE_FD, pci_addr)
ioctl_unsafe!(container_fd, VFIO_IOMMU_GET_INFO, &iommu_info)
```

Excluding the Status and Info calls, the functionality consists of initialising the IOMMU for the device groups by setting the container on the groups, enabling Type1 for the IOMMU and fetching the device file descriptor. With the device file descriptor, we gain access to the device regions through the VFIO device API, allowing us to `mmap` the NVMe BAR into memory.

### 4.2.3 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

### 4.2.4 DMA

To enable DMA we need to set a bit in the PCIe device registers. This is done in the command register of the PCIe configuration space.

**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.

**Listing 4.4:** Mapping memory for DMA

```
let mut iommu_dma_map = vfio_iommu_type1_dma_map {
    argsz: mem::size_of::<vfio_iommu_type1_dma_map>() as u32,
    flags: IoctlFlag::VFIO_DMA_MAP_FLAG_READ
           | IoctlFlag::VFIO_DMA_MAP_FLAG_WRITE,
    vaddr: ptr as u64,
    iova: ptr as u64,
    size,
};

ioctl_unsafe!(
    *container_fd,
    IoctlOperation::VFIO_IOMMU_MAP_DMA,
```

```
    &mut iommu_dma_map
)?;

let iova = iommu_dma_map.iova as usize;
```

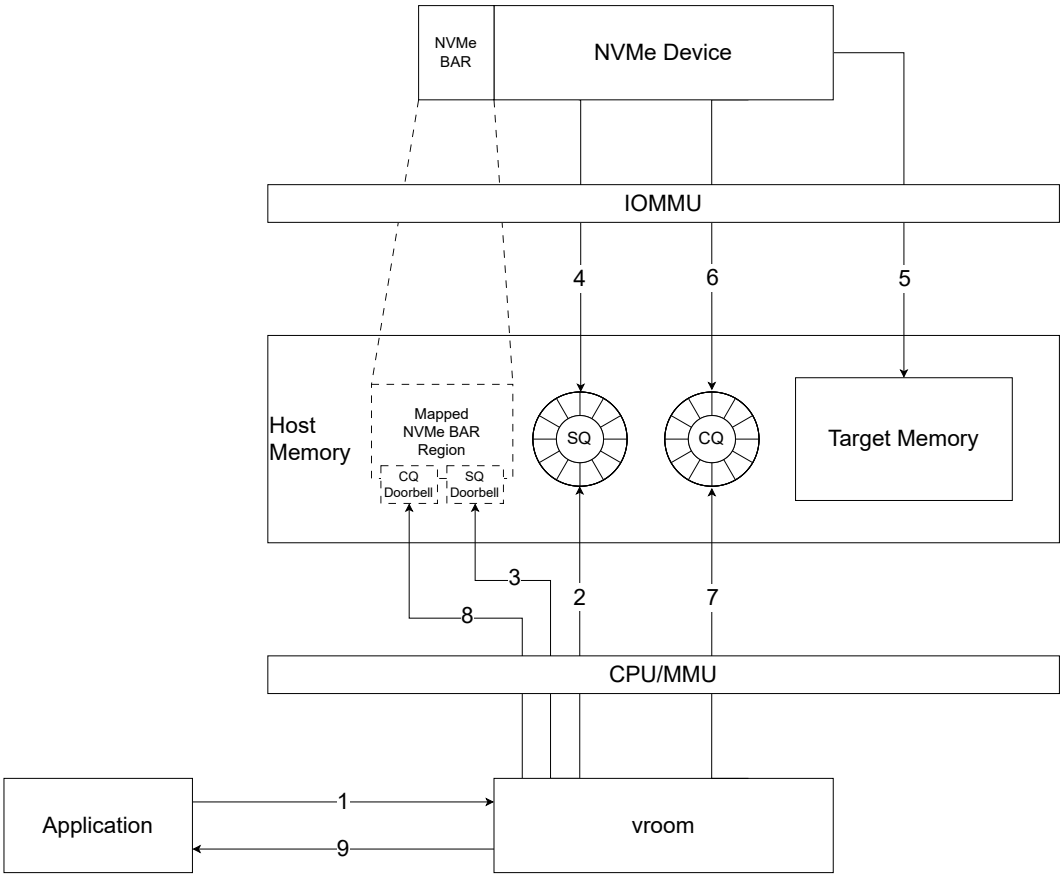



**Figure 4.1:** I/O operation using vroom with enabled IOMMU

On Figure 4.1 an I/O operation timeline graph can be seen. The timeline is described as followed:

1. **I/O function call:** The application calls a read/write method on vroom
2. **Command Submission:** Vroom creates a `NvmeCommand` struct and places it on the Submission Queue head.
3. **Ring SQ Doorbell:** Vroom places the submission queue head address in the doorbell register. The doorbell register is part of the NVMe BAR region, which is mapped to memory.
4. **Take Command:** The NVMe takes the command from the SQ.

5. **Perform I/O:** The NVMe uses the IOMMU to access the host memory via DMA and performs the read/write command.
6. **Complete I/O:** The NVMe places a `NvmeCompletion` struct instance on the head of the Completion Queue.
7. **Polled CQ:** By polling the CQ, vroom can process the CQ entry.
8. **Ring CQ Doorbell:** After processing the CQ entry, vroom rings the CQ Doorbell, in order to notify the NVMe controller, that the Completion Queue has been processed.
9. **Notify Application:** Vroom notifies the Application of the success of the I/O operation. The application can continue running.

**Unmapping DMA** Unmapping DMA happens when the process exits, yet for performance and application reasons we implement the unmap_dma function which can be used to unmap a DMA. Using the `VFIO_IOMMU_UNMAP_DMA` `ioctl` operation we can unmap the memory, and finally free it by using `munmap`.

### 4.2.5 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.

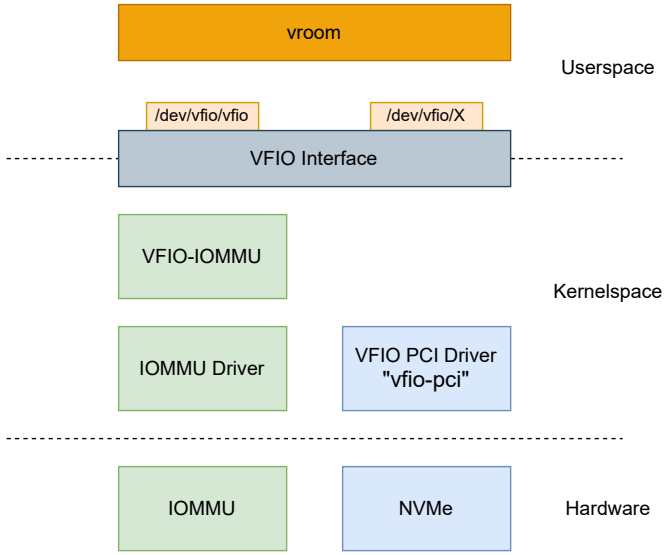
### 4.2.6 Groups and Containers

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.

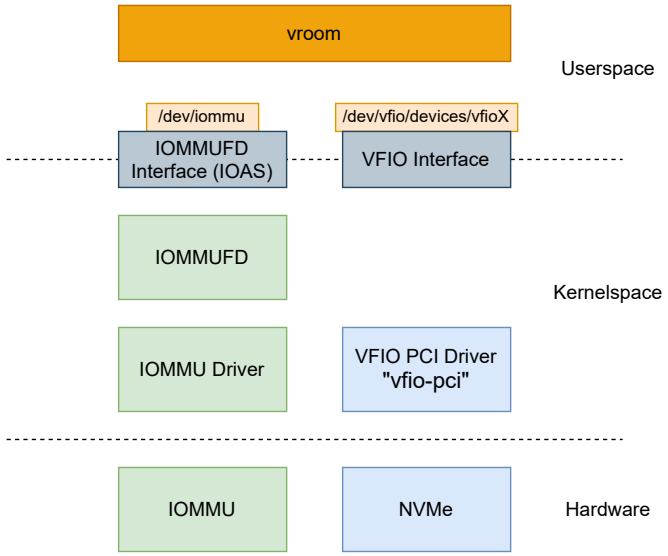
## 4.3 IOMMUFD

The IOMMU File Descriptor user API (IOMMUFD) offers a way of controlling the IOMMU subsystem using file descriptors in user-space [8]. It replaces the VFIO IOMMU

API using containers with the IOMMUFD API. 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 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 container way. 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. I/O Address Spaces (IOAS) are the equivalent to a VFIO container in legacy VFIO.




**(a)** VFIO with Containers




**(b)** VFIO with IOMMUFD (IOAS)

**Figure 4.2:** Layer diagrams of VFIO with VFIO Container API and IOMMUFD, partly adopted from [25]

# 5 Evaluation

In this chapter, we analyse the performance impact of the IOMMU, directly comparing it to the physical address approach. To compare the both approaches fairly, we dont include allocation and mapping times and perform them upfront. The main focus lies on the IOMMU itself and how it performs with different page sizes. All performance tests use the Container IOMMU API as it currently remains the widely adopted VFIO variant. Due to the page size limitation of using physical addresses, we cannot compare the IOMMU to physical addresses using 4 KiB pages.

## 5.1 Setup

We use two systems two benchmark the driver's performance. Both systems run Ubuntu 23.10 with Linux kernel version 6.5.0-42. Despite the NVMe specifications maximum capability of 65536 I/O queues, our SSDs support a more reasonable amount of 64 I/O queues, which seems to be a typical amount. We use 1 thread per 1 I/O queue in our multithreaded tests. Turbowrite is a Samsung technology that drastically speeds up write latencies in the so-called "Turbowrite" buffer with the size of 42 GB of the NVMe, as shown in [17]. In order to use the Turbowrite NVMe to its maximum potential, most tests are conducted in said buffer to avoid the NVMe being the bottleneck instead of the IOMMU. As both systems are servers with NUMA architectures, we use `numactl -N 0 -m 0` for all tests to maintain consistency. The NVMe SSDs are formatted to 512-byte blocksize. Additionally, all standard tests are run with the `iommu=nomerge` kernel parameter to prevent the IOMMU from merging entries together, which can influence the results.

| CPU | Memory | NVMe | Capacity | Count |
|---|---|---|---|---|
| Intel Xeon E5-2660v2 | 251 GiB | Samsung Evo 970 Plus | 1 TB | 1 |
| AMD EPYC 7713 | 1007 GiB | Samsung PM9A3 | 1.92 TB | 8 |

**Table 5.1:** Specifications of systems used in performance testing

| CPU | Clock | Cores | Virtualization | Year |
|---|---|---|---|---|
| Intel Xeon E5-2660v2 | 2.2 GHz | 10 | VT-d | 2012 |
| AMD EPYC 7713 | 2.0 GHz | 64 | AMD-V | 2021 |

**Table 5.2:** CPUs of the systems

| NVMe | Maximum Queue Count | Maximum Queue Size | Turbowrite | Usage |
|---|---|---|---|---|
| Samsung Evo 970 Plus | 64 | 16384 | Yes | Consumer |
| Samsung PM9A3 | 64 | 16384 | No | Enterprise |

**Table 5.3:** NVMe(s) of the systems

As Linux as well as our IOMMU supports 4 KiB, 2 MiB and 1 GiB page sizes we will test and analyse how it affects the overall performance.

## 5.2 Overall Latency and Throughput

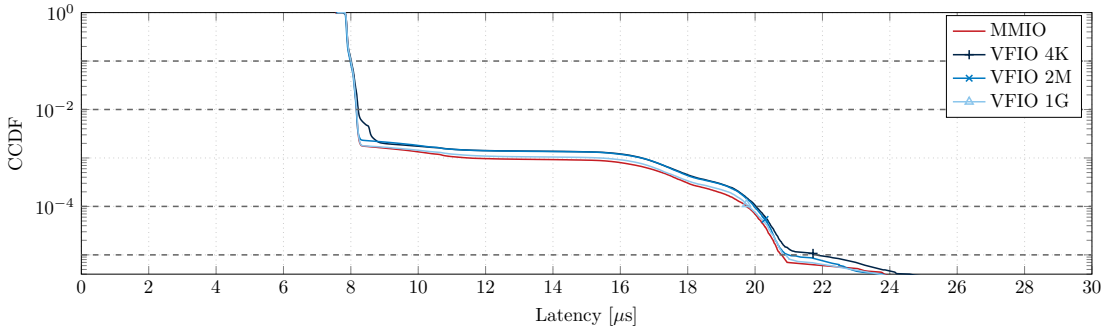
First, we will compare the VFIO implementation to the MMIO implementation using latency and throughput tests. In these tests, we repeatedly write/read from a 4 KiB buffer in the memory. Each I/O operation uses a 4 KiB unit size.

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.

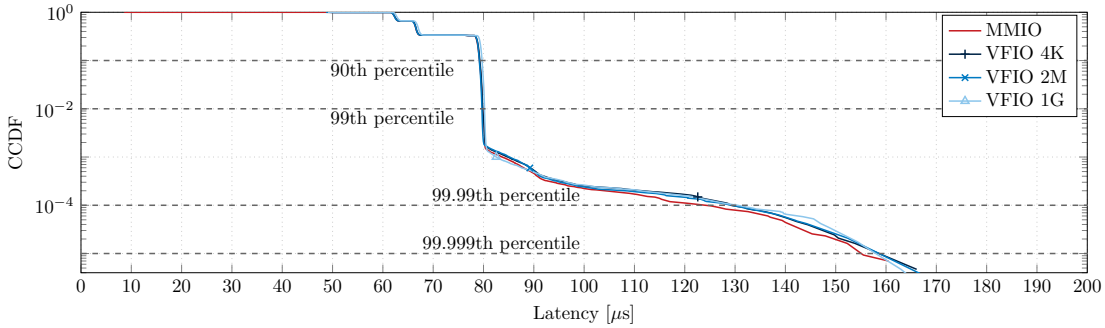
This test uses one buffer from which the NVMe driver reads/writes to. This buffer and the Queues can fit on the IOTLB. Fetching addresses from the IOTLB is very efficient and thus, no significant performance impact occurs.

## 5.3 Determining IOTLB size

As the size of the IOTLB is not stated in hardware and VT-d specifications, we use a latency test to analyse 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 aim to analyse the latencies of the fastest operation the NVMe can perform. The fastest operation is using random writes with the smallest blocksize of 512 B on an empty NVMe drive. We repeatedly perform this operation on a number of pages that is an increasing power of two. Taking the median and comparing them we can figure




**(a)** Random write




**(b)** Random read

**Figure 5.1:** Tail latencies

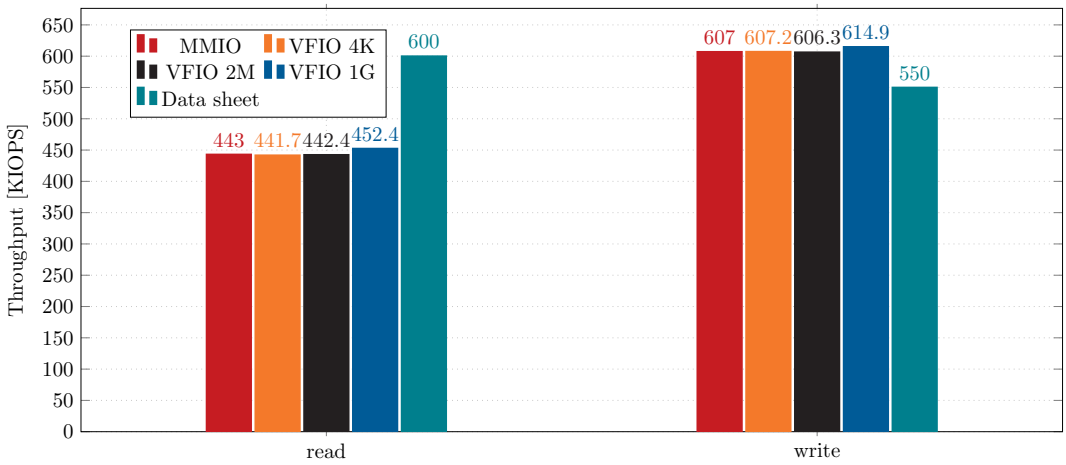



**Figure 5.2:** Write and Read Operations with queue depth 32 and 4 threads

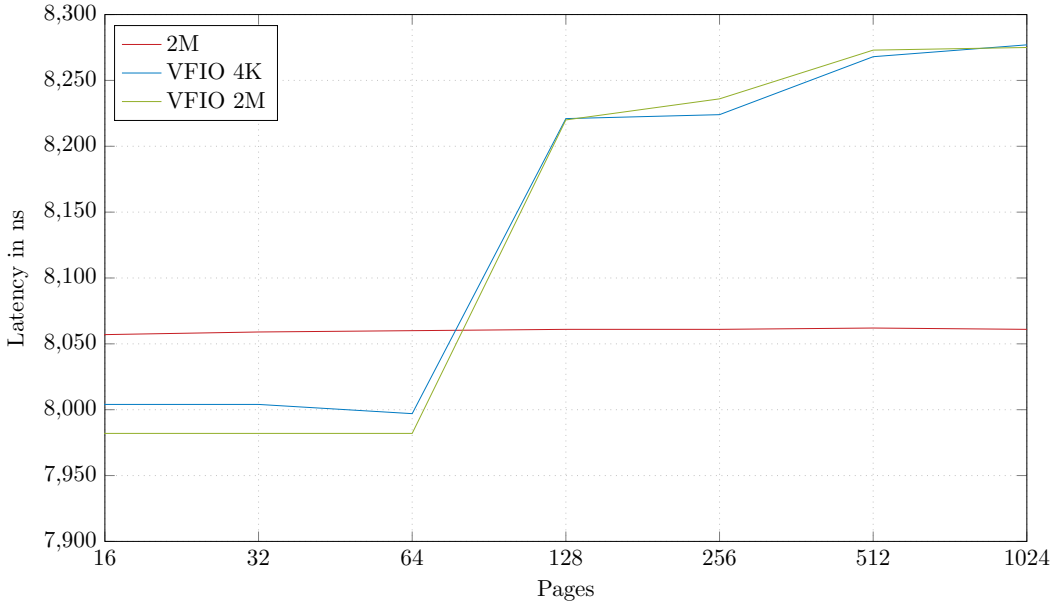



**Figure 5.3:** Latencies of random writes on an emptied SSD with increasing host memory pages on the Intel system

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 without the IOMMU with 2 MiB pages and the IOMMU with 4 KiB, 2 MiB and 1 GiB pages. In order to test 1 GiB pages, we first need to increase the ulimit settings, as memlock limits the amount of locked-in-memory address space.

**Results of Intel Xeon** In the resulting graph Figure 5.3 we can observe a performance spike of around 300 nanoseconds for each write between 128 and 256 allocated pages. In the case of 4 KiB pages, this is a memory size of only 512 KiB. Using this information, we can assume that the IOTLB has the same size for each pagesize, as well as it being 128 entries of size.

**Results of AMD Epyc** On the AMD IOMMU, we can see a performance spike that occurs at 64-128 pages for 2 MiB and 1 GiB page sizes and at 256-512 pages. We can therefore assume that the IOTLB size depends on the pagesize unlike on the Intel CPU. The performance itself only decreases by about 60 ns, which is a five fold performance

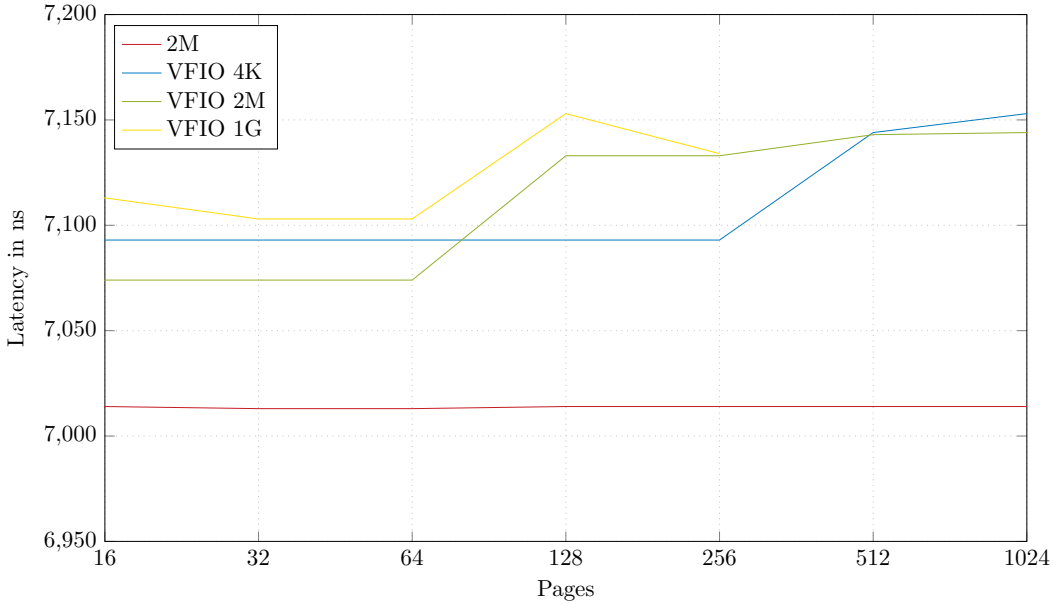
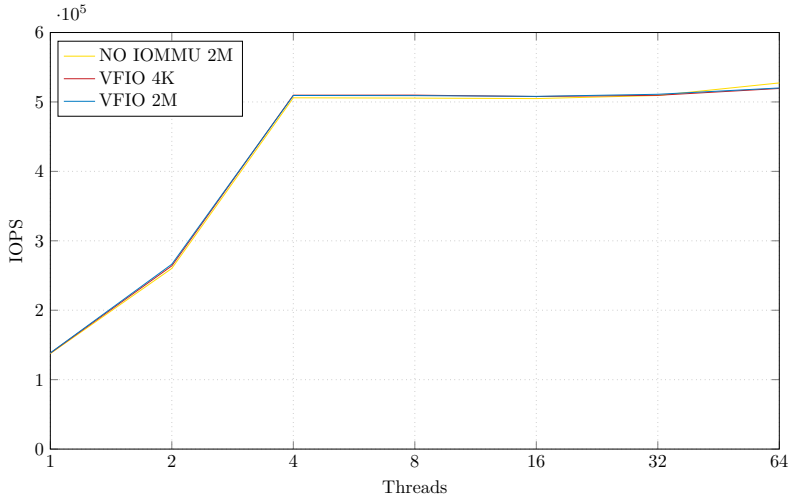



**Figure 5.4:** Latencies of random writes on an emptied SSD with increasing host memory pages on the AMD system

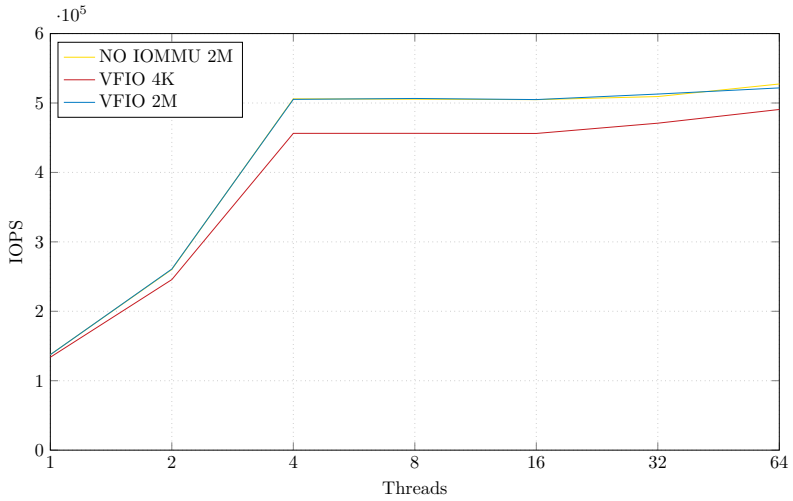
increase of page walks compared to the intel cpu system.

## 5.4 Multithreaded Random Writes

When using multithreaded random writes with queue depth 1, in Figure 5.5 we can mostly see the same performance for 1 page, until 32 threads. This is because we allocate a buffer for each thread. For each thread, the submission queue, the completion queue and the buffer take up each one entry of the IOTLB. Therefore, with 32 threads we use 96 pages, exceeding the 64 entries we can store in the IOTLB. Using a 2 MiB buffer, we see around 10% less performance using 4 KiB pages, as even with one thread, the page count exceeds the possible entries in the IOTLB. The performance of the 2 MiB Vfio implementation stays the same as with 1 4 KiB buffer, as the 2 MiB buffer fits in one IOTLB entry.




**(a)** 1 4 KiB buffer per thread




**(b)** 1 2 MiB buffer per thread

**Figure 5.5:** QD1 Random Write Throughput with multithreading

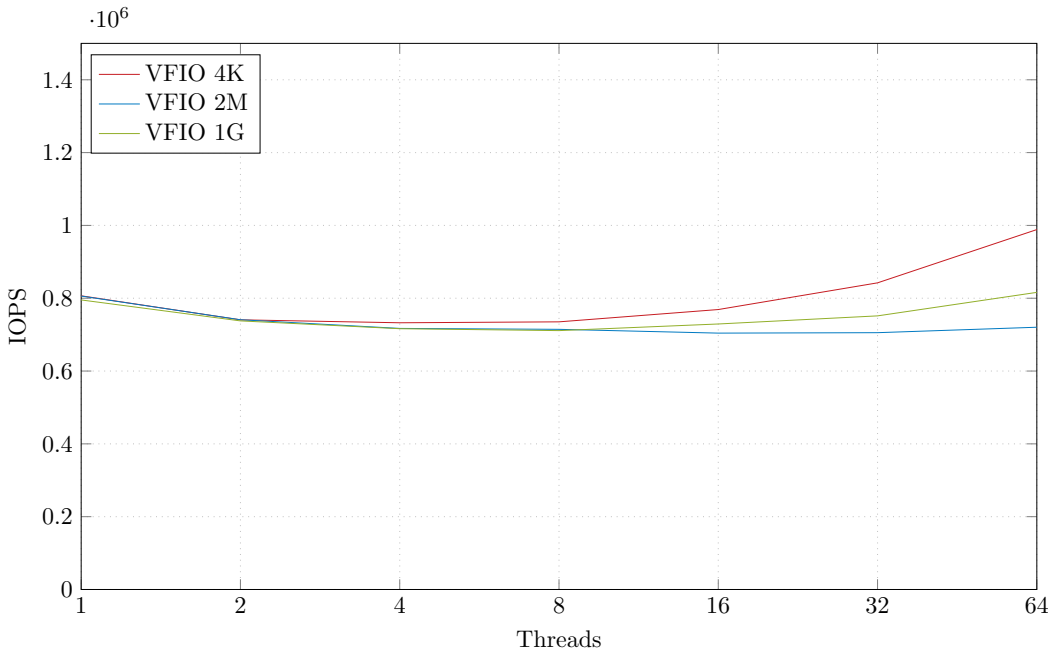



**Figure 5.6:** Latencies of random writes on an emptied SSD with queue depth 32 and increasing amount of threads

## 5.5 Increasing Queue Depth

To further test performance we can increase the queue depth. This metric describes how many outstanding commands can be placed on each queue.

## 5.6 Using multiple SSDs

All tests with multiple SSDs are run on the system with the AMD CPU.

## 5.7 IOMMU modes

There are a couple of kernel parameters that can be set at boot-time to influence the behaviour of the IOMMU. The availability is influenced by the iommu manufacturer, e.g. theres `amd_iommu` and `intel_iommu`, as well as the CPU architecture. Many of these manufacturer dependent options are either very specific, or shared behaviour is ported

to the general iommu parameter, e.g. `amd_iommu=fullflush` and `intel_iommu=strict`. We will be mainly looking at the general options `iommu`.

**Strict** To enable strict IOMMU mode, `iommu.strict=1` has to be set. Using strict mode, unmapping operation cause a complete IOTLB invalidation. Using relaxed mode, unmapping operations can be deferred and batched. This increases performance as an invalidation completely clears the IOTLB buffer, but reduces device isolation.

**Passthrough** Passthrough mode can be enabled using `iommu.passthrough=1`. Using passthrough DMA operations bypass the translation of the IOMMU, and instead directly access physical memory.

# 6 Conclusion

**IOMMU** In this thesis, we improved vroom's safety by implementing IOMMU support and come to the same conclusion as SPDK. The advantages of using the VFIO such as access rights and bigger address spaces 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. The ability to improve security drastically and increasing address space, while not compromising on performance are the reason the MMU succeeded, and it is likely that the IOMMU will as well.

**Rust in driver development** The viability of using Rust to develop drivers has been shown oftentimes, and it has proved that a modern, memory-safe language like Rust can compete with C in systems development. Using Rust does not only provide more safety, but also a modern ecosystem, a package manager and zero cost abstractions. Using Rust for drivers ensures in-process memory safety.

**Future Work** Future Work on the driver could include expanding the NVMe capabilities. Currently the driver is fixed to one namespace. Furthermore, the driver does not support a block device layer and a file system. Including sysfs support could also be a next step. To further push the throughput it could be investigated if and how many threads could operate on one I/O queue.

# List of Figures

# List of Tables

# Listings

# Bibliography


[1] *About DPDK*. DPDK. URL: https://www.dpdk.org/about/ (visited on 08/08/2024).

[2] 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.

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

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

[5] 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).

[6] 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).

[7] *External Technical Root Cause Analysis — Channel File 291*. Crowdstrike, Aug. 6, 2024. URL: https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf (visited on 08/08/2024).

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

[9] S. Huber. "Using the IOMMU for Safe and Secure User Space Network Drivers." MA thesis. Technical University of Munich, 2019.

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

[11] 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).

[12] Intel. *Intel Virtualization Technology for Directed I/O Architecture Specification Revision 4.1*. Mar. 22, 2023. URL: https://www.intel.com/content/www/us/en/content-details/774206/intel-virtualization-technology-for-directed-i-o-architecture-specification.html (visited on 08/02/2024).

[13] *mmap(2) - Linux manual page*. URL: https://man7.org/linux/man-pages/man2/mmap.2.html (visited on 08/02/2024).

[14] *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).

[15] *numactl(8) — Linux manual page*. URL: https://man7.org/linux/man-pages/man8/numactl.8.html (visited on 08/09/2024).

[16] O. Peleg and A. Morrison. *Utilizing the IOMMU Scalably*. USENIX ATC '15. 2015. URL: https://www.youtube.com/watch?v=kL0Roes_cy0 (visited on 08/06/2024).

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

[18] L. Proven. "Linux 6.1: Rust to hit mainline kernel." In: *The Register* (Oct. 5, 2022). URL: https://www.theregister.com/2022/10/05/rust_kernel_pull_request_pulled/ (visited on 08/02/2024).

[19] 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).

[20] K. Sarcar. *What is NUMA?* URL: https://docs.kernel.org/mm/numa.html (visited on 08/09/2024).

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

[22] *Submitting I/O to an NVMe Device*. SPDK. URL: https://spdk.io/doc/nvme_spec.html (visited on 08/05/2024).

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

[24] *VFIO - "Virtual Function I/O"*. URL: https://docs.kernel.org/driver-api/vfio.html (visited on 07/08/2024).

[25] C. Xia and Y. Cao. *Introducing New VFIO and IOMMU Framework to DPDK*. DPDK Summit 2023. Sept. 13, 2023. URL: https://www.youtube.com/watch?v=ZhI0HEv50e0 (visited on 08/02/2024).