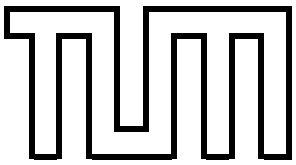


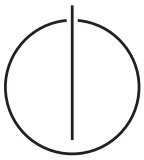
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

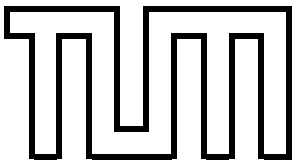
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Effects of Linux VFIO for User Space I/O

Adrian Simon Würth





SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

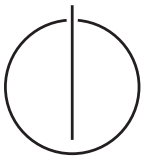
TECHNISCHE UNIVERSITÄT MÜNCHEN

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 14, 2024 Adrian Simon Würth

Abstract

Peripheral devices like SSDs need access to main memory in order to perform I/O operations. In high-performance drivers this is done using Direct Memory Access. 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 examine how impactful the IOMMU is on the whole driver's performance. In this thesis, we implement IOMMU support for a userspace NVMe driver written in Rust and examine its performance, directly comparing DMA with physical addresses and IOMMU I/O virtual addresses. We demonstrate that essentially identical performance may be achieved with 2 MiB pages, along with enhanced system security and the ability to run the driver without root privileges. We also add support for IOMMUFD, which can be used as a modern backend for VFIO.

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 [17]. 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" [21]. This meltdown affected 8.5 million Microsoft Windows devices and severely impacted public institutions, including critical infrastructure like hospitals and airports [5]. In the root cause analysis paper, Crowdstrike, the cybersecurity company that deployed the faulty code, revealed that improper compile time validation and missing runtime array bounds checks were a big part of the error [9].

The damage that can be done by a single ring 0 driver like Crowdstrike's Falcon software shows how critical it is to ensure 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 other languages like C++, now allows Rust code in its codebase [20].

However, it's also essential to consider Rust's safety limits. While using Rust for a driver improves the overall safety of the process while not compensating for performance, direct memory and I/O operations have to be implemented in a memory-unsafe way. A userspace driver using physical DMA addresses enables a device to have full access to the memory and potentially do detrimental I/O operations. Malicious firmware attacks are a rising threat. To enforce safety at the device level, we need to use the IOMMU, a safe way of doing direct memory access. The IOMMU acts as a layer of isolation between devices and memory. Using virtual addresses, the IOMMU provides a bigger virtual address space and enforces 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, an NVMe driver written in Rust [19], and comparing it to using physical addresses. We use the Linux framework VFIO to implement the IOMMU functionality, which has the additional benefit of enabling the driver to run without root privileges. We will also look at IOMMUFD, a modern user API for managing I/O page tables

from userspace, which can replace the backend for VFIO.

2 Background

2.1 vroom

vroom is a userspace NVMe driver written in Rust. As of this writing, it offers high 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 [23]. When using vroom without the IOMMU, the BAR of the NVMe is exposed in the pseudo-filesystem `sysfs` e.g., for the device with PCI address `0000:01:00.0` under the path `/sys/bus/PCI/devices/0000:01:00.0/resource0`. Direct memory access is performed using physical addresses on hugepages.

An 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 provided 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 is a dummy driver that occupies the PCI driver such that the kernel or another application cannot bind to the device.

2.2 Memory Management Unit

Memory Management Units (MMU) for the CPU have been used since the 1980s. After their first integrated application featuring on Intel's 80286 chip [13], they have since become the de facto standard for addressing computer memory. By providing processes with a virtual address space instead of physical addresses, every process is isolated and only has access to memory assigned to it virtual address space. Each translated address points to a region of memory called a page. These pages can have different sizes, with the default being 4 KiB pages on modern x86-64 architectures.

The translations of these pages are stored in so-called page tables. As one page table does not offer enough address space, multiple tables are linked together, consisting of pointers to a lower-level page table. Each table stores parts of the address, and

the entry can be determined by using the offset. One page table walk thus includes fetching multiple tables from memory, resulting in a high latency. To circumvent this, a Translation Lookaside Buffer (TLB) is used to cache translations.

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

2.3 I/O Memory Management Unit

The advantages and success of the CPU's MMU and the introduction of the PCIe bus specification have incentivized hardware manufacturers to apply this concept to peripheral device buses. In 2006, Intel introduced their "Virtualization Technology for Directed I/O" (Intel VT-d) and AMD their "AMD I/O Virtualization Technology" (AMD-Vi/IOMMU). In this thesis, the term IOMMU references both technologies. The IOMMU was originally only used for "solving the addressing problems of devices with limited address space" [25], but nowadays is used mainly for virtualization and device isolation.

The IOMMU works similarly to the MMU, but instead of mapping memory to a process's virtual address space, it maps it to an I/O virtual address space for device access. The addresses used are called I/O Virtual Addresses (IOVA). On a 4-level page table structure as the IOMMU uses for 4 KiB pages, one address resolution results in 4 memory accesses.

The IOMMU, like the MMU has a TLB, which is called the I/O Translation Lookaside Buffer (IOTLB). The size of the IOTLB is not officially documented by Intel nor AMD [11].

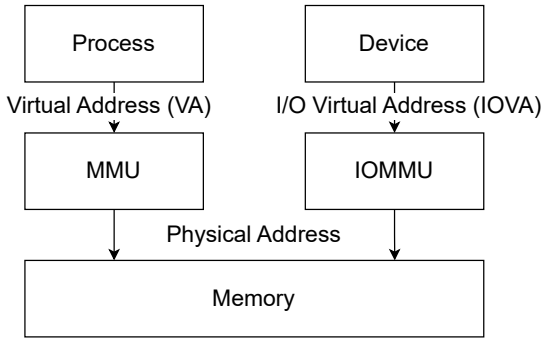


Figure 2.1: MMU and IOMMU relation to physical memory, adapted from [18]

The IOMMU paging structures of Intel's VT-d consist of 4 KiB 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 2 MiB its 21.

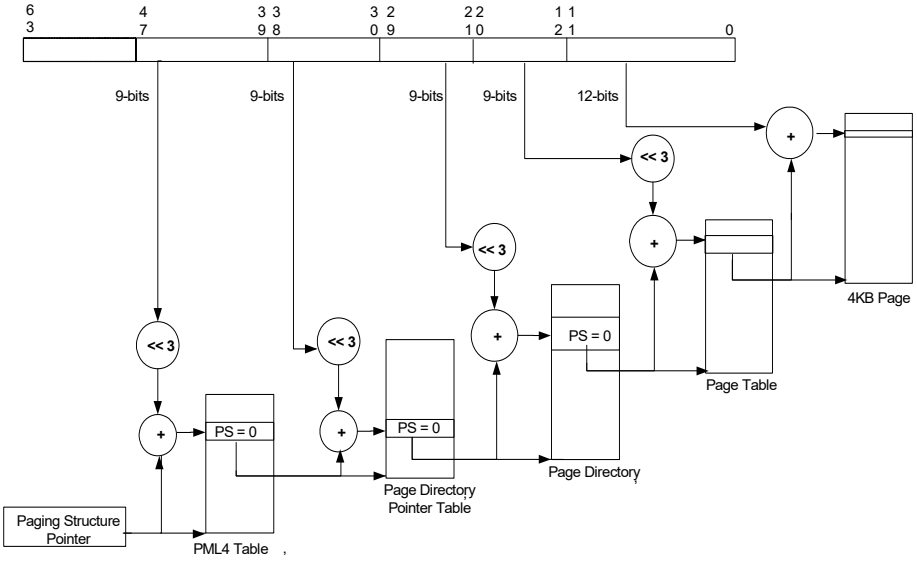


Figure 2.2: Intel VT-d Paging structure for translating a 48-bit address to a 4 KiB page, grabbed from [14]

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) [6]. Using the IOMMU, the request is intercepted and translated to the physical address.

2.5 Hugepages

As the demand for bigger memory mappings, e.g., for big files, increased, the amount of TLB cache misses rose proportionally. With modern CPUs TLB typically having space for only 4096 4 KiB pages, only an address space of 16 MiB could be stored and accessed quickly [8]. To increase the 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. This makes a huge difference, especially in high-performance computing.

- **Persistent Hugepages**: Persistent Hugepages are reserved in the kernel and cannot be swapped or used for another purpose. These hugepages can be mounted as a (pseudo) filesystem called `hugetlbfs`, which can be mounted to a directory. The amount and size of the pages can be specified either during boot on the kernel command line with, e.g., `hugepagesz=1g hugepages=16` or dynamically using the Linux `proc` virtual filesystem [12].
- **Transparent Hugepages**: Transparent Hugepages (THP) 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 provide a way of utilizing the TLB effectively without reserving vast 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 [24].

vroom currently uses hugepages for DMA and locks them with the Linux syscall `mlock` to prevent the kernel from swapping them out. This does not ensure that the kernel can not migrate the page to another physical location. This is the reason why persistent Hugepages have to be used. The kernel cannot move these pages like 4 KiB pages. Other userspace drivers like SPDK or DPDK also rely on this to perform DMA without the IOMMU.

2.6 Peripheral Component Interconnect Express

PCIe is a standard for peripheral device buses. Each device on the PCI bus has a unique PCI address, segmented into three parts, as seen in Figure 2.3. The maximum data payload size of PCIe is 4 KiB.

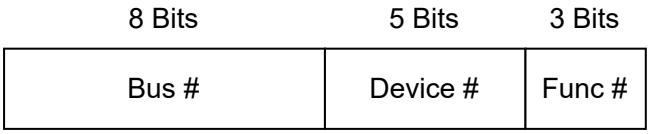


Figure 2.3: Segmented PCI identifier

Each device on the bus uses a PCIe configuration space, which includes registers for controlling the device's behavior, e.g., enabling DMA in the command register. It also includes Base Address Registers (BAR), which are used to access the device's actual controller. The configuration space can be seen on Figure 2.4. The marked fields are needed for vroom.

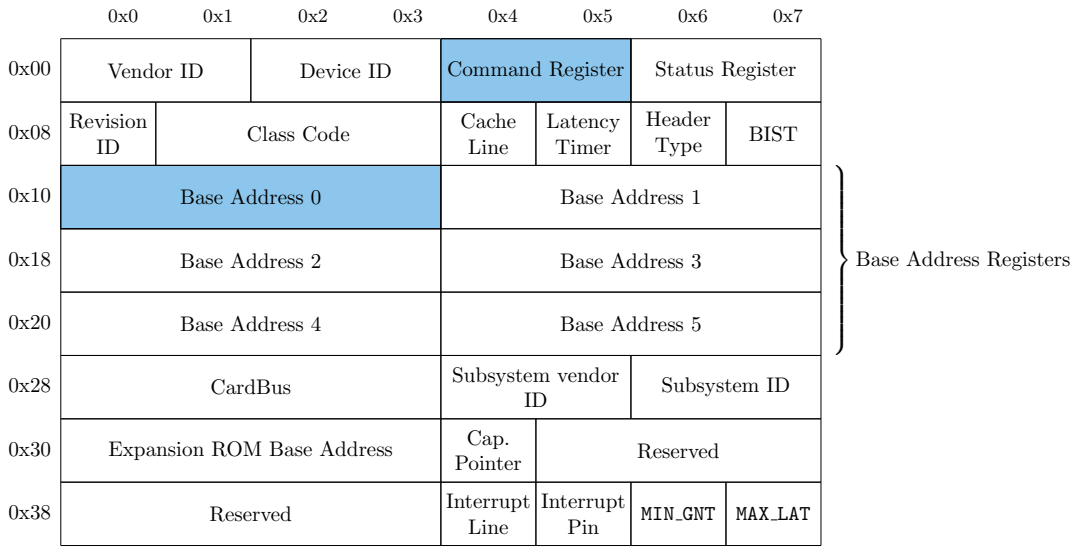


Figure 2.4: PCIe configuration space, adapted from [19]

2.7 Rust

While userspace drivers can theoretically and practically be written in any language, as proven by the network driver Ixy [15], 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 critical factor is that, like C, Rust does not use exceptions. Being forced to handle errors ensures no rogue exception can take down critical code infrastructure. Additionally, Rust provides low-level access while offering a high-level development experience through zero-cost abstractions.

3 Related Work

3.1 Ixy

Ixy is a network interface card (NIC) driver for Intel's 82599 10GbE NICs (ixgbe family) [8]. Ixy has been implemented in many languages, e.g., C, Go, and Rust. Ixy.rs is the Rust implementation of the Ixy driver [7]. Stefan Huber implemented IOMMU support for the Ixy.rs driver. It was concluded that while using the IOMMU with 2 MiB pages, the performance matches the performance without the IOMMU. On the other hand, using 4 KiB pages leads to a potential 75% performance loss. Additionally, it was found that the tested Intel Xeon E5-2620 v3 6-core CPU IOMMU TLB has a maximum size of 64 entries [11]. Rolf Neugebauer et al. determined the same 64 IOTLB size on the tested Intel Xeon E5-2630v4 2.2GHz [16].

3.2 Data Plane Development Kit

The Data Plane Development Kit (DPDK) is a framework for developing userspace network card drivers. It allows for high-performance network applications. It can run using direct memory access with physical addresses or with 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.

3.3 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. Primarily 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 that is widely used, modern, and highly performant. Therefore, it is a protocol for which many drivers, including userspace drivers, have been written. The Storage Performance Development Kit (SPDK) provides "a collection of tools and libraries for writing high

performance, scalable, user-mode storage applications" [22]. It includes a userspace NVMe driver, which is fast and production-ready. While this driver supports using the driver without the IOMMU, the SPDK Documentation recommends using the IOMMU as it 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.

4 Implementation

The VFIO implementation by Stefan Huber for Ixy.rs was used as a reference but changed to fit the project structure and use case. The implementation for IOMMU support includes the initialization of the IOMMU, the steps needed to create mappings to the I/O virtual address space and the access to the device registers. The complete code of the driver, including IOMMU support, can be found on GitHub [26].

4.1 Virtual Function I/O

Virtual Function I/O (VFIO) is an IOMMU agnostic framework for exposing devices to userspace. VFIO acts like the kernel module to userspace drivers, allowing unprivileged, regulated access to physical memory and device registers.

As seen on Figure 4.2a, VFIO consists of an IOMMU API for management of IOMMU mappings (VFIO backend) and a device API for device access, which uses the backend to perform the access. With the new introduction of IOMMUFD, the native backend of VFIO is considered to be the "legacy" backend. Legacy VFIO uses the type1 IOMMU API for x86 architectures or the SPAPR IOMMU API for ppc64 architectures. As our implementation is for x86, we will equate legacy VFIO with the type1 API.

The `vfio-pci` driver (device API) is used for interaction with the device, i.e., reading, writing, and mapping the device registers. When VFIO is used with IOMMUFD, only the backend gets replaced, as IOMMUFD still relies on device API to access device registers. We will first focus on the implementation of legacy VFIO, and then compare it to the implementation of IOMMUFD. The layers of VFIO with both backends can be seen on Figure 4.2.

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

1. **Binding NVMe to** `vfio-pci`: Before initializing the devices, the NVMe device needs to be unbound from the kernel driver and bound to `vfio-pci`.

2. **IOMMU container initialization**: As the first step of the actual driver, we need to initialize the container. This is done using the VFIO IOMMU API, which is an interface for the IOMMU driver. We can get the container file descriptor with

VFIO. The group needs to be assigned to the container. In this step, we also can attain the device file descriptor, which can be used to read/write/`mmap` the PCIe registers through the IOMMU.

3. **Device register access**: Using the device fd, we can enable DMA by setting a bit in the PCIe command register.

4. **NVMe initialization**: We use `mmap` to map the NVMe base address register into memory for configuring the NVMe device.

5. **DMA (Un-)Mapping**: Using the container fd, we can create a mapping in the IOMMU and, therefore, place it in the virtual address space of the NVMe controller for DMA.

Interaction with the VFIO interface works using `ioctl` system calls. `ioctl` or control device syscall, uses a file descriptor (`fd`), operation id (`op`), and optional arguments to perform actions on devices that are not covered by other system calls. The operation IDs used for VFIO are defined as constants or enums in the `vfio.h` header file in the Linux kernel. To use these constants in Rust, the constants need to either be defined manually or with a crate like bindgen, which automates bindings for C and C++ libraries [3]. We chose the manual implementation to keep the binary and dependency list as small as possible. Many `ioctl` calls used for VFIO also take in a mutable reference to a struct, which is used for specific input and output. These structs are also defined in `vfio.h` and can be ported over to Rust using the `#[repr(C)]` attribute, which ensures the same struct alignment as in C.

To model the IOMMU we implement the struct `Vfio` and the enum `VfioBackend`, as seen in Listing 4.1. The shared functionality, e.g., accessing the device registers, is implemented on the struct `Vfio`, while the enum `VfioBackend` takes care of the backend-specific behaviour, i.e., mapping DMA addresses.

Listing 4.1: Structs used to model VFIO

```
pub struct Vfio {
    pci_addr: String,
    device_fd: RawFd,
    page_size: Pagesize,
    iommu: VfioBackend,
}

enum VfioBackend {
    Legacy {
        container_fd: RawFd,
    },
    IOMMUFD {
        ioas_id: u32,
        iommufd: RawFd,
    },
}
```

4.1.1 Groups and Containers

VFIO works with groups and containers. Each group can contain one or multiple devices. As many devices use DMA between each other, a single IOMMU group has to be created, as these devices cannot function in an isolated environment. The other way round can also be the case, with one device exposing two interfaces, which get their own group each. Therefore, groups are the smallest unit of granularity able to function. While groups are supposed to provide the highest amount of isolation, the need for shared memory between devices often exists. This need can be solved by using containers. Containers consist of one or more groups. The groups in one container share the same I/O virtual address space created by the IOMMU, allowing both to access the same memory. A new container can be created by opening the file `/dev/vfio/vfio`. The groups of devices bound to `vfio-pci` can be found under the path `/dev/vfio/$GROUP`.

4.1.2 Binding NVMe to `vfio-pci`

To use the IOMMU for the driver, we first need to initialize the VFIO kernel module using `modprobe` and bind the `vfio-pci` driver to the NVMe device. By changing the owner of the container and group file to an unprivileged user, vroom can use the VFIO

driver to create memory mappings and interact with the device without root.

4.1.3 IOMMU container initialization

To initialize 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:

Listing 4.2: `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 initializing 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 access the device registers.

4.1.4 Device register access

Previously, device access was done through the `sysfs` (pseudo-)filesystem. `sysfs` exposes the device registers under the path `/sys/bus/pci/devices/$PCI_ADDRESS/`. As seen in Figure 2.4, we need to access the command register and the NVMe BAR0 register. In `sysfs` these are the `config` and `resource0` files. The `config` file points to the address 0x0 of the PCI configuration space. By adding the command register offset (0x4), we can access the command register. Using the offset 0x2, we can set the bit for bus mastering, allowing the device to perform DMA and disable interrupts by setting the bit at offset 0x4 at the `INTERRUPT_DISABLE` flag. To map the BAR0 register to memory, we can use `mmap` with a file descriptor to `resource0`.

To do the same using VFIO, we use the VFIO device API. We can access the device registers using the `VFIO_DEVICE_GET_REGION_INFO` `ioctl` operation on the device fd. This operation requires the struct `vfio_region_info` as the third parameter, which needs to be initialized with a given index from `vfio.h`. These indizes are similar to the files in sysfs. We use the indizes `VFIO_PCI_CONFIG_REGION_INDEX` and `VFIO_PCI_BAR0_REGION_INDEX` to access the config and BAR register respectively. The

struct is used as a input/output struct. After After performing the syscall, the other fields are set, e.g., size or offset, and can be used to read/write or memory map device registers.

Using `VFIO_PCI_CONFIG_REGION_INDEX` as the index, we again get the PCIe configuration space address 0x0. By adding the command register offset, we can read the 2-byte command register or the DMA bit and write the modified bytes back into the register. After this, we can map the NVMe base address register to memory using the `VFIO_PCI_BAR0_REGION_INDEX` index. The offset and size can be directly passed into `mmap` to map the BAR0 register to memory, as seen on Listing 4.3.

Listing 4.3: Mapping the BAR0 NVMe register to memory

```
let mut region_info = vfio_region_info {
    argsz: mem::size_of::<vfio_region_info>() as u32,
    flags: 0,
    index: Self::VFIO_PCI_BAR0_REGION_INDEX,
    cap_offset: 0,
    size: 0,
    offset: 0,
};

ioctl_unsafe!(
    self.device_fd,
    IoctlOp::VFIO_DEVICE_GET_REGION_INFO,
    &mut region_info
)?;

let len = region_info.size as usize;

let ptr = mmap_unsafe!(
    ptr::null_mut(),
    len,
    libc::PROT_READ | libc::PROT_WRITE,
    libc::MAP_SHARED,
    self.device_fd,
    region_info.offset as i64
)?;
```

4.1.5 DMA (Un-)Mapping

When using physical addresses for DMA, a number of steps have to be performed to ensure that the page is not moved. Firstly, hugepages have to be used, as the kernel cannot move these. Additionally the allocated memory is locked using `mlock` to lock the page. Thus, a hugepage file is created in the directory where hugetlbfs is mounted, in our case `/mnt/huge`. By then using `mmap` with the file descriptor and then locking it, we can create a statically mapped page in memory. To get the physical address for DMA, the address translation entry in the MMU needs to be fetched from `/proc/self/pagemap`. As `mmap` uses 4 KiB by default, we have to specify the use of hugepages with the `MAP_HUGETLB` flag. The resulting pagesize depends on the default hugepage size of the system but can be specified with either the `MAP_HUGE_2MB` or the `MAP_HUGE_1GB` flag.

As seen in Listing 4.4, VFIO greatly simplifies this, as using IOVAs alleviates the need for ensuring the physical address does not change. This is handled by VFIO. Firstly, we can either either allocate or map a file to the process virtual space using `mmap`. As we do not have the restriction of using hugepages, we can also perform the mapping using the default 4 KiB pagesize. To get the IOVA for a corresponding page in memory, we need to use the `vfio_iommu_type1_dma_map` struct with the VFIO operation `VFIO_IOMMU_MAP_DMA`. To correctly map an address, we need to specify the address mapping in the struct. `vaddr` is the address in the process virtual address space, i.e., the address returned by `mmap`. By setting the field `iova`, we can conveniently use the same address for the IOVA. If we would not do this, the IOVAs would have to be managed manually, e.g., to avoid mapping to an address twice. Finally, the size has to be specified, which is the same as the length passed to `mmap`. In the `flags` field, we can specify if the memory is accessible with read and/or write. By default, we simply set both. This IOVA can then be used just like the physical address by the NVMe controller.

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,
    IoctlOp::VFIO_IOMMU_MAP_DMA,
    &mut iommu_dma_map
)?;

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

Unmapping DMA Unmapping DMA happens when the process exits, yet for performance and application reasons, we implement the `unmap_dma` function to remove a DMA mapping from the IOMMU. Using the `VFIO_IOMMU_UNMAP_DMA` `ioctl` operation, we can unmap the memory and finally free it using `munmap`.

4.1.6 NVMe initialization

Using the mapped NVMe BAR and the ability to create DMA mappings, we can now initialize the NVMe controller. Mainly this consists of allocating the queues, and mapping them to be accessed by the NVMe controller through DMA. The IOVAs of the admin queues can be written to the mapped BAR, where also additional configuration, like enabling the controller, setting queue entry sizes happens. Through the mapped BAR, we can set the admin queue addresses and additional configuration like, e.g, setting queue entry sizes. Using the admin queues, an I/O queue pair can be created. The driver is ready to operate and perform I/O operations.

4.1.7 I/O operations with VFIO

After initialization, the NVMe is ready to use. A sequential, single-threaded I/O operation is shown in Figure 4.1.

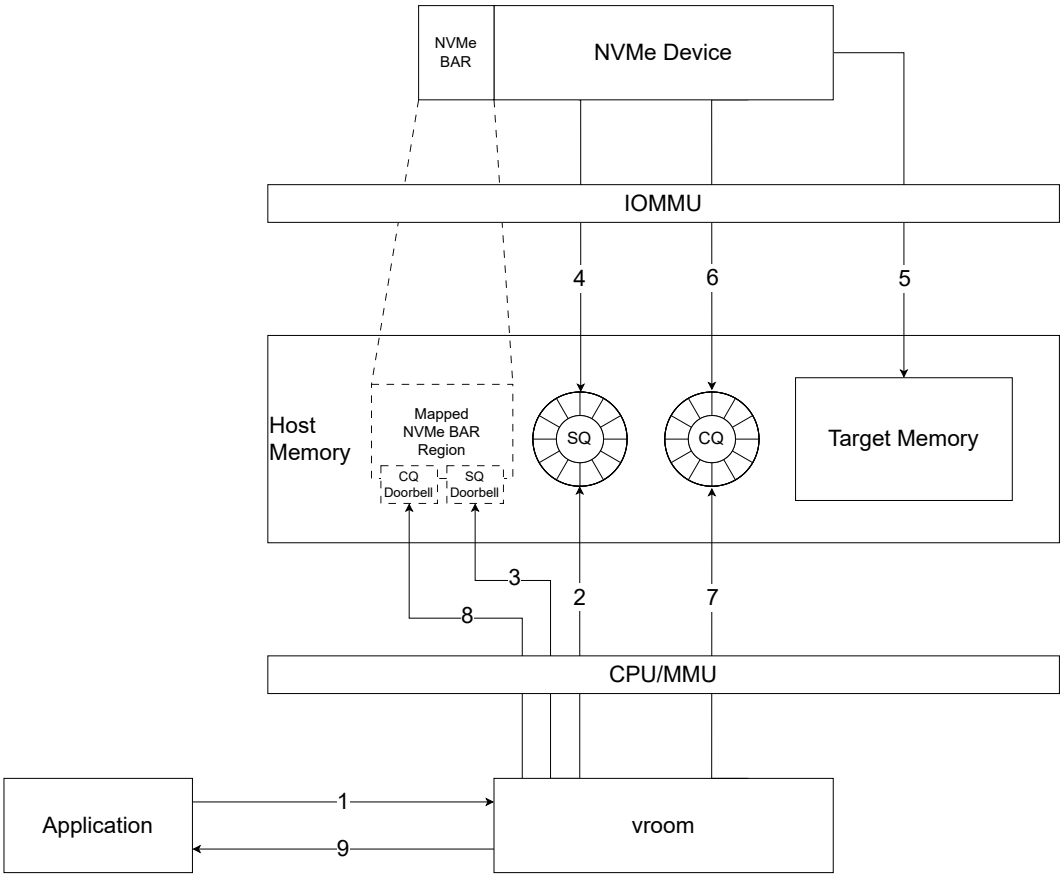


Figure 4.1: I/O operation using vroom with enabled IOMMU

The sequence of events in Figure 4.1 are 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 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.

4.2 IOMMUFD

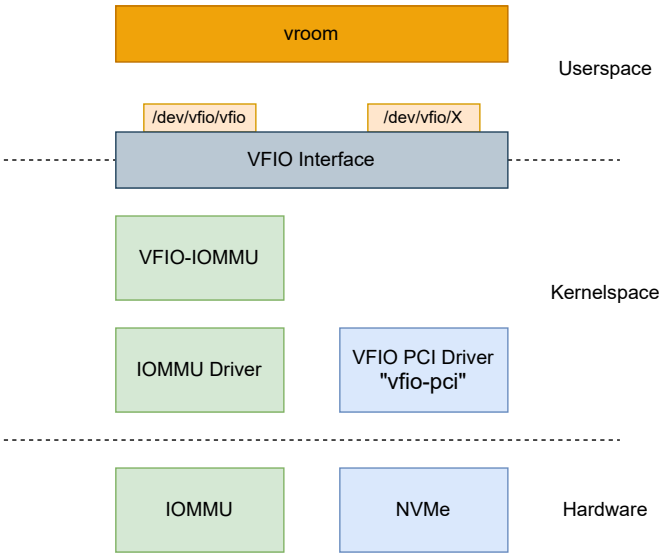
The IOMMU File Descriptor user API (IOMMUFD) offers a way of controlling the IOMMU subsystem using file descriptors in user-space [10]. IOMMUFD offers a more granular managing of the IOMMU, using devices instead of groups. Additionally, it supports a more user-friendly interface, allowing the user to modify IOAS easily. Although IOMMUFD could potentially be used as a standalone to provide simple IOAS management functionality like mapping or unmapping, e.g., the device registers still need to be accessed through the `vfio-pci` driver. Thus, IOMMUFD is used in conjunction with VFIO, replacing its backend, i.e. the interaction with 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. Instead of using containers or groups, IOMMUFD uses so-called I/O address spaces (IOAS) and character device file descriptors. Just like containers, IOAS can be used to provide shared memory mappings for multiple devices. The implementation of IOMMUFD is similar to VFIO, but there are some key differences. As with VFIO, to interact with IOMMUFD, we use the syscall `ioctl`. The needed bindings, flags and operations are defined in the Linux kernel under the path `include/uapi/linux/iommufd.h`. Again, we manually port the needed structs and constants to Rust.

The first change is the acquisition of the group/device and the container/iommu fd. In VFIO, a container can be created using the file `/dev/vfio/vfio`. For IOMMUFD, first the iommu fd needs to be acquired from `/dev/iommu`. By then using the `IOMMU_IOAS_ALLOC ioctl`, a new IOAS can be allocated. The device file descriptor, which was previously attained with `VFIO_GROUP_GET_DEVICE_FD` with the group can now be obtained through opening the character device `/dev/vfio/devices/vfioX` [25]. In order to use the device with VFIO, it still has to be bound to IOMMUFD, using `VFIO_DEVICE_BIND_IOMMUFD`.

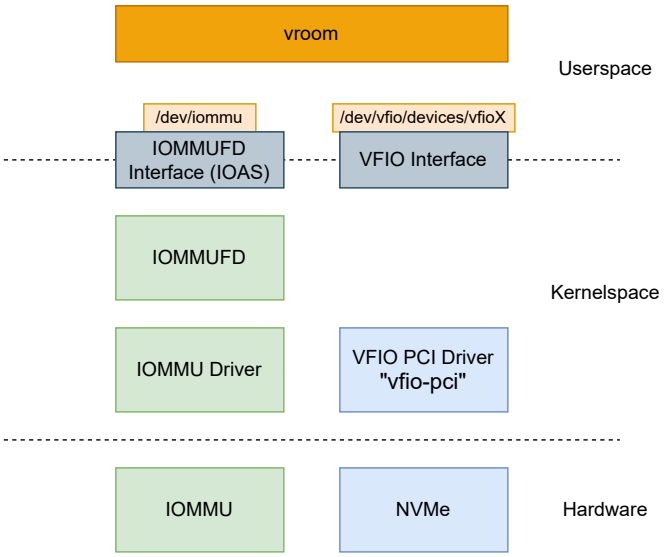
The IOAS can then be assigned to the device by using `VFIO_DEVICE_ATTACH_IOMMUFD_PT`. As with containers, this operation can be performed on multiple devices for a shared IOAS. The equivalent in VFIO is `VFIO_GROUP_SET_CONTAINER`.

When using VFIO with IOMMUFD, the interaction with `vfio-pci` stays the same. Primarily the whole functionality of reading, writing and mapping to the device registers is unchanged, except that the character device fd is used.

As for (un-)mapping DMA, the `IOMMU_IOAS_MAP` and `IOMMU_IOAS_UNMAP` are used.



(a) VFIO with Containers



(b) VFIO with IOMMUFD (IOAS)

Figure 4.2: Layer diagrams of VFIO with VFIO Container API and IOMMUFD, adapted from [27]

4.3 Linux Systemcalls

A variety of Linux Systemcalls (syscalls) are used in vroom. The syscalls that are used by vroom are mmap, ioctl, pread, pwrite (and mlock for the non IOMMU version). While there are crates that implement the syscall functionality, we only use the `libc` crate to avoid inflating the dependency list and executable size. As these require C-like syntax and an unsafe block in Rust, wrapper macros are used to provide locality of behaviour and secure error handling. In Listing 4.5 the macro for the mmap syscall can be seen. As part of our error handling, we introduce an error enum variant for each syscall. To not hide the inherit unsafety of these macros, we add the suffix "_unsafe".

Listing 4.5: 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)
        }
    }};
}
```

5 Evaluation

In this chapter, we analyse the performance impact of the IOMMU, directly comparing it to the physical address approach. To compare both approaches fairly, we allocate and map the memory upfront. The focus lies on the IOMMU itself and how it performs. All performance tests use legacy VFIO instead of IOMMUFD as it currently remains the widely adopted way of using the IOMMU.

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 and are NUMA systems which 2 nodes each. We adhere to NUMA-locality.

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	128	16384	Yes	Consumer
Samsung PM9A3	128	16384	No	Enterprise

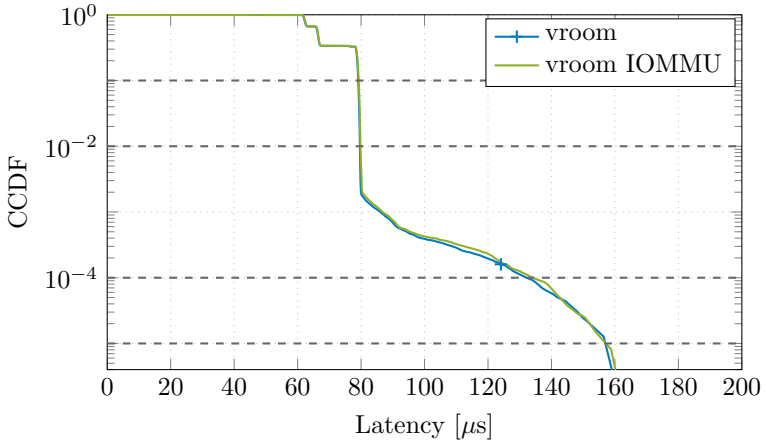
Table 5.3: NVMe(s) of the systems

Despite the NVMe specifications maximum capability of 65536 I/O queues, our SSDs support a more reasonable amount of 128 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 [19]. The NVMe SSDs are formatted to 512-byte blocksize. All writes are performed on an empty SSD to avoid any overhead through garbage collection on the NVMe. As the NVMe can optimize reads on an empty SSD, all reads will be performed on a full SSD. The tests mainly use random writes/reads, as the NVMe can drastically optimize sequential requests, which can lead to altered results. Unless stated otherwise, we configure the submission and completion queue length to the maximum amount. Additionally, all standard tests are run with the `iommu.strict=1` kernel parameter. When this parameter is set, the IOMMU invalidates the complete IOTLB when an unmapping happens. As we unmap IOVAs inbetween tests, this ensures that for each test the IOTLB is flushed.

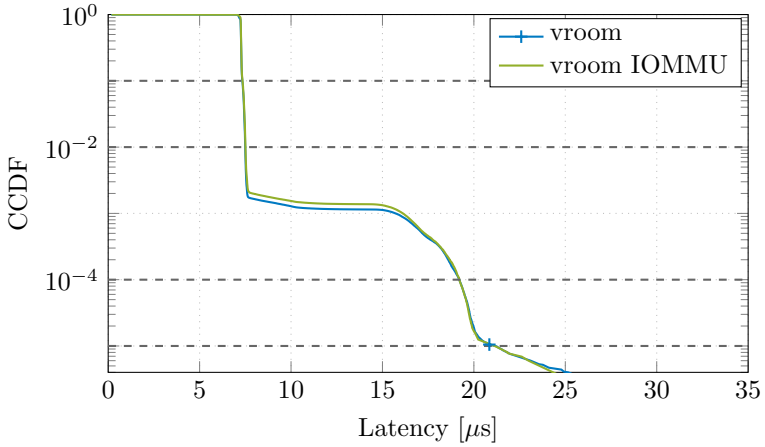
5.2 Results

For the following results, we will compare the latencies and throughput of vroom without the IOMMU and with the IOMMU, both utilizing 2 MiB pages. All latency and throughput tests are run with a 1 GiB buffer in memory. For 2 MiB pages this equates to 512 pages being accessed.

5.2.1 Latencies

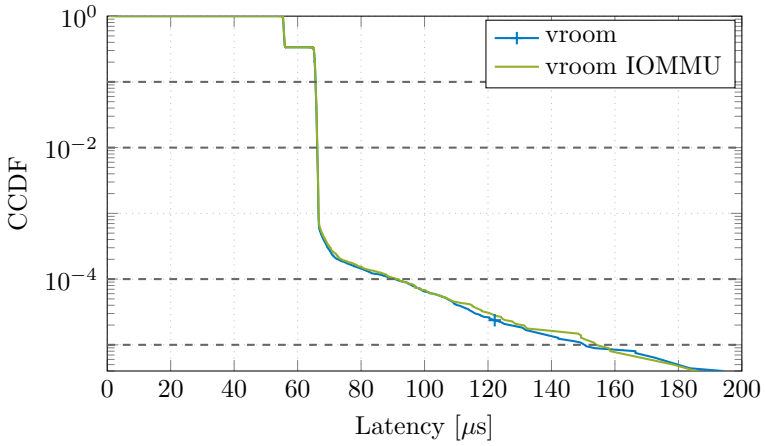


(a) Random read

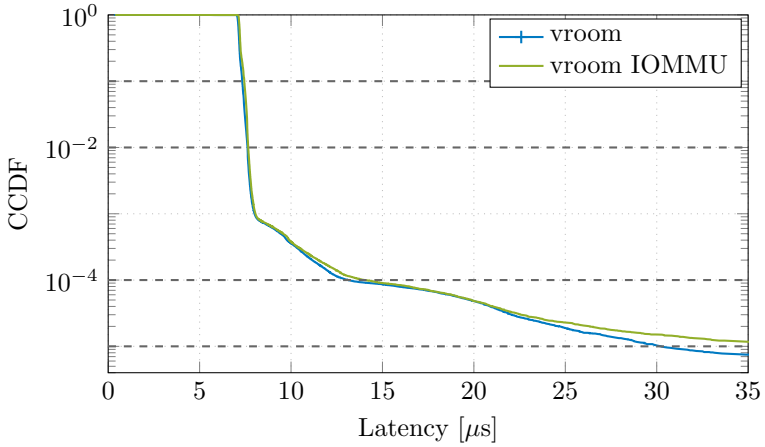


(b) Random write

Figure 5.1: Tail latencies on Intel System



(a) Random read

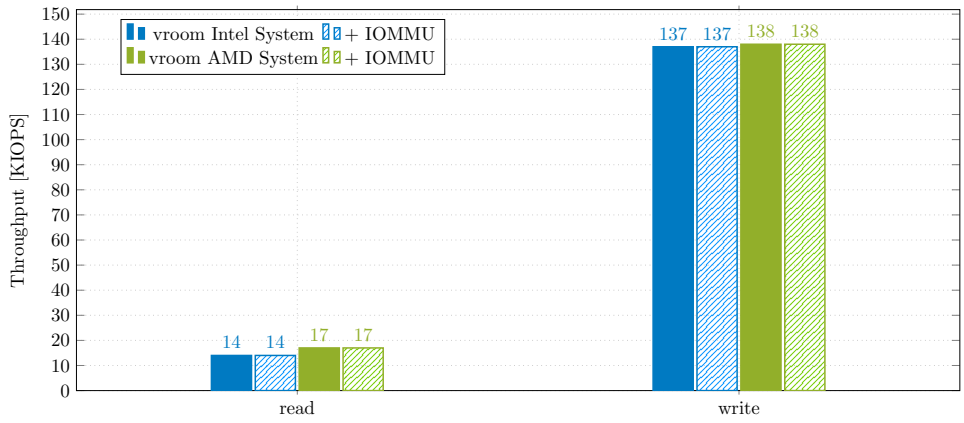


(b) Random write

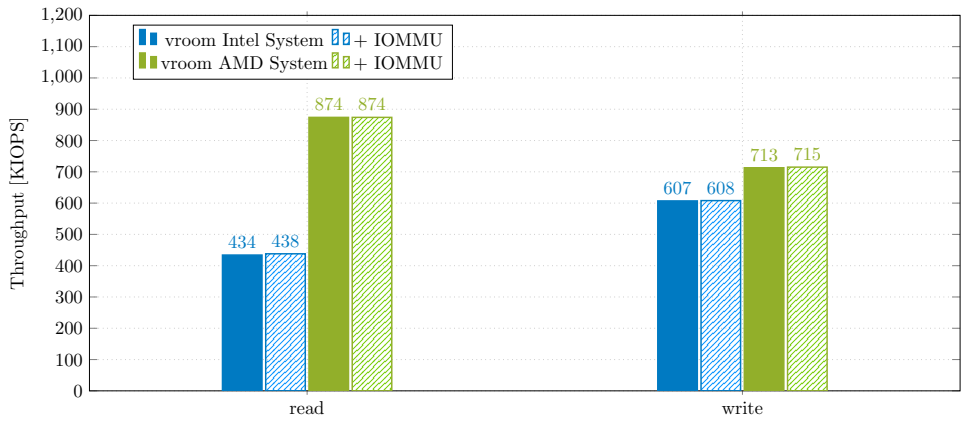
Figure 5.2: Tail latencies on AMD System

5.2.2 Throughput

All throughput tests are done with a 1 GiB buffer in memory. As seen on Figure 5.3, the IOMMU has no noticeable impact to performance.



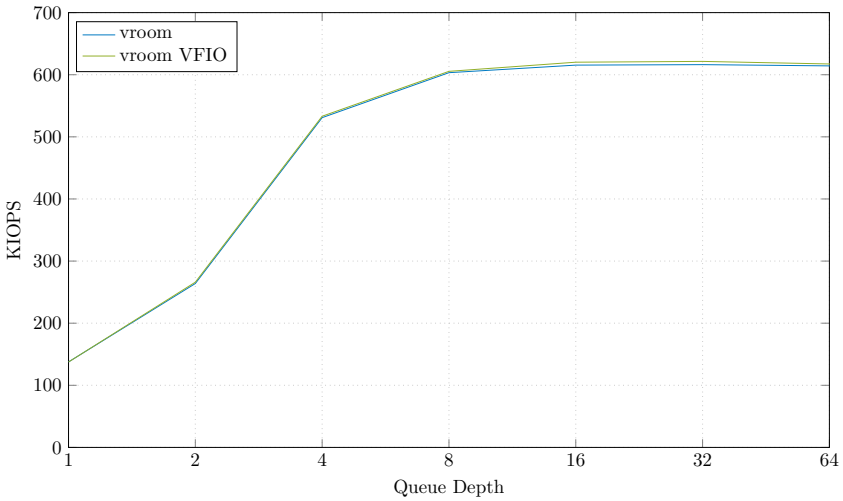
(a) Queue depth 1 and 1 thread



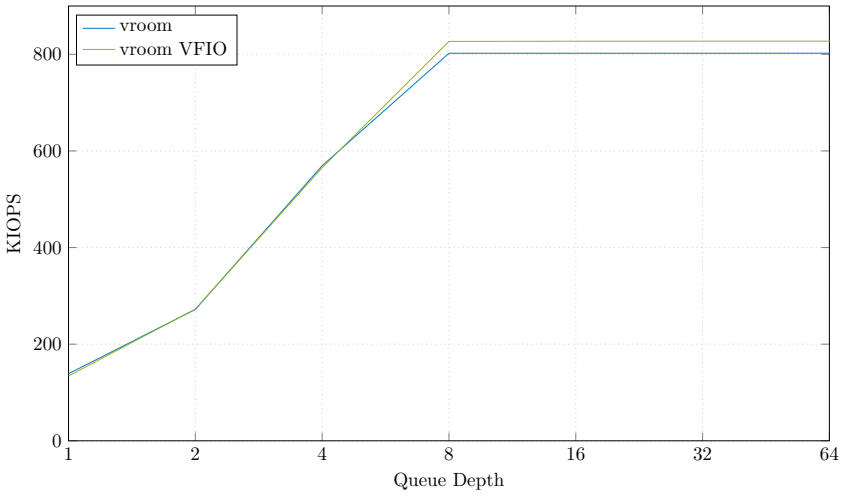
(b) Queue depth 32 and 4 threads

Figure 5.3: Throughput over 60s

We also take a look at throughput performance with larger queue depths. The queue depth describes how many outstanding requests are put onto a submission queue, leading to lower latencies. Again, the lines are mostly overlapping. A slight performance difference is noticeable on the AMD system, where the maximum throughput of vroom with IOMMU is around 3% higher than without IOMMU.



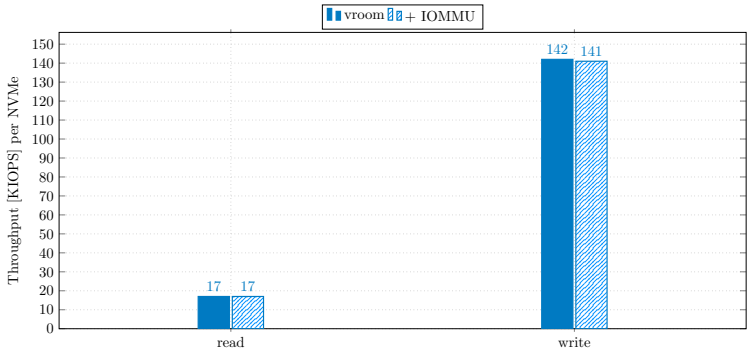
(a) Intel System



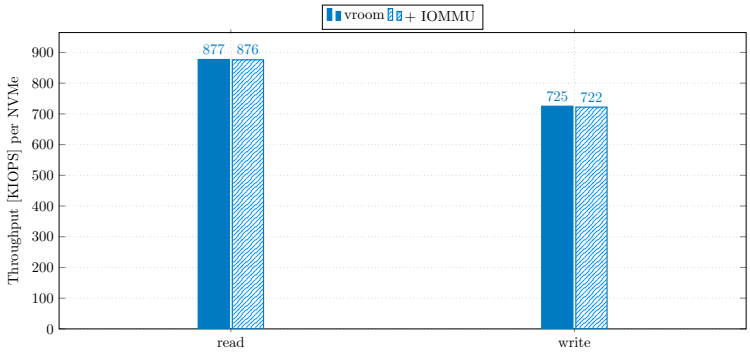
(b) AMD System

Figure 5.4: Throughput over 60s of random writes with increasing Queue Depth

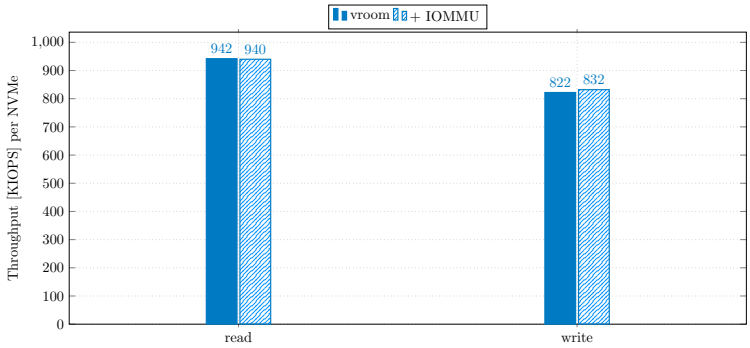
We can further push the throughput by using 8 NVMes in parallel. We tested multiple NVMe configurations. Again, no significant impact of the IOMMU occurs.



(a) Queue depth 1 and 1 thread



(b) Queue depth 32 and 4 threads



(c) Queue depth 128 and 16 threads

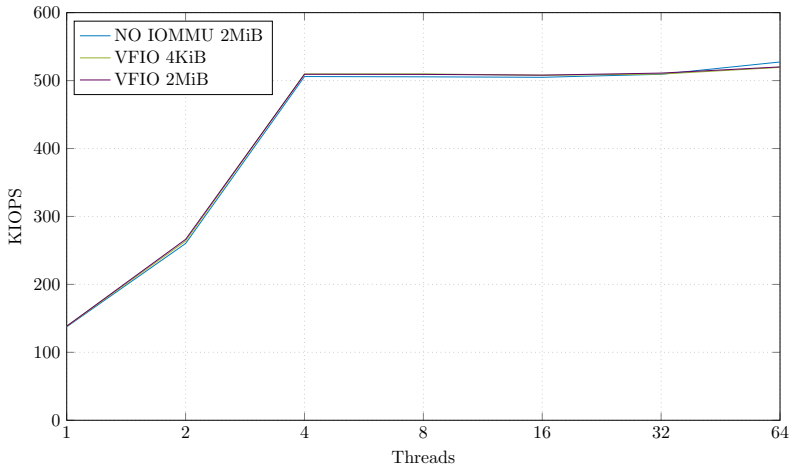
Figure 5.5: Throughput over 60s on 8 NVMe SSDs

5.3 Impact of 4 KiB pages

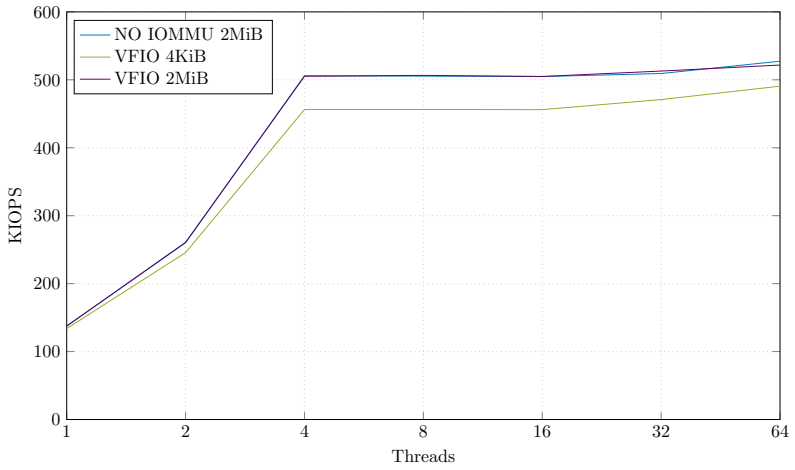
5.3.1 Throughput

As Linux as well as our IOMMUs supports 4 KiB, 2 MiB and 1 GiB page sizes we will first test and analyse how it affects the latencies and overall performance. Especially using 4 KiB pages, a performance impact should be noticeable. As we use a typical unit size of 4 KiB using 4 KiB pages should result in TLB-thrashing, and every operation in a page walk. To test this hypothesis, we focus on the Intel system and only write to the Turbowrite buffer of the Samsung Evo 970 Plus, to reach the maximum performance and lowest latencies.

When comparing the performance of vroom without the IOMMU against vroom with IOMMU using 4 KiB and 2 MiB pages on a 2MiB buffer, a performance difference of around 10% between 4 KiB pages and 2 MiB pages can be observed. This stems from the aforementioned IOTLB-thrashing. Noticeable is that no performance impact can be seen when using a 4KiB buffer, as all pages can fit into the IOTLB. The lines of both implementations using 2 MiB pages overlap with both buffer sizes.



(a) 1 4 KiB buffer per thread



(b) 1 2 MiB buffer per thread

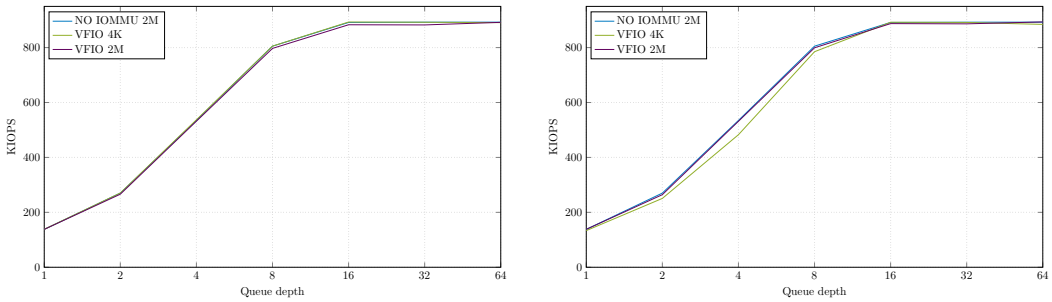
Figure 5.6: QD1 random write throughput over 20s with multithreading on the Intel system

5.3.2 PCIe limitations

When testing increasing queue depth, as seen on Figure 5.7, around a 10% performance decrease can be seen using queue depth 4. Further increasing the queue depth leads to a decrease of this gap, with the throughput capping at around 890K IOPS.

The Intel System SSD is mounted on a PCIe 3.0 4x width bus with a maximum

payload of 256 bytes. This PCI bus has a maximum throughput of 3.938 GB/s. Using the SSD to its full capability, i.e. using random writes with high queue depths in the Turbowrite buffer, can result in the bus being the bottleneck. With the highest throughput measured being 890K IOPS with one I/O operation containing 4096 bytes of data, we achieve 3.64 GB/s. Including the headers for each TLP and submission- and completion queue entries, we achieve 3.908 GB/s. This roughly equates the PCIe bus limits and leads us to conclude, that the missing overhead of using 4 KiB pages on high queue depths stems from this bottleneck.



(a) 1 4 KiB buffer per thread

(b) 1 2 MiB buffer per thread

Figure 5.7: Singlethreaded random write throughput over 20s with increasing queue depth on the Intel system

5.3.3 Determining IOTLB size

As the size of the IOTLB is not stated in hardware and VT-d or AMD-V specifications, we use a latency test to analyze the behaviour of the IOMMU. In order to isolate the effect of the IOMMU we track the latencies of the fastest operation the NVMe can perform. The fastest operation is using random writes with the smallest blocksize of 512 B.

If we then write from a single block from each page to the NVMe, repeat it 65536 times on an increasing page count that are a power of two, we can figure out where a latency spike occurs. The page count right before the latency spike should equal the IOTLB entry count. We configure the queues, buffer and prp-list to each take up one page, resulting in 6 allocated pages before the actual workload, to not shift the latency spike over. This test is done using without the IOMMU with 2 MiB pages and the IOMMU with 4 KiB, 2 MiB and 1 GiB pages. As for 1 GiB pages, the graphs are limited by the available memory.

Results of Intel Xeon E5-2660v2 In the resulting graph Figure 5.8 we can observe a performance spike of around 250 nanoseconds for each write between 64 and 128 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 64 entries of size**. This matches the page size Stefan Huber and Rolf Neugebauer found [11][16].

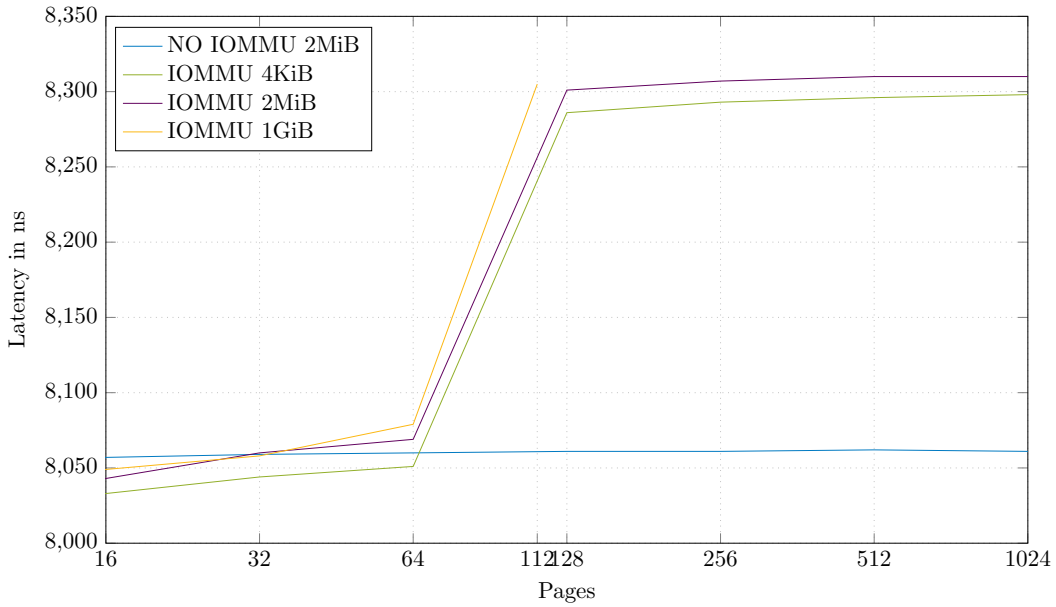


Figure 5.8: Latencies of random writes on an emptied SSD with increasing host memory pages on the Intel system

Results of AMD EPYC 7713 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. This leads us to suspect an **IOTLB size of 64 for** 2 MiB **and** 1 GiB **and 256 for** 4 KiB **pages**. The performance itself only decreases by about 60 ns, which is a five fold performance increase of page walks compared to the intel cpu system.

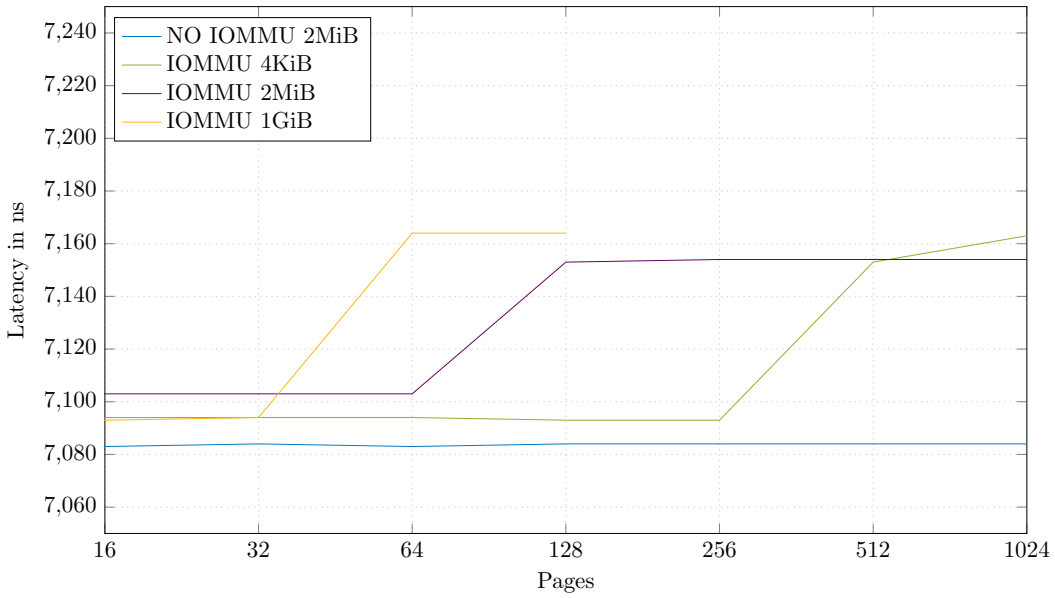


Figure 5.9: Latencies of random writes on an emptied SSD with increasing host memory pages on the AMD system

6 Conclusion

In this thesis, we investigated the effects of using VFIO, or more generally, the IOMMU on the userspace NVMe driver vroom. As part of our implementation, we support using legacy VFIO and IOMMUFD as the VFIO backend.

Our findings include the IOTLB size of two IOMMU models from Intel and AMD, which, when exceeded, can introduce address translation overhead for 4 KiB pages. When not exceeding the IOTLB, or using hugepages, the IOMMU performs extremely well, and no overhead can be measured. The advantages of using the IOMMU, such as access rights and bigger address spaces, as well as the ability to run the driver without root privileges with VFIO, make it the preferred method of addressing memory.

Rust in driver development The viability of using Rust to develop drivers has been shown often, and it has proved that a modern, memory-safe language like Rust can compete with C in systems development. Using Rust provides more safety and a modern ecosystem, a package manager, and zero-cost abstractions. This makes Rust an excellent choice for driver development.

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 or file system. Also, it could be investigated if and how many threads could operate on one I/O queue to further push the throughput. Finally, a performance investigation into IOMMUFD could be conducted, as we only tested the legacy VFIO implementation.

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] S. Ellmann. "Investigating Effects of Hardware Isolation in High-Speed Network Environments." MA thesis. Technical University of Munich, 2021.

[7] S. Ellmann. "Writing Network Drivers in Rust." Bachelor's thesis. Technical University of Munich, 2018. URL: https://www.net.in.tum.de/fileadmin/bibtex/publications/theses/2018-ixy-rust.pdf (visited on 08/12/2024).

[8] P. Emmerich, M. Pudelko, S. Bauer, S. Huber, T. Zwickl, and G. Carle. "User Space Network Drivers." In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE. 2019, pp. 1–12. URL: https://www.net.in.tum.de/fileadmin/bibtex/publications/papers/ixy-writing-user-space-network-drivers.pdf (visited on 08/11/2024).

[9] *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).

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

[11] S. Huber. "Using the IOMMU for Safe and Secure User Space Network Drivers." MA thesis. Technical University of Munich, 2019. URL: https://www.net.in.tum.de/fileadmin/bibtex/publications/theses/2019-ixy-iommu.pdf (visited on 08/11/2024).

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

[13] 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).

[14] 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).

[15] *ixy-languages GitHub*. URL: https://github.com/ixy-languages/ixy-languages (visited on 08/11/2024).

[16] R. Neugebauer, G. Antichi, J. F. Zazo, Y. Audzevich, S. López-Buedo, and A. W. Moore. "Understanding PCIe performance for end host networking." In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: Association for Computing Machinery, 2018, 327–341. ISBN: 9781450355674. DOI: 10.1145/3230543.3230560. URL: https://doi.org/10.1145/3230543.3230560.

[17] *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).

[18] 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).

[19] T. Pirhonen. "Writing an NVMe Driver in Rust." Bachelor's thesis. Technical University of Munich, 2024. URL: https://db.in.tum.de/~ellmann/theses/finished/24/pirhonen_writing_an_nvme_driver_in_rust.pdf (visited on 08/11/2024).

[20] 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).

[21] 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).

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

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

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

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

[26] *vroom source code*. URL: https://github.com/adwuerth/vroom (visited on 08/10/2024).

[27] 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).