



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 13, 2024

Adrian Simon Würth

Abstract

Peripheral devices like SSDs must access memory through Direct Memory Access for I/O operations. 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 replacement for the VFIO IOMMU Interface.

Contents

Abstract	iii
1 Introduction	1
2 Background	2
2.1 vroom	2
2.2 Memory Management Unit	2
2.3 I/O Memory Management Unit	3
2.4 Direct Memory Access	4
2.5 Hugepages	4
2.6 Peripheral Component Interconnect Express	5
2.7 Character and Block Devices	6
2.8 Non-Uniform Memory Access	7
2.9 Rust	7
3 Related Work	8
3.1 Ixy	8
3.2 Data Plane Development Kit	8
3.3 Storage Performance Development Kit	8
4 Implementation	10
4.1 Virtual Function I/O	10
4.1.1 Groups and Containers	12
4.1.2 Binding NVMe to vfio-pci	12
4.1.3 IOMMU initialization	12
4.1.4 Device register access	13
4.1.5 NVMe initialization	14
4.1.6 DMA (Un-)Mapping	15
4.1.7 I/O operations with VFIO	16
4.2 IOMMUFD	17
4.3 Linux Systemcalls	20

5	Evaluation	21
5.1	Setup	21
5.2	Impact of 4 KiB pages	22
5.2.1	Determining IOTLB size	24
5.3	Latencies	28
5.4	Throughput	29
5.5	Increasing Queue Depth	30
5.6	Using multiple SSDs	31
5.7	PCIe limitations	32
5.8	IOMMU modes	32
6	Conclusion	34
	List of Figures	35
	List of Tables	36
	Listings	37
	Bibliography	38

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" [22]. 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 [21].

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 [20], 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 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 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 [25]. 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 prevented access to memory regions without permission. 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 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 must be performed. Thus, a page table walk

is a performance-costly operation. 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" [27], 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].

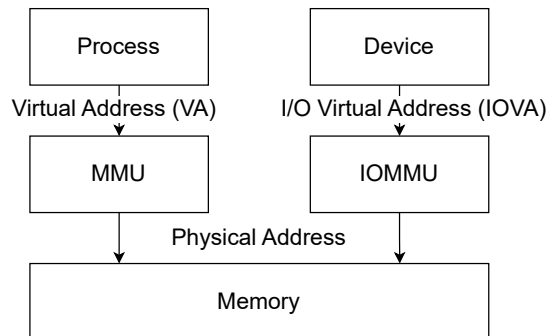


Figure 2.1: MMU and IOMMU relation to physical memory, adapted from [19]

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.

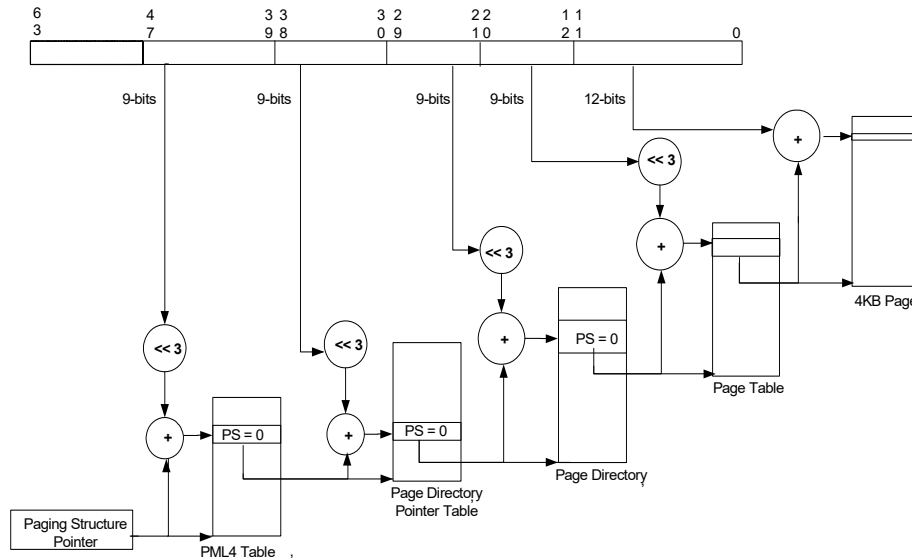


Figure 2.2: Intel VT-d Paging structure for translating a 48-bit address to a 4 KiB page, grabbed from [14]

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 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 [26].

Vroom currently uses hugepages for DMA and locks them with the Linux syscall `mlock` to prevent the kernel from swapping them out. Hugepages have to be used to prevent the kernel from moving data from one physical page to another physical page. The kernel cannot move these pages like 4 KiB pages. This is called ‘page pinning’. 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.

8 Bits	5 Bits	3 Bits
Bus #	Device #	Func #

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.

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
0x00	Vendor ID		Device ID		Command Register		Status Register	
0x08	Revision ID	Class Code			Cache Line	Latency Timer	Header Type	BIST
0x10	Base Address 0				Base Address 1			
0x18	Base Address 2				Base Address 3			
0x20	Base Address 4				Base Address 5			
0x28	CardBus				Subsystem vendor ID		Subsystem ID	
0x30	Expansion ROM Base Address				Cap. Pointer	Reserved		
0x38	Reserved				Interrupt Line	Interrupt Pin	MIN_GNT	MAX_LAT

Base Address Registers

Figure 2.4: PCIe configuration space, adapted from [20]

2.7 Character and Block Devices

Unix/Linux uses 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 typically have large data volumes and are organized in blocks, like HDDs and SSDs. 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.

2.8 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, and I/O buses [23]. 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. With `numactl`, it is possible to specify the locality of memory and CPU to run a process on one node [18].

2.9 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” [24]. 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.

4.1 Virtual Function I/O

Virtual Function I/O (VFIO) is an IOMMU agnostic framework for exposing devices to userspace. VFIO consists of two parts, the `vfio-pci` driver and an IOMMU API. The IOMMU API can either be the type1 IOMMU API for x86 architectures or the SPAPR IOMMU API for ppc64 architectures. We will only use the type1 IOMMU API. 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 [27]. The layers of VFIO can be seen on Figure 4.2a. VFIO practically acts like the kernel module to userspace drivers, allowing unprivileged, regulated access to physical memory and device registers.

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 initialization:** As the first step of the actual driver, we need to initialize the IOMMU. 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 exposing it to the NVMe controller for DMA.

Interaction with the VFIO interface works by 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 aren't 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 them 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]. To keep the binary and dependency list as small as possible we chose the manual implementation. Many `ioctl` calls used for VFIO also take in a mutable reference to a struct, which is used for specific input and/or 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.

We implement the struct `Vfio` and the enum `VfioIommu`.

Listing 4.1: Structs used to model VFIO

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

enum VfioIommu {
    Container {
        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 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 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.

4.1.4 Device register access

Using the `VFIO_DEVICE_GET_REGION_INFO` ioctl operation on the device `fd`, we can access the device registers. This operation requires the struct `vfio_region_info` as the third parameter, which needs to be initialized with a given index from `vfio.h`. After performing the syscall, the other fields, e.g. `size` or `offset` can be used to either read/write or map device registers.

We first need to enable DMA by setting a bit in the PCIe command register. Using `VFIO_PCI_CONFIG_REGION_INDEX` as the index, we can get the offset for the PCIe configuration space address `0x0`. By then adding the command register offset (`0x4`) 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 BAR register to memory. This is done using the `VFIO_PCI_CONFIG_REGION_INDEX` index. Using the offset and size, we can use `mmap` to map the `BAR0` register to memory.

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,
    IoctlOperation::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 NVMe initialization

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

4.1.6 DMA (Un-)Mapping

In order to provide a section of memory on which the device can perform DMA operations, the user needs to either allocate some memory in the processes address space or map an existing file into the process virtual address space. This can be achieved by using `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;
```

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 remove a DMA mapping from the IOMMU. Using the `VFIO_IOMMU_UNMAP_DMA` `ioctl` operation we can unmap the memory, and finally free it by using `munmap`.

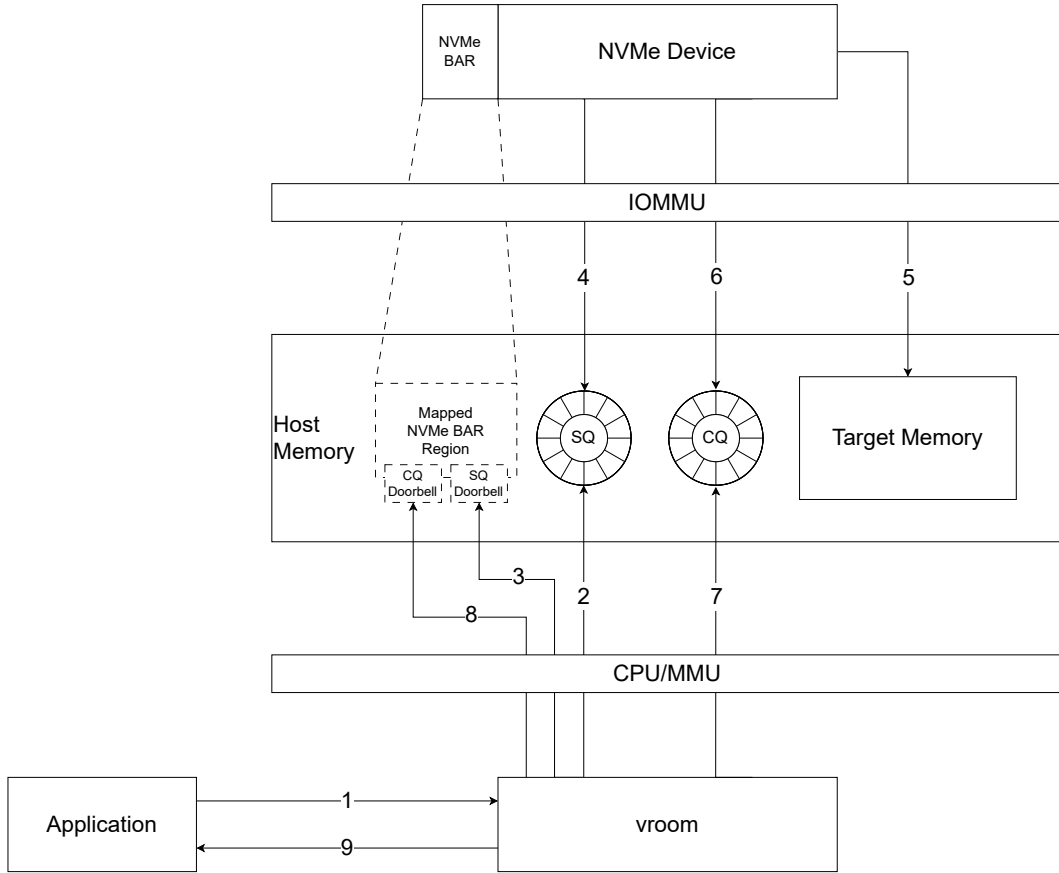


Figure 4.1: I/O operation using vroom with enabled IOMMU

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. The sequence of events 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

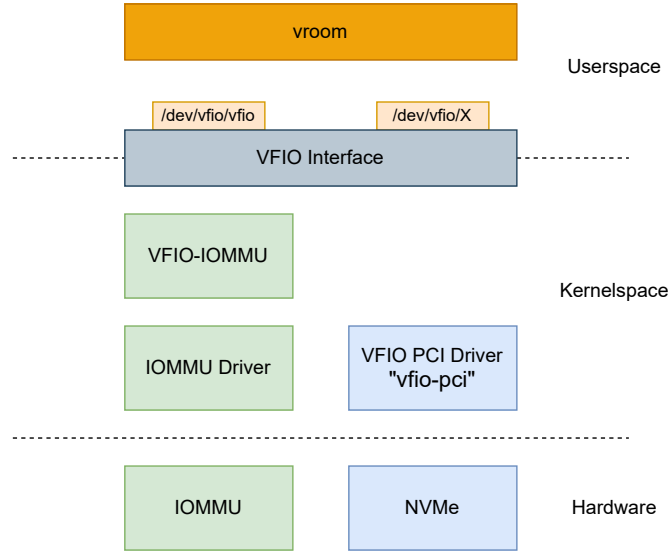
The IOMMU File Descriptor user API (IOMMUFD) offers a way of controlling the IOMMU subsystem using file descriptors in user-space [10]. IOMMUFD has only been recently added to the Linux Kernel in December 2022. E.g. Debian 12 does not include it, Fedora 40 does, but it is not enabled in the kernel configuration. 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.

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` [27]. In order to use the device with VFIO, it still has to be bound to IOMMUFD, using `VFIO_DEVICE_BIND_IOMMUFD`.

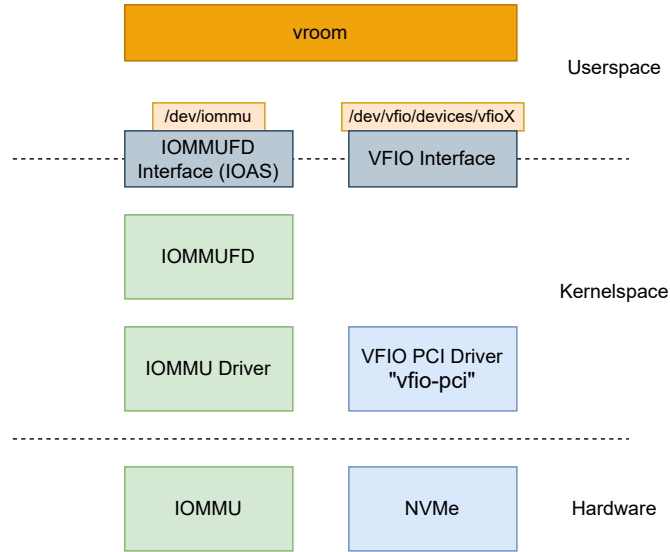
The IOAS can then be assigned to the device by using `VFIO_DEVICE_ATTACH_IOMMUFD_PT`. As with containers, this operation can be performed on multiple devices for a shared IOAS. The equivalent in VFIO is `VFIO_GROUP_SET_CONTAINER`.

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, partly adopted from [29]

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 inherent 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 do not include allocation and mapping times and perform them upfront. The focus lies on the IOMMU itself and how it performs with different page sizes. All performance tests use the Container IOMMU API instead of IOMMUFD 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 to benchmark the driver's performance. Both systems run Ubuntu 23.10 with Linux kernel version 6.5.0-42 and are NUMA systems with 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 [20]. 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. 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.

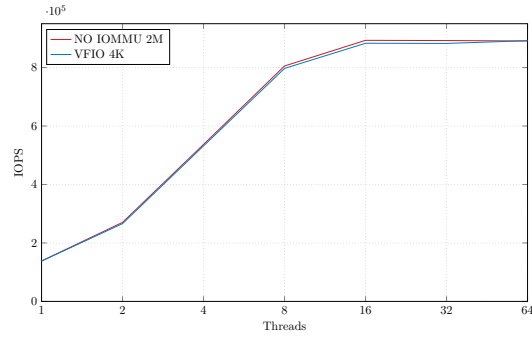
We also disable Intel's pstate and AMD's frequency scaling driver options which can influence the results.

Additionally, all standard tests are run with the `iommu.strict=1` kernel parameter to ensure a flushed IOTLB.

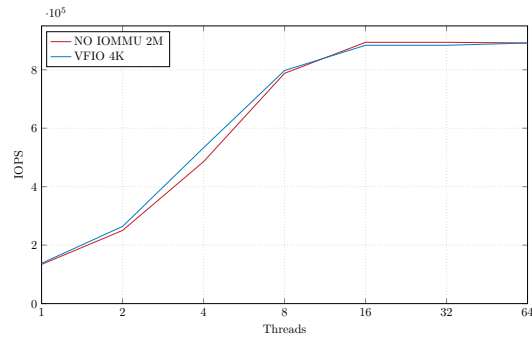
5.2 Impact of 4 KiB pages

When comparing the performance of vroom without the IOMMU against vroom with IOMMU using 4 KiB and 2 MiB pages, a performance difference can be noticed.

5 Evaluation

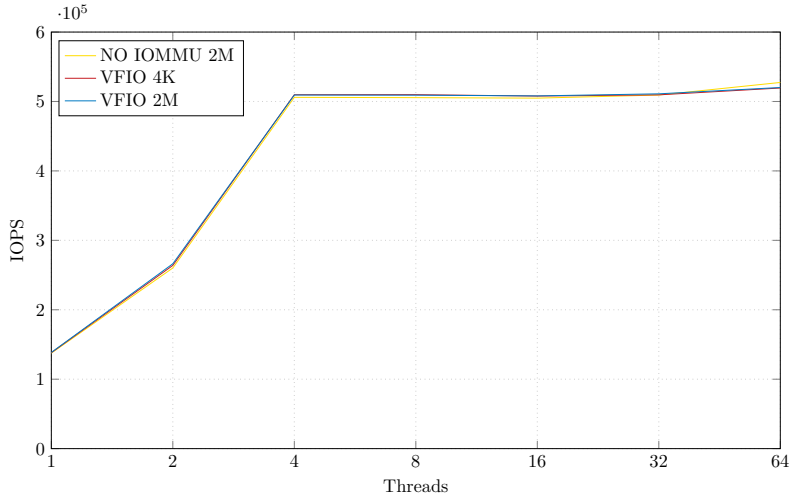


(a) 1 4 KiB buffer per thread

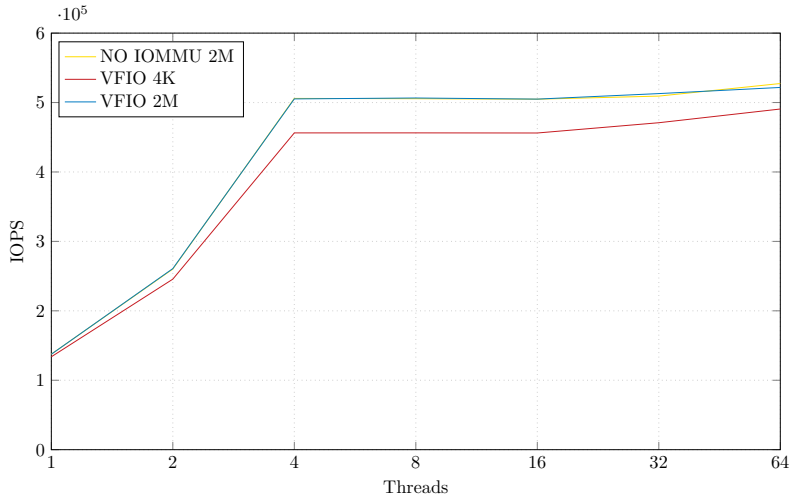


(b) 1 2 MiB buffer per thread

Figure 5.1: Singlethreaded random write throughput with increasing queue depth



(a) 1 4 KiB buffer per thread



(b) 1 2 MiB buffer per thread

Figure 5.2: QD1 random write throughput with multithreading

5.2.1 Determining IOTLB size

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.

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 4096 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. This test is done using without the IOMMU with 2 MiB pages and the IOMMU with 4 KiB, 2 MiB and 1 GiB pages.

Results of Intel Xeon In the resulting graph Figure 5.3 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.

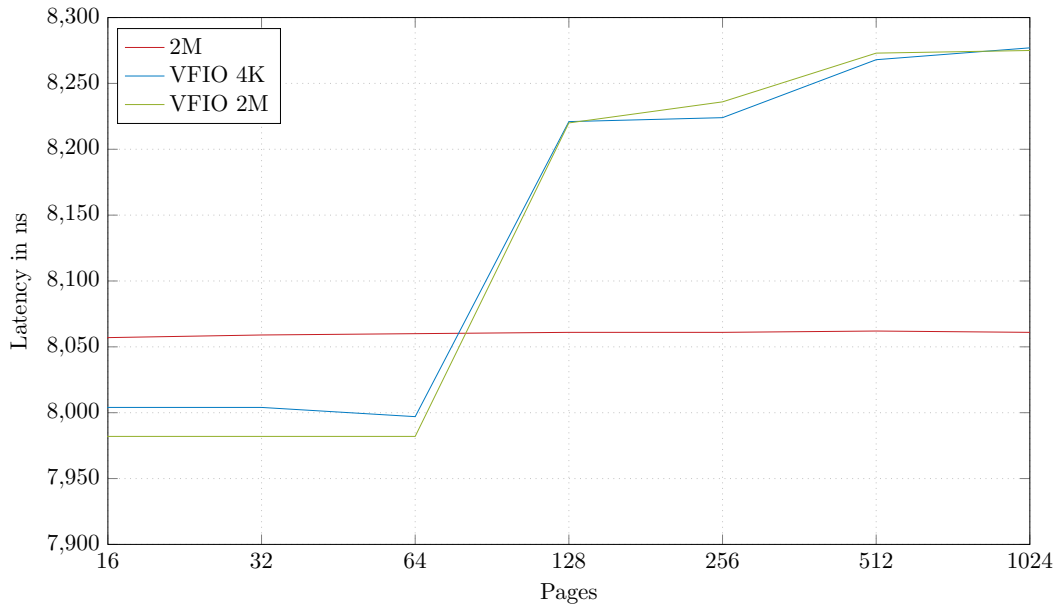


Figure 5.3: Latencies of random writes on an emptied SSD with increasing host memory pages on the Intel system

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 increase of page walks compared to the intel cpu system.

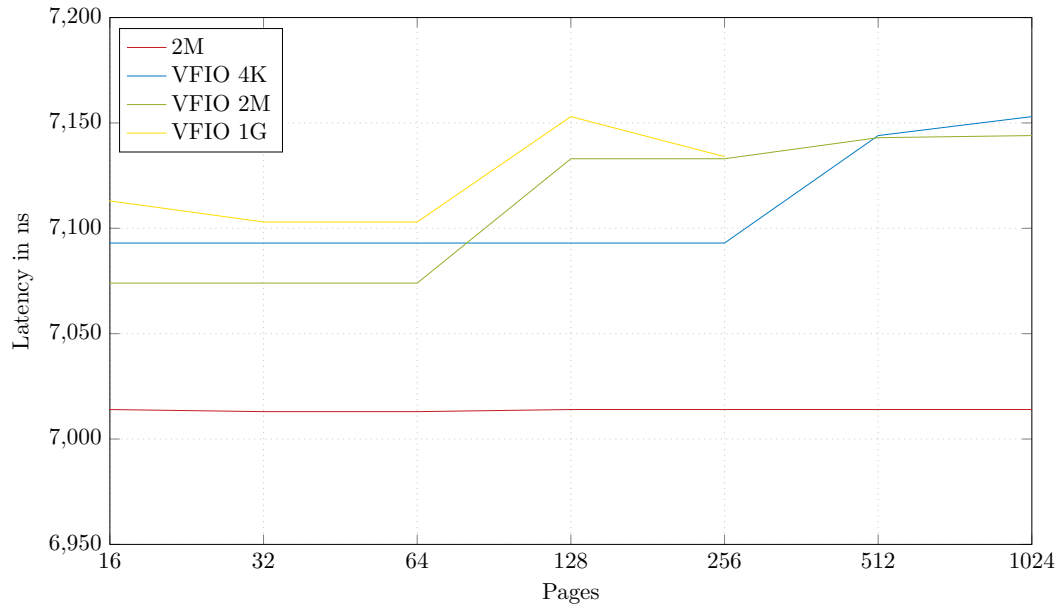
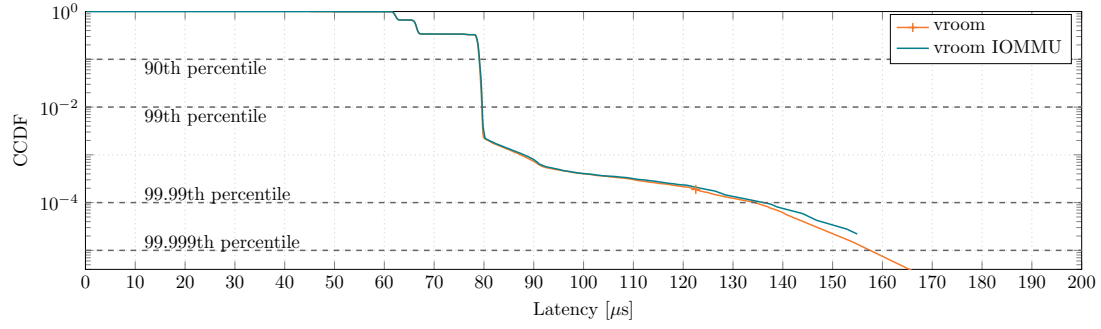
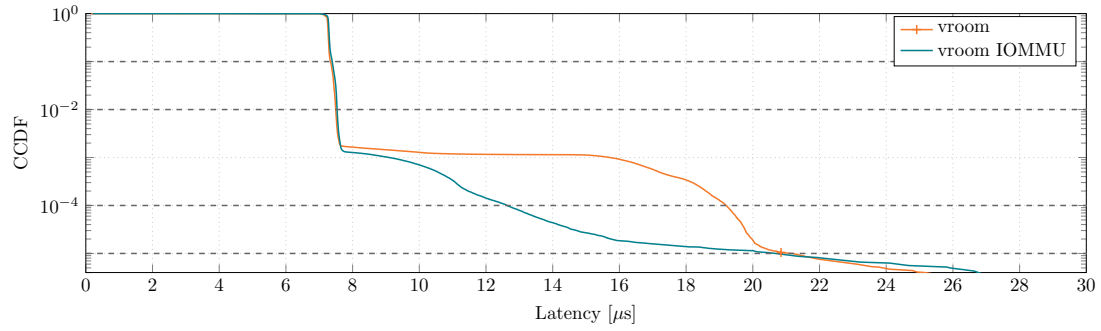


Figure 5.4: Latencies of random writes on an emptied SSD with increasing host memory pages on the AMD system

5.3 Latencies



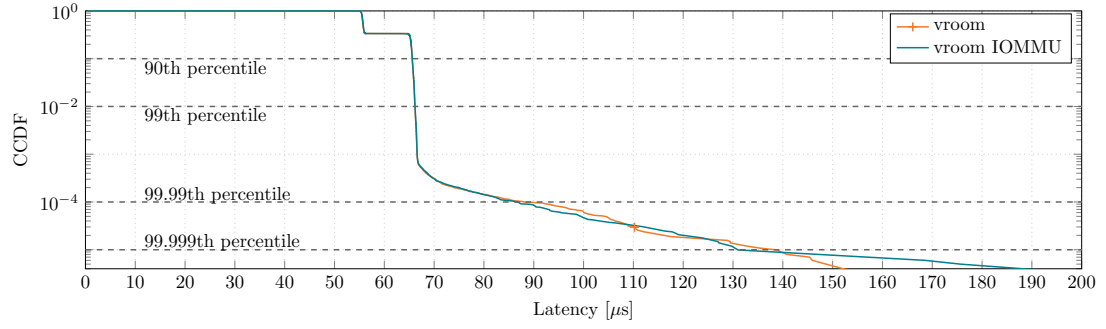
(a) Random read



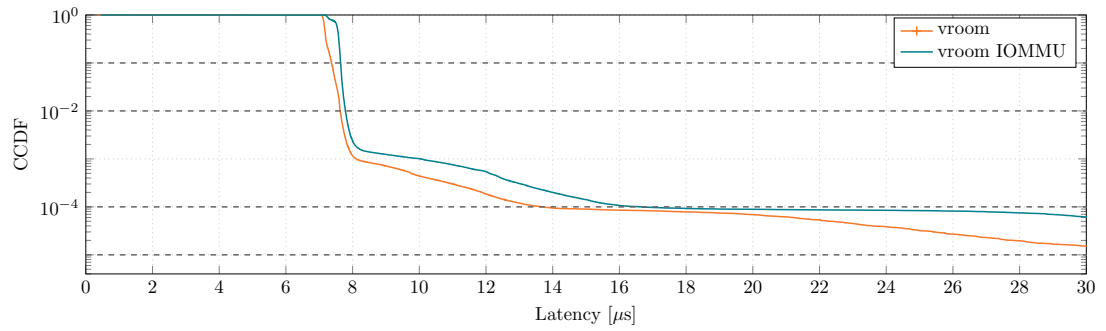
(b) Random write

Figure 5.5: Tail latencies on Intel System

5 Evaluation



(a) Random read



(b) Random write

Figure 5.6: Tail latencies on AMD System

5.4 Throughput

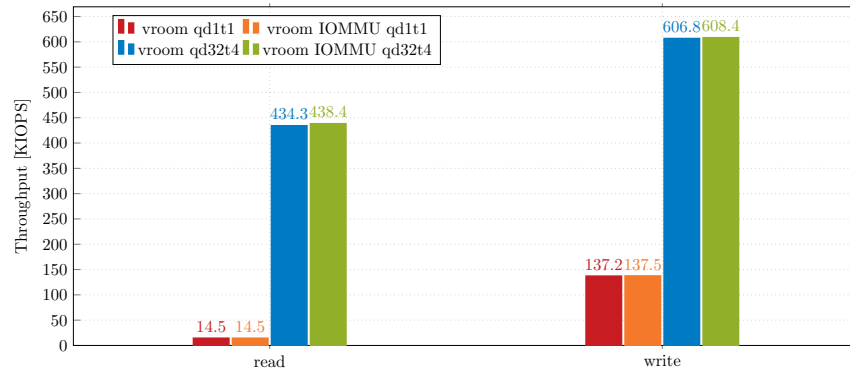


Figure 5.7: Throughput on Intel system

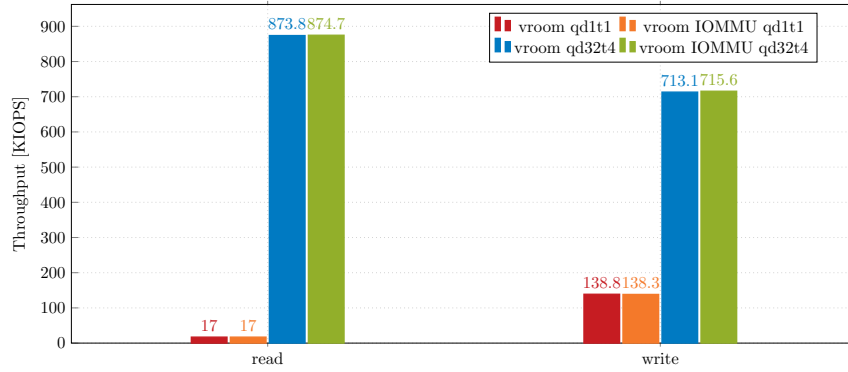


Figure 5.8: Throughput on AMD system

5.5 Increasing Queue Depth

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.

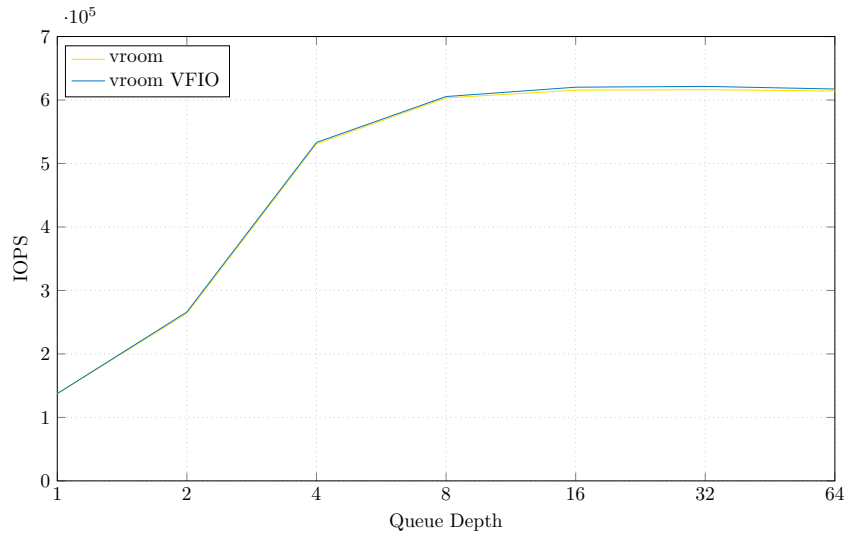


Figure 5.9: Throughput with increasing Queue Depth on the Intel system

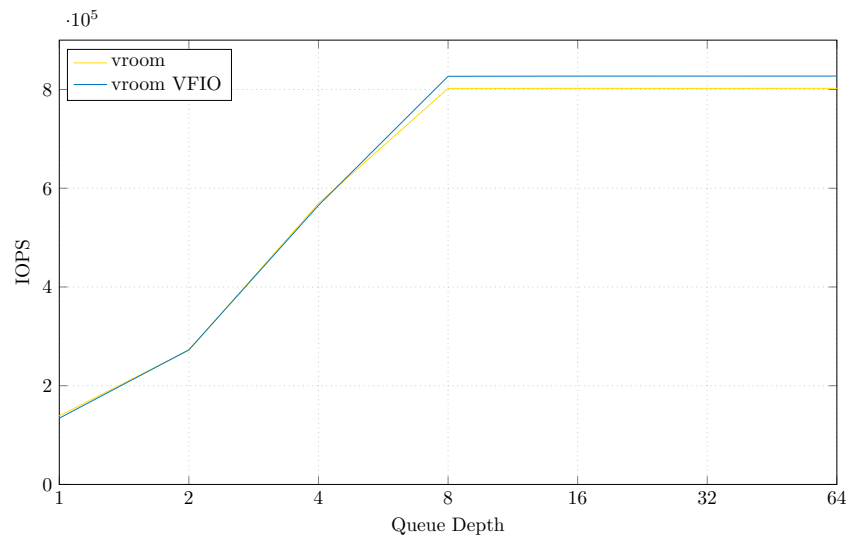


Figure 5.10: Throughput with increasing Queue Depth on the AMD system

5.6 Using multiple SSDs

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

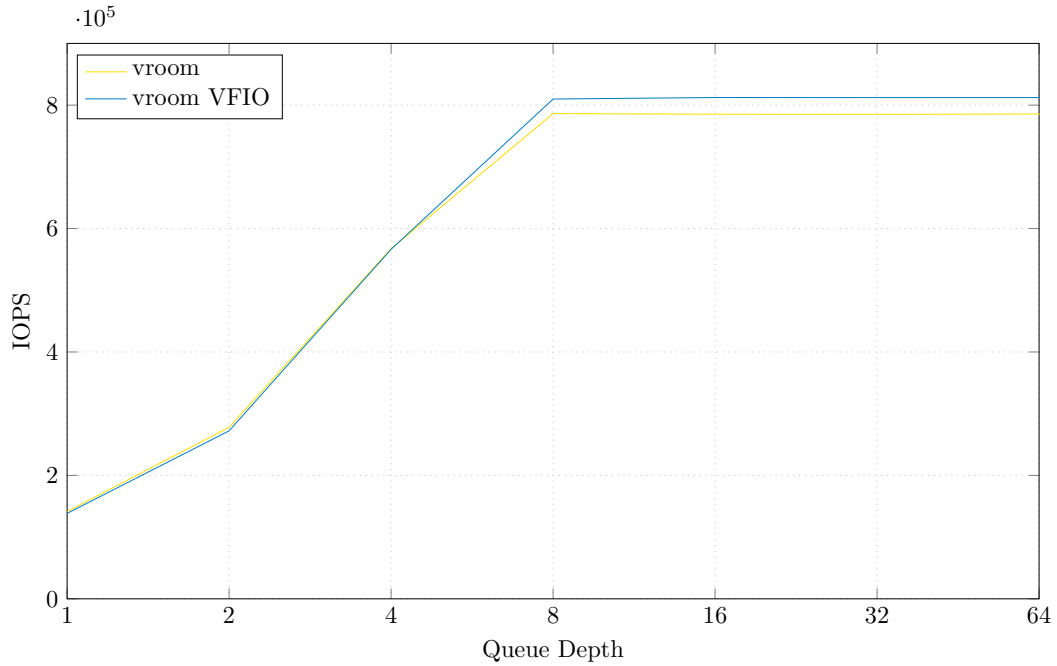


Figure 5.11: Throughput on 8 NVMe SSDs with 1 thread each

5.7 PCIe limitations

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 stems from this bottleneck.

5.8 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. there's `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 came to the same conclusion as SPDK. The complete code of the driver, including IOMMU support, can be found on GitHub [28]. In our tests, we only saw a noticeable performance impact on the older and slower Intel Xeon CPU, which makes us conclude that the IOMMU performance is becoming less of a factor. We figured out the IOTLB size of two IOMMU models, which, given the right conditions, can introduce address translation overhead. There was a noticeable overhead because of the IOMMU found, but it could only be observed with 4 KiB pages. This issue is solved by using 2 MiB pages. Using hugepages, the IOMMU performs the same as the physical addressing. 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, outweigh 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 passthrough 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 increase address space while not compromising on performance is 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 not only provides 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 or file system. Including sysfs support could also be a next step. It could be investigated if and how many threads could operate on one I/O queue to further push the throughput.

List of Figures

2.1	MMU and IOMMU relation to physical memory, adapted from [19] . .	3
2.2	Intel VT-d Paging structure for translating a 48-bit address to a 4 KiB page, grabbed from [14]	4
2.3	Segmented PCI identifier	5
2.4	PCIe configuration space, adapted from [20]	6
4.1	I/O operation using vroom with enabled IOMMU	16
4.2	Layer diagrams of VFIO with VFIO Container API and IOMMUFD, partly adopted from [29]	19
5.1	Singlethreaded random write throughput with increasing queue depth	23
5.2	QD1 random write throughput with multithreading	24
5.3	Latencies of random writes on an emptied SSD with increasing host memory pages on the Intel system	26
5.4	Latencies of random writes on an emptied SSD with increasing host memory pages on the AMD system	27
5.5	Tail latencies on Intel System	28
5.6	Tail latencies on AMD System	29
5.7	Throughput on Intel system	29
5.8	Throughput on AMD system	30
5.9	Throughput with increasing Queue Depth on the Intel system	30
5.10	Throughput with increasing Queue Depth on the AMD system	31
5.11	Throughput on 8 NVMe SSDs with 1 thread each	32

List of Tables

5.1	Specifications of systems used in performance testing	21
5.2	CPUs of the systems	21
5.3	NVMe(s) of the systems	22

Listings

4.1	Structs used to model VFIO	11
4.2	ioctl calls needed for IOMMU initialization	12
4.3	Mapping the BAR0 NVMe register to memory	14
4.4	Mapping memory for DMA	15
4.5	Syscall mmap macro, with own error variant	20

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 Access (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] *numactl(8) — Linux manual page*. URL: <https://man7.org/linux/man-pages/man8/numactl.8.html> (visited on 08/09/2024).
- [19] 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).
- [20] 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).
- [21] 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).

- [22] 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).
- [23] K. Sarcar. *What is NUMA?* URL: <https://docs.kernel.org/mm/numa.html> (visited on 08/09/2024).
- [24] *Storage performance Development Kit*. URL: <https://spdk.io/> (visited on 07/22/2024).
- [25] *Submitting I/O to an NVMe Device*. SPDK. URL: https://spdk.io/doc/nvme_spec.html (visited on 08/05/2024).
- [26] *Transparent Hugepage Support*. URL: <https://docs.kernel.org/admin-guide/mm/transhuge.html> (visited on 07/23/2024).
- [27] *VFIO - "Virtual Function I/O"*. URL: <https://docs.kernel.org/driver-api/vfio.html> (visited on 07/08/2024).
- [28] *Vroom source code*. URL: <https://github.com/adwuerth/vroom> (visited on 08/10/2024).
- [29] 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=ZhIOHEv50e0> (visited on 08/02/2024).