SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Effects of Linux VFIO for User Space I/O

Adrian Simon Würth

SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Effects of Linux VFIO for User Space I/O

Effekt von Linux VFIO auf User Space E/A

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

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

Munich, July 22, 2024

Adrian Simon Würth

Abstract

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

Contents

1 Introduction

The IOMMU was first implemented as a way to increase address space for 32-bit devices. After almost every device was using 64-bit addresses, the IOMMUs purpose shifted to providing a layer of isolation between devices and the CPU. [1] The isolation layer prevents devices from accessing memory that they are not allowed to access. An additional benefit of the IOMMU is that root privileges are not needed to access memory. We demonstrate the effects of using the IOMMU by implementing it on the NVMe driver vroom. [5] We use Rust to implement the driver and the userspace application as it offers high performance and memory safety without garbage collection. As even the Linux Kernel, which has stuck to C for more that 30 years not admitting any other languages like C++, not permits Rust code in its codebase, we can see Rusts impact on the systems development community. The IOMMU used in this thesis is an Intel IOMMU using the VT-d extension.

2 Background

2.1 vroom

vroom currently uses hugepages and locks them using mlock to prevent the Kernel from swapping them out. This only enables the use of 2MiB hugepages, which can be disadvantageous for certain applications that would benefit from smaller page sizes. A NVMe driver consists of submission and completion queues, implemented as ring buffers. The driver adds commands to the submission queue, which the NVMe controller reads and executes. The executed command gets placed on a corresponding completion queue. For accessing the devices memory as well as the device accessing the host memory, it is necessary to either use the physical addresses and compromise on safety and use root privileges or use the IOMMU for virtualization, which can introduce performance overhead. We unbind the kernel driver and bind it to pci-stub. Pci-stub does not do anything but occupy the pci-driver such that the kernel or another application can not bind to the device.

2.2 I/O Memory Management Unit

The I/O Memory Management Unit intercepts the PCI Bus and the CPU/Memory. The IOMMU works similarly to the MMU in the CPU, mapping physical addresses to virtual addresses. Using DMA Remapping (DMAR) the CPUs MMU is bypassed and the DMA request is handled by the IOMMU. The IOMMU does this by performing page table walks, through which a physical address is translated to a virtual address. To manage the huge address spaces required by modern computing, multi-level page tables are used. This results in a huge virtual address space size, but a single address resolution using multiple memory accesses. In the case of Intels VT-d the page table structures consists of 4, 3 and 2 tables for 4kib, 2mib and 1gib pages respectively as seen on Figure 2.1. In order to not perform such an expensive operation on every address resolution, a buffer is used to cache previous translations. This buffer is called the I/O Translation Lookaside Buffer (IOTLB). Using the buffer results in a fraction of the time needed to perform a page walk. If an address is not stored in the TLB, it is called a TLB miss, and the IOMMU has to perform a page table walk.

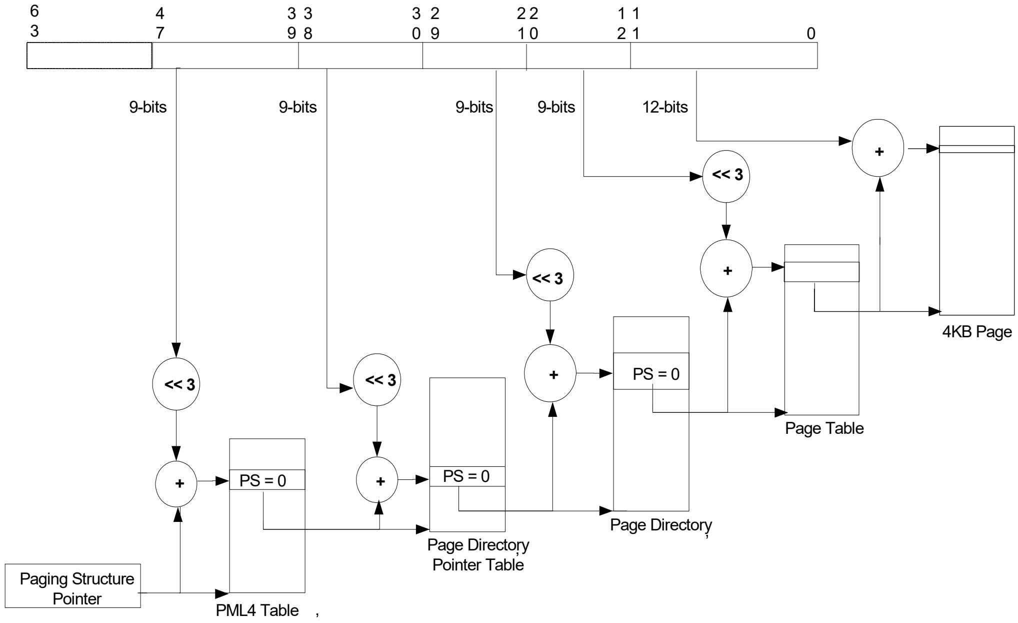


Figure 2.1: VT-d Paging structure for translating a 48-bit address to a 4-KByte page

2.3 I/O Translation Lookaside Buffer

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

2.3.1 Character and Block Devices

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

2.3.2 Rust

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

3 Related Work

3.1 Storage Performance Development Kit

The Storage Performance Development Kit (SPDK) provides “a collection of tools and libraries for writing high performance, scalable, user-mode storage applications” [6]. It includes an user-space NVMe driver which is fast and production-ready. While this driver supports the use of the driver without the IOMMU, the SPDK Documentation recommends using the IOMMU as using VFIO and the IOMMU is the "future proof...long-term foundation" for SPDK [3].

4 Implementation

4.1 Virtual Function I/O

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

4.1.1 Initialising the IOMMU

To use the IOMMU for the driver, we first need to initialize the VFIO kernel module and bind the VFIO driver to the NVMe device: This has to be done using root privileges. By chowning the VFIO container, the driver can use the VFIO driver to interact with the device without root.

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

Listing 4.1: Initializing VFIO using bash

```
#!/bin/bash
modprobe vfio-pci
nvme_vd="$(cat␣/sys/bus/pci/devices/$nvme/vendor)␣$(cat␣/sys/bus/pci/
    ↪ devices/$nvme/device)"
echo $nvme > /sys/bus/pci/devices/$nvme/driver/unbind
```

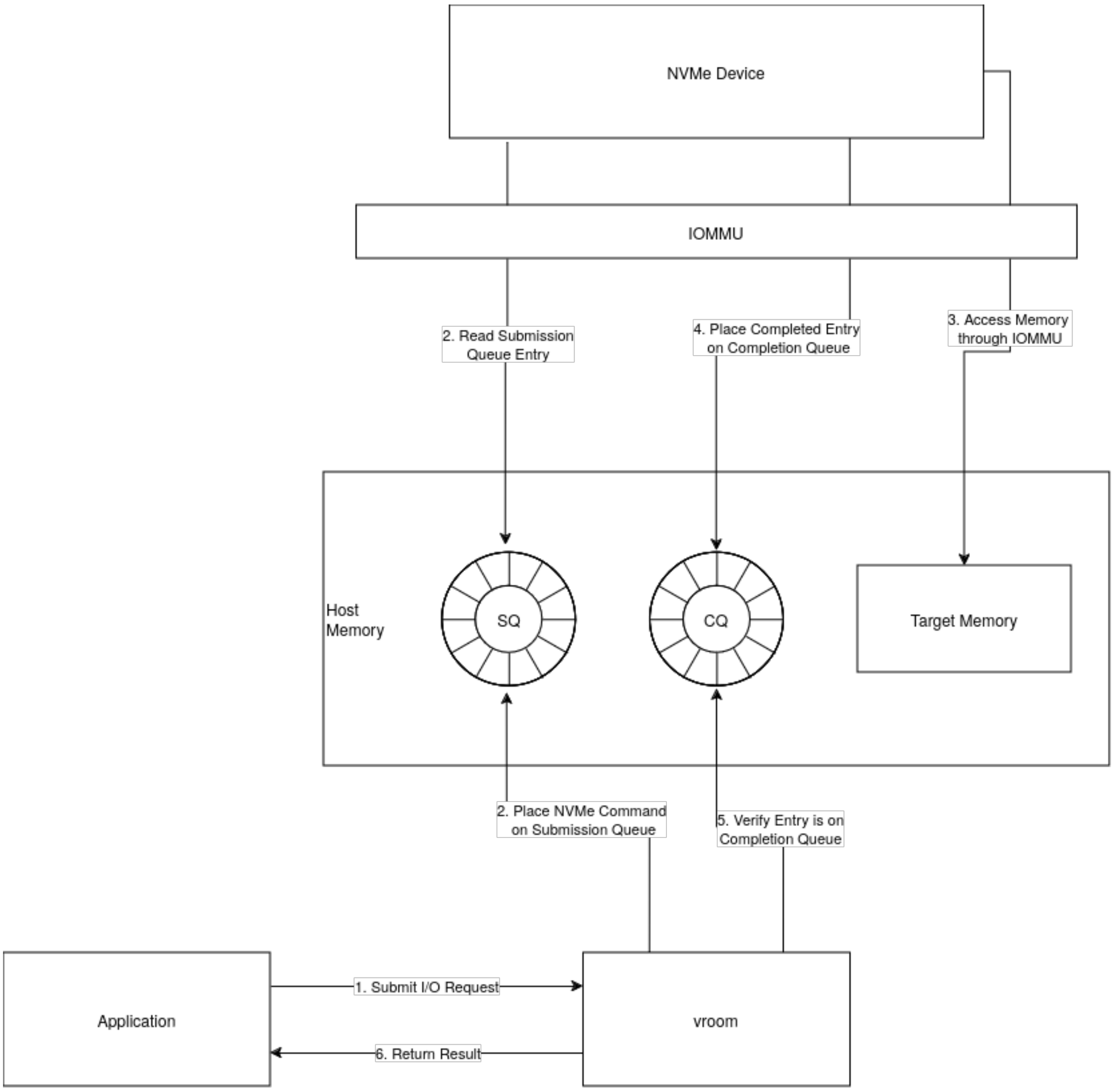

NVMe Device
IOMMU
2. Read Submission Queue Entry
4. Place Completed Entry on Completion Queue
3. Access Memory through IOMMU
Host Memory
SQ
CQ
Target Memory
2. Place NVMe Command on Submission Queue
5. Verify Entry is on Completion Queue
Application
1. Submit I/O Request
6. Return Result
vroom

Figure 4.1: I/O operation vroom

```
echo $nvme_vd > /sys/bus/pci/drivers/vfio-pci/new_id
chown $user:$group /dev/vfio/*
```

After this, the real work of the driver happens using these steps:

1. Mapping the NVMe device memory into host memory using VFIO resource info.
2. Allocating Admin SQ, CQ and I/O SQ, CQ
3. Creating a mapping on the IOMMU using VFIO
4. Configuring the NVMe device
5. Passing I/O Queue addresses to NVMe device using admin queues

4.1.2 Enabling DMA

4.1.3 Mapping DMA

In order to provide a section of memory on which the device can perform DMA operations, the user needs to allocate some memory in the processes address space. This is done 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.

4.1.4 Unmapping DMA

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

4.1.5 Regions

Using regions, we can directly mmap device memory into host memory for easy access to the NVMe controller. VFIO provides structs for using mmap to directly map the

NVMe device into memory. Using VFIO_DEVICE_GET_REGION_INFO we can attain the length and the offset needed for mmap.

4.1.6 Groups

VFIO uses group to distinguish between groups of devices which can be isolated from the host system. In the ideal case, every device would only be part of one group in order to increase security by providing single-device isolation. Groups are the smallest unit size on a system to ensure secure user access.

4.1.7 Containers

To further reduce overhead from the IOMMU Containers are used in VFIO, which can hold multiple groups. These containers can be used to ease translation and reduce TLB page faults. In our implementation we use one group and container each for our NVMe device.

4.2 IOMMUFD

The IOMMU File Descriptor user API (IOMMUFD) offers a way of controlling the IOMMU subsystem using file descriptors in user-space [4]. It allows management of I/O address spaces (IOAS), enabling mapping user space memory on the IOMMU. IOMMUFD has only been recently added to the Linux Kernel in December of 2022. E.g. Debian 12 does not include it, Fedora 40 does, but it is not enabled in the kernel configuration. Considering that it is not widely available or enabled on many distributions, our driver offers both options of using the IOMMU. The performance tests are done using the 'legacy' VFIO-only way, but it can be assumed that the framework/user api does not have a noteworthy performance impact. The device file descriptor, which was previously attained with VFIO_GROUP_GET_DEVICE_FD can now be gotten through opening the character device /dev/vfio/devices/vfioX. By using this character device pointer we can claim the ownership over the VFIO device. That way VFIO does not rely on group/container/iommu drivers.

4.2.1 IOAS

5 Evaluation

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

5.1 Setup

All benchmarks are run on a system with an Intel Xeon E5-2660 with 251 GiB of RAM running Ubuntu 23.10 with a 1 TB Samsung Evo 970 Plus NVMe SSD.

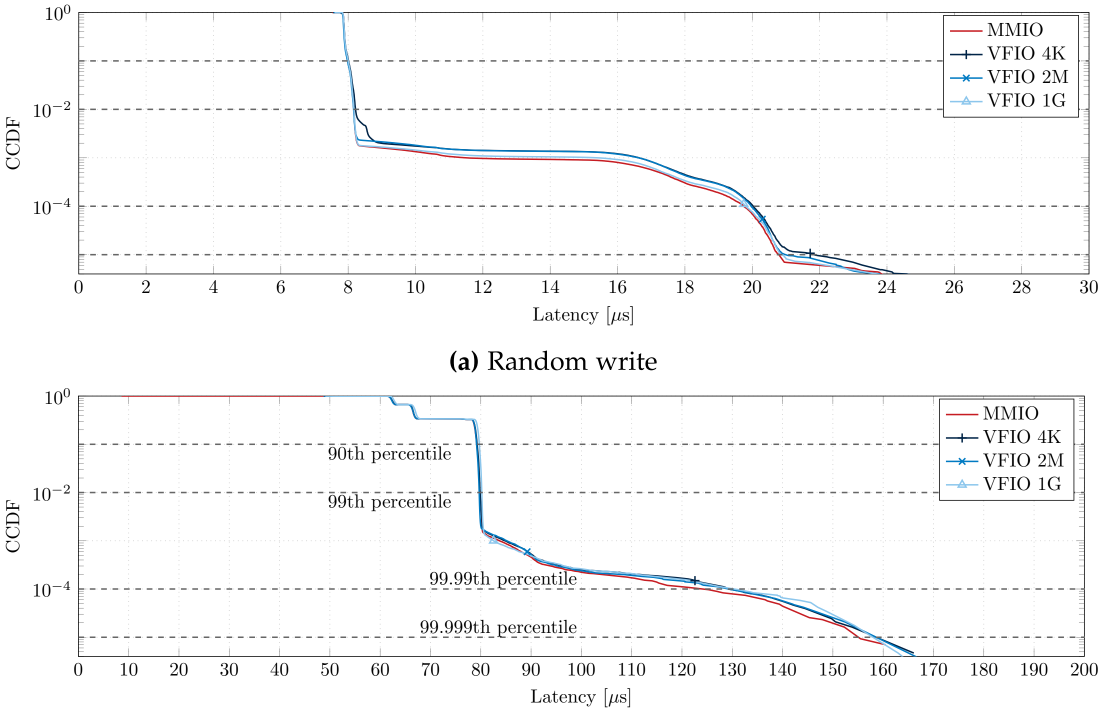
During our tests we will use 4KiB unit sizes for read and write accesses. As Linux as well as our IOMMU supports 4KiB, 2MiB and 1GiB page sizes we will test and analyse how it affects the overall performance.

5.2 Overall Latency and Throughput

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

5.3 IOTLB

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



(a) Random write

(b) Random read

Figure 5.1: Tail latencies

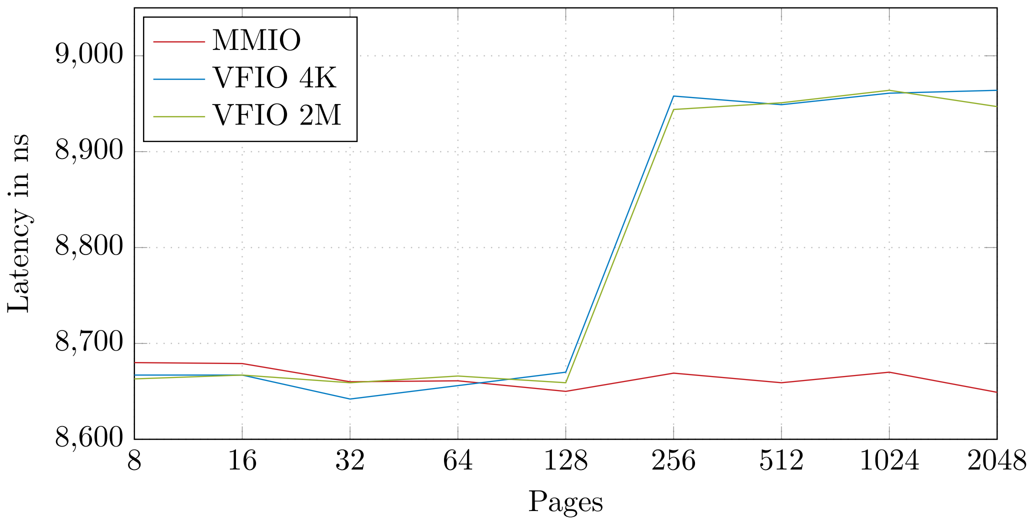


Figure 5.2: Pagesize Medians

comparing them we can figure out where a latency spike occurs and can then derive the IOTLB size.

Nvme Driver: 2 2mib sub queues, 2 2mib comp queues, 1 2mib buffer, 1 4kib prp_list => 512 * 2 + 512 * 2 + 512 + 1 = 2561 4KiB Pages, or 5 MiB Pages, depending on pagesize

5.4 Mapping

6 Conclusion

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

List of Figures

List of Tables

Listings

Bibliography

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

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

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

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

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

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