SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Writing an NVMe Driver in Rust

Tuomas Pirhonen

SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Writing an NVMe Driver in Rust

NVMe-Treiber in Rust

Author:	Tuomas Pirhonen
Supervisor:	Prof. Dr. Thomas Neumann
Advisor:	Simon Ellmann, M.Sc.
Submission Date:	April 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, April 15, 2024 Tuomas Pirhonen

Abstract

Today's SSDs are capable of performing millions of I/O operations per second (IOPS). However, while capable, traditional Linux I/O APIs and even newer asynchronous APIs often fall short of achieving the lowest possible latency and highest throughput due to their dependence on kernel-based I/O paths, which introduce significant overheads. The Storage Performance Development Kit (SPDK) offers a solution through its user space driver model, eliminating this overhead, but at the cost of increased complexity and potential safety concerns due to its C codebase.

Recognising these challenges, we present a novel user space driver written in Rust, a language that promises memory safety without sacrificing performance, employing zero-copy I/O and simple abstractions. With this, we aim to enable an easier way to assess individual NVMe features and I/O path optimisations. We show that, despite the stripped-down design of the driver, we achieve SPDK-like throughput and latency. Our work undertakes a comparative analysis between vroom, our proposed NVMe driver, and SPDK, as well as the Linux I/O APIs, intending to simplify access to high-performance storage technologies.

Contents

1 Introduction

In the age of PCIe Gen 5.0 NVMe SSDs, we have storage devices capable of exceeding one million I/O operations per second (IOPS). However, with Linux's standard file I/O API (p)`read` and (p)`write`, achieving this level of throughput is not possible, requiring over 100 threads to "get good throughput on a modern SSD" [9]. Modern asynchronous I/O libraries, such as `libaio` and `io_uring`, have offered improvements by diminishing the kernel gap, thus reaching closer to the physical limits of the SSD. However, these still pay performance penalties by interacting with the operating system kernel, like context switching and interrupt handling [2]. Maximising the throughput of an SSD requires circumventing the kernel entirely, which has led to the adoption of user space drivers such as SPDK's NVMe driver.

With this in mind, why write our own driver when SPDK exists? The SPDK codebase itself is complex and extensive, posing a barrier of entry into understanding its inner workings as well as understanding how to optimise storage I/O paths, with SPDK's `hello_world.c`[1] example coming in at 511 lines. It is also written in C, where critical errors, such as memory leaks or segmentation faults, are easily created in a lapse of judgement. While C remains the language used for the Linux kernel, it isn't necessarily the ideal language for driver development. In 2017, Cutler et al. analysed bugs in the Linux kernel which enabled arbitrary code execution and found that of 65, 40 bugs stemmed from invalid memory accesses, such as use-after-frees [1]; it was found that 39 of these bugs stemmed from device drivers [4].

In this thesis, we posit that it is feasible to develop a driver that achieves comparable performance to SPDK, with a simplified API and less code, while leveraging the benefits of a memory-safe programming language. Thus, we want to offer a platform that simplifies the exploration of NVMe and SSD capabilities and creates a driver where unsafe code is kept to a minimum. To this end, we will explore the development and evaluation of a user space storage driver written in Rust, a language that guarantees memory safety without any performance downfalls.

In chapter 2, we will review all the relevant background information, looking at how communicating with PCIe devices works, the NVMe specification, and the Rust programming language. Then, we will elaborate on other relevant I/O APIs and NVMe

[1] `https://github.com/spdk/spdk/blob/0680c7a27bd3950f0b7abb21effde66d5da7976e/examples/nvme/hello_world/hello_world.c`

driver implementations in chapter 3. We present the implementation in chapter 4. We go over the driver's architecture, as well as driver-specific implementations, after which we show how I/O operations are handled.

Finally, we analyse the driver's performance in chapter 5, looking at its throughput and latency while also comparing it to SPDK and the aforementioned Linux I/O APIs.

2 Background

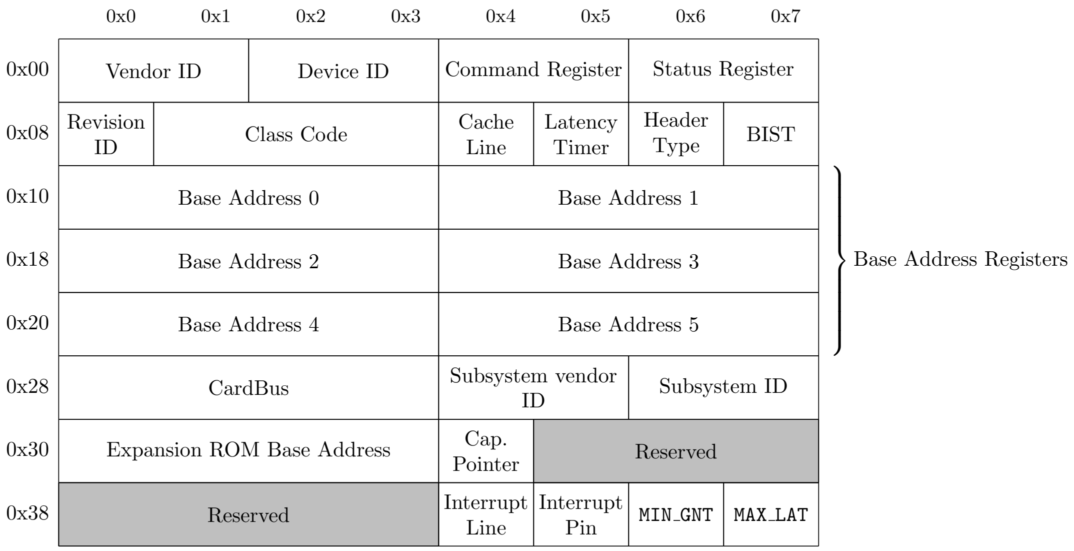
Before getting into the details of our driver, some concepts central to any NVMe driver need to be explained in detail first: how communicating with PCIe devices works with Memory-Mapped I/O and Direct Memory Access, the NVMe specification itself, and our programming language of choice, Rust.

2.1 PCI Express

Nowadays, all peripherals, from NVMe SSDs to USB hubs, are connected to the computer via the Peripheral Component Interconnect Express (PCIe) interface. PCIe builds upon PCI and PCI-X, defining the specifications for how the computer and peripheral devices interact.

These devices are connected via lanes over the PCIe bus, with transmit and receive channels. For higher throughput, the specification allows for devices with up to 32 lanes, each lane providing a raw bandwidth of 32 Gbit/s as of PCIe Gen 5.0 [15]. PCIe uses a TCP-like protocol for reliable data transmission with features like flow control, congestion avoidance, and acknowledgements, as well as packets.

Configuring PCIe devices is done by accessing the PCI configuration space, depicted by Figure 2.1, a standardised register space containing information and memory addresses of the PCI(e) device. For instance, we can disable interrupts or enable direct memory access (DMA) by configuring the "Command Register" accordingly.



	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				Base Address Registers
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	

Figure 2.1: Structure of PCIe configuration space

2.2 Memory-Mapped I/O

Memory-mapped I/O (MMIO) is a method of performing I/O operations between the CPU and peripheral devices in a computer. Using MMIO, the memory and register of a PCIe device share the same address space as main memory, allowing these to be addressed the same way as main memory, i.e. using CPU instructions like `mov`. Historically, peripherals were accessed using port-mapped I/O with specific `in` and `out` instructions to access the device's I/O ports via an I/O bus or pins.

The subsystem `uio` in Linux exposes the device's memory, Base Address Registers (BARs), as well as other required interfaces as files in the pseudo-filesystem `sysfs`.

2.3 Direct Memory Access

Passing data between hosts and PCIe devices is typically done through direct memory access (DMA); this allows peripheral devices to initiate access to main memory independently of the CPU. I/O performance can be improved by not involving the CPU in expensive memory transfers, giving it headroom to perform other operations.

Unlike on older hardware architectures, PCIe does not require a DMA controller (third-party DMA); instead, we can enable "bus mastering" (first-party DMA) for PCIe devices, which is necessary to allow the device to issue memory or I/O requests.

Typically, DMA requires the use of physical addresses; as such, bus masters can write to any address in main memory. The modern way of performing DMA is to use virtual addresses in conjunction with an input-output memory management unit (IOMMU). Using the IOMMU, DMA operations between a PCIe device and main memory are translated from bus addresses to virtual addresses, which then get translated to a physical address [20]. On Linux, the Virtual Function I/O (VFIO) driver framework enables the use of IOMMU for non-privileged, safe user space device drivers [11].

2.4 Non-Volatile Memory Express

Non-Volatile Memory Express (NVMe) itself "is an open collection of standards and information to fully expose [. . .] non-volatile memory in all types of computer environments"[1]. Relevant for us is the NVMe specification, an open logical device interface specification for accessing non-volatile storage media attached via the PCIe bus. NVMe was designed to capitalise on the low latency and parallelism of SSDs, providing improvements over older storage interfaces such as SATA or SAS in speed and latency. This specification defines several key components:

- **NVMe commands**: the basic units of work that the host system uses to communicate with the NVMe device. These commands may involve I/O operations or administrative tasks. These submission entries contain the command opcode, an identifier, and values about the command, e.g. data pointers for read/write operations. The structure of such a command can be seen in Figure 2.2
- **Submission Queues (SQ)**: The host system places commands here to be processed by the NVMe device. Each NVMe device can support multiple SQs, enabling parallel command processing.
- **Completion Queues (CQ)**: The NVMe device places completion entries, notifying that commands have been processed. Each completion queue is associated with a submission queue; however, the specification allows multiple submission queues to be associated with a single completion queue.

The specification supports 1 administrative submission and completion queue pair and up to 65535 I/O submission and completion queues, in theory allowing for high scalability and the ability to handle high volumes of I/O requests. Both the submission and completion queue operate as ring buffers, supporting up to 65536 entries and 65535

[1] `https://nvmexpress.org/about/`

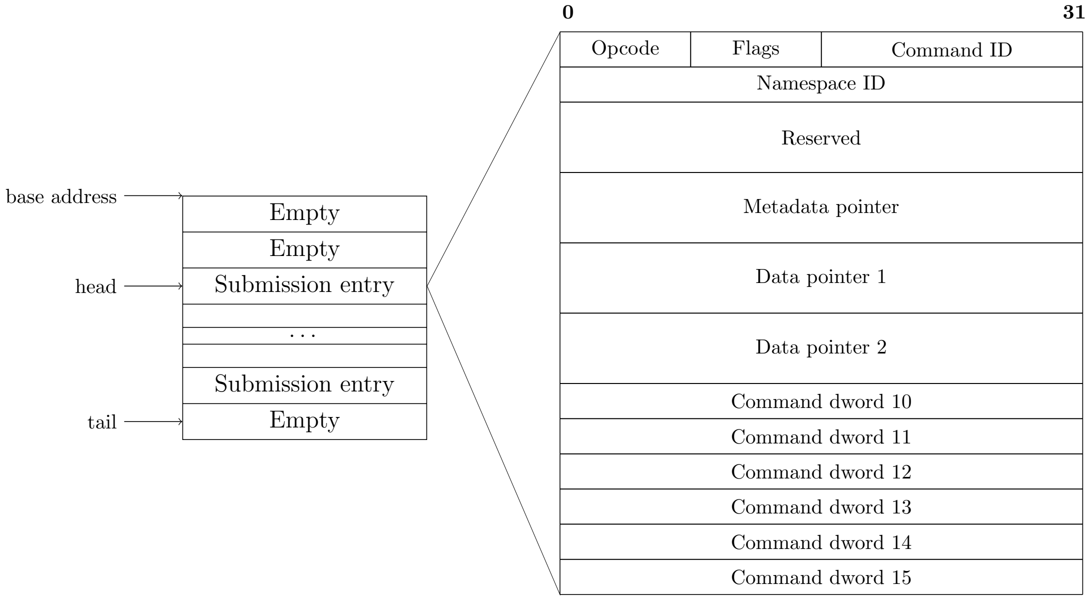


Figure 2.2: Example queue and structure of the NVMe submission entry

outstanding requests. Due to the command identifier being 16 bits in size, the NVMe controller supports up to 65536 outstanding requests at a time.

Submission and completion queues operate similarly to a Consumer-Producer pattern, e.g. in the case of an SQ, the host produces requests and adds them to the queue for the NVMe device to consume. For the SQ, we keep track of a `tail` pointer as the producer, pointing to the following free slot and a `head` pointer, pointing to the next slot to be consumed. We only keep track of the `head` pointer for the CQ.

Submitting requests to the NVMe device is done by constructing and then inserting a submission entry into the queue and updating the corresponding doorbell register to the new `tail` value, notifying the NVMe device of the newly submitted request. Upon completion, the NVMe controller will post a completion entry into the completion queue belonging to the submission queue. The completion entries contain the command identifier, the status of the operation, and other command-specific information. After processing the completion, the host then updates the doorbell register of the completion queue to the new value of `head` to signal that the completion entry has been acknowledged and processed. Depending on how the driver configures the NVMe controller, the device may send an interrupt signal upon completion.

Depending on the amount of data, different data pointers are passed to the command

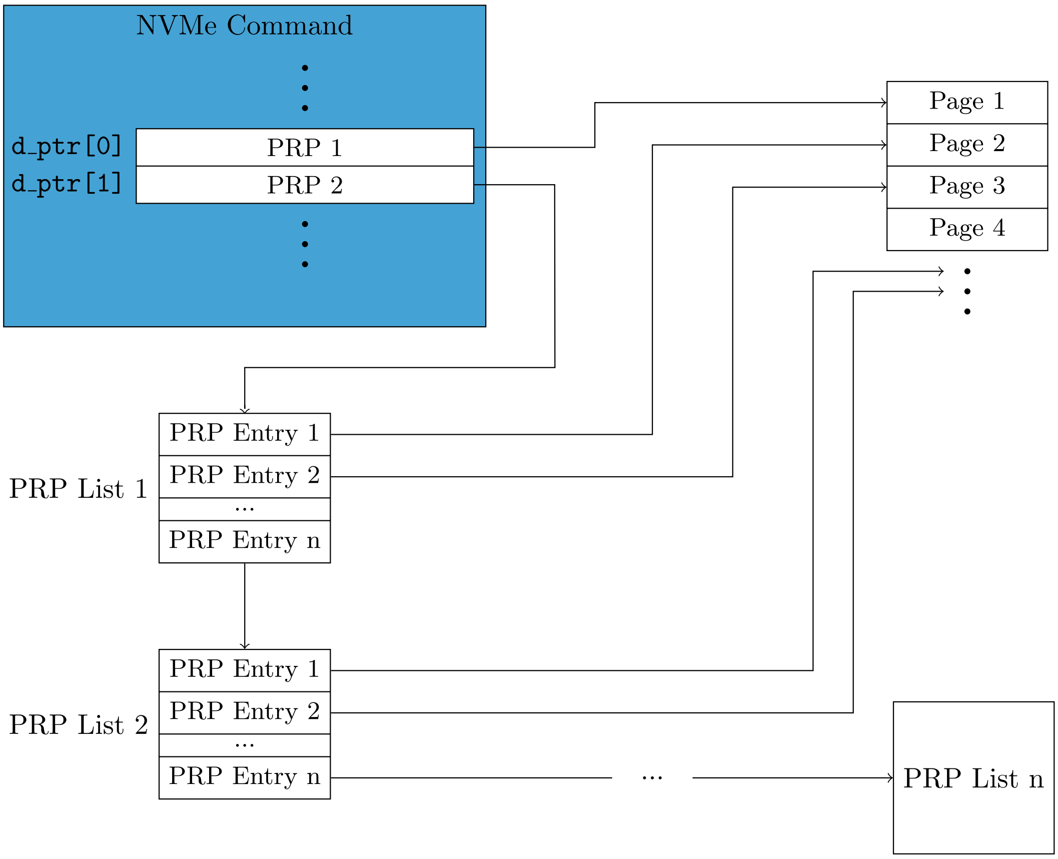


Figure 2.3: Visualisation of the PRP lists in NVMe commands

for reading and writing. The addresses passed to the data pointers are called Physical Region Page (PRP), which is nothing more than a pointer to a physical memory page. Another data structure for describing data buffers is the Scatter Gather List (SGL); however, these are not supported by all NVMe SSDs, while PRPs are.

For I/O operations sized less or equal to one page (by default 4 KiB), we pass the physical address to Data pointer 1 (`d_ptr[0]`). For requests which cross one memory boundary, we set `d_ptr[0]` to the physical address and Data pointer 2 (`d_ptr[1]`) to the address of the second block of data, e.g. `d_ptr[0] + nvme_page_size` for contiguous memory. Once the I/O operation spans more than two pages, a pointer to a PRP list is passed to `d_ptr[1]`; this is depicted in Figure 2.3. A PRP list is a list of PRP entries where the last entry points to the following PRP list, essentially an array of pointers.

2.5 Rust

Memory bugs remain amongst the most exploited vulnerabilities [12], and companies such as Google have begun to transition away from C and C++ towards using memory-safe languages, like Java, Rust or Go [7]. Meanwhile, an effort exists to integrate Rust into the Linux kernel spearheaded by the Rust for Linux project[2], with Linux adopting support for the programming language with release 6.1.

Rust[3] is a modern systems programming language focusing on safety, speed and concurrency. It was designed to provide memory and thread safety guarantees through a unique ownership model without any performance pitfalls. These safety checks are done at compile time, eliminating common bugs like null-pointer dereferences and data races. Furthermore, Rust is a compiled language without a garbage collector, leading to much better performance than interpreted languages and less overhead than languages with a garbage collector.

With all these factors in mind, Rust seems to be an ideal programming language for developing (user space) device drivers where safety and efficiency are paramount.

[2] `https://rust-for-linux.com/`

[3] `https://www.rust-lang.org/`

3 Related Work

Writing device drivers is nothing new; as such, we will go over the user space NVMe drivers implemented by SPDK, Redox and Redleaf, the latter two also written in Rust like vroom. Additionally, we will look at what I/O APIs Linux offers for high throughput.

3.1 SPDK

Originally developed by Intel, the Storage Performance Development Kit (SPDK) provides a large set of tools libraries for high-performance storage applications, at the centre of which is its NVMe driver. The driver itself is poll-based, zero-copy, and runs in user space. By eliminating interrupts and system calls entirely, SPDK achieves the highest performance out of all modern storage APIs [2]. However, its added complexity raises the question of whether it's possible to write a user space, poll-based driver in Rust that achieves similar performance while also focusing on simplicity.

3.2 Redox

Redox is a Unix-like operating system written entirely in Rust to be a "robust, reliable and safe general-purpose operating system" [17]. Its development began in 2015 by Jeremy Soller and is still being actively worked on at the time of writing.

Redox is based on a microkernel architecture, so many operating system functionalities run in user space, especially drivers. Its design focuses on minimalism and modularity, emphasising implementing as much of the operating system in user space as possible, improving system stability and security. Currently, Redox's NVMe driver employs an interrupt-driven architecture and supports asynchronous I/O to the NVMe device, using `Future`'s for interrupt handling.

There have been no performance evaluations of Redox's NVMe driver at the time of writing.

3.3 RedLeaf

Like Redox, RedLeaf [16] is also a microkernel operating system written in Rust. Developed by the University of Utah's Mars Research Group in 2020, RedLeaf's NVMe user space driver shares a similar structure to Redox's driver. Their benchmark results show that the RedLeaf driver can achieve an I/O throughput within 1% of and, in some cases, even exceeding what SPDK achieves [13]. It is important to note that Redleaf's driver was only tested in its sequential reading and writing performance.

3.4 Linux

Linux provides various APIs for handling I/O operations, the most prominent for asynchronous I/O being `libaio` and `io_uring`; the latter is still being actively worked on and improved upon.

With the use of `libaio`, applications can access block devices asynchronously. This centres around the two system calls `io_submit()` and `io_getevents()`. As the names suggest, the former submits I/O requests to the kernel, while the latter retrieves I/O completions. With two system calls required per request, there comes significant overhead from context switches, with data being copied from kernel to user space and vice versa.

`io_uring`, on the other hand, "implements a shared memory-mapped, queue-driven request/response processing framework" [2] by implementing a submission and completion ring which is mapped into user space and shared with the kernel. An application can add submissions to the submission ring without any system calls; however, by default, it notifies the kernel about new entries in the ring with `io_uring_enter()`, similar to updating the submission queue doorbell in the NVMe specification. As the completion queue is mapped into user space, it is also possible to poll for completions by polling for new entries in the completion ring, the alternative is to wait for new completions with `io_uring_enter()`. `io_uring` can also spawn a kernel thread, which polls for new submissions; thus, it can handle I/O without system calls. In this mode, `io_uring` can achieve performance within 10% of SPDK, albeit at a higher CPU usage, requiring CPU cores for the polling threads to achieve higher throughput [2].

4 Implementation

Our driver implementation takes architectural inspiration from the driver implemented in Redox, as well as notes from SPDK; additionally, many utility functions are adapted from those in `ixy.rs` [3]. All NVMe commands and directives are implemented based on Revision 1.4 of the NVMe Specification [14]. The source code can be found at `https://github.com/bootreer/vroom`.

4.1 User Space Drivers

Like SPDK, vroom runs entirely in user space and implements zero-copy I/O operations, as well as a poll-based architecture; SPDK[1] being the de-facto standard for NVMe devices in high throughput environments. To write a user space driver, we use all the concepts explained in chapter 2. The driver can access the device after memory mapping device files from user space, which offers more flexibility, simplicity, and stability over kernel drivers; with access to debugging tools and overall fewer restrictions, the development of user space drivers is much easier, and the ability to use any programming language is also guaranteed. User space drivers are also less likely to cause system crashes or kernel panics due to bugs; faults in user space can often be handled gracefully, improving overall system stability. By avoiding context switches, these drivers can reduce overall latency and increase throughput.

Given these advantages, we chose to develop a user space NVMe driver rather than kernel space.

4.2 Memory-Mapped I/O

Listing 4.1 shows how to implement memory-mapping a PCIe resource in Rust. In this example, we open the `resource0` file with read and write access and pass the file descriptor and its length to `libc::map`. As `libc::mmap` directly calls `mmap(2)`, the function call is wrapped in an `unsafe` block. The BARs are mapped as shared memory, so changes to the mapped memory are also written back to the file and vice-versa. If

[1] `https://spdk.io`

Listing 4.1: Memory mapping a PCIe resource in Rust

```
pub fn map_resource(pci_addr: &str) -> Result<(*mut u8, usize), Error> {
    let path = format!("sys/bus/pci/devices/{}/resource0", pci_addr);

    let file = fs::OpenOptions::new().read(true).write(true).open(&path)?;
    let len = fs::metadata(&path)?.len() as usize;

    let ptr = unsafe {
        libc::mmap(
            ptr::null_mut(),
            len,
            libc::PROT_READ | libc::PROT_WRITE,
            libc::MAP_SHARED,
            file.as_raw_fd(),
            0,
        ) as *mut u8
    };

    if ptr.is_null() || len == 0 {
        Err("pci␣mapping␣failed".into())
    } else {
        Ok((ptr, len))
    }
}
```

the function returns a null pointer or the length of the file is 0, we return an error. Otherwise, the pointer and the file length are returned as a pair.

4.3 Direct Memory Access

We use DMA to enable the transfer of data between the host system and the NVMe device. We initialise DMA memory for all submission and completion queues, as well as buffers that the device can read from and write to. As PCIe devices access memory via physical addresses independently of the CPU, we require the buffers we use for DMA to stay in main memory. We can use `mlock(2)` to guarantee a memory page is locked in main memory; however, the mapping is not static for 4 KiB pages, the standard page size on Linux. Instead, we make use of 2 MiB huge pages for this, where the physical addresses are pinned in Linux [5]. Enabling the usage of huge pages on the operating system is done with the shell script `setup-hugetlbfs.sh`, which creates a mount point for huge pages and writes a number of huge pages to `sysfs` files. Now we can allocate memory by creating the file in the newly mounted directory and then memory map the file with `mmap(2)` and lock it in memory with `mlock(2)` by using the appropriate bindings in the `libc` crate. We then derive the physical memory address of the page through `/proc/self/pagemap`.

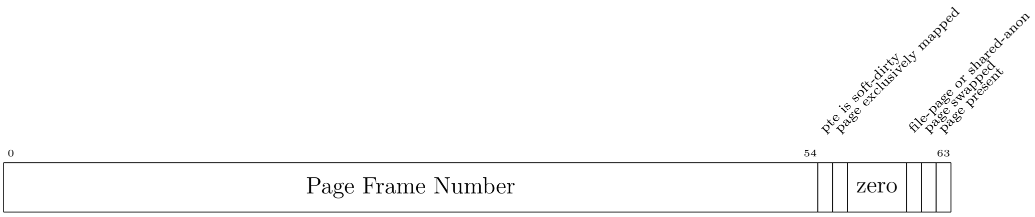


Figure 4.1: Fields of a pagemap entry when the page is present in main memory

The pagemap contains one 64-bit value for each virtual page; our huge page is in main memory, so the pagemap entry is structured as depicted Figure 4.1. Finding the relative index of the page is done by taking the virtual address of it and dividing it by the page size, which we use to locate the corresponding pagemap entry. Constructing the physical address from the pagemap entry and the virtual address is done by taking the Page Frame Number (Bits 0-54), multiplying that by the page size to get the physical address of the page, to which we add (`addr % pagesize`), the offset within the physical page. For 2 MiB pages, this offset is 0. Listing 4.2 shows how this is done in Rust.

One of the reasons vroom needs to run as root is due to requiring `CAP_SYS_ADMIN` to read the Page Frame Numbers ever since the Rowhammer vulnerability exploit, i.e. from Linux 4.0 onwards.

For DMA, we define the struct `Dma<T>`, with which we can allocate a memory-mapped huge page, encapsulating its virtual address type `*T` and physical address. We also define the trait `DmaSlice`, which allows us to iterate over chunks of the DMA memory and use a subsection of the memory via slicing. This trait is used for zero-copy I/O operations.

4.4 Architecture

Our overall goal was to design a lightweight driver where external dependencies are kept to a minimum. As such, we only require the crates `libc` for bindings to C library

Listing 4.2: Translating a virtual address to its physical address

```
fn virt_to_phys(addr: usize) -> Result<usize, Error> {
    let pagesize = unsafe { libc::sysconf(libc::_SC_PAGESIZE) } as usize;

    let mut file = fs::OpenOptions::new()
        .read(true)
        .open("/proc/self/pagemap")?;

    file.seek(io::SeekFrom::Start(
        (addr / pagesize * mem::size_of::<usize>()) as u64,
    ))?;

    let mut buffer = [0_u8; mem::size_of::<usize>()];
    file.read_exact(&mut buffer)?;

    let phys = unsafe {
      mem::transmute::<[u8; mem::size_of::<usize>()], usize>(buffer)
    };
    Ok((phys & 0x007F_FFFF_FFFF_FFFF) * pagesize + addr % pagesize)
}
```

functions and byteorder for reading different-sized integers and to avoid unnecessary unsafe code from working with raw bytes.

Listing 4.3: NvmeDevice struct definition

```
pub struct NvmeDevice {
    pci_addr: String,
    addr: *mut u8, // BAR address
    len: usize, // BAR length
    dstrd: u16, // Doorbell stride
    admin_sq: NvmeSubQueue, // Queues
    admin_cq: NvmeCompQueue,
    io_sq: NvmeSubQueue,
    io_cq: NvmeCompQueue,
    buffer: Dma<u8>, // 2MiB of buffer
    prp_list: Dma<[u64; 512]>, // PRP list
    pub namespaces: HashMap<u32, NvmeNamespace>,
    pub stats: NvmeStats,
    q_id: u16,
}
```

The entry point to accessing an NVMe device on our driver is the NvmeDevice struct (Listing 4.3). Like in Redox, we designed this struct to be able to handle all NVMe operations, i.e. Administrative and I/O capabilities; however, as we haven't implemented async, we handle the request submission and completion polling from beginning to end, meaning requests currently operate synchronously.

To enable multithreaded I/O processing, we have defined NvmeQueuePair, so, akin to SPDK, each thread can have its own queue pair to handle reading and writing without needing locking access to a single instance of NvmeDevice. With the queue pair, submissions and completions can be handled independently of one another. Each queue pair encapsulates a unique identifier, a completion and submission queue. With unique identifiers, we guarantee each queue pair only writes to its corresponding doorbell registers, thus enabling lock-free multithreaded I/O to the NVMe device.

4.5 Driver Initialisation

In this section, we will go over the initialisation process of the driver, looking at what happens within the function init() in lib.rs and the functions it calls; this function returns an instance of NvmeDevice if nothing goes wrong.

Before any configuration and initialisation are done, we check if the PCIe device has the class id 0x0108: 0x01 for mass storage device, 0x08 the NVMe subclass. We then unbind the kernel driver from the NVMe device by writing the PCIe address of the device to the unbind file in sysfs. This is followed by enabling the bus master and disabling interrupts by setting the appropriate bits in the PCI command register, thus enabling DMA and disabling interrupts entirely, as our driver is poll-based. At this point, we also initialise all the relevant structs required for the driver, e.g. admin and I/O queues.

The BARs of the NVMe device are then mapped into main memory, as described in section 4.2. We then follow the initialisation procedure described in the NVMe specification. First, we disable the controller by setting the EN (enable) bit to 0 in the CC (Controller Configuration) register. We wait for the ready bit in the CSTS (Controller Status) register to be set to 0, after which we can configure the controller. Configuring the controller involves setting register values to those we require, such as the attributes of the admin queues and command entries. All relevant offsets for the registers are stored in the enum NvmeRegs32 and NvmeRegs64. After all the configuration is set, the controller is re-enabled by setting the EN bit to 1. Now, the NVMe controller is ready to process admin submissions.

Listing 4.4: Writing to a 32 bit register

```
fn set_reg32(&self, reg: u32, value: u32) {
    assert!(reg as usize <= self.len - 4, "memory␣access␣out␣of␣bounds");
    unsafe {
      std::ptr::write_volatile(
        (self.addr as usize + reg as usize) as *mut u32, value
      );
    }
}
```

Although we use unsafe functions to access the BARs of the NVMe device, with an assertion guard, we can make sure that all accesses to the registers are not out of bounds; shown by Listing 4.4 for writing to 32-bit registers.

After this, we request an I/O completion queue, followed by a request for an I/O submission queue so the NvmeDevice can do I/O operations. Finally, we identify the namespaces, storing these in a HashMap in the NvmeDevice.

4.6 I/O Operations

As all commands are processed through submission and completion queues, we have defined the structs NvmeSubQueue, and NvmeCompQueue, which store a NvmeCommand array, and NvmeCompletion array each on a huge page. The submission entry being 64 bytes in size means that at most 1024 entries fit on a 2 MiB huge page; for simplicity, we have not implemented queues spanning multiple pages. Thus, the maximum number of outstanding submissions we currently support is 1023 per queue.

Name	Zero-Copy	Struct
`read()`	Yes	`NvmeDevice`
`write()`	Yes	`NvmeDevice`
`read_copied()`	No	`NvmeDevice`
`write_copied()`	No	`NvmeDevice`
`batched_read()`	No	`NvmeDevice`
`batched_write()`	No	`NvmeDevice`
`submit_io()`	Yes	`NvmeQueuePair`

Table 4.1: I/O methods in vroom

Table 4.1 contains an overview of all the methods responsible for reading or writing. We pass references to `impl DmaSlice` to the zero-copy methods, which are iterated over in chunks of 8196 KiB, as larger chunks require the use of PRP lists. We avoid building PRP lists for each request for now, as these require DMA-able memory. Additionally, `submit_io()` returns the number of requests added to the submission queue, as it is the caller's responsibility to make sure completions are handled with `complete_io()`.

As for the non-zero-copy methods, we pass `[u8]` slices, which are copied chunk-wise to the DMA buffer in `NvmeDevice` for writes or copied from for reads. For these methods, we can iterate over the data buffer in larger chunks, more specifically 512 KiB chunks, as we have already constructed a PRP list for the DMA buffer in the initialisation of `NvmeDevice`. The batched read and write methods split the requests into smaller chunks and submit these in batches, taking advantage of the parallel nature of NVMe SSDs.

Aside from the data buffer, the user also passed the logical block address (LBA) for the I/O operation. The length of data written is derived from the size of the buffer.

5 Evaluation

We need to benchmark its performance to evaluate whether vroom loses out on performance due to a more straightforward design and fewer optimisations. In this chapter, we analyse vroom's performance, i.e. throughput and latency, while comparing it to other I/O engines.

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, which has a 1 GB cache; the throughput and bandwidth limits of the SSD are noted in Table 5.1.

In the following sections we will compare our driver's performance with the datasheet numbers in Table 5.1, as well as against other storage engines: `libaio`, `io_uring`, SPDK and the Linux file I/O API `pread/pwrite` (`psync`). Each storage engine is tested by running a read or write workload over 900 seconds for single-threaded I/O and 60 seconds for multithreaded I/O, with I/O unit sizes of 4 KiB. We tested random I/O instead of sequential I/O, as achieving good random read and write performance is generally more challenging.

The NVMe controller is aware when a device is empty and thus processes read requests without performing any read operations. On an empty drive, the reported read performance will be much higher. Hence, all read tests are done on a full drive, i.e. each LBA has been written to at least once.

Mode	Configuration	Throughput
Sequential Read	-	3500 MB/s
Sequential Write	-	3300 MB/s
Random Read	QD 1 Thread 1	19 KIOPS
	QD 32 Thread 4	600 KIOPS
Random Write	QD 1 Thread 1	60 KIOPS
	QD 32 Thread 4	550 KIOPS

Table 5.1: Samsung Evo 970 Plus performance limits as per the datasheet [19]

Listing 5.1: `fio` configuration

```
[global]
io_engine={spdk,io_uring,libaio,psync}
rw={randread,randwrite,read,write}
blocksize=4k
direct=1
norandommap=1
runtime=900

numjobs={1,4}
queue_depth={1,32}
group_reporting=1
```

For writes, the SSD is cleared beforehand, such that it is in a comparable state for each test. Full NVMe SSDs report worsened write performances due to wear-levelling and write amplification, where the NVMe controller performs garbage collection and internally reorders data.

We use the Flexible I/O tester `fio`[1] to test the performance capabilities of `libaio`, `io_uring` and `psync`, and in some cases SPDK. In Figure 5.4, Figure 5.5 we use numbers from SPDK's own `spdk_perf_tool` tool, as their `fio` plugin introduces some overhead; for Figure 5.6, Figure 5.7, Figure 5.8 we use log files `fio` creates.

We use the `fio` configuration in Listing 5.1, with the Linux storage engines performing I/O to the NVMe as a block device. The same parameters were used when testing with `spdk_perf_tool`.

Our driver, vroom, is tested using our own time-based workload simulating the same workloads started by `fio`, performing reads and writes to random block-aligned LBAs.

5.2 Throughput

In this section, we will analyse the throughput capabilities of our NVMe driver and how changing parameters affect performance. Afterwards, we will compare vroom's throughputs with those of the other I/O engines.

Observing the throughput trend over time in Figure 5.1, we see the write throughput begin at a heightened rate, at around 800 thousand IOPS (KIOPS) and after approximately 40 seconds, decreasing to approximately 200 KIOPS. The SSD has a so-called

[1] `https://github.com/axboe/fio`

"TurboWrite" buffer region of 42 GB, which allows for faster writes by simulating a Single Level Cell NAND [18]. This allows for lower write latencies, so heightened write throughputs, as long the buffer is not fully saturated. Samsung states in their data sheet that random writes after this have a throughput of 300 KIOPS [19]. Averaged over 900 seconds, we achieve 255 KIOPS when writing to an empty drive. On the other hand, read throughput stays relatively constant throughout the entire test, at around 440 KIOPS, 160 KIOPS below the datasheet value.

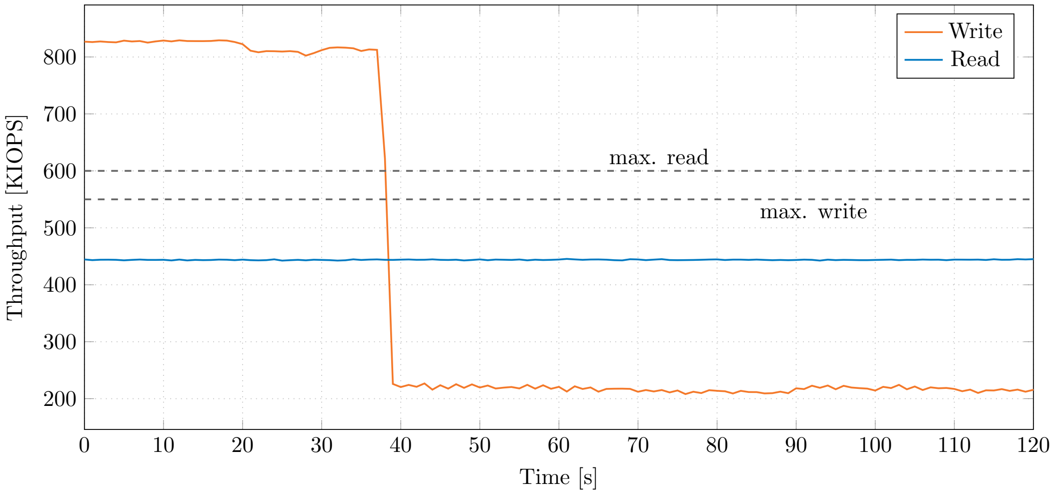
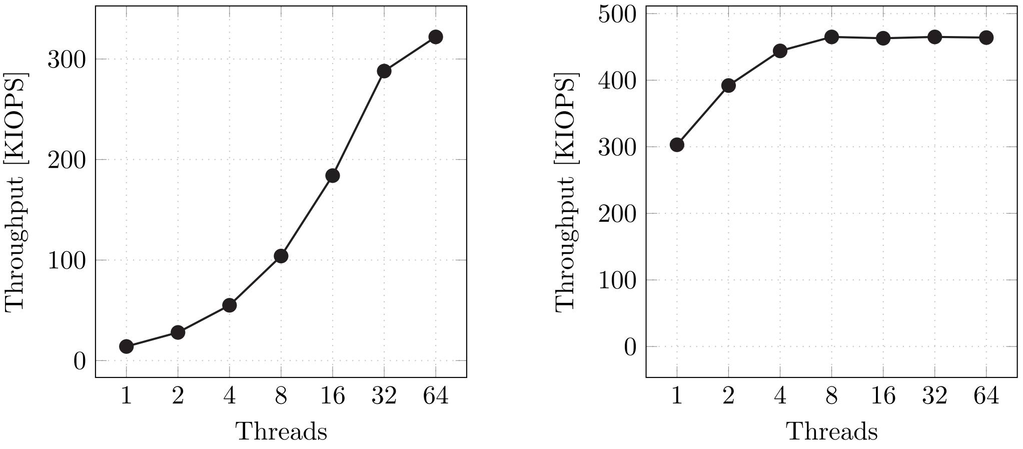


Figure 5.1: vroom's random read and write throughputs over time, QD32T4

To investigate the effects the number of threads and queue depth (QD) have on throughput, we performed multiple tests with different parameters on the read throughput rather than write to minimise variance.

Observing how the number of threads impacts the throughput in Figure 5.2, we see the throughput double each time we double the number of threads until 16 threads; afterwards, we see gains become more minor. With a queue depth of 32, we see an increase in throughput until 8 threads, after which the throughput plateaus at around 460 KIOPS. As our CPU has 40 hardware threads, we would expect diminishing returns or even regression by increasing the number of threads beyond 40.



(a) Queue depth 1

(b) Queue depth 32

Figure 5.2: Threads vs. IOPS; vroom random read

When increasing the queue depth, we can notice a similar growth where the IOPS doubles each time the queue depth doubles in size until QD16 in Figure 5.3; at QD64, the throughput is higher than at 64 threads with QD1 by approximately 80 KIOPS. By increasing queue depth, we achieve higher throughput than increasing the amount of threads when using a queue depth of 32. Here, we reach a plateau with a queue depth of 256 with a throughput of around 480 KIOPS. With this Samsung SSD, there is no need to implement queues deeper than 256. Increasing queue depth leads to better performance than increasing the threads when comparing “queue depths”, i.e. QD32T4 ($\approx$ QD128= $32 \cdot 4$)) has a lower throughput than QD128T1.

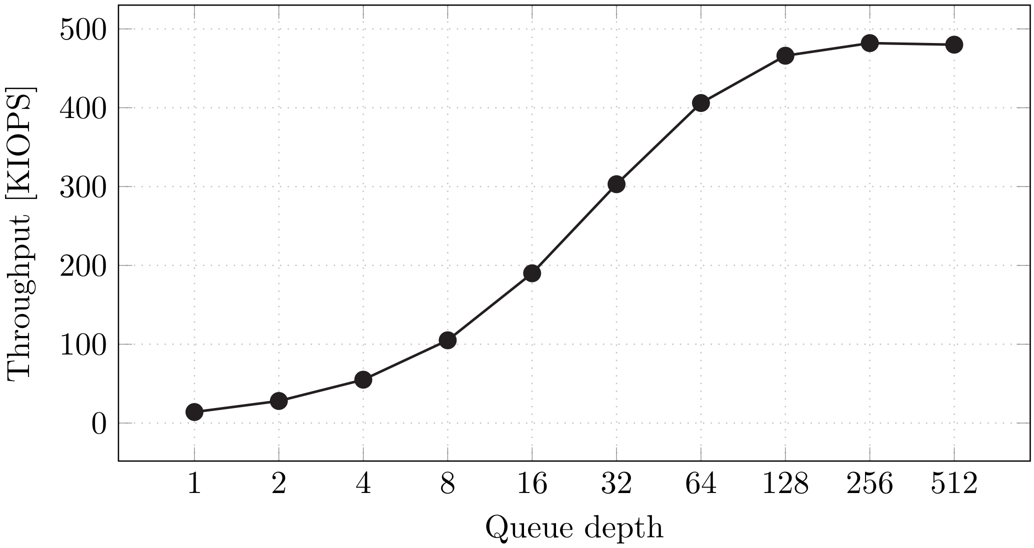


Figure 5.3: Queue depth vs. IOPS; vroom random read

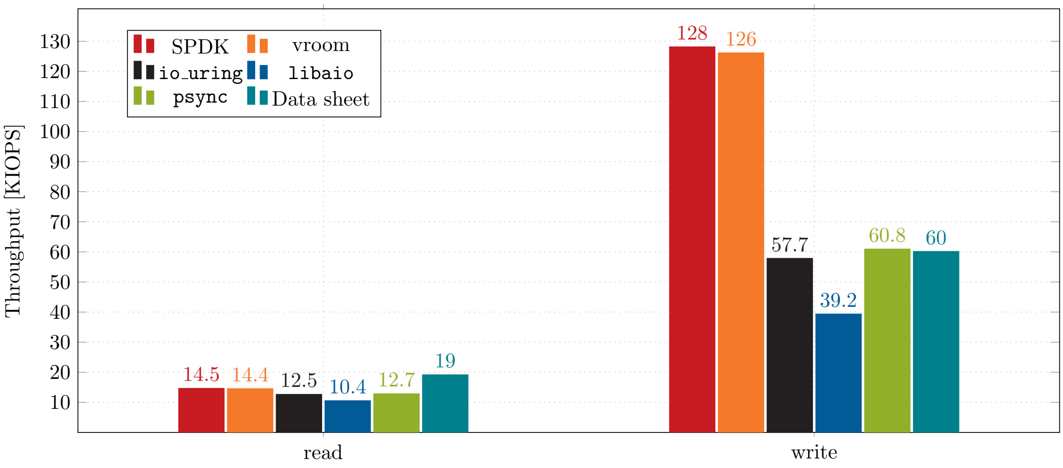


Figure 5.4: Random read (900 s) and write (900 s) throughput; Queue depth 1

Comparing the throughput of I/O engines in terms of synchronous I/O, i.e. single-threaded with a queue depth of 1, we see the user space drivers performing above the kernel-based APIs. Looking at read throughput in Figure 5.4, none of the I/O engines

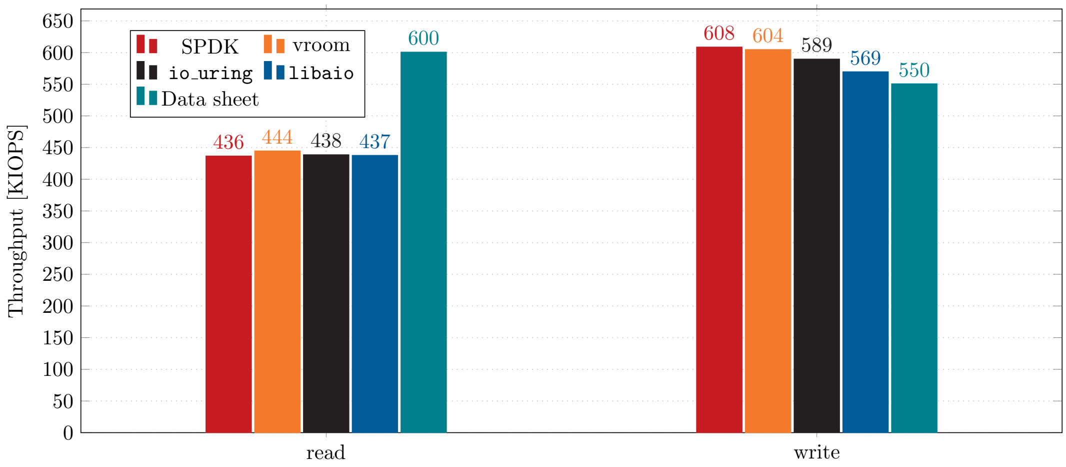


Figure 5.5: Random read (60 s) and write (60 s) throughput; Queue depth 32, 4 Threads

come within 20% of the maximum achievable throughput of the SSD, performing between 10 KIOPS and 15 KIOPS, with SPDK and vroom performing comparably at 14.5 KIOPS and 14.4 KIOPS, respectively. In terms of write performance, we see `psync` and `io_uring` right around 60 KIOPS, while `libaio` performs considerably below the limit of the SSD. Both SPDK and vroom achieve IOPS numbers doubling the limit stated by Samsung, at 128 KIOPS and 126 KIOPS, respectively.

As Samsung has not specified the test parameters and environment used, likely they did not perform these read tests on a fully utilised drive or write tests on an empty one. This means they would likely expect higher IOPS than we observe for reads, as the NVMe controller would access unwritten areas, increasing the overall IOPS number. For writes, the NVMe controller performs garbage collection when overwriting non-empty areas, introducing some overhead and, thus, resulting in fewer IOPS overall.

As NVMe SSDs are able to process a multitude of requests in parallel, we also analyse multithreaded read and write performances; specifically, we used 4 threads, each with a queue depth of 32, without explicitly batching requests. `psync`, as it only allows synchronous I/O, cannot be tested with these parameters.

Like in the previous test, all storage engines perform pretty closely in terms of read as we see in Figure 5.5, here all within 1% of one another, with vroom being the most performant at 444 KIOPS; however none come close to the QD32T4 600 KIOPS limit in the datasheet. Interestingly, once we introduce deeper queues and multithreading, the system call overhead does not affect `libaio` and `io_uring` to the same degree as for synchronous I/O when reading.

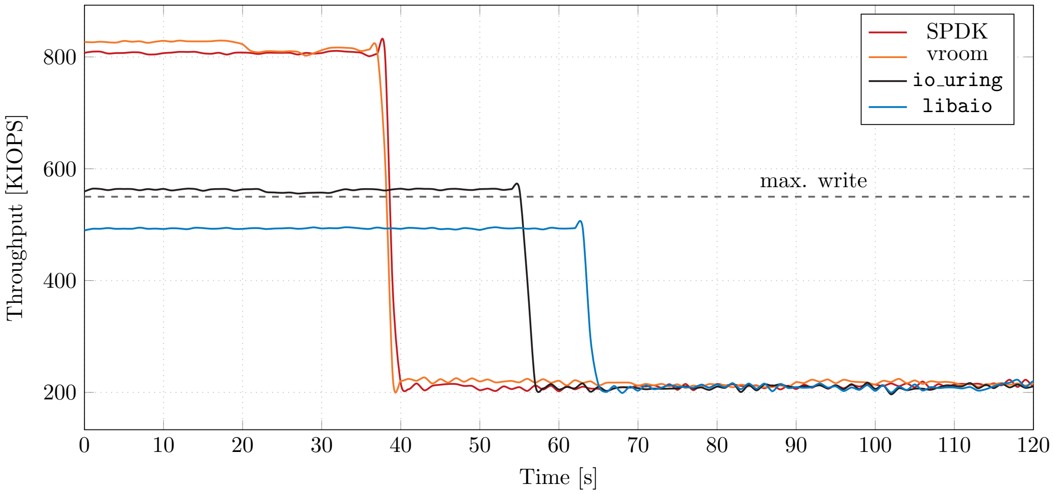


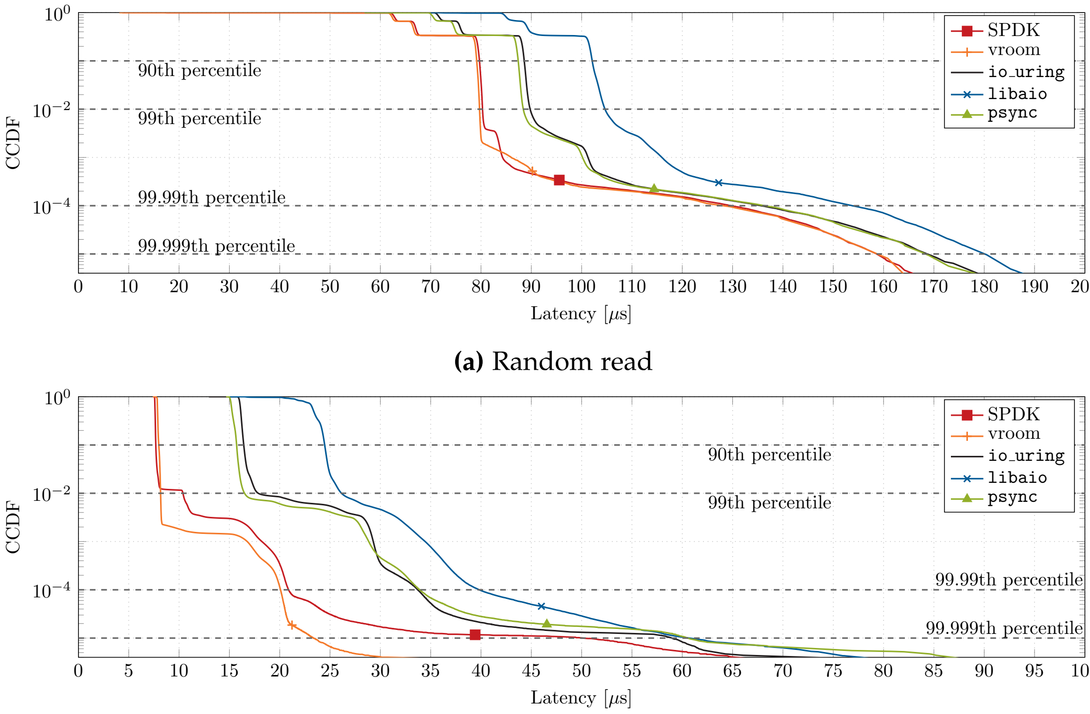
Figure 5.6: QD32T4 random write throughput over time

Over 900 seconds, all I/O engines achieve similar throughputs on average. As observed in Figure 5.1, the overall write throughput decreases immensely once the SSD's write buffer is saturated. Over longer write workloads, the performance disparities become negligible. Due to this, we look at the average write throughput over 60 s rather than 900 s. Averaged over 60 s, all I/O engines surpass the 550 KIOPS "ceiling", with the user space drivers performing marginally better than the kernel bound storage engines. Similarly to Figure 5.4, `libaio` achieves the lowest throughput for writes.

When we look at how the throughput changes over time in Figure 5.6, similar to the observation of Figure 5.5, we see SPDK and vroom both ahead in terms of peak throughput, achieving around 800 KIOPS, while `io_uring` and `libaio` have a throughput of approximately 570 KIOPS and 500 KIOPS, respectively. Here, it is clear that once the "TurboWrite" buffer is fully saturated, the SSD becomes the bottleneck, where each I/O engine performs nearly identically with a throughput of around 200 KIOPS. Also note that Figure 5.6 uses IOPS logs from `fio`, so realistically, the throughput of SPDK would likely be higher than vroom initially.

5.3 Latency

As expected, the user space drivers achieve higher throughput numbers than the rest. SPDK and our driver achieve IOPS numbers within 1% of one another while being noticeably more performant than the Linux I/O APIs, especially when writing. System



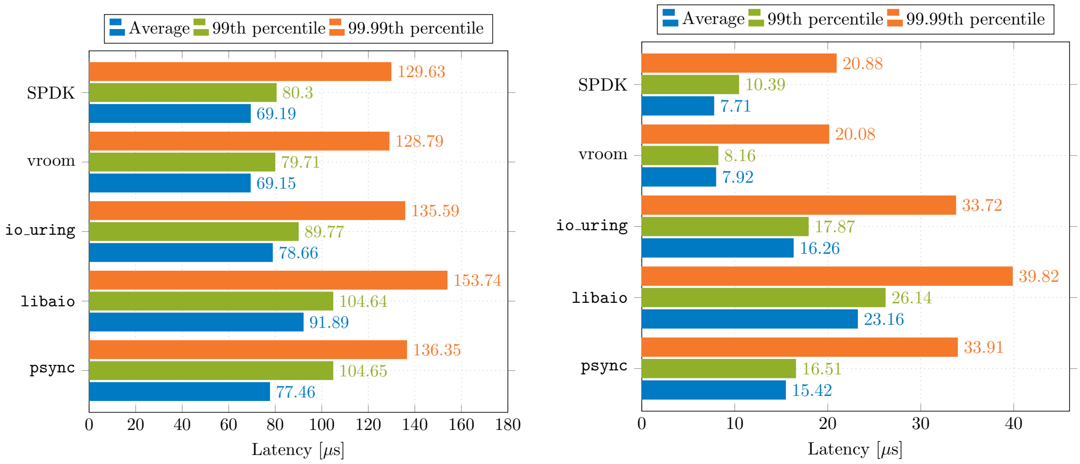
(a) Random read

(b) Random write

Figure 5.7: Tail latencies

calls, and the consequent context switches cause this difference. This is especially visible in Figure 5.7, where the tail distribution of the I/O latencies is plotted; here, all I/O engines share a similar distribution, however, offset by a certain amount. The system call overhead from `psync` and `io_uring` introduces around 10 µs over SPDK and vroom, while `libaio` seemingly has some extra internal overhead adding in total 20 µs over vroom and SPDK.

Figure 5.8 confirms our takeaways from the observations of Figure 5.7: we also see vroom and SPDK achieving nearly identical latency values. `io_uring` and `psync` are, on average, slower by 9 µs and 8 µs, respectively, while `libaio` trails `io_uring` by approximately 13 µs for reads and 7 µs for writes.



(a) Random read

(b) Random write

Figure 5.8: Latencies

6 Conclusion

In this thesis, we have presented vroom, a novel user space and poll-based NVMe driver written in Rust. Our evaluations have shown here that developing a user space NVMe driver in a higher-level programming language with SPDK-like performance is feasible while simultaneously containing fewer lines of driver code overall. We reach comparable throughput speeds as SPDK while outperforming the Linux kernel I/O APIs by avoiding the kernel altogether.

Future Work There are still many aspects in the driver where optimisations can be investigated, such as the effect of prefetching pages for I/O operations or implementing IOMMU support, so root privileges are not required to start the driver and measure the performance impact it may have. Investigating the performance impacts of interrupts compared to polling and comparing different interrupt methods are also open topics.

As vroom does not support all NVMe features, we can still add many features to the driver, like supporting SGLs and comparing them to PRP lists or using the NVMe controller's onboard memory buffer.

With `libaio` not being actively worked on, the last commit being made two years ago [10], and `io_uring` exhibiting enough security concerns that Google disabled the use of the storage engine on all production servers [6], and the containerd runtime disabling `io_uring` system calls [8] altogether, there lacks an I/O engine which is safe, performant and simple to use at the same time; extending vroom by a block device and filesystem layer could serve to fill this gap.

List of Figures

List of Tables

Listings

Bibliography

[1] C. Cutler, M. F. Kaashoek, and R. T. Morris. "The benefits and costs of writing a POSIX kernel in a high-level language." In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, 89–105. ISBN: 9781931971478.

[2] D. Didona, J. Pfefferle, N. Ioannou, B. Metzler, and A. Trivedi. "Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring." In: *Proceedings of the 15th ACM International Conference on Systems and Storage*. SYSTOR '22. Association for Computing Machinery, 2022, 120—127. ISBN: 9781450393805. DOI: `10.1145/3534056.3534945`.

[3] S. Ellmann. *ixy.rs source code*. 2018. URL: `https://github.com/ixy-languages/ixy.rs` (visited on 04/10/2024).

[4] P. Emmerich, S. Ellmann, F. Bonk, A. Egger, E. G. Sánchez-Torija, T. Günzel, S. di Luzio, A. Obada, M. Stadlmeier, S. Voit, and G. Carle. "The Case for Writing Network Drivers in High-Level Programming Languages." In: *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2019, pp. 1–13. DOI: `10.1109/ANCS.2019.8901892`.

[5] P. Emmerich, M. Pudelko, S. Bauer, and G. Carle. "User Space Network Drivers." In: *Proceedings of the Applied Networking Research Workshop*. ANRW '18. Montreal, QC, Canada: Association for Computing Machinery, 2018, 91–93. ISBN: 9781450355858. DOI: `10.1145/3232755.3232767`.

[6] Google. *Learnings from kCTF VRP's 42 Linux kernel exploits submissions*. 2023. URL: `https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html` (visited on 04/12/2024).

[7] Google. *Secure by Design: Google's Perspective on Memory Safety*. 2024. URL: `https://security.googleblog.com/2024/03/secure-by-design-googles-perspective-on.html` (visited on 04/10/2024).

[8] V. Goyal. *containerd commit a48ddf4*. 2023. URL: `https://github.com/containerd/containerd/commit/a48ddf4a208b24eadea82f0eac62e236f2acf004` (visited on 04/12/2024).

[9] G. Haas, M. Haubenschild, and V. Leis. "Exploiting Directly-Attached NVMe Arrays in DBMS." In: *CIDR*. 2020.

[10] *libaio source code*. URL: `https://pagure.io/libaio` (visited on 04/12/2024).

[11] Linux Kernel documentation. *VFIO - "Virtual Function I/O"*.

[12] MITRE Corporation. *2023 CWE Top 10 KEV Weaknesses*. 2024. URL: `https://cwe.mitre.org/top25/archive/2023/2023_kev_list.html` (visited on 04/10/2024).

[13] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev. "RedLeaf: Isolation and Communication in a Safe Operating System." In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 21–39. ISBN: 978-1-939133-19-9. URL: `https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram`.

[14] NVM Express, Inc. *NVM Express Base Specification Rev. 1.4*. 2019. URL: `https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf` (visited on 04/10/2024).

[15] PCI-SIG. *PCI Express Base Specification Rev. 5.0, Version 1.0*.

[16] *RedLeaf*. URL: `https://mars-research.github.io/projects/redleaf/` (visited on 04/03/2024).

[17] *Redox*. URL: `https://redox-os.org/` (visited on 04/03/2024).

[18] Samsung Electronics Co., Ltd. *Samsung Solid State Drive TurboWrite Technology*. 2013. URL: `https://images-eu.ssl-images-amazon.com/images/I/914ckzwNMpS.pdf` (visited on 04/10/2024).

[19] Samsung Electronics Co., Ltd. *Samsung V-NAND SSD 970 EVO Plus Data Sheet*. 2021. URL: `https://download.semiconductor.samsung.com/resources/data-sheet/Samsung_NVMe_SSD_970_EVO_Plus_Data_Sheet_Rev.3.0_10129514059241.pdf` (visited on 04/10/2024).

[20] Storage Performance Development Kit. *Direct Memory Access (DMA) From User Space*. URL: `https://spdk.io/doc/memory.html` (visited on 04/11/2024).