Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Exploring the Performance of the io_uring Kernel I/O Interface

Author: Brynjar Ingimarsson (2721481)

1st supervisor: Animesh Trivedi
2nd reader: Tiziano De Matteis

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

June 25, 2024

"If debugging is the process of removing bugs, programming must be the process of putting them in"
by Edsger W. Dijkstra

Abstract

The demand for faster and more capable computer systems grows every year, as their role in society becomes ever more important. In particular, large-scale applications are increasingly being deployed in cloud environments due to their flexibility and scalability. Most of these applications are data-intensive, and thus storage devices are an important aspect of cloud computing. The cloud computing market size was valued at $590 billion in 2023, and the amount of data created has grown from 2 zettabytes in 2010 to 120 zettabytes in 2023.

The advent of solid-state storage devices (SSDs) has sparked a revolution in high-performance storage. A modern SSD can provide millions of IOPS in throughput and single-digit microsecond latencies. The introduction of SSDs has exposed various overheads and limitations of existing software stacks, leading to new designs for operating systems, file systems, and database systems. In addition, more efficient storage interfaces have been introduced for both hardware and software. To enable high-performance storage, Linux has introduced `io_uring`, a new asynchronous I/O interface for user applications.

In this thesis, we make a systematic study of the `io_uring` interface. The interface offers many configuration options that affect how I/O is submitted and completed. The effect of these options is largely ignored in previous studies, and existing documentation provides limited details. In this thesis, we study `io_uring` configuration options in detail and provide guidelines for application developers. We study several `io_uring` options with microbenchmarks and provide a detailed understanding of their internal workings. Finally, we show that our guidelines can improve the performance of RocksDB, a popular key-value store.

The artifacts of this thesis are available online on GitHub at `https://github.com/Ingimarsson/iouring-perf-analysis`.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

In today's world, computer systems are an integral part of society. In particular, cloud computing has grown exponentially in the last decade, as businesses increasingly rely on computer systems to make their operations more efficient. The cloud computing market size was estimated at $590 billion in 2023 and is predicted to grow (1). The amount of data created in the world is also growing exponentially, and one study estimates that 120 zettabytes of data were created in 2023, compared to just 2 zettabytes in 2010 (2). This rapid growth constantly calls for more efficient hardware and software.

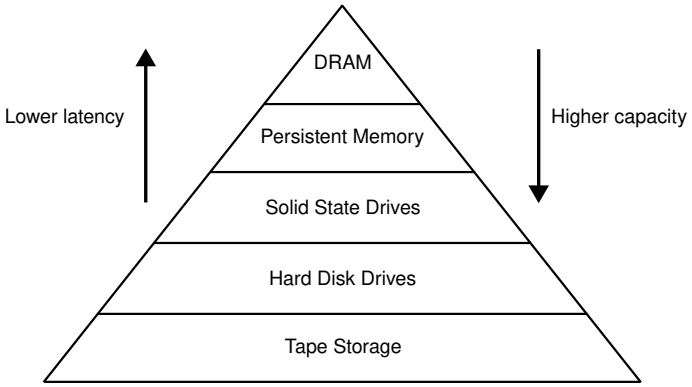


Figure 1.1: Modern memory hierarchy, showing the relation between latency and capacity

An important part of the digital world is persistent storage, as the data created must be stored somewhere. For example, individuals often connect their devices to cloud providers, so that their photos and documents are synced between devices and backed up in case of data loss. Figure 1.1 shows the different levels of the memory hierarchy, with persistent storage in the lower half. It depends on the application which type of persistent storage is most suitable. High-performance applications may require solid-state drives, while hard

disk drives may be more cost-efficient for others. For archival storage, a tape drive may offer the most benefits (3).

In the 2010s, it became more economical to produce large storage devices with flash memory chips, as opposed to hard disk drives based on spinning platters. The introduction of these devices, known as solid-state drives (SSDs) has brought rapid advancements in storage, including new database designs, operating systems, file systems, and interfaces on both the hardware side and software side. Table 1.1 compares two modern SSDs and HDDs. While the HDD is more economical for capacity, the SSD provides an order of magnitude higher throughput (IOPS) and five orders of magnitude lower latencies. The SSD also provides much better throughput for random access, while the HDD performs much better with sequential access (4).

	Seagate Exos X10 (5)	Intel Optane P5800X (6)
Capacity (GB)	10,000	800
Sequential read (IOPS)	62,500	1,800,000
Random read (IOPS)	170	1,500,000
Average latency (μs)	4,600.0	1.5

Table 1.1: Comparison of a Seagate Exos X10 (HDD) and Intel Optane P5800X (SSD)

1.1 Context

A storage stack refers to all the components involved in storing data, from the hardware device to the user application. Figure 1.2 shows a typical storage stack that starts at the SSD **(1)**, where multiple flash chips are connected to a controller, which exposes a block interface to the host. A block interface is an I/O interface where data is read or written in blocks of fixed size, often 512 or 4096 bytes, or a multiple of that. The operating system must include a driver **(2)** for this interface, in addition to an optional I/O scheduler that reorders requests for an optimal operation **(3)**. However, applications do not typically submit block requests, but rather interact with files that belong to file systems. A file system **(4)** maps file offsets to blocks on the device, and allocates blocks when files grow. Above the file system, data may be cached in the OS page cache **(5)**, so that frequent access to the same data is quickly served. A user application **(6)** uses generic system calls to interact with files, abstracting away all the details of the storage stack (7).

On a Linux system, disk I/O is traditionally done by blocking system calls such as `open`, `read`, and `write`. These calls block until the operation has been completed, even though

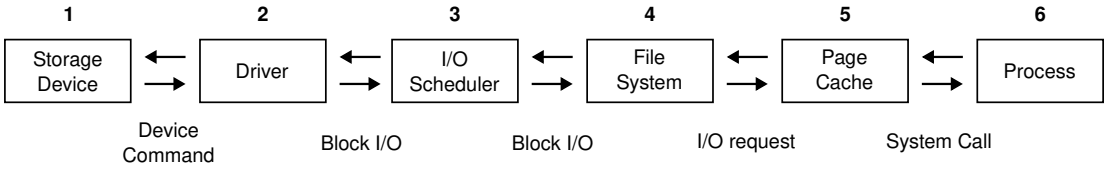


Figure 1.2: The main components in a typical OS storage stack

most of the duration was spent waiting for the device, and that duration could have been spent on other work. There exist several variants of the traditional I/O system calls, for example, `readv` and `writev`, which are vectored operations that take in a vector of offsets and lengths to operate on, and `pread` and `pwrite`, which are POSIX-compliant versions of the operations. For some applications, it may be beneficial to run some other tasks while an I/O request is in flight, which is possible with an asynchronous interface. Figure 1.3 shows the difference between a synchronous and an asynchronous request. Note that although no other work can be done by the process during synchronous I/O, the OS can schedule other processes in the meantime to make use of the available CPU time.

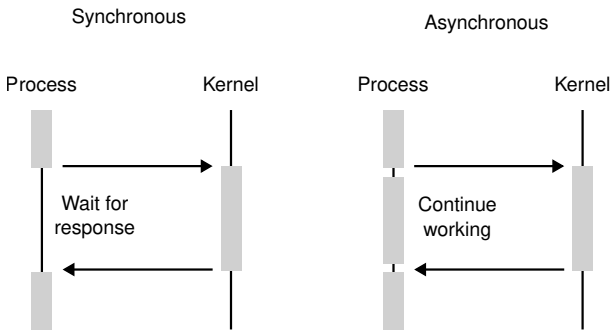


Figure 1.3: Comparison of a synchronous and asynchronous system call

On Linux, synchronous operations were the only option for a long time. Application developers often introduced I/O worker threads to run I/O requests on another thread, so that the main thread would not block during I/O. To offer asynchronous I/O, the `aio` interface was introduced in the Linux kernel. However, it has many limitations, such as not supporting non-direct operations that go to the page cache (8). In 2019, the `io_uring` interface was introduced to Linux, which offers an improved asynchronous I/O interface to applications.

The `io_uring` interface is based on ring buffers (queues), which allow for lock-free and zero-copy operation. To use `io_uring`, an application first creates an instance of `io_uring`,

also known as a ring. The application then maps two ring buffers into memory, which are used for a submission queue (SQ) and completion queue (CQ). To submit I/O, the application places a request in the submission queue and tells the kernel that there are new requests to submit. The application can then move on to other work, and it can poll the completion queue to see if a corresponding response has arrived. The interface also supports two polling modes that can prevent a high number of system calls and hardware interrupts, in addition to many other configuration options that affect how I/O is submitted and completed.

1.2 Problem Statement

Several studies have explored the `io_uring` interface and demonstrated improved performance compared to other I/O interfaces (8, 9). However, we find that the performance effect of most `io_uring` configuration options has not been well studied, and existing documentation only provides vague hints about when to use them. Thus, the goals that we want to achieve with this thesis are the following.

- Provide guidelines for application developers about what `io_uring` configuration can be beneficial under what circumstances, and what configuration can harm performance
- Provide a detailed explanation of the internals of each configuration option, to motivate what factors can affect their performance
- Confirm that our guidelines are beneficial by applying them to an existing real-world data-intensive application

The man page of `io_uring` lists 17 configuration options (10), in addition to 7 flags that can be set on submission events (11). As each option requires significant time to study, we select only a few options that we categorize as likely to affect performance. The documentation of many flags indicates that they may not be relevant to performance, as we will explain in chapter 3.

1.3 Research Questions

To solve the problems mentioned previously, we present the following research questions to help us understand the problem.

- **RQ1** - What methods are available for evaluating and measuring `io_uring` performance under different configurations?
- **RQ2** - What `io_uring` configuration options can affect the throughput and latency of application I/O and under what circumstances?

1.4 Research Methodology

To answer our research questions, we make use of the following methodology.

- **M1** - Microbenchmarking: we run small benchmarks that perform a simple task such as random reads, for this we use the `fio` storage benchmarking tool, with our custom `liburing` engine
- **M2** - Profiling: we collect various data during the run of our experiments to gain insights into internal behavior, for this we use `eBPF` programming
- **M3** - Real-world benchmarking: we implement a storage engine for a real-world key-value store based on our results and measure the effect on performance metrics

1.5 Contributions

The contributions of this thesis are threefold. First, we provide four guidelines about how the configuration of `io_uring` can significantly affect performance. Second, we provide detailed explanations of how the Linux kernel implements `io_uring` configuration options internally, which further supports our guidelines. Finally, we show that the `io_uring` support in RocksDB, a popular key-value store, can provide 12% higher throughput and 13% lower latency after applying our guidelines. To the best of our knowledge, these contributions have not appeared in publicly available work before.

1.6 Societal Relevance

As the cloud computing industry keeps growing, so does its carbon footprint. In 2025, it is predicted that the cloud industry will use 20% of global electricity, and produce 5.5% of global carbon emissions (12). In this thesis, we show that selecting the appropriate configuration for `io_uring` can result in better performance. This means using computing resources more efficiently, and thus allowing cloud operators to run less hardware. In

addition, improving I/O performance can result in more efficient operation of businesses, allowing them to grow faster.

1.7 Thesis Structure

In chapter 2, we provide background on storage systems, the Linux storage stack, the `io_uring` interface, and kernel concepts that we will come across in our exploration. In chapter 3 we explain how we choose configuration options to evaluate, how we use `fio` to run microbenchmarks, and how we implement a custom `liburing` engine. In chapter 4 we run detailed microbenchmarks for each configuration option, in addition to various kernel profiling, and exploring the kernel internals of each feature. In chapter 5 we implement a `liburing` backend for RocksDB and evaluate its performance. In chapter 6 we describe several areas of research that relate to our work. Finally, in chapter 7 we conclude our thesis by listing our guidelines and answering the research questions. In chapter 8 we describe our artifacts and how to reproduce our experiments, and explain how to program with `liburing` and `eBPF`.

1.8 Plagiarism Declaration

I confirm that this thesis is my own work, and is not copied from any other source (e.g. a person, the internet, or a machine) unless explicitly stated. This work has not been submitted for assessment anywhere else. I acknowledge that plagiarism is a serious academic offense that should be dealt with if found.

2

Background

In this section, we discuss the necessary background on storage and systems topics that this thesis builds on. First, we discuss flash storage, its history, internal architecture, performance characteristics, and host interfaces. We then discuss Linux and in particular, its block layer, including the main components that are involved with I/O requests, and we also give a detailed explanation of interrupts and context switches, as these concepts occur frequently in this thesis. We then discuss asynchronous I/O and the history of asynchronous interfaces. Finally, we introduce `io_uring`, its main design features, and usage from the viewpoint of a user process.

2.1 Flash Storage

Flash storage is a popular type of NVM (Non-Volatile Memory), a class of semiconductor-based memory that persists data after losing power, and involves no moving mechanical parts. Flash memory has been available since the 1980s, and was previously mainly found in embedded systems, as it could not compete with mechanical disks (HDDs) in terms of capacity and price until recently. An SSD (Solid State Drive) consists of one or more flash chips that are connected to an FTL (Flash Translation Layer), that exposes a block interface to the host (4).

Inside an SSD, flash chips are arranged into channels, and within each flash chip, memory cells are divided into dies, planes, blocks, and pages. Read operations can read any page, but write operations can only write to pages in sequential order within blocks. The erase operation is used to reset all memory cells in an entire block. This means that flash chips do not support random access with respect to write operations. Pages are typically 4 kB

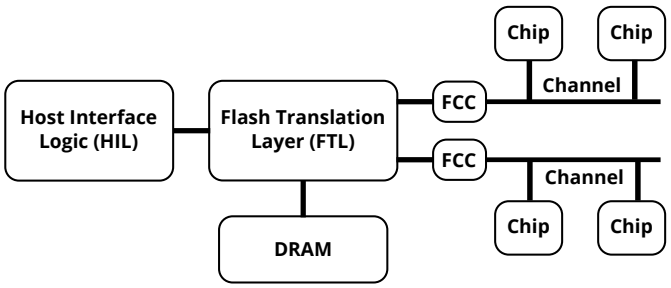


Figure 2.1: Internal structure of an SSD with flash chips on multiple channels

in size, and a typical block might contain 32-128 pages. Figure 2.1 shows the internal structure of an SSD.

An operation on an SSD can be seen as three separate steps.

1. **Controller** - The controller translates a request from the host into a flash chip operation.

2. **Channel Bus** - The flash operation and data are transferred over a channel bus into control and data registers on a flash chip.

3. **Flash Chip** - The flash chip executes an operation based on the contents of its registers.

Each plane in a flash chip usually has its own control and data registers, and the flash chip can execute multiple operations on different planes independently. As the transfer time over a channel bus may be shorter than the flash operation time, multiple operations can be interleaved. Because of this, most SSDs need to have multiple requests in flight to fully saturate their performance, i.e. SSDs have internal parallelism. A typical NAND flash chip might offer 25 μs reads, 200 μs writes, 1.5 ms erases and 100,000 erase cycles (4). Because of the speed of flash chips and internal parallelism, SSDs can provide millions of IOPS in throughput and single-digit microsecond latencies. This has exposed bottlenecks in software and driven the need for lightweight I/O interfaces such as `io_uring`.

The limitation of only being able to write to erased pages results in what is known as the semantic gap of flash storage. The host expects a generic block interface where any page can be read or written, but internally the writes must happen sequentially within blocks. The FTL is responsible for mapping logical block addresses (LBAs) from the host into physical block addresses on the flash chips. On a write request, the FTL will find a

free page on a flash chip, and update its mapping table. This means that if the LBA was previously written, that data is still taking space on a flash page, but there is no mapping to it anymore, making it a zombie page. To reclaim zombie pages, the FTL runs a garbage collector that merges and erases multiple blocks, leaving only active pages. The FTL also distributes flash operations to improve parallelism, and ensures equal wear on the chips, due to the limited erase cycles (4, 13).

Early SSDs came with a SATA or SCSI host interface. However, as SSD throughput and latency improved, these interfaces became insufficient, as they involve an HBA (Host Bus Adapter) between devices and the host, with expensive translations between interfaces. The NVMe interface is designed for high-performance SSDs, and connects directly to a host PCIe bus. It is also better designed for parallelism, and supports up to 64k queue pairs, where each queue can have a depth of up to 64k (13).

2.2 Linux Kernel

Operating systems typically provide extensive facilities for disk I/O, including drivers for the hardware device, a block layer with queueing and scheduling of block requests, file systems, caching layers, and a virtual file system. These facilities allow application developers to work with the abstraction of files, which are generic containers of binary data identified by a filename. This makes programs highly portable, and leaves complex tasks to the OS, such as crash recovery and sharing devices between processes. In this section, we explain the components involved in disk I/O on Linux, along with interrupts and context switches, which appear frequently in this thesis.

2.2.1 Linux Storage Stack

Applications typically perform disk I/O with system calls such as `open`, `read`, and `write`. These calls interact with files in the VFS (Virtual File System), which combines multiple physical file systems into a single interface. A file on any file system is identified by a unique path in the VFS. A physical file system may store data on a block device (e.g. `ext4`, `xfs`), connect to a remote system over a network (e.g. `nfs`, `samba`), or provide access to special system files (e.g. `sysfs`, `proc`). File systems can also make use of the OS page cache, to cache frequently accessed data.

Below file systems is the block layer, which allows sending generic block requests, known as `struct bio` in the Linux kernel, to block devices. For each block device, the kernel sets up a multi-queue block layer, also known as `blk-mq`. A block request first goes into a

software staging queue, where a scheduler may reorder requests for better performance or QoS guarantees. The block request is then moved to a hardware dispatch queue, where the device driver can consume requests from. There is typically one staging queue per core, to prevent expensive lock operations across cores. There may also be a dispatch queue per core if the device supports enough hardware queues, otherwise, dispatch queues will be shared by two or more staging queues. The staging and dispatch queue split also allows for less locking overhead (7).

The Linux kernel provides several I/O schedulers that operate on the per-core staging queues in the block layer. Storage devices have various performance characteristics, and it is possible that when multiple applications submit I/O, the interleaved sequence of operations results in poor performance, or applications receive unequal shares of performance. For example, on SSDs, mixing reads and writes can severely hurt the latency of reads. The `kyber` I/O scheduler aims to throttle writes in order to guarantee better read latencies. Applications are typically more sensitive to read latencies, as data may be needed to perform the next step of computation, while write requests typically don't stop an application from continuing (9).

As we have previously described, SSDs can provide millions of IOPS in throughput and single-digit microsecond latencies. This has exposed various overheads and bottlenecks in the Linux kernel block layer. Before the introduction of the multi-queue block layer, the Linux kernel was unable to fully saturate some high-performance storage devices (14). In addition, SSDs have necessitated the development of high-performance I/O interfaces such as `io_uring`.

2.2.2 Context Switches

On Linux, every process has its own context, which includes a private virtual memory space, and the values of registers. When an application makes a system call, most of the context remains, but execution switches into kernel mode, a special per-process kernel stack is used, and the kernel's address space becomes accessible. If the system call is blocking, e.g. a synchronous I/O operation, or a sleep system call, then the scheduler will be invoked to switch to a different process. In that case, the context of the application is saved, and the new process context is loaded. The jump from user to kernel mode is typically known as a mode switch, and a jump between processes is known as a context switch, but sometimes both are referred to as context switches. Context switches are expensive, as they involve saving register state, a TLB (Translation Lookaside Buffer) flush, along with other CPU overhead (7).

2.2.3 Interrupts

Interrupts are events that can be generated by hardware devices or the CPU itself, and force the CPU into executing different code, known as interrupt handlers. In multi-core x86 systems, each CPU core has a local APIC (Advanced Programmable Interrupt Controller) that manages interrupts. The kernel must register interrupt handlers for each type of interrupt, identified by an interrupt vector. To balance the work of interrupts between CPU cores, the kernel runs a process called `irqbalance`. The `irqbalance` kernel process periodically checks the number of interrupts that have occurred per core, and moves them between cores to balance the load of each core. When an interrupt comes in, the CPU jumps to kernel mode with a mode switch. A mode switch results in a performance overhead as we described in the previous subsection. For disk I/O, there is typically one interrupt generated on the completion of each request, letting the host know that the result is available, that can be passed to a process. For SSDs that support millions of IOPS in throughput, the amount of interrupts can result in a significant overhead. Another type of interrupts are IPIs (Inter-Processor Interrupts), which are used by the Linux kernel to communicate across CPU cores, for example, to deliver a signal to a process, or to invoke the process scheduler on a different core (7).

2.3 Asynchronous I/O Interfaces

The standard POSIX I/O system calls such as `read` and `write` are both synchronous and blocking. The term synchronous means that a call to the function does not return until the operation is completed. For disk I/O, this means that when the operation is called, a request is sent to the disk, the disk performs the operation, a response is sent to the host, and only then does the system call return, with the result available. A synchronous system call can be either blocking or non-blocking, where non-blocking means that the system call will fail fast, e.g. if the file is locked, a "try again later" error code is returned, rather than waiting for the lock. The term asynchronous means that the call returns immediately after sending a request to the device, and the program can keep running while the device runs the operation. A separate call may then be needed to check if the result has arrived.

Synchronous I/O has two big problems. First, the application can not do anything else while the device runs the operation, even though the host CPU is free. The OS then typically tries to schedule a different task during that time. Second, applications can not submit multiple I/O requests in parallel, i.e. concurrency is not possible. To solve these problems, a common solution is to create more threads, where each thread can have one

in flight request, providing concurrency. However, the number of threads determines the degree of concurrency, which limits scalability as maintaining a high number of threads can be expensive.

More generally, there has been a long debate between using multiple threads or event-based systems for concurrency. In an event-based system, function calls are pushed to an event queue, and an event loop continuously pops from the queue and runs the functions. Event systems are cooperatively scheduled, i.e. only when an event finishes running, a new event is run. In an event loop, programmers do not need to think about synchronization. On the other hand, threads are preemptively scheduled, so their state must be saved when descheduled. Threads have been considered to scale poorly because of the overhead of context switching and saving state, in addition to schedulers relying on $O(n)$ operations where n is the number of threads. However, opponents of event systems have shown that threads can scale as well as events with a good threading implementation. An event loop also runs on a single thread, and thus having multiple threads is the only way to obtain true CPU concurrency (15, 16).

On Linux, asynchronous I/O has been available since version 2.6 with the AIO interface. However, the interface has limited adoption, and is generally considered sub-optimal. Its main limitations are that it can only be used for direct I/O, as otherwise it falls back to synchronous submission, and if meta-data is required before the operation, it also becomes synchronous. In addition, each operation involves memory copying that adds overhead, and each operation requires two system calls, for submission, and completion, adding more overhead. The limited success of AIO is what led to the introduction of a new asynchronous I/O interface for Linux, namely `io_uring`.

2.4 The io_uring Interface

The `io_uring` interface was introduced in Linux version 5.1 in 2019 to address the shortcomings of AIO. `io_uring` is an asynchronous I/O interface that can be used for networking and storage I/O. At the center of `io_uring`'s design are two ring buffers, a submission queue (SQ) and a completion queue (CQ). The ring buffers reside in memory that is shared by the user process and kernel, eliminating the need for expensive copying between user and kernel space. Figure 2.2 shows the structure of ring buffers. Another benefit of the ring buffer data structure is that operating on it does not require locking. 57 different opcodes are supported by `io_uring` for different operations, including `readv` and `writev` (17). One

criticism of `io_uring` is that it exposes a significant attack surface in the Linux kernel, and has been difficult to integrate with the Linux security modules (18).

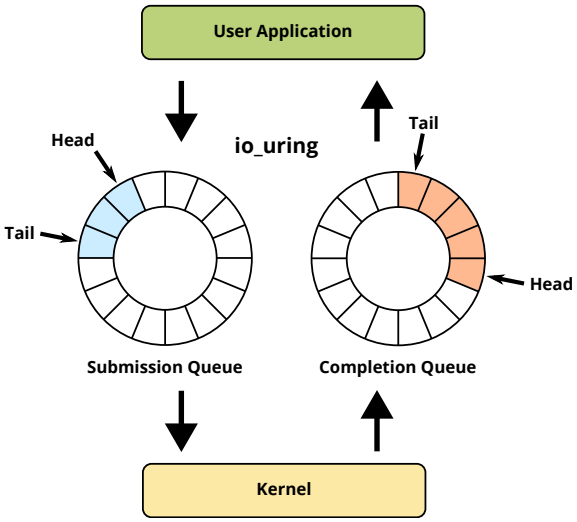


Figure 2.2: Ring buffers as used by io_uring, with head and tail pointers

2.4.1 Basic Usage

To use `io_uring`, an application starts by calling the `io_uring_setup` system call with the desired queue depth, which returns a file descriptor for the ring. Two `mmap` system calls are then needed to map the submission queue (SQ) and completion queue (CQ) into the process. The application can add events to the submission queue (SQE), which describe an operation, e.g. a `readv` operation with a pointer to a buffer and file descriptor to use. Then, an `io_uring_enter` system call is needed to notify the kernel that there are new events. This also makes it possible to submit multiple events in one system call. When an operation is finished, the kernel adds an event to the tail of the completion queue (CQE) with the result. The application can see new CQEs without the need for a system call, but it can also call `io_uring_enter` with the `GETEVENTS` flag to wait (i.e. sleep) until a certain number of events (specified by the `min_complete` argument) are available (19).

2.4.2 Polling Modes

We previously explained that to notify the kernel of new SQEs, a call to `io_uring_enter` is needed. However, `io_uring` also offers a polling mode called SQ polling, which creates a kernel thread that constantly polls the SQ for new events, eliminating the need for system calls. Modern storage devices also support polling for completed operations, instead of

notifying the host with an interrupt. For this, `io_uring` supports I/O polling. I/O polling is only possible for direct I/O, and is hinted to provide lower latency than interrupt-driven I/O, at the cost of using more CPU resources (17, 19).

2.5 Summary

Solid-state drives (SSDs) are a type of storage medium based on non-volatile memory. SSDs have become popular for mass storage in the last decades, as they can provide millions of IOPS in throughput and low latencies. The introduction of SSDs has exposed various bottlenecks and overheads in the Linux kernel and has led to the redesign of the Linux block layer. In addition, SSDs have driven the need for new high-performance I/O interfaces such as `io_uring`.

Although Linux has supported asynchronous I/O for a long time with the AIO interface, it has limited adoption and is considered sub-optimal. To overcome the limitations of AIO and support high-performance asynchronous I/O, the `io_uring` interface was introduced to Linux in 2019. `io_uring` is based on ring-buffers that reside in shared memory for lock-free and zero-copy operation. In addition, `io_uring` supports several high-performance features, such as batched submission and polling modes.

3

Design of Experimental Setup

In chapter 1, we declare our goals for this thesis, which are to evaluate different `io_uring` features, and to come up with guidelines for getting the best performance out of `io_uring` based on these features. In this section, we explain the design of our experiments. In particular, the choice of tooling, the choice of `io_uring` features to evaluate, how we evaluate each feature, the hardware configuration for our experiments, and the implementation of custom storage backends.

3.1 Benchmarking Tools

The `fio` storage benchmarking tool is popular for storage research due to its flexibility, extendability, and low CPU overhead. We use `fio` as we can build our own `io_uring` engine to support the features that we evaluate. The `fio` tool comes with multiple I/O engines, such as `sync` (POSIX), `libaio`, and `io_uring`. It also supports different I/O patterns such as `read`, `write`, `randread`, and `randwrite`. We can specify the block size, queue depth, the number of workers, and even the statistical distribution for random access. To mimic real applications, we can add a think time between I/O operations, or we can replay a trace from a real application.

For our evaluation, we also need insights into the Linux kernel while our benchmarks run, to understand what happens inside the kernel when a given feature is enabled. Many tracing tools are available on Linux, such as `strace` to understand what system calls a process makes (including arguments), and `perf` can count various events and collect traces. In recent years, the `eBPF` kernel feature has become popular to debug and observe kernel internals. We will make extensive use of `eBPF` in this thesis, e.g. to count function calls,

get stack traces of kernel functions, and confirm that the kernel takes a certain code path. We give a more detailed explanation of `eBPF` in the appendix in chapter 8.

3.2 Evaluation Plan

The `io_uring_setup` system call that is used to create a ring, supports 17 different flags to configure the ring (10). All flags are boolean options, that are either enabled or disabled, except for three flags, for specifying an SQ polling CPU affinity, a maximum number of CQEs for clamping, and a file descriptor for sharing a work queue. In addition, the `io_uring_enter` system call supports 7 different flags on SQEs, all of which are boolean, but may depend on the flags that the ring was created with (11). In addition to configuration options, the performance that an application gets from `io_uring` might also depend on its own structure and behavior.

However, for this thesis, we prioritize the features that we think are most relevant, as doing a detailed evaluation of a single flag can be a significant effort. Based on information from the `io_uring` man pages, and other resources that we find online, we believe the following configuration flags are most likely to show interesting results for performance.

- **COOP_TASKRUN** - Turn off IPIs (Inter-Processor Interrupts) for notifying a process of new completion events, which can be an overkill for many applications. Setting the flag will improve performance according to the man page (10).
- **IOSQE_ASYNC** - Request that an SQ is submitted asynchronously from a worker thread, rather than trying non-blocking submission first and then asynchronous on failure. The man page suggests it should be used if an application can assume most requests will block (11).
- **IOSQE_FIXED_FILE** - Use a registered file rather than a normal file descriptor (20). The man page does not hint an effect on performance, but online sources mention that normal file descriptors have an overhead (21).
- **REGISTER_BUFFERS** - Keep data buffers of user process persistently mapped into the kernel. Normally the buffers are mapped and unmapped into the kernel for each I/O operation (20).
- **SQPOLL** - Use submission queue polling, i.e. a kernel thread that constantly polls the submission queue, so that the application does not need to make system calls to submit (10). Previous research suggests that it can improve performance (8, 9).

- **IOPOLL** - Use polling on the device, i.e. instead of an interrupt from the device. The device must support polling and the NVMe driver must have polling queues setup rather than default queues (10). Previous research suggests that it can improve latency (8, 9).

- **SINGLE_ISSUER** - This option can be used if SQEs are always issued from a single thread. The man page hints that this allows the kernel to make some optimizations (10).

As an example of a flag that does not suggest improved performance, we can look at `IORING_SETUP_SQE128`, which is needed to make SQEs 128 bytes instead of 16 bytes, to fit certain operations. It is only used in specific applications that need these non-standard operations (10). For this thesis, we evaluate the first four flags listed, due to the time required to evaluate each flag.

To evaluate a configuration option, we start by running the simplest possible `fio` benchmark, e.g. comparing random reads with and without the flag, with a single thread, on an idle system, and look at throughput and latency for different queue depths. We then take a deeper look at the flag, its internal implementation in the kernel, and look for hints about what factors affect the performance from that flag. Based on that, we run benchmarks with system conditions that we believe affect the flag's performance, and use `eBPF` to confirm that the conditions have the effect that we suspect.

3.3 Hardware Configuration

We perform all our evaluation on the same system. The hardware configuration of the system is shown in Table 3.1, and the software versions used are listed in Table 3.2. Note that for all experiments we leave CPU turbo boosting disabled, and we also leave dual threading disabled (giving 20 instead of 40 cores), unless stated otherwise, in order to get more predictable performance.

CPU	2 x Intel Xeon Silver 4210R (10 core, 20 thread)
Memory	256 GB (4 x 64 GB) - 2933 MHz DDR4
Storage	7 x Intel Optane SSD 900P Series (280 GB)

Table 3.1: Hardware configuration used for evaluation

Distro	Ubuntu 22.04.1
Kernel	Linux 6.3.8
fio	3.35

Table 3.2: Software versions used for evaluation

3.4 Implementing a fio Engine with liburing

In this section we explain how we implement a `liburing` engine for `fio`. Although `fio` comes with an existing `io_uring` engine, we implement our own engine with `liburing` to make it easier to implement configuration options and because for real-world applications, it is recommended to use `liburing` rather than the low-level interface. The appendix in chapter 8 and documentation in the artifacts repository provide more details on how to compile `fio` with our custom engine. To write an engine, we must provide a number of function pointers through the `ioengine_ops` struct. We explain the most important functions here.

- First `init` is called to initialize our engine, where we create our ring with the desired flags to enable different features.
- Then `u_init` is called for each I/O unit, the number of I/O units is the same as the queue depth. We only use this function to create a mapping between unit ID and its struct.
- Then `post_init` is called, where we create `iovec` structs for each I/O unit, that will be needed for our `readv` and `writev` operations.
- In `queue`, we acquire a free SQE slot in the ring buffer, and prepare a `readv` or `writev` request.
- In `commit`, we submit the SQEs using the `io_uring_submit` function.
- To retrieve events, `getevents` is called, there we call `io_uring_wait_cqe`, and return the number of finished events.
- For each event returned from `getevents`, a call is made to `event` to finish that event, where we pass the result to the I/O unit.

In our engine, we have added options to enable the features that we want to evaluate. The features can be enabled with the flags `-coop_taskrun`, `-force_async`, `-registerfiles`, and `-sqthread_poll`.

3.5 Summary

In this chapter, we come up with a plan for our experiments. First, we explore what configuration options are available for `io_uring` and what their impact is on performance based on existing work. We then prioritize the configuration options and select 4 options for detailed study. We also explore what tools are available for our experiments, and the hardware setup that we will run experiments on. Finally, we describe how we implement a custom storage engine for `fio` with `liburing`. Although `fio` has an existing `io_uring` storage engine, we implement our own engine as `liburing` is recommended for real-world applications. In addition, the configuration options that we selected are not supported by the existing engine, and using `liburing` makes it simpler to add support for them.

4

Experiments

In this section, we explore the four `io_uring` configuration options that we selected in the previous section. The options were chosen based on their likeliness to affect performance as hinted by man pages and other resources. For each option, we start by exploring its implementation to understand what might affect its performance. We then run experiments with `fio` using our `liburing` engine, where we have added support for the options. For each option, we try to understand in what scenario it provides the most benefits, and in what scenarios it can degrade performance.

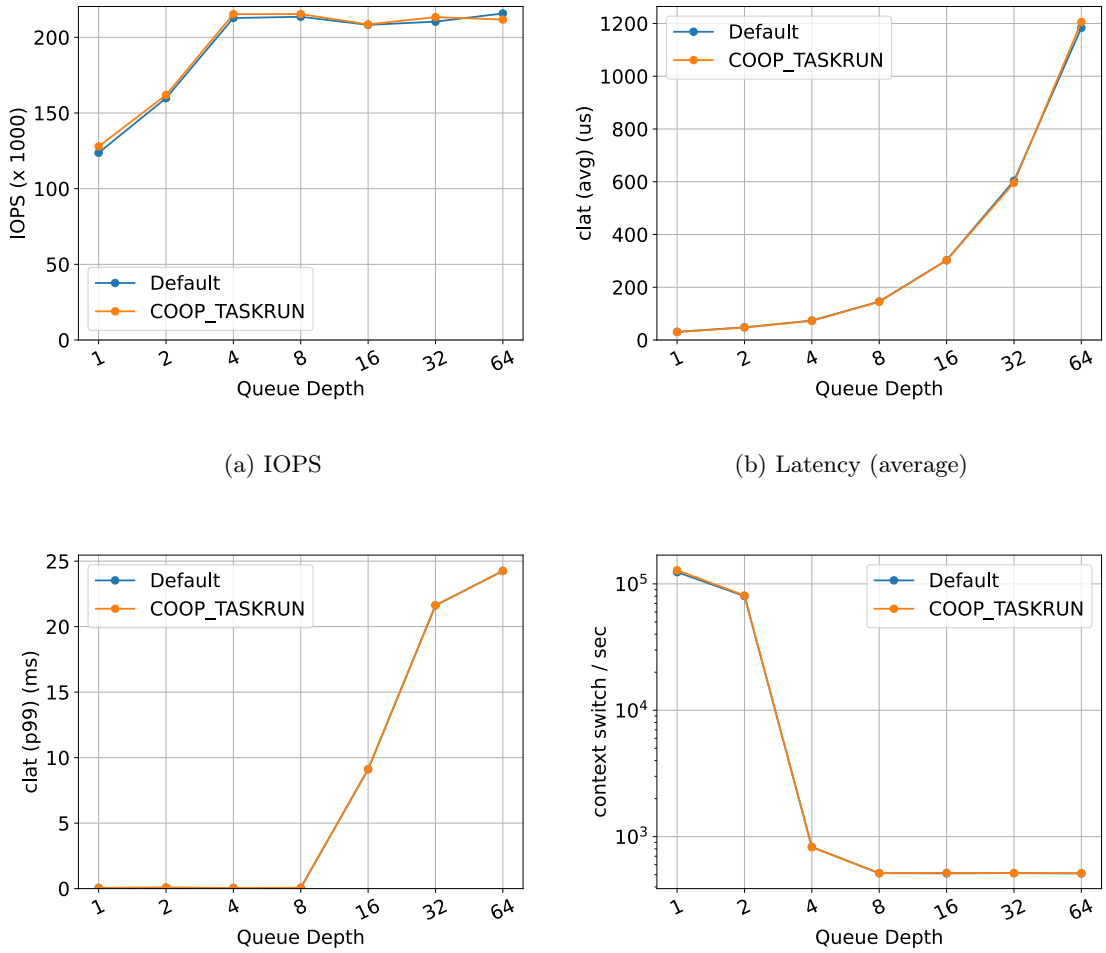
4.1 Cooperative Task Running (COOP_TASKRUN)

In the `io_uring` manpage, the flag `COOP_TASKRUN` is described as preventing a storm of IPIs that are sent for completion events, it is implied that this lowers CPU load with a potential downside on latency.

> By default, `io_uring` will interrupt a task running in userspace when a completion event comes in. This is to ensure that completions run in a timely manner. For a lot of use cases, this is overkill and can cause reduced performance from both the inter-processor interrupt used to do this, the kernel/user transition, the needless interruption of the tasks userspace activities, and reduced batching if completions come in at a rapid rate. Most applications don't need the forceful interruption, as the events are processed at any kernel/user transition. The exception are setups where the application uses multiple threads operating on the same ring, where the application waiting on completions isn't the one that submitted them. For most other use cases, setting this flag will improve performance. Available since 5.19.

4.1.1 First Experiments

For our first experiment, we run `fio` with and without the `COOP_TASKRUN` flag on a single core. We use `fio` with the `liburing` engine to run a random read benchmark, using a single Intel Optane device and a single thread. Figure 4.1 shows the IOPS, average and p99 latency, and context switches per second. We do not observe any difference in performance with or without the `COOP_TASKRUN` flag.



(a) IOPS

(b) Latency (average)

(c) Latency (p99)

(d) Context switches per second

Figure 4.1: IOPS, average and p99 latency, and context switches per second for the initial `COOP_TASKRUN` experiment

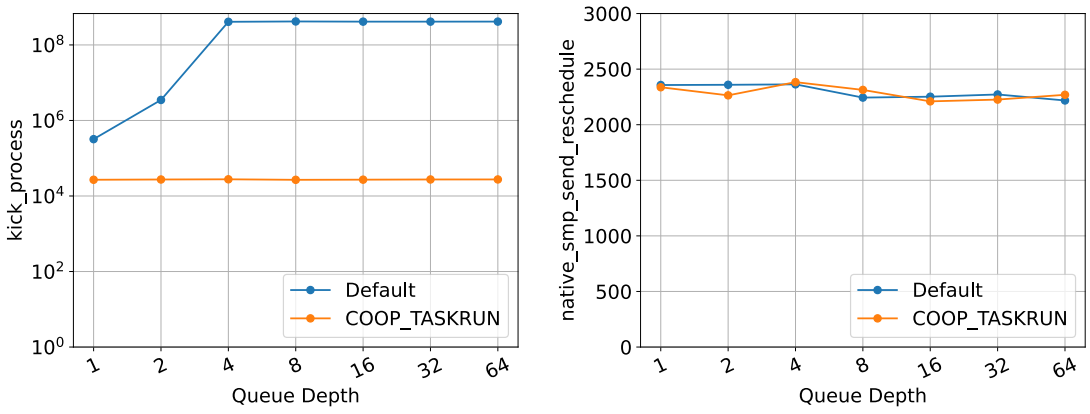
We also run the same experiment on multiple cores by adding more threads, and we also experiment with running CPU intensive tasks in parallel to force more context switches.

However, these experiments also showed no difference in performance. The full results are omitted but can be found in the artifacts repository (experiment 2 and 3).

4.1.2 Understanding What the Flag Does

Next, we start reading the kernel source code to see what the `COOP_TASKRUN` flag does. We see that it changes something called a notify method, a property that is passed when calling a `task_work_add` function. We figure out that `io_uring` creates something called a task work, a callback that finishes each I/O completion. This task work is put in a queue on the submitting process `task_struct`, and all task works are processed when the process exits to user mode. The `task_work_add` function takes in an argument called notify method, which `COOP_TASKRUN` changes to `SIGNAL_NO_IPI` from `SIGNAL`. We see that when SIGNAL is set, `task_work_add` will call `kick_process`, which kicks a process that is running in userspace back into kernelspace, so that it will run its task work. The kick is done by sending a reschedule IPI to the CPU running the process. There are two conditions for `kick_process` to send an IPI, that the process is running on a remote core, and the process is currently the active process on that core.

We use eBPF to count the number of calls to `kick_process` and the function that sends a reschedule IPI (`native_smp_send_reschedule`). Figure 4.2 shows that the `COOP_TASKRUN` flag is indeed preventing calls to `kick_process` (first figure), but regardless of the flag, no IPIs are actually being sent (second figure).



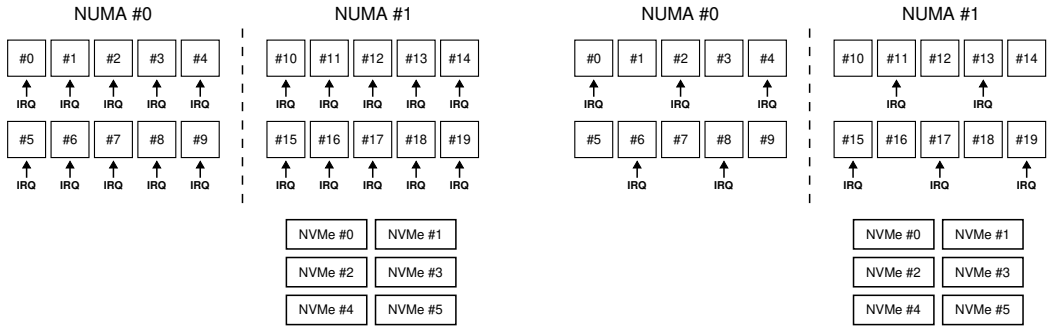
(a) Calls to kick_process

(b) Calls to send reschedule IPI

Figure 4.2: eBPF counts for the number of calls to `kick_process` and number of IPIs sent with `native_smp_send_reschedule`

4.1.3 Investigating IRQ Affinity

This motivates us to experiment with an environment where I/O can not happen locally on a single core. As our system has 20 CPU cores, the Linux kernel sets up one NVMe hardware queue for every core, giving us 20 hardware queues. We configure our system such that only 11 NVMe hardware queues are available, in that case Linux spreads the queues (and thus NVMe IRQs) across the CPUs as shown in Figure 4.3.



(a) Interrupts on all cores

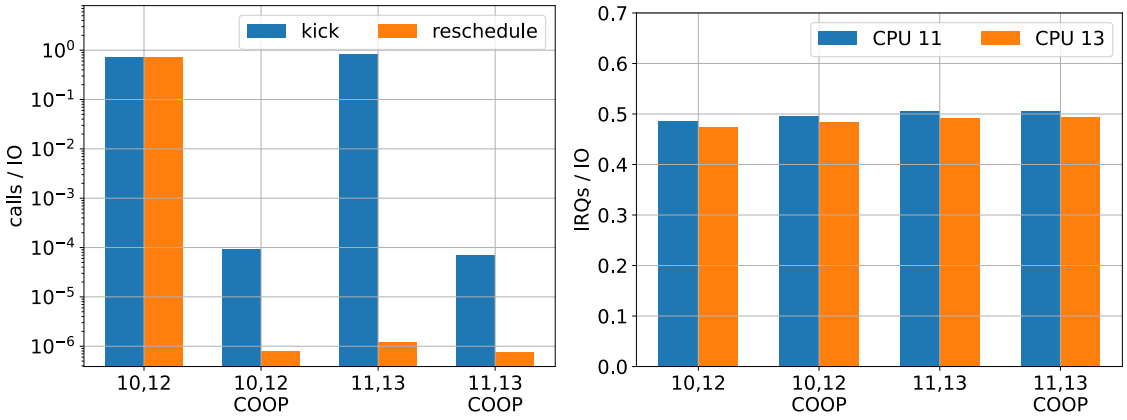
(b) Interrupts on half the cores

Figure 4.3: IRQ handler placement (a) before and (b) after lowering the number of available NVMe hardware queues

We configure our system so that only odd numbered cores have an IRQ registered. In this experiment we use a single Intel Optane device and experiment with running `fio` on cores 11 and 13 (cores with interrupts), and cores 10 and 12 (cores without interrupts). We use two cores as that still allows saturating the Optane device, and we are interested in seeing what activity happens between CPUs.

Now, we count the calls to `kick_process` and `native_smp_send_reschedule` again using eBPF, and we also count the number of NVMe IRQs coming in per core. First running `fio` on CPU (11, 13) and then on CPU (10, 12). We run a `fio` random read benchmark with two threads pinned to the two CPU cores using `taskset`.

The results are shown in Figure 4.4. For CPU (11, 13) we see the same result as before, `kick_process` is only called when the `COOP_TASKRUN` flag is not set, but no reschedule IPIs are sent regardless of the flag. However, for CPU (10, 12) we see a high number of reschedule IPIs being sent by default, and `COOP_TASKRUN` prevents them from being sent. We also see that for all experiments, NVMe interrupts are always coming in on cores 11 and 13.



(a) eBPF kick and IPI counts

(b) Interrupt counts per CPU

Figure 4.4: eBPF counts for number of function calls and the number of NVMe IRQs received per core

We now investigate whether there is a performance difference for these four cases, i.e. with or without `COOP_TASKRUN`, running on IRQ or non-IRQ cores. The results are shown in Figure 4.6. Starting with throughput (IOPS), we see that running on interrupt cores (11, 13) we get the same throughput regardless of the `COOP_TASKRUN` flag, which is the same result as we got in our first experiments. When running on non-interrupt cores (10,12) we see slightly higher throughput without `COOP_TASKRUN`, and significantly higher throughput with `COOP_TASKRUN`. The same result is reflected in the latency.

We see that the improvement in throughput is mostly at $QD \geq 4$, at low queue depths the COOP flag makes little difference. We investigate this further with eBPF, and find out that `kick_process` is only called if a function called `wake_up_state` returns false. That function wakes up a sleeping task, in which case it is enough to place it on the run queue again, while `kick_process` is needed for currently running tasks. Figure 4.5 shows the complete logic as we now understand it, with the following six steps:

1. `io_uring` receives a completion event from the NVMe drive and creates a `task_work` that will finish the IO. If the `COOP_TASKRUN` flag is set, the notify mode will be `TWA_SIGNAL_NO_IPI`, but by default it is `TWA_SIGNAL`.

2. After the `task_work_add` function has inserted the `task_work` into the linked list of task works on the task `task_struct`, it will notify the task. The `TIF_NOTIFY_SIGNAL`

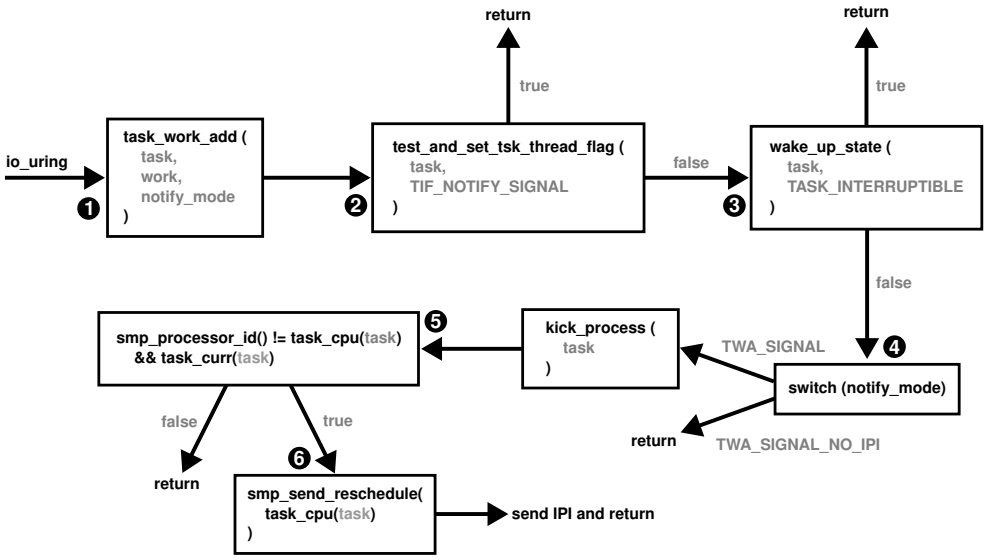


Figure 4.5: A flow diagram of `io_uring`'s call to `task_work_add` showing where `COOP_TASKRUN` changes the behavior.

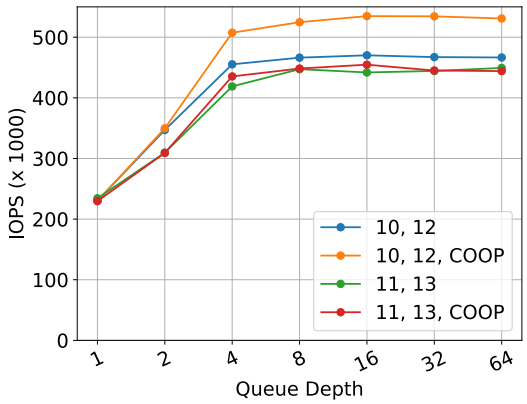
flag is added to the task, but if it was already set, then `task_work_add` does nothing and returns.

3. Otherwise, if the signal flag was not previously set, `task_work_add` tries to wake up the task, but only if it's in the `TASK_INTERRUPTABLE` state. In that case, the task will be put in the run queue again, and when it gets scheduled, the `task_work` will run. `task_work_add` returns unless the wake up call was unsuccessful.

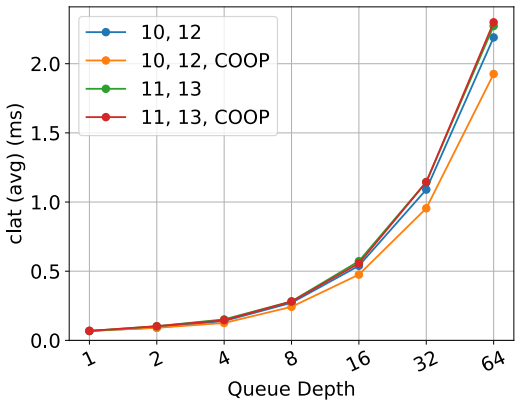
4. The wake up call might have failed because the task was already in the RUNNABLE state (or some other state), in that case, a `kick_process` call is needed to notify the task, but if `COOP_TASKRUN` and thus `TWA_SIGNAL_NO_IPI` is set, then nothing is done and `task_work_add` returns.

5. Kicking a process is only done if it is running on a different core than `kick_process` is being called on, and only if that task is currently in the `RUNNABLE` state on that remote core, otherwise nothing happens.

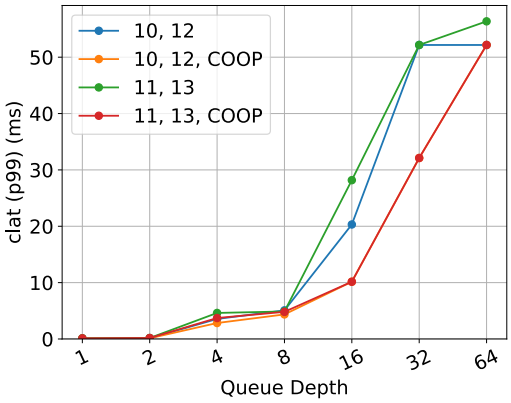
6. Finally, a reschedule IPI is sent to the remote core and `task_work_add` returns. The remote core will invoke the IPI handler, which for reschedules does nothing but invoke the scheduler again. Most likely, the same task will be run again, and before entering user mode again, the new `task_work` will be run.



(a) IOPS



(b) completion latency (clat) - average

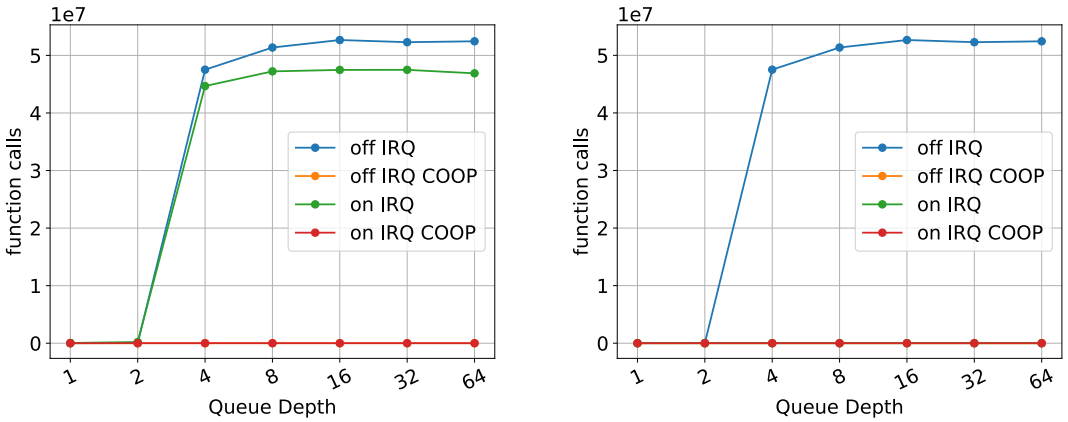


(c) completion latency (clat) - p99

Figure 4.6: IOPS, average and p99 latency for the four cases

We now do more eBPF counts to confirm our understanding of `task_work_add` and the effect that `COOP_TASKRUN` has. Figure 4.7 shows the counts of `kick_process` and reschedule IPIs as a function of queue depth. Like before, we have four cases to examine, running fio on interrupt cores (10, 12) or non-interrupt cores (11, 13), and running with or without `COOP_TASKRUN`. We can see that `kick_process` is not being called at $QD \leq 2$, but at $QD \geq 4$ it is called. This matches with the performance seen in Figure 4.6.

Figure 4.8 explains why `kick_process` is not being called at $QD \leq 2$. It shows the number of calls to `wake_up_state` for the four cases, with two lines for each case, the number of calls that return `true` and `false`. We see that at $QD \leq 2$, `wake_up_state` is



(a) Calls to `kick_process`

(b) Calls to `native_smp_send_reschedule`

Figure 4.7: Number of calls to `kick_process` and `native_smp_send_reschedule` as a function of queue depth for the four cases

almost always returning `true` (for all four cases), meaning that the task was sleeping and `kick_process` thus not necessary.

In this subsection we have demonstrated that when running an I/O process on non-interrupt cores, specifying `COOP_TASKRUN` will give significant performance improvement. We suspect that one reason for the improved performance is that the IRQ handler `nvme_irq` still runs on the interrupt cores, thus freeing up CPU time on the non-interrupt cores.

4.1.4 Understanding Interrupt vs Non-interrupt Cores

We now have an environment where every other CPU has an interrupt (only odd numbered cores), and we have seen that running `fio` on a non-interrupt core gives better performance, in particular when IPIs are disabled with `COOP_TASKRUN`. We can now explain how the path of an `io_uring` read changes when running on a non-interrupt core and the effect of `COOP_TASKRUN`.

First, let's assume that our process submits I/O commands on an interrupt core (e.g. 10 and 12), Figure 4.9 shows the following order of events:

1. **Process submits SQ:** fio submits a read request using the `io_uring_enter` syscall, this system call returns to userspace after running (i.e. after step 2).

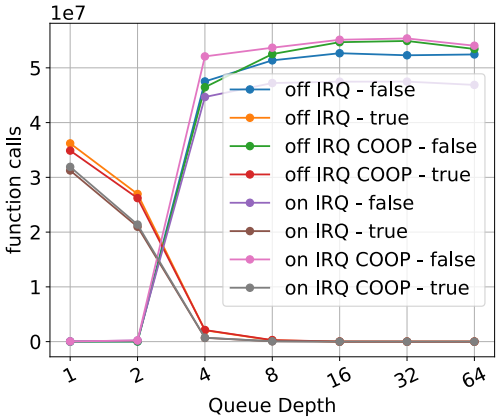


Figure 4.8: Number of calls to `wake_up_state` as a function of queue depth and grouped by return value (`true` or `false`) for the four cases, showing that for $QD \leq 2$ the return value is always `false`, after that it is always `true`

2. **Command is sent to NVMe device:** the block layer adds an NVMe read command to the NVMe hardware queue and then rings the NVMe doorbell.

3. **NVMe device raises IRQ:** the device raises an IRQ to signal that it has finished a command. Linux's interrupt handler maps the IRQ vector to the `nvme_irq` handler and calls it. The handler adds a task work, `tctx_task_work`, to the process that submitted the IO. This task work will do the final processing of the command completion event.

4. **Returning to usermode:** if `nvme_irq` preempted user mode, it will return back to user mode after running. At every return to usermode, the task works that have been added to the processes' task work queue are called, in our case `tctx_task_work`.

5. **Reading NVMe completion queue:** the result of the read command is read from the NVMe completion queue.

6. **Back to usermode:** after the task work queue of our process has been emptied, we finally return to usermode.

Let's now look at the second case, when our process submits I/O commands on a non-interrupt core (e.g. 11 and 13). Figure 4.10 shows two cases, depending on whether the IPI preempts user- or kernel mode execution. In any case, the difference starts when the NVMe device raises an IRQ (**3**), which comes in on a different core than the submission (**2**). Assuming that the submitting process is still running (not sleeping), an IPI will be sent back to the submitting CPU (**4**). The IPI may either come during user- og kernel

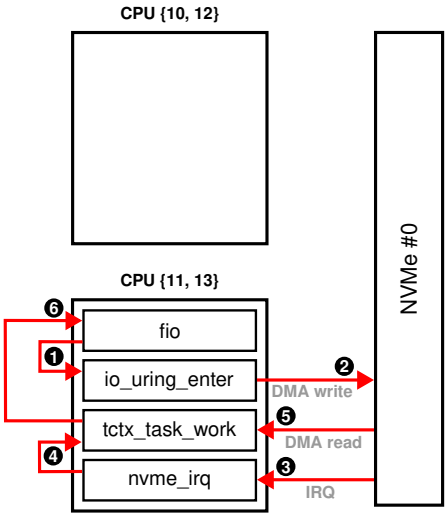


Figure 4.9: Flow of events running on an interrupt core

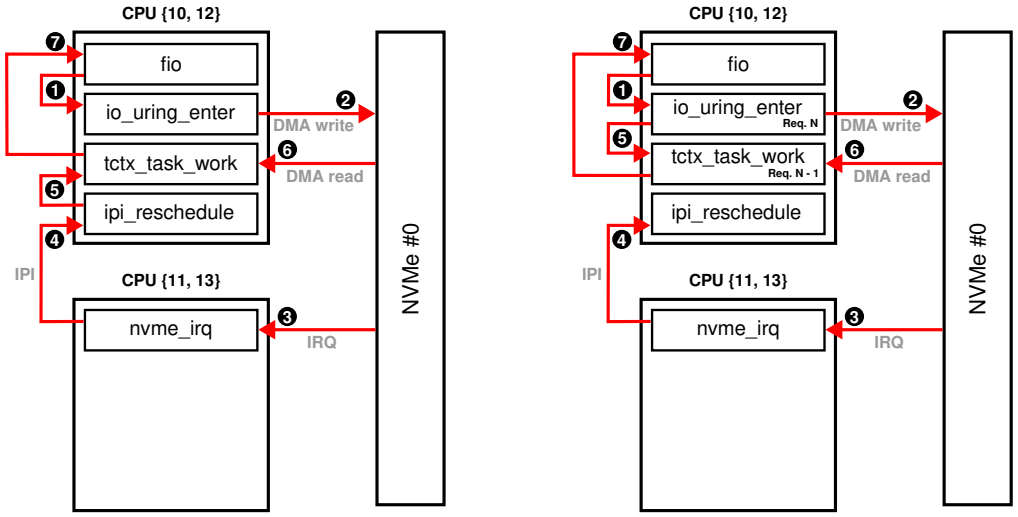
mode execution. When preempting kernel mode, the `task_work` will not run when the IPI handler exits, but when the previous kernel context returns to user mode (see Figure 4.11). In a sense, the IPI is then useless, as the `task_work` runs at the same point regardless of the IPI.

4.1.5 Quantifying the Benefits of No IPIs

We have seen that `COOP_TASKRUN` improved the performance by around 20% under the conditions of our last experiment. We are now interested in quantifying the cost of IPIs, as it may vary with system conditions. In normal operation, `fio` does little other than submitting I/O as fast as possible.

The cost of an IPI may be different depending on whether it is preemting user- or kernel space. In addition, an IPI handler preemting kernel space will not execute `task_work` on exit, the handler exits to the previous kernel space operation, e.g. a syscall, and when that syscall exits to userspace, the task works will be processed. In that sense, some IPIs may be useless. Figure 4.11 shows where `task_work` is run during mode switches (red dots).

We expect that if an application spends more time in usermode, there will be a higher ratio of IPIs that preempt usermode (Figure 4.12). This assumes that the IPIs are sent uniformly and not in bursts. We are interested in whether IPIs that hit user mode are more expensive.



(a) IPI preempts user mode (4) and causes task work to run (5)

(b) IPI preempts kernel space (4), no task work is run directly, only when previous kernel execution returns to user mode (useless IPI)

Figure 4.10: Flow of events running on a non-interrupt core, two scenarios

To investigate this, we use the `thinkcycles` option of `fio`, which will simulate a computation of N cycles after each I/O completion. We first attempted to use the `thinktime` option of `fio`, but that one results in system calls (`get_microtime` and `wait`), which alters our results. The `thinkcycles` option does a pure CPU computation.

We start by measuring the total number of IPIs, and also the number of IPIs that are preempting user space. We do this by counting the number of times `exit_to_user_mode_prepare` is called from the stack of a reschedule IPI handler. We can then calculate the ratio of IPIs that are hitting user mode.

As we increase the thinktime, we do see an increase in the ratio of IPIs that preempt user mode (Figure 4.13). However, there is a sudden drop at 128-256 thinkcycles where almost all IPIs are preempting kernel mode. We can not explain this drop.

Next, we investigate the performance for different thinkcycles values. We can see throughput (IOPS) dropping as we increase the thinkcycles in Figure 4.14, but the rate of change is the same with and without `COOP_TASKRUN`. The second graph shows the IOPS improvement of `COOP_TASKRUN`, which does not increase with added thinkcycles. We see similar results for latency in Figure 4.15.

(a) IPI preempts kernel

(b) IPI preempts user

Figure 4.11: The difference between an IPI that preempts kernel- and userspace, the red dot shows where task work is run.

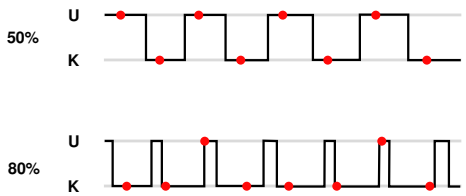


Figure 4.12: An application that spends more time in usermode will have a higher ratio of IPIs arriving in usermode.

A possible explanation is that although IPIs on userspace may be more expensive, whatever increased cost comes from IPIs hitting userspace, it is small compared to the added thinkcycles, and thus IOPS drops equally fast with and without `COOP_TASKRUN`.

However, this does show that as I/O applications get more CPU intensive, the `COOP_TASKRUN` flag starts to make less difference.

4.1.6 IRQ Affinity and NUMA Topology

Our previous experiment showed that running an I/O process on a non-interrupt core can give better performance, especially when using the `COOP_TASKRUN` flag. We now want to quantify the improvement depending on what non-interrupt cores are used in a NUMA topology.

In our system we have 2 NUMA domains and hyperthreading, so there are four levels when considering communication between cores:

1. Local
2. Twin thread

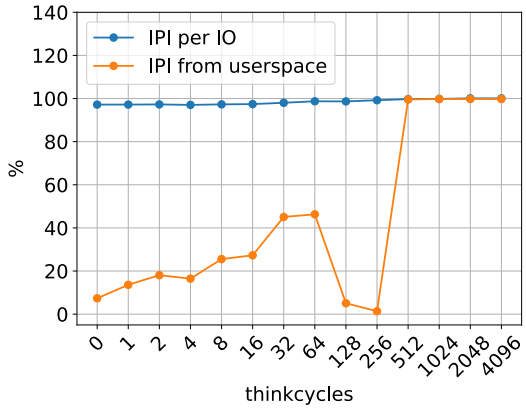


Figure 4.13: The ratio of IPIs that preempt user mode as a function of thinkcycles

3. Core on same NUMA node

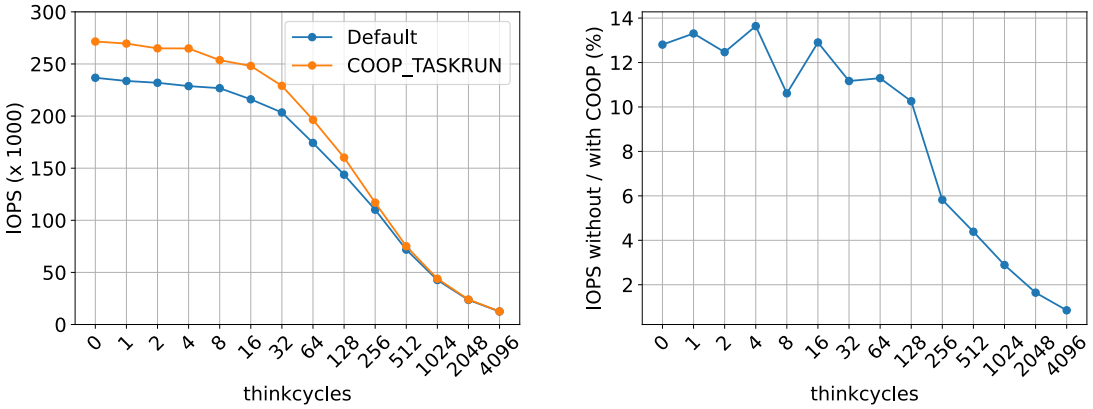
4. Core on other NUMA node

In this experiment, we configure the system with only a single NVMe IRQ, and then experiment with running fio on a core from each of these levels. We experiment with having the NVMe IRQ on a core in the NUMA node that has the Optane drive (Figure 4.16 (a), and then on the remote NUMA node (Figure 4.16 (b).

The results are shown in Figure 4.17 and Figure 4.18. The four different colors indicate where the benchmark is run in the NUMA topology compared to the interrupt core. The four groups indicate which NUMA domain the NVMe interrupt core is in, and whether COOP_TASKRUN is used.

For throughput (IOPS), we see that for $QD = 16$ the best throughput is when fio runs on a CPU in the same NUMA domain as the interrupt core, but the core is not the same as the interrupt core (local) or the interrupt core's adjacent hyperthread (thread), and it does not matter which NUMA domain fio and the interrupt are in. However, for $QD = 1$, the best performance is slightly better when the NVMe interrupt is registered on NUMA 0.

Having an NVMe interrupt on NUMA domain 0 (remote from NVMe drive) is only possible if we specify 2 hardware queues. When specifying a single queue, the interrupt always gets registered on NUMA 1. This is why the red bar is missing for the NUMA 0 bars.



(a) Throughput (IOPS) as a function of thinkcycles

(b) Relative improvement to IOPS when using `COOP_TASKRUN` (percent)

Figure 4.14: The effect of increasing thinkcycles on throughput (IOPS) with and without `COOP_TASKRUN`

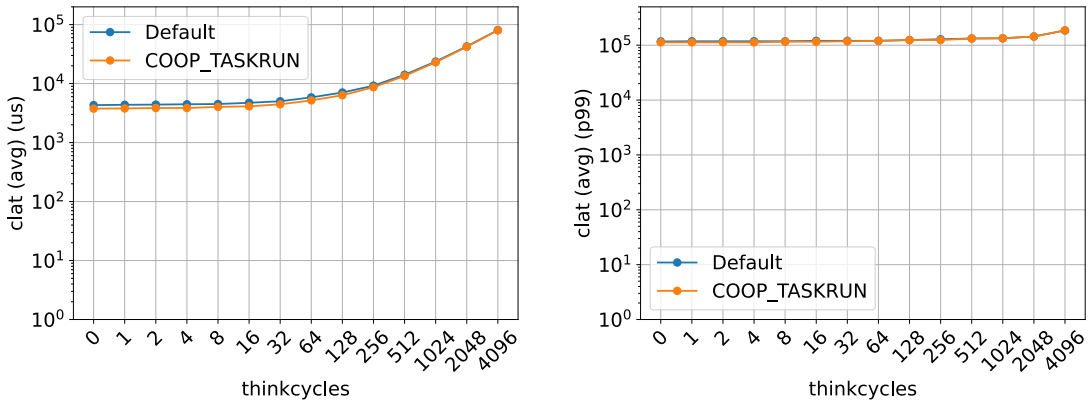
4.1.7 Ring Completions on a Separate Thread

This experiment is about finding a case where `COOP_TASKRUN` makes things worse. We go off a hint in the man page that it should not be used when completions happen on a different thread from submissions. First we try modifying `fio` for this experiment, by having each worker thread spawn an additional thread for its completions. We also attempt to add a flag `-single-ring` that would share an `io_uring` with all worker threads, then a completion could happen on a different thread from submission.

This design did not succeed, implementing dual-thread workers was too incompatible with `fio`'s architecture. The main problems were that `fio` is designed with completions happening (`getevents`) on the same thread, so we would need to get the completions back to the submitting thread with IPC, and the `getevents` and `event` callbacks are not suitable for multiple threads using the same ring.

We then write a simple `io_uring` program that does N reads and reports the latency of each request. It forks a new thread that handles completions on a shared ring. First, we try to store the submission time in nanoseconds in the SQE user data field, and then reporting the latency on the consuming thread, but we run into the following problem:

1. We need a mechanism to limit the rate of submissions, otherwise it can submit too many requests, causing a segfault



(a) Average latency as function of thinkcycles

(b) p99 latency as function of thinkcycles

Figure 4.15: The effect of increasing thinkcycles on average and p99 latency with and without `COOP_TASKRUN`

2. The clocks on each thread are not synced to nanoseconds, so the reported latency is wrong

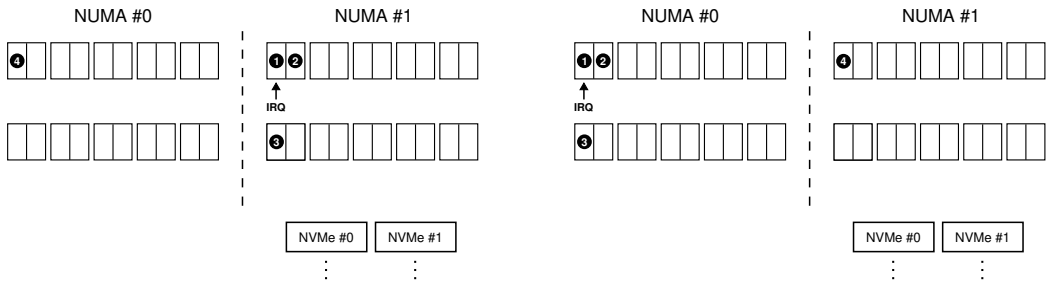
Then we modify the program to a "ping-pong" approach: thread 1 submits on ring 1, then waits for completion on thread 2. thread 2 waits for completions on ring 1 and then submits on ring 2. Then we can measure submission and completion time on the same thread (but it will be the time of 2 requests). The program also fixes the CPU affinity of both threads to separate cores. The custom benchmark is called `io_uring_completion_thread` and can be found in the artifacts repository (see appendix).

4.1.8 First Run and Investigating Absence of kick_process Calls

We are ready to experiment with our new benchmark. We configure the machine to have a single NVMe hardware queue, and we pin our two benchmark threads to separate cores, neither having an NVMe interrupt. This will cause completion events to send an IPI, as we saw in our previous experiment.

	kick_process	reschedule	wake_up_state (F)	wake_up_state (T)
Default	2,842	280	242	2,000,028
COOP	2,726	74	63	2,000,025

Table 4.1: eBPF counts during first benchmark run



(a) Single IRQ on NUMA domain of NVMe drives

(b) Single IRQ on remote NUMA compared to NVMe drive

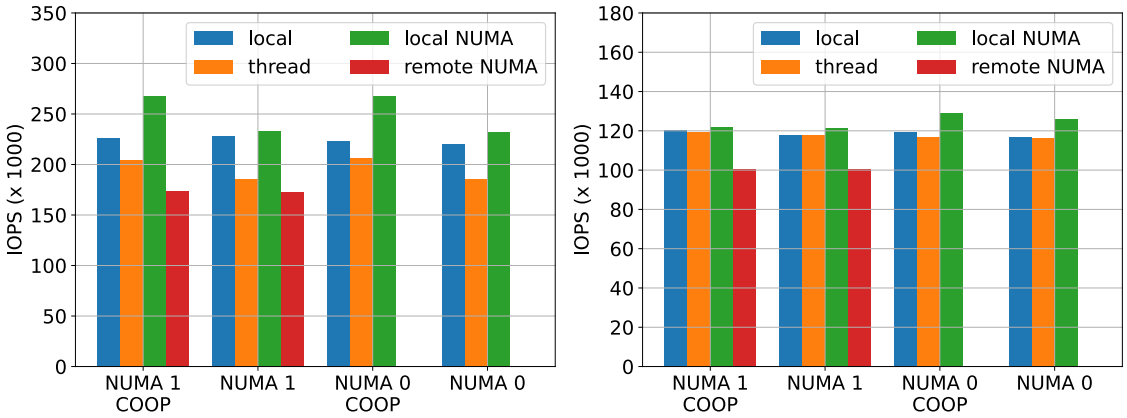
Figure 4.16: The four levels in our NUMA hierarchy, with IRQ on either a local or remote NUMA node

We start by using eBPF to verify that our benchmark is triggering the expected kernel code path. However, counting `kick_process` calls (Table 4.1), we do not see any calls regardless of whether `COOP_TASKRUN` is used. We then check the `wake_up_state` function and see that it always returns `true`, it must return `false` for `kick_process` to be called. It means that the task was sleeping (in `INTERRUPTABLE` state) and it was enough for the kernel to wake the task up to make `task_work` run, i.e. no kick is needed. This is the same result as in Figure 4.6, where we saw that `COOP_TASKRUN` only makes a difference at higher queue depths. At higher queue depths the task is always busy and never goes to sleep, and thus a `kick_process` is needed to kick the task into kernel mode, and then `task_work` will run before entering user mode again.

The benchmark is currently submitting I/O at $QD = 1$. We modify our benchmark to support variable queue depth with the same "ping pong" approach as before, but the submission thread will start by submitting N operations before completing ("ponging") the first response. Then we have N operations in flight at each moment. Figure 4.19 shows how a queue depth of N is maintained.

4.1.9 Investigating Absence of IPIs

Running our benchmark with a higher queue depth, we now see a high number of `kick_process` as expected. However, we do not see any IPIs with eBPF, even though our two benchmark threads are running on non-interrupt cores. We need to investigate why `kick_process` does not send an IPI. From our previous investigations (see Figure 4.5) we know that there



(a) IOPS (QD=16)

(b) IOPS (QD=1)

Figure 4.17: IOPS when running `fio` in 4 different levels of our NUMA hierarchy, for two different queue depths.

can only be two reasons: the task to kick is on the same CPU as `kick_process` is running, or the task is not active at that moment (sleeping).

We start by investigating the first condition. We know that the tasks to kick, our benchmark threads, run on CPU 0 and 2. The single NVMe interrupt handler is on CPU 26 and thus `kick_process` should be running there as well, making the first condition true. We use eBPF to count on what CPU `kick_process` runs along with its stack trace (who called it).

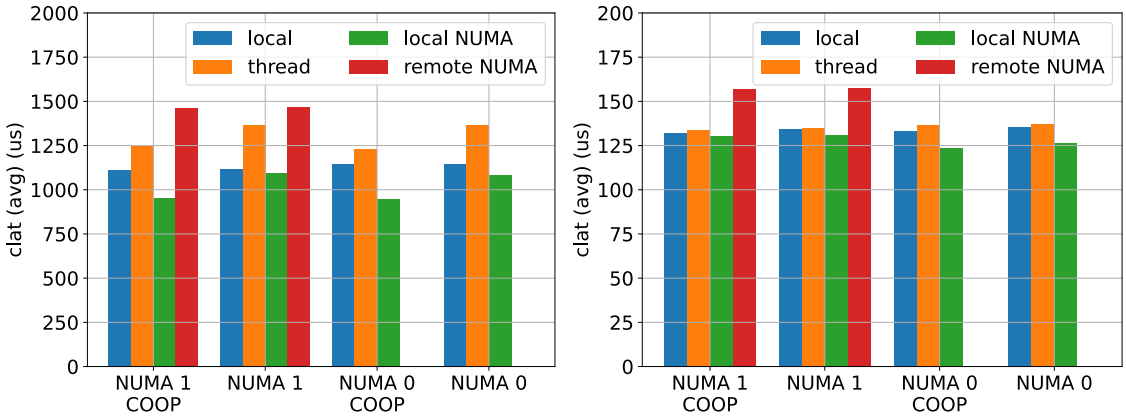
Listing 4.1: CPU ID of kick_process (1 queue)

```
...
@[32]: 1527
@[4]: 2851
@[0]: 752962
@[2]: 755993
```

Listing 4.2: CPU ID of kick_process (2 queues)

```
...
@[27]: 98
@[10]: 105
@[6]: 1522
@[26]: 448726
```

Surprisingly, we see that `kick_process` is running on CPU 0 and 2, same as our benchmark threads, which contradicts what we have previously seen. `kick_process` call counts per CPU are shown in Listing 4.1. We then use eBPF to see the kernel stack when it is called, to see where it gets called from. The stack is shown in Listing 4.3. We can see that it is run from a `softirq`. However, if we look at the stack from our previous experiments, shown in Listing 4.4, we see `kick_process` being called directly from the `nvme_irq`.



(a) Average latency (QD=16)

(b) Average latency (QD=1)

Figure 4.18: Average latency when running `fio` in 4 different levels of our NUMA hierarchy, for two different queue depths.

Listing 4.3: Stack trace from kick_process (1 queue)

```
kick_process
__io_req_task_work_add
io_complete_rw
blkdev_bio_end_io_async
...
do_softirq
...
secondary_startup_64_no_verify
```

Listing 4.4: Stack trace from kick_process (2 queues)

```
kick_process
__io_req_task_work_add
io_complete_rw
blkdev_bio_end_io_async
...
nvme_irq
...
handle_irq_event
...
common_interrupt
...
secondary_startup_64_no_verify
```

After comparing the two stack traces and exploring the kernel source code, we see that the divergence is caused by `blk_mq_complete_request_remote`. If this function returns true, the operation will be finished remotely. It returns false unless a few conditions are met, one of which is `rq->q->nr_hw_queues == 1`. Thus if there is only one NVMe hardware queue, the kernel calls `blk_mq_raise_softirq` to complete the request remotely. When we run our benchmark with 2 queues, we do indeed get IPIs (Listing 4.4). We do not know why this special case exists in the kernel.

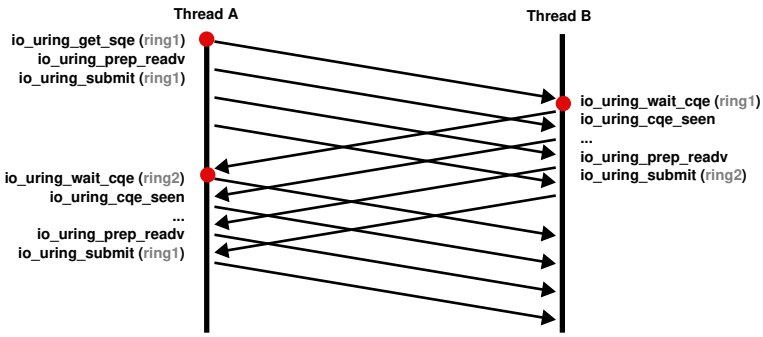


Figure 4.19: Read commands sent between two threads in a "ping pong" dynamic, showing the `liburing` functions used for the two rings

4.1.10 A Case Where COOP_TASKRUN Breaks Things

We have now figured out what conditions are needed for our benchmark to show a difference with `COOP_TASKRUN`, i.e. having a higher queue depth, and at least 2 hardware queues. We now run our actual experiment, to see if `COOP_TASKRUN` makes a difference with a separate thread consuming CQEs. We measure the time it takes to complete $1 \cdot 10^9$ requests. The results are in Figure 4.20 and we see no difference in using the flag. In both cases, the duration is highest for $QD = 1$ and decreases until $QD = 4$.

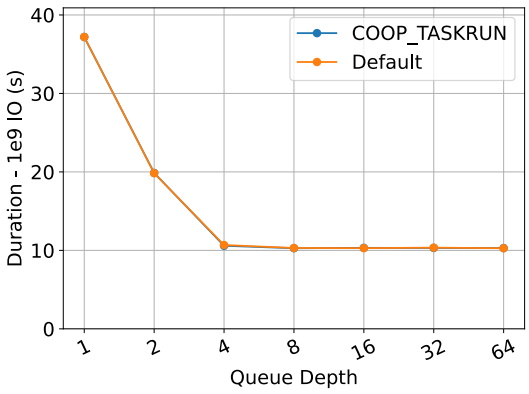
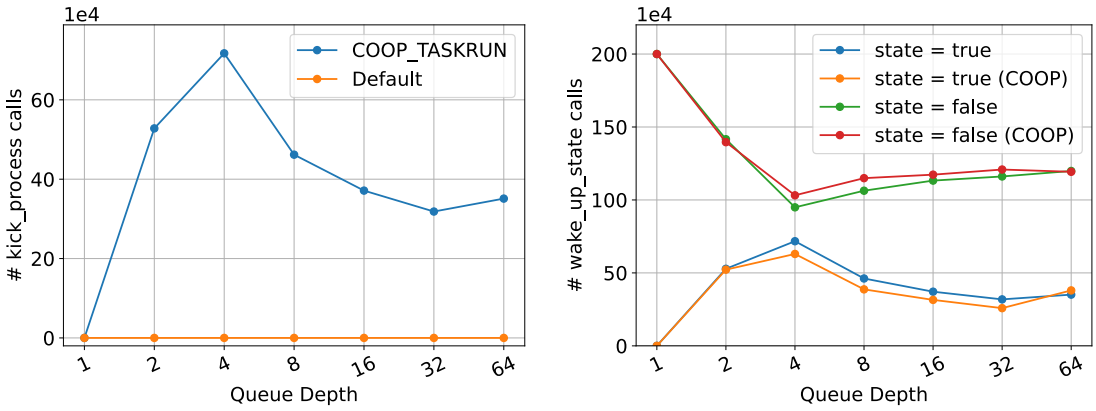


Figure 4.20: Duration of the custom benchmark with and without COOP_TASKRUN

Unlike our previous experiments, using `COOP_TASKRUN` is not resulting in higher throughput. We verify that IPIs are being sent, figure Figure 4.21 shows that `kick_process` is being called only when `COOP_TASKRUN` is omitted, and the number of `kick_process` calls is determined by how often `wake_up_state` returns `false`, same as in our previous experiments.



(a) `kick_process` counts

(b) `wake_up_state` counts

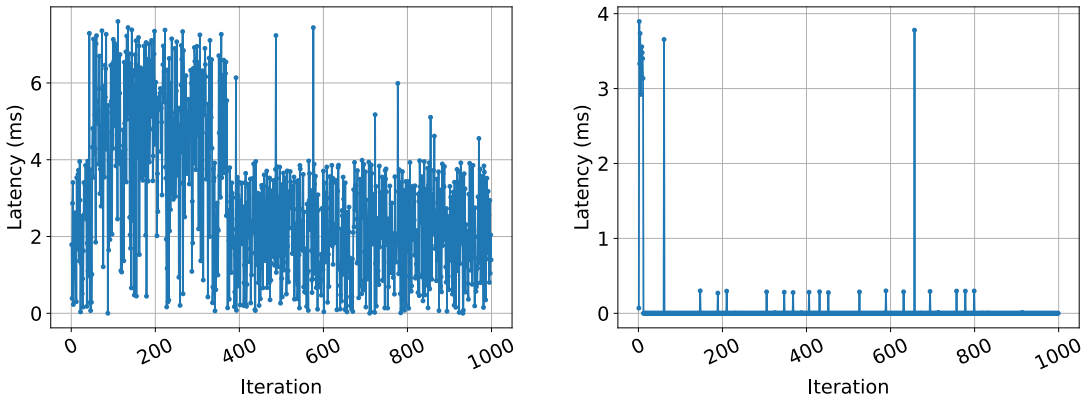
Figure 4.21: eBPF function call counts for completion thread experiment

We also use eBPF to confirm our assumption that a `task_work` for a completion event always runs on the submitting task, even if the completion takes place on a different core. To do this, we have thread A do two submissions (pings) for every completion (pong), and indeed we observe twice the number of `tctx_task_work` (the task work of `io_uring`) on the submitting core.

We have an idea of when `COOP_TASKRUN` could break things: If the submitting thread calls `io_uring_submit` to submit an SQE, and then moves on to pure CPU work that does no system calls. In that case a `kick_process` would be needed to make the submitting thread go into kernel mode and process the outstanding `task_work` that will make the CQE visible to the consuming thread.

We modify our custom benchmark to submit only a single SQE, and then do a long busy loop. We measure the time on the consuming thread that it takes `io_uring_wait_cqe` (liburing) to return the CQE.

Running the benchmark 1000 times, we get the latency measurements shown in Figure 4.22. We observe a significant difference in latency, where not using `COOP_TASKRUN` (the default configuration) gives a latency in the microsecond range as expected, but with `COOP_TASKRUN` the latency is up to 7 milliseconds (7000μs). With `COOP_TASKRUN`, we also notice a strange period where the latency doubles (ca. #100 - #400), but outside that the latency seems bounded by 4 milliseconds. Table 4.2 shows the average of all iterations, and also if we filter out points above 4 milliseconds.



(a) Latency with `COOP_TASKRUN`

(b) Latency without `COOP_TASKRUN`

Figure 4.22: Latency of `main-fail` over 1000 iterations

	# samples	average (μs)	median (μs)	standard deviation (μs)
All results	1000	2916.9	2712.3	1883.3
Filtered	775	2107.5	2116.8	1169.7

Table 4.2: Latency of 1000 iterations with COOP_TASKRUN, before and after filtering

We have found a case where `COOP_TASKRUN` hurts the latency significantly. This matches with our assumption, that while the submitting thread is in a CPU busy-loop, no kernel mode transition occurs to finish the I/O by running `task_work`. Still, the requests are finishing eventually, so we use eBPF to find out what causes the submitting task to get its `task_work` executed. Listing 4.5 shows the stack trace for this case.

This originates from the APIC timer interrupt. The kernel uses this interrupt as a "system tick", to invoke the scheduler so that no task runs for too long. We can verify the frequency of the timer with eBPF (Listing 4.6).

Indeed, we measure 250 times per second on the CPU where our benchmark runs, but less on cores that go into sleep mode. This means that our `task_work` will be delayed at worst by 4 milliseconds, and average 2 milliseconds, which matches with our measurement.

Listing 4.5: Stack trace of task_work corresponding to IO completion

```
tctx_task_work
get_signal
arch_do_signal_or_restart
exit_to_user_mode_prepare
```

```
irqentry_exit_to_user_mode
irqentry_exit
sysvec_apic_timer_interrupt
asm_sysvec_apic_timer_interrupt
```

Listing 4.6: Counting APIC timer interrupts over 10 seconds

```
$ bpftrace -e 'kprobe:__sysvec_apic_timer_interrupt {@[cpu] = count();}'
             -e 'interval:s:10 { exit(); }'
             -c './main-fail /dev/nvme2n1 1 coop'
...

@[11]: 96
@[37]: 147
@[0]: 2503
```

A delay in the millisecond range is significant for NVMe drives that offer microsecond range latencies. We conclude that `COOP_TASKRUN` should not be used when completions are consumed on another thread.

4.1.11 Conclusion

We have discovered that the `COOP_TASKRUN` flag affects cases where `io_uring` completion events must be communicated between cores. Our initial experiments did not show any difference, because every CPU core on our system had an NVMe interrupt vector, and thus completion events could always happen locally. Limiting the NVMe vectors and submitting I/O on non-interrupt cores showed that `COOP_TASKRUN` can significantly improve performance. We also observed that the choice of CPU cores for user application threads and interrupt vectors in a NUMA topology can affect performance. Finally, we confirmed that if consuming completion events is not done on the same process thread as submissions, then `COOP_TASKRUN` can result in significantly worse (1000 x) latency. Apart from that case, we recommend always setting the flag, as it either does nothing, or improves performance.

4.2 Forced Asynchronous Submission (IOSQE_ASYNC)

`io_uring` provides multiple options for the submission of I/O requests. By default, `io_uring` tries to submit requests in non-blocking mode, which does not make use of worker threads. However, if requests fail, `io_uring` may try to submit them again but through a worker

thread. In addition, `io_uring` provides modes for polling for submissions. In this experiment, we explore the `IOSQE_ASYNC` flag, which forces `io_uring` to do submissions asynchronously, by running them on a worker thread. The man page provides the following description of the `IOSQE_ASYNC` flag.

> Normal operation for io_uring is to try and issue an sqe as non-blocking first, and if that fails, execute it in an async manner. To support more efficient overlapped operation of requests that the application knows/assumes will always (or most of the time) block, the application can ask for an sqe to be issued async from the start. Available since 5.6.

In this experiment, our goal is to understand under what conditions this flag is beneficial. We will explore what can cause non-blocking submission to fail, whether asynchronous submission can improve performance and under what conditions, and explore how the `io_uring` worker pool works.

An `io_uring` SQE (submission queue event) is normally submitted in a non-blocking mode, which means that the user application is not blocked while the I/O request is in flight. However, before the request is considered in flight, the submission goes through the kernel's block layer, e.g. page cache, file system, and I/O scheduler. This submission path is executed in the context of the application, and the syscall that submitted the request (`io_uring_enter`) only returns after the request has been submitted to the device. In that sense, the submission of a request is still blocking, it is only the in flight part that is non-blocking. As we will see, the submission can block for an undetermined time, for example due to inode lock contention within the file system, or due to expensive page cache operations.

4.2.1 Function of IOSQE_ASYNC

As the description from the man page is rather vague, we begin by exploring the kernel source code to see how the `IOSQE_ASYNC` flag is used. We are interested in what is meant by a "submission failure", how worker threads are handled, what kind of requests are submitted as non-blocking, and how setting the `IOSQE_ASYNC` flag changes the submission of a request.

The request submission starts with a `io_uring_enter` syscall from the user application, with the `to_submit` parameter set to a positive value. Let's go through what happens on the kernel side during this system call.

1. The entry point of this syscall is the `SYS_io_uring_enter` function. If the `to_submit` parameter is set, then `io_submit_sqes` is called, which then calls `io_submit_sqe` for each SQE to submit.

2. In `io_submit_sqe`, the action depends on the `REQ_F_FORCE_ASYNC` flag, which is equivalent to the `IOSQE_ASYNC` flag. If the flag is not set, the SQE is passed to `io_queue_sqe`, but otherwise to `io_queue_sqe_fallback`, which then calls `io_queue_iowq` to pass the request to a worker thread.

3. We assume that the flag is not set and `io_queue_sqe` is called. This calls `io_issue_sqe` with a `IO_URING_F_NONBLOCK` flag. If the issue call returns non-zero, the SQE is passed to `io_queue_async`, which may be the failure case that the man page refers to.

4. In `io_issue_sqe`, the request opcode is looked up in the `io_issue_defs` array. Each opcode has an associated `issue` callback. For example, the `readv` opcode has `io_read` as its `issue` callback. `io_issue_sqe` calls the associated callback and forwards the return value back.

5. In `io_read`, a `kiocb` object is created and passed to the VFS through the `io_iter_do_read` function. A `kiocb` object (kernel I/O callback) represents a general I/O request in the kernel. We notice that `io_read` checks whether the `IO_URING_F_NONBLOCK` flag is set, and in that case adds a `IOCB_NOWAIT` flag to the `kiocb` object.

6. We can see an example of how `IOCB_NOWAIT` is used by looking at the `ext4_dio_read_iter` function. Every file inode in the Linux kernel has a lock. If the flag is set, the lock is checked with a `trylock` operation, which immediately fails if the lock is set, and `ext4` returns an `EAGAIN` error, telling the caller to try again later. If the flag is not set, a `lock` operation is used instead of `trylock`, which will block until the lock is acquired. This lock operation can therefore block the `io_uring_enter` system call for an undetermined time.

7. Assuming that `EAGAIN` is returned, this error is propagated back up to `io_read`, `io_issue_sqe`, and `io_queue_sqe`. From `io_queue_sqe`, the SQE is passed to `io_queue_async` as described in step 3.

8. In `io_queue_async`, the `io_arm_poll_handler` function is called which "arms" a VFS poll handler through the `vfs_poll` function. Registering a poll handler results

in the process being woken up when the file descriptor becomes ready to perform I/O, similar to the `poll` system call. A `task_work` is added that will perform the I/O when the process is woken up by the poll handler.

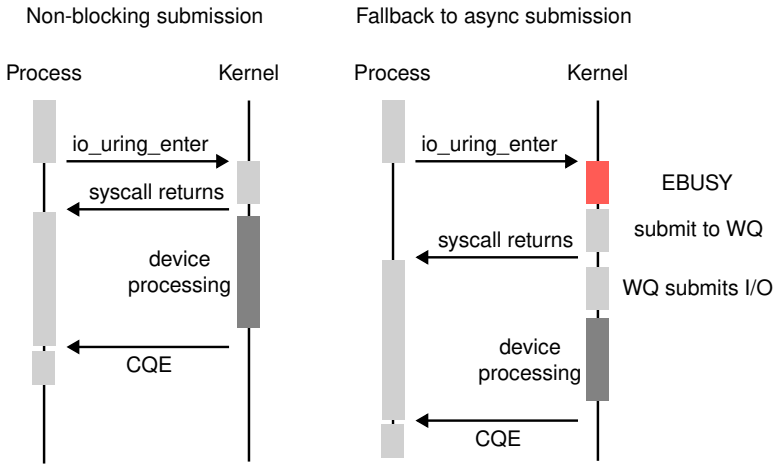


Figure 4.23: Comparison of a a successful non-blocking submission and submission that falls back to an asynchronous worker

Figure 4.23 shows the difference between a successful non-blocking submission, and a submission that falls back to an asynchronous worker thread. In the latter case, the non-blocking submission fails with an `EAGAIN` error, and the SQE is submitted asynchronously. This part happens in-line, so the `io_uring_enter` system call takes longer to return.

4.2.2 Worker Pools

We also explore how `io_uring` uses worker thread pools, known as `io-wq`, to make asynchronous requests. For example when the `IOSQE_FORCE_ASYNC` flag is set and an SQE is passed to `io_queue_iowq`. The SQE is then passed to `io_wq_enqueue` which adds a `struct io_wq_work` into a worker thread's work queue. If the current worker count is below the `max_workers` limit, a new worker will be created. The worker threads consume from their work queue and are destroyed after a 5 second timeout. This means that workers are recycled and not created for every SQE.

Each `io_uring` ring can have two worker pools, for bounded and unbounded work. If the request's file descriptor has the type `S_ISREG` (a regular file) or `S_ISBLK` (a block device), then the SQE goes to the bounded worker pool. Other file descriptors, such as sockets (`S_ISSOCK`) go to the unbounded worker pool. For the unbounded worker pool, the maximum worker count (`max_workers`) is the maximum limit of processes on the system

(RLIMIT_NPROC), but for the bounded pool, it is the lower value of four times the CPU count (num_online_cpus), or the queue depth of the SQ ring.

4.2.3 File Inode Locks

As we discovered in the previous section, io_uring sets the IOCB_NOWAIT flag on kiocb objects. We found out that this flag results in file systems attempting to take inode lock, and if that fails, forcing io_uring to register a poll handler and submitting the request asynchronously when the file descriptor becomes available. In a later section, we will explore how lock contention affects the performance of io_uring, and whether the IOSQE_ASYNC flag becomes beneficial in that case.

The inode lock is stored in the i_rwsem attribute of the struct inode structure. The lock is a reader/writer semaphore, meaning that multiple readers can hold the lock, but a writer must hold the lock exclusively. This ensures that if a file is opened by multiple processes for writing, the writes will happen serially within the kernel.

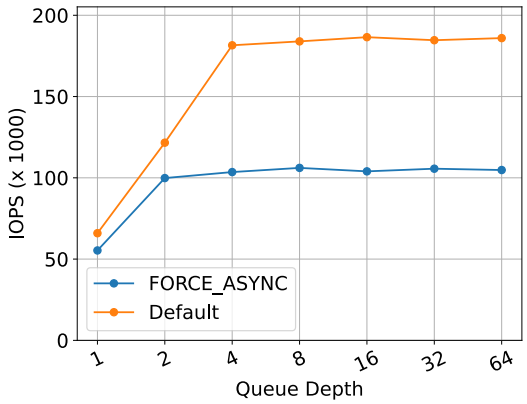
In the following section, we will use eBPF to confirm that IOCB_NOWAIT is set for non-blocking submissions, but not when submissions come from a worker thread. We will also use eBPF to confirm that EAGAIN is returned if there is lock contention during non-blocking submission.

4.2.4 First Experiment with IOSQE_ASYNC

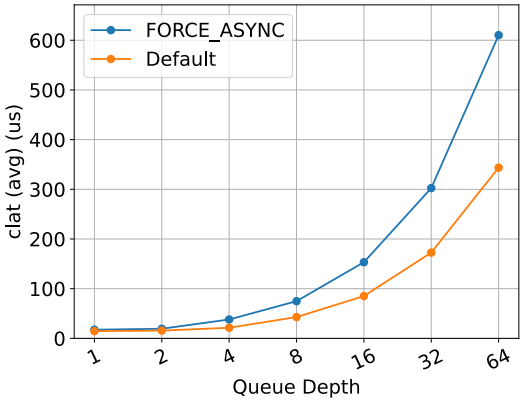
For our first experiment, we explore what difference IOSQE_ASYNC makes for read requests. We have implemented a flag to enable IOSQE_ASYNC in our liburing based fio engine. We run fio with random reads and a single worker on a single Intel Optane NVMe device, and we set the CPU affinity to core 0.

Figure 4.24 shows the IOPS, average, and p99 latencies for different queue depths, with and without the IOSQE_ASYNC flag. We see that forcing asynchronous submissions results in a much lower throughput and higher latencies. It is likely that the single CPU core is the bottleneck, and creating and communicating with workers may be more CPU intensive.

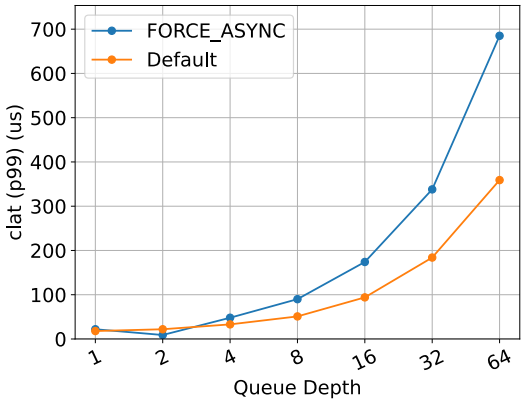
We now run the same experiment again but without the CPU affinity. We still run fio with a single worker so we expect a similar performance without worker threads. Figure 4.25 shows that without CPU affinity, using IOSQE_ASYNC results in a higher throughput and lower latencies, unlike the previous experiment. It is likely that in this case, the worker threads are running on different CPU cores. Thus IOSQE_ASYNC can help to offload I/O submission to different CPU cores, even if our application itself is single threaded.



(a) IOPS



(b) completion latency (clat) - average

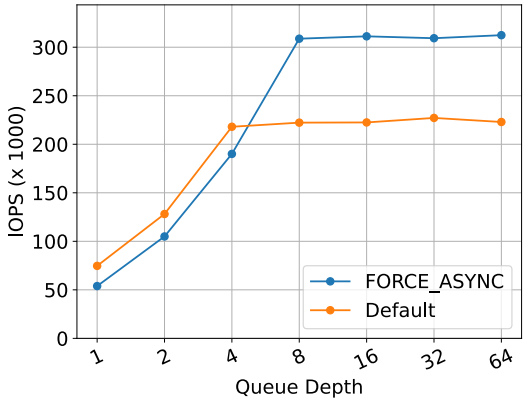


(c) completion latency (clat) - p99

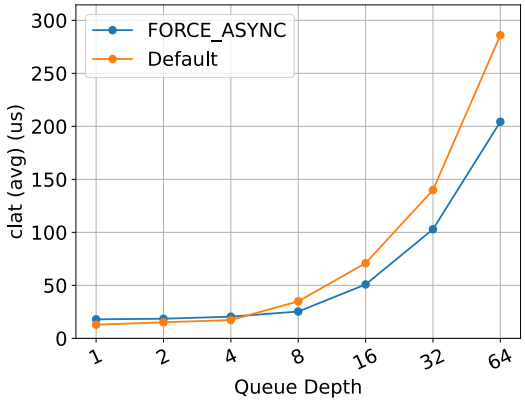
Figure 4.24: IOPS, average and p99 latency with and without `IOSQE_ASYNC`, single `fio` thread pinned to a single core, random reads

4.2.5 Counting Worker Thread Operations with eBPF

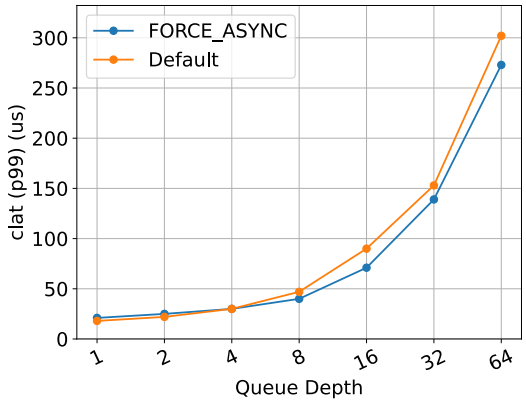
To better understand how `IOSQE_ASYNC` uses worker threads in our previous experiments, we can run eBPF to count important worker operations. Table 4.3 shows counts for the creation of workers, which happens through the `create_io_worker` function, and number of work that is processed by workers, through the `io_worker_handle_work` function. This confirms that worker threads are only used when `IOSQE_ASYNC` is set. It also shows that the number of threads created is much lower than the number of requests, showing that the threads are heavily recycled.



(a) IOPS



(b) completion latency (clat) - average



(c) completion latency (clat) - p99

Figure 4.25: IOPS, average and p99 latency with and without `IOSQE_ASYNC`, single `fio` thread without pinning, random reads

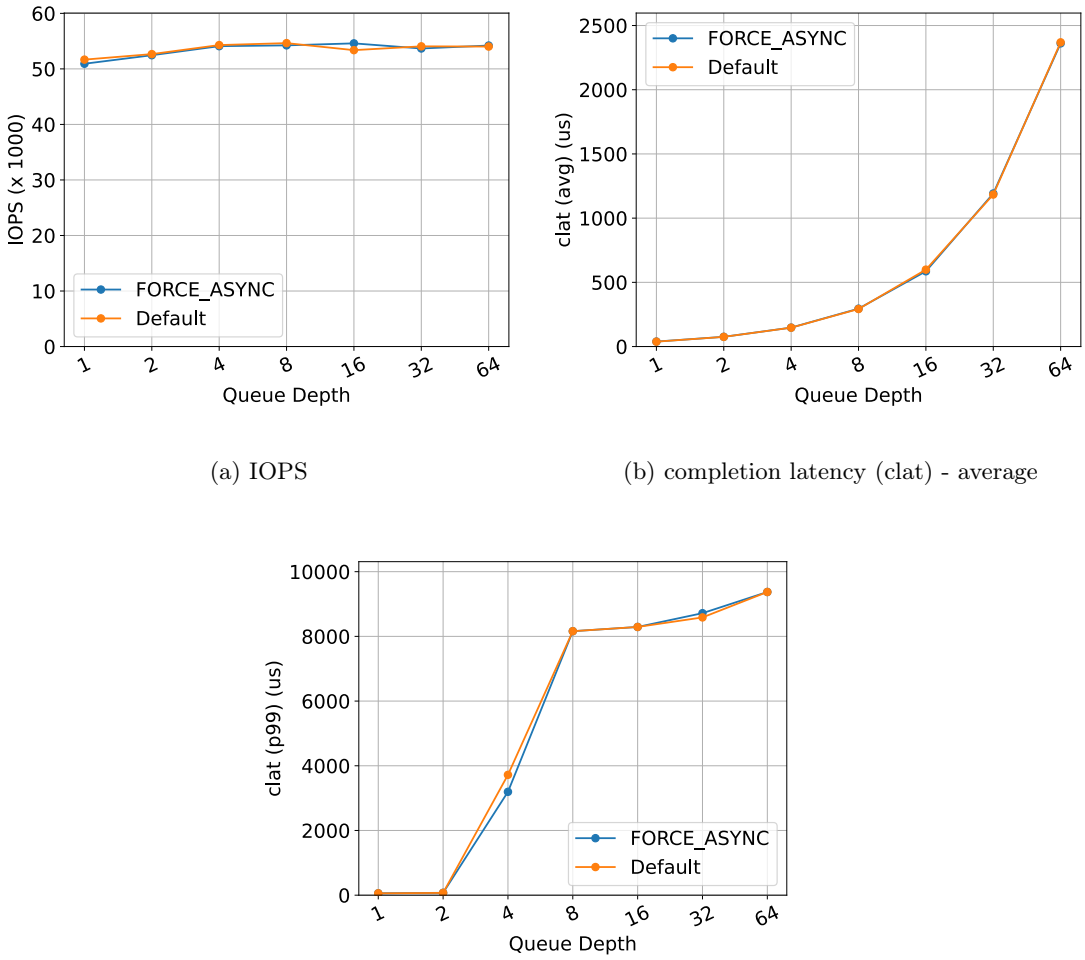
4.2.6 Experimenting with IOSQE_ASYNC and Lock Contention

The man page for `io_uring` hints that `IOSQE_ASYNC` is beneficial when we know that non-blocking submission will frequently fail. In a previous section, we explored the usage of `IOSQE_ASYNC` in the `io_uring` source code, and noticed that if a `trylock` operation fails on an inode lock, `io_uring` will fallback to asynchronous submission with the help of VFS polling.

To introduce lock contention, we run `fio` with two workers doing random writes on the same file. We create an `ext4` file system on an Intel Optane NVMe drive. Figure 4.26

	create_io_worker	io_worker_handle_work
single core	0	0
single core + async	64	29,360,383
all cores	0	0
all cores + async	5,178	53,682,699

Table 4.3: eBPF function call counts for worker threads



(a) IOPS

(b) completion latency (clat) - average

(c) completion latency (clat) - p99

Figure 4.26: IOPS, average and p99 latency with and without `IOSQE_ASYNC`, two `fio` threads doing random writes to the same file on an `ext4` file system

shows IOPS, average and p99 latencies for different queue depth, with and without the `IOSQE_ASYNC` flag. We are unable to draw a meaningful conclusion from this experiment,

	Default	FORCE_ASYNC
Success	3,051,615	3,139,685
Failure (EAGAIN)	3,051,615	0

Table 4.4: eBPF counts for `EAGAIN` errors in the `io_issue_sqe` function, with and without the `FORCE_ASYNC` flag

we suspect that is because write performance on Intel Optane devices is less predictable than read performance.

We also run the experiment for $QD = 64$ with an eBPF script to count the return value of `io_issue_sqe`. Table 4.4 shows the counts for two different return values, zero (success), and `EAGAIN` (try again later). We see that if `FORCE_ASYNC` is not set, every SQE first results in an `EAGAIN`, and then success, which can be attributed to inode lock contention. If `FORCE_ASYNC` is set then all SQEs are issued from a worker thread, where the inode lock operation is allowed to block.

4.2.7 IOSQE_ASYNC and Lock Contention with Readers and Writer

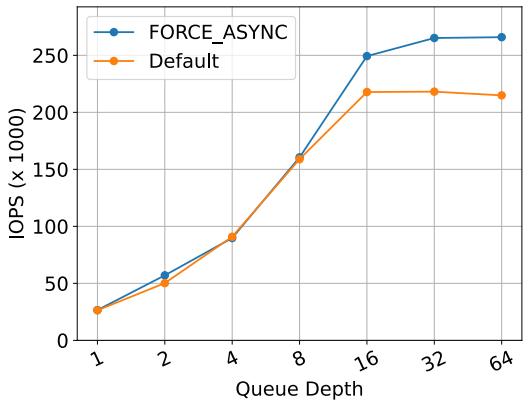
As we could not draw a conclusion from the previous locking experiment, we modify the experiment to observe lock contention in `fio` reader threads, while a `fio` writer thread runs on the same file in the background. As before, we use an `ext4` file system on an Intel Optane device. The `fio` worker performs random read, while the background worker performs random writes.

	Default	FORCE_ASYNC
No IOCB_NOWAIT	0	19,075,609
With IOCB_NOWAIT	10,818,768	0

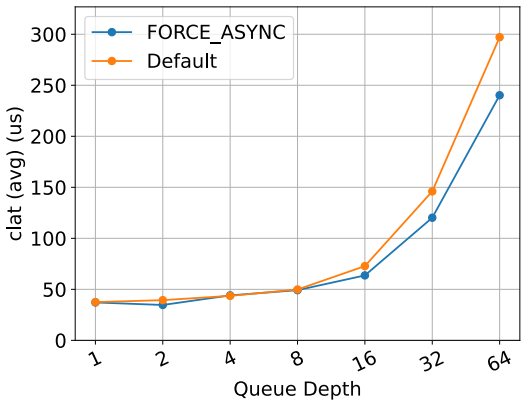
Table 4.5: eBPF counts for whether `IOCB_NOWAIT` flag is set on `kiocb` objects passed to `ext4_file_read_iter`, depending on whether `FORCE_ASYNC` is set on SQE, showing that `IOCB_NOWAIT` is not set for async requests

Figure 4.27 shows the IOPS, average and p99 latencies for different queue depths, with and without the `IOSQE_ASYNC` flag. This time, we observe that `IOSQE_ASYNC` gives up to 15% higher IOPS at higher queue depths, and lower latencies. However, at queue depths 1 and 2, async submission produces worse results.

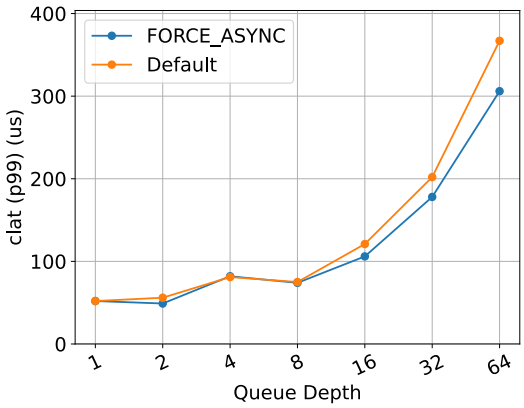
Finally, we use `eBPF` to observe whether the `IOCB_NOWAIT` flag is set on `kiocb` objects. As we discovered in a previous section, this flag controls whether file systems do a `trylock`



(a) IOPS



(b) completion latency (clat) - average



(c) completion latency (clat) - p99

Figure 4.27: IOPS, average and p99 latency with and without `IOSQE_ASYNC`, one `fio` thread doing random reads to a file on an `ext4` file system, while a background thread does random writes to the same file

operation on the inode lock and return `EAGAIN`, or do a `lock` operation and block until getting a lock. Table 4.5 shows the count of calls to the `ext4_file_read_iter` function, depending on whether the `IOCB_NOWAIT` flag is set on the passed `kiocb` object. As we can see, `IOCB_NOWAIT` is never set during async submission, but always set for non-blocking submission.

4.2.8 Conclusion

In this experiment we have investigated the `IOSQE_ASYNC` flag. We have confirmed that setting this flag forces I/O requests to be submitted from an `io_uring` kernel worker thread. We have explored how `io_uring` manages worker pools, how it decides what requests should be attempted as non-blocking first, and we discovered a scenario that causes `io_uring` to fallback from non-blocking to asynchronous submission, namely file system inode lock contention.

Our experimental results reveal two cases where `IOSQE_ASYNC` can be beneficial to performance. First, when an application is limited to running on a single thread, but the system has other CPU cores that are under-utilized, `IOSQE_ASYNC` can help offload work to other CPU cores. Second, as hinted by the man page, when we know non-blocking submission is likely to fail, `IOSQE_ASYNC` can improve performance, compared to having frequent failures in non-blocking submission. A downside of `IOSQE_ASYNC` is that worker threads use more CPU resources, and thus in a CPU bound environment, the flag can cause decreased performance.

4.3 Registered Files (IOSQE_FIXED_FILE)

`io_uring` provides a feature called registered files, also known as fixed files, or direct descriptors. When submitting a requlat SQE, the file must be opened by the application first, and the file descriptor (`fd`) specified on the SQE. As we will explore in this section, submitting I/O on regular descriptors requires some setup work in the kernel for each I/O. Registered files are intended to prevent that repetitive setup work.

To submit I/O on a registered file, the file must first be registered in the ring, by using the `io_uring_register` system call with the `IORING_REGISTER_FILES` flag. After the file has been registered, an SQE can be created with the `IOSQE_FIXED_FILE` flag, and instead of a regular file descriptor, the index of the registered file is used.

In this section, our goal is to understand the benefits of using registered files, how the kernel handles them internally, and how that differs from using regular files.

4.3.1 Function of Registered Files

We are interested in what effect registered files have internally in the kernel. We first look at the submission side, and see that the `IOSQE_FIXED_FILE` flag is used internally as `REQ_F_FIXED_FILE`. For each I/O submission, the `io_submit_sqe` function makes a call to

`io_assign_file` to retreive a `struct file` for the given file descriptor on the SQE, and assigns it to the `kiocb` object. The `io_assign_file` function handles regular files and registered files differently.

- Regular files are handled through `io_file_get_normal`, which calls `fget`, a function that belongs to the kernel's file system layer, and returns a `struct file` for a given file descriptor.
- Registered files are handled through `io_file_get_fixed`, which looks up the registered file's index (stored in the file descriptor field) in the rings internal file table and returns a `struct file`.

The `fget` function looks up a file descriptor in the current task's table of open files. The task's open files are stored in its `task_struct` under the `files_struct` property, which contains an array of `struct file` that is indexed by file descriptor. As the file table can be shared by multiple processes on different threads, the RCU (read-copy-update) synchronization mechanism is used. In addition to looking up the file descriptor, `fget` also increments the reference count (`f_count`) on the file, using an atomic operation. When an operation has finished, the `fput` function is called to decrease the reference count.

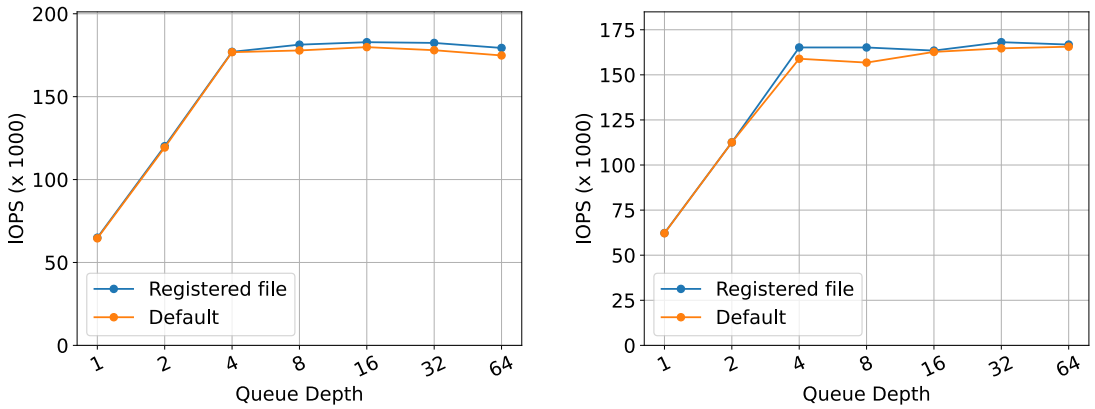
We use eBPF to confirm that using registered files prevents repeated calls to `fget`. Table 4.6 shows the number of `fget` calls, with and without using registered files. The counts were collected while running `fio` with the conditions that are described in the next section.

	fget call count
Registered file	731
Regular file	12,321,840

Table 4.6: eBPF counts for `fget`, with and without using registered files

4.3.2 Experimental Results

We have added support for registered files in our `fio liburing` I/O engine. We run `fio` with a single worker on a single Intel Optane NVMe device, and we set the CPU affinity to core 0. Figure 4.28 shows the IOPS for different queue depths, with and without registered files, for I/O on a raw block device and an `ext4` file system, using a single Intel Optane device. Figure 4.29 shows the average latency for the same cases. For I/O on a raw block



(a) Block device

(b) File system (`ext4`)

Figure 4.28: Throughput (IOPS) for a raw device and `ext4` file system, with and without a registered file, random reads

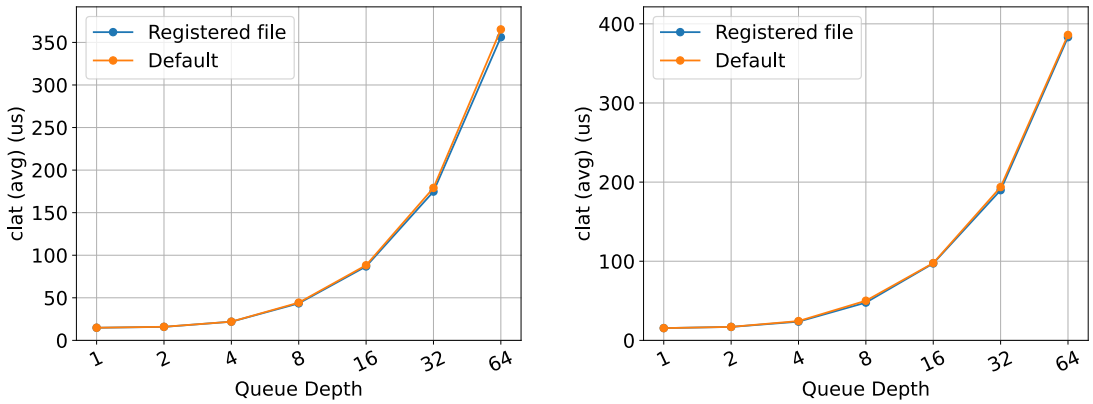
device, we observe up to 2.6% increase in throughput with a registered file, and 2.5% decrease in average latency. For the file system (`ext4`), we do not get conclusive results.

4.3.3 Other Factors

In our experiment, we observed a 2.6% increase in IOPS when using registered files. We are now interested in whether there may be cases that result in a higher performance increase, or in a performance decrease. We notice the following hint in the `liburing` documentation for the `io_uring_register_files` function. That if the process has created any threads, the cost of reference counting goes up (22).

> Registered files have less overhead per operation than normal files. This is due to the kernel grabbing a reference count on a file when an operation begins, and dropping it when it's done. When the process file table is shared, for example if the process has ever created any threads, then this cost goes up even more. Using registered files reduces the overhead of file reference management across requests that operate on a file.

This motivates us to look into the case where a process file table is shared. As we explored earlier, the `fput` function increments a file's reference count by using an atomic operation (`atomic_long_inc`), which contains a memory barrier. This memory barrier



(a) Block device

(b) File system (`ext4`)

Figure 4.29: Average latency for a raw device and `ext4` file system, with and without a registered file, random reads

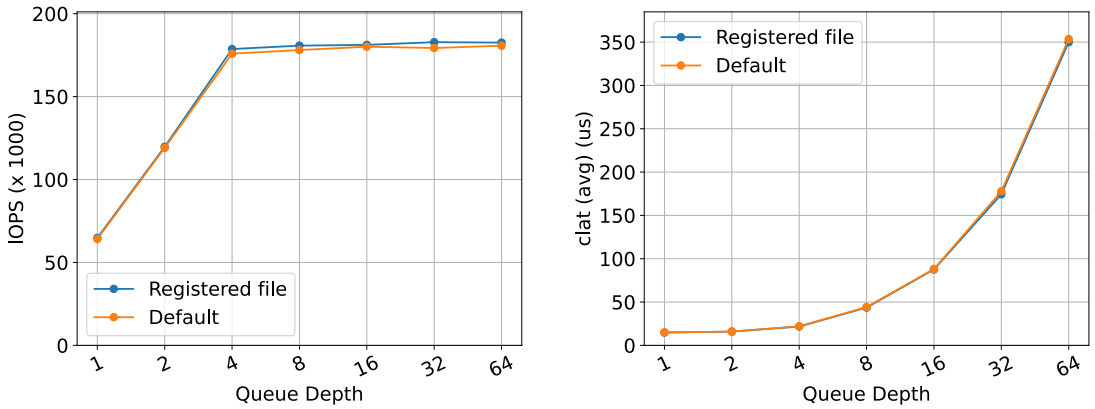
might cause a performance decrease if the cache line containing the counter is being used on multiple CPU cores.

We extend our fio engine to have each worker thread fork an additional thread that does an infinite no-op busy-loop. We use the same setup as in the previous experiment, running on a raw block device, but we set the CPU affinity to two CPU cores, allowing the fio worker and busy loop to run on separate cores.

Figure 4.30 shows the IOPS and average latency for different queue depths, with and without registered files. The results show a similar performance difference as in the previous experiment, so the introduction of the busy-loop thread did not affect performance. This does not prove that the memory barrier does not affect performance, but it might be that our busy-loop thread is too simple to affect it.

4.3.4 Conclusion

Registered files provide a slight performance increase. However, we are unable to find a case where registered files degrade performance, and we could also not find a case where the performance increase is more significant, despite a hint that we found in documentation.



(a) IOPS

(b) completion latency (clat) - average

Figure 4.30: IOPS and average latency, with and without a registered file, with additional thread, random reads

4.4 Submission Queue Polling (SQ_POLL)

Submission queue polling is a feature of `io_uring` that can replace the need for expensive `io_uring_enter` system calls to submit SQEs. By configuring the ring with the `SQ_POLL` flag, a separate kernel thread is created that constantly polls the submission queue for new entries. Previous work has found that SQ polling can provide significant performance improvements.

The nature of polling requires one or more CPU cores to be dedicated to polling for new events, any hick-ups such as interrupts or scheduling of other processes will block the submission of new events. In addition, a polling CPU core can be either overloaded, (requiring more CPU cores for polling), or underutilized (e.g. wasting a whole CPU core polling). In this section, we explore the performance benefits of SQ polling, the internal implementation of the polling thread, and configuration options for the polling thread. We also explore how an imbalance between user threads and polling threads can lead to wasted CPU resources.

4.4.1 Behavior of Polling Threads

To make use of submission queue polling, the `io_uring_setup` system call is called with the `IORING_SETUP_SQPOLL` flag. In addition, we may set the `IORING_SETUP_SQ_AFF` flag and set the CPU affinity of the kernel thread. The polling thread continuously peeks the

submission queue (SQ) and submits any new SQEs that it finds.

In a previous experiment, we explored asynchronous worker threads, that are created using the `create_io_thread` function. SQ polling threads do not share the async worker system, apart from also being created using the `create_io_thread` function. This happens only one time, in the `io_sq_offload_create` function, which is called during the `io_uring_setup` system call. We will use eBPF to confirm that this happens when submission queue polling is enabled.

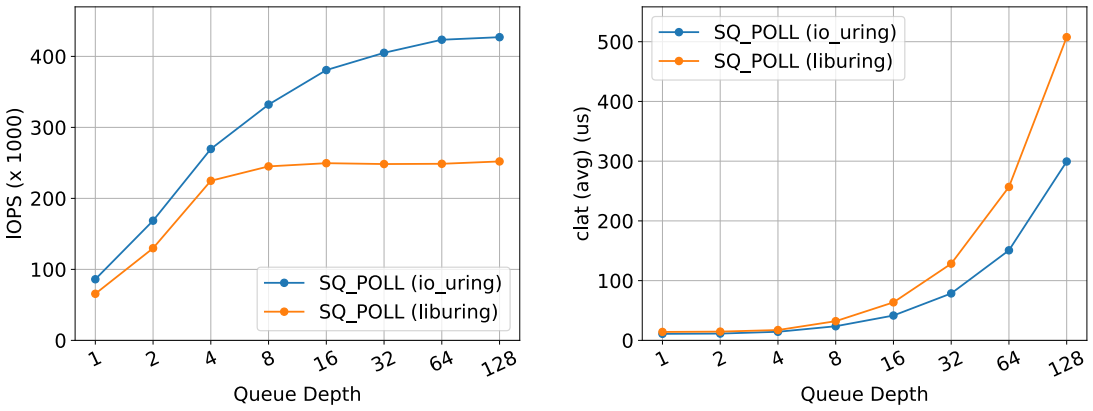
SQ polling also has a timeout feature, so that if the process does not submit SQEs in a certain period, the polling thread goes to sleep. The timeout value can be specified in `sq_thread_idle` (in milliseconds) when setting up the ring. If the value is not specified, the default value is 1 second. If the SQ polling thread goes to sleep, the user process is responsible for waking it up again. The ring's `flag` property will have the `IORING_SQ_NEED_WAKEUP` set if a wake-up is needed, and then `io_uring_enter` must be called be the user process to wake up the thread.

4.4.2 Limitations of liburing

When attempting to implement SQ polling in our liburing fio engine, we came across a problem. When fio commits an I/O unit, we use liburing's `io_uring_submit` function. Without SQ polling, this function returns the number of SQEs that were submitted, and fio can subtract them from the number of queued SQEs. However, when using SQ polling, this function may report a higher number than the actual submit count. This unexpected behavior has been documented before (23).

This is a problem for applications that need to keep track of the number of submitted requests, such as fio. One solution is to use the `io_uring_sq_ready` function from liburing to get the number of requests that have not been submitted yet. However, this function relies on a memory barrier for synchronization, making it more expensive.

We have implemented SQ polling in our fio engine with the help of `io_uring_sq_ready`. However, when comparing our `liburing` engine to the built-in `io_uring` engine, we see that our engine does not provide the same performance. Figure 4.31 shows IOPS and average latency for different queue depths, running with a single fio thread and a kernel polling thread running on another core. We can see that the throughput is up to 40% lower and latency is also higher. We are unable to optimize our `liburing` engine, and we will thus use the built-in `io_uring` engine for our SQ polling experiments.



(a) IOPS

(b) Completion latency (clat) - average

Figure 4.31: IOPS and average latency for different queue depths comparing our `fio liburing` engine to the built-in `io_uring` engine

4.4.3 Evaluation of Submission Queue Polling

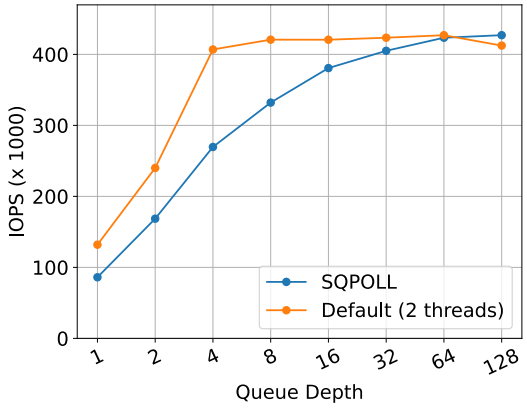
For our first experiment, we will run fio with a single thread and SQ polling enabled (thus using 2 threads including the kernel thread) and compare it with running fio with two normal threads (without SQ polling). We run a random read benchmark on a single Intel Optane device, and in both runs we pin the threads to the same CPU cores.

Figure 4.32 shows IOPS, average and p99 latency for the two cases. We can see that both configurations converge to roughly the same throughput, although running two normal threads is faster with lower queue depths. However, SQ polling results in significantly lower latency at higher queue depths. The average latency at $QD \geq 64$ is around 50% lower when using SQ polling.

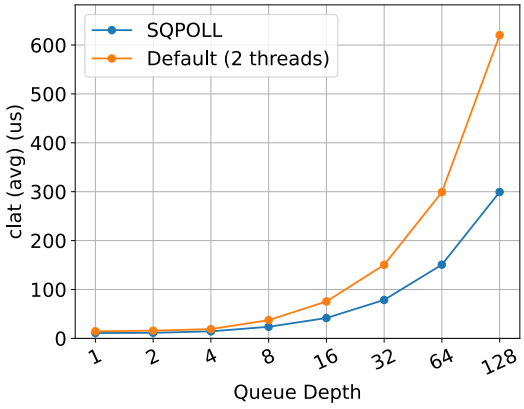
4.4.4 Exploring Polling Efficiency

For the second experiment, we are interested in the balance between the user application thread and the kernel polling thread. Assuming that the two threads are pinned to separate CPU cores, one of them is likely to become a bottleneck before the other, and thus leaving wasted CPU time on the other core.

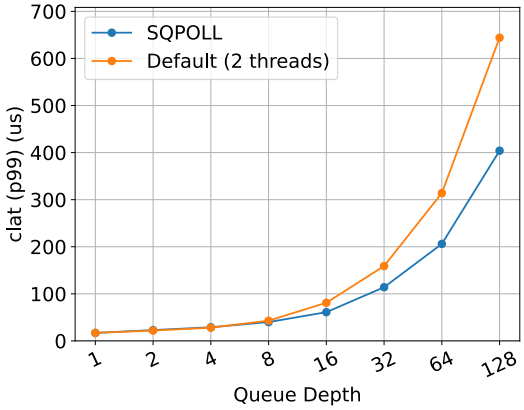
Fio provides the `thinkcycles` configuration parameter to simulate CPU load, as most real world applications are a mix of CPU and I/O work, unlike fio which normally submits I/O continuously with minimal CPU work on the user space side. For this experiment,



(a) IOPS



(b) Completion latency (clat) - average



(c) Completion latency (clat) - p99

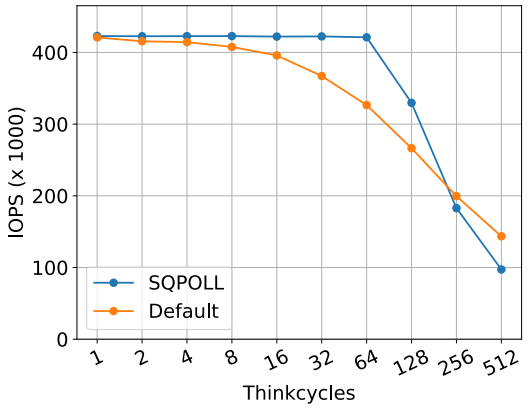
Figure 4.32: IOPS, average and p99 latency for different queue depths, comparing one `fio` thread with `SQPOLL` (a kernel thread) with 2 normal `fio` threads, random reads

we run fio with the same configuration as before, but this time we gradually increase the `thinkcycles` value. We keep the queue depth fixed at 64.

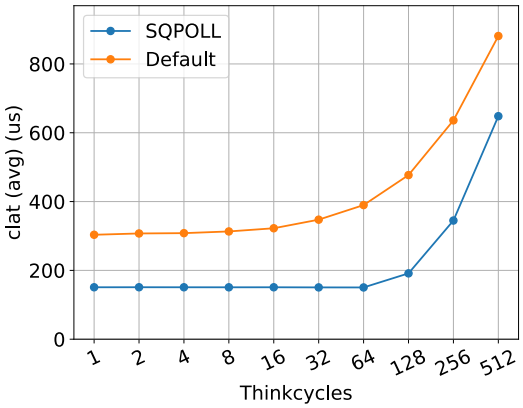
Figure 4.33 shows IOPS, average and p99 latency for different `thinkcycles` values, comparing a single thread with SQ polling, and two normal threads. First, the IOPS results show that adding thinkcycles with SQ polling does not impact throughput until a certain threshold (after 64 thinkcycles), but adding thinkcycles impacts the two normal threads immediately. In both cases, we see much lower latency with SQ polling, as in the previous experiment.

	kthreads created	io_uring_enter
Normal	0	12,855,461
SQ polling	1	7

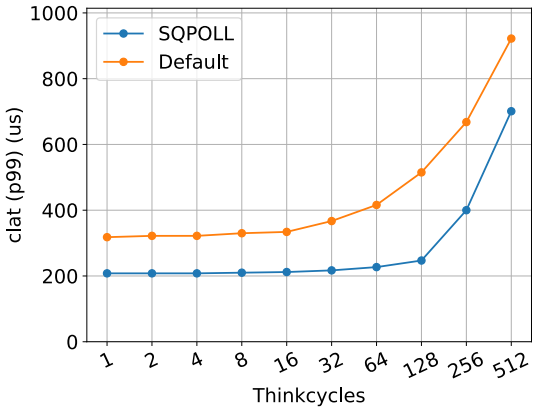
Table 4.7: eBPF counts for kthreads created and `io_uring_enter` system calls, comparing SQ polling with normal operation



(a) IOPS



(b) Completion latency (clat) - average



(c) Completion latency (clat) - p99

Figure 4.33: IOPS, average and p99 latency for different `thinkcycle` values, comparing one `fio` thread with `SQPOLL` (a kernel thread) with 2 normal `fio` threads, random reads

The fact that adding thinkcycles did not impact throughput until reaching a threshold means that the CPU core running the user thread (fio) was not fully utilized. In other words, `io_uring` does more work within the kernel polling thread than in the user thread.

Finally, we also used eBPF to confirm that a kernel polling thread is being created, and that the number of system calls drops. Table 4.7 shows that when using SQ polling, a single kernel thread is created, and the number of `io_uring_enter` system calls drops almost to zero.

4.4.5 Conclusion

Our two experiments show that SQ polling provides very different performance characteristics. First, SQ polling may not necessarily provide better throughput, but it shows significantly better latency. At lower queue depths, SQ polling may provide worse throughput. Secondly, SQ polling can have different results depending on the CPU intensity of the user thread, too little CPU intensity may leave one CPU core underutilized. On the other hand, too much CPU intensity may leave the kernel polling thread underutilized, in that case, it may be beneficial to share the kernel polling thread with multiple `io_uring` instances.

5

Evaluation with RocksDB

In the previous section, we explored several `io_uring` features with small `fio` benchmarks and found that adjusting the `io_uring` configuration can improve I/O throughput and latency under certain conditions. However, `fio` benchmark results may not be representative of real-world applications, as it is only designed to submit as much I/O as possible, with minimal CPU workload. In this section, we implement a `liburing` storage backend for the RocksDB key-value store and evaluate how the same `io_uring` features affect its performance.

5.1 How Does RocksDB Work?

RocksDB is a high-performance embeddable database for key-value data that is optimized for flash storage. It was developed by Facebook in 2012, and is based on LevelDB, a previous embeddable key-value store from Google. RocksDB is primarily based on a data structure known as Log-Structured Merge (LSM) tree, which results in flash-friendly sequential I/O operations. It provides a flexible API that supports a broad range of applications, and can be used as a backend for many databases (e.g. MySQL, Cassandra, and MongoDB), but also for search indexes, and caching services. It also includes a powerful benchmarking tool called `db_bench`. This makes RocksDB a popular target for storage research, and as it supports writing custom storage backends, we decide to use it for our evaluation (24, 25).

The basic key-value operations provided by RocksDB are `put`, `get`, and `delete`. Both keys and values are treated as byte arrays and can be of variable length. In addition, an `iterator` operation is provided to scan over a range of keys. RocksDB arranges keys in sorted order, and applications can optionally specify their own comparator function.

RocksDB also provides a `multiget` operation, that allows fetching multiple keys in one operation, which can be faster than a loop of `get` operations (24).

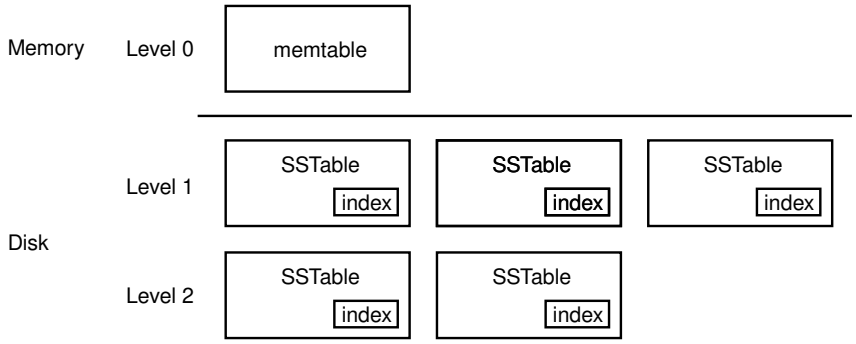


Figure 5.1: The levels of an LSM tree

LSM trees are the primary data structure used by RocksDB. An LSM tree consists of a series of levels, where each level is in sorted order, and the levels are periodically merged. When inserting into an LSM-tree, the value is written into an in-memory part known as a memtable. When the memtable has reached a maximum size, it is written to disk into an SSTable (Sorted String Table) in level 0. An SSTable stores the data in sorted order, and also contains an index for fast lookup. When a level in the LSM tree has reached a maximum size, some of its SSTables are moved one level down, and merged with overlapping SSTables in that level. LSM trees are well suited for write-heavy workloads, and as SSTables are always written sequentially, they result in good performance on flash storage. Figure 5.1 shows the structure of an LSM tree, with a memtable stored in memory, and increasingly larger levels of SSTables stored on disk (25).

5.2 Implementing a RocksDB Backend with liburing

RocksDB provides an API to write plugins, including a file system API. To write a file system plugin, we must extend the `FileSystem` class, and implement various metadata operations, in addition to file abstractions such as `SequentialFile`, `RandomAccessFile`, and `WritableFile`. For our `liburing` plugin, we extend the `FileSystemWrapper` class, which includes all operations from the default storage backend, allowing us to extend only the operations that we need. For the benchmark that we will run, we only need to

customize the `RandomAccessFile`, so we start with a copy of the POSIX version, and then customize the `io_uring` initialization to support the features that we want to evaluate.

The default RocksDB POSIX storage backend already uses `io_uring` for some operations, using functions from `liburing`. We discover that a ring is created with a constant queue depth of 256, and no configuration flags passed. In the default backend, we only find `io_uring` usage in the `MultiRead` function, which is used when applications use the RocksDB `multiget` operation. Running `db_bench` with the `readrandom` benchmark does not result in `io_uring` calls, however, the `multireadrandom` benchmark does. The fact that the RocksDB key-value API is synchronous may explain why there is limited async support in the file system layer.

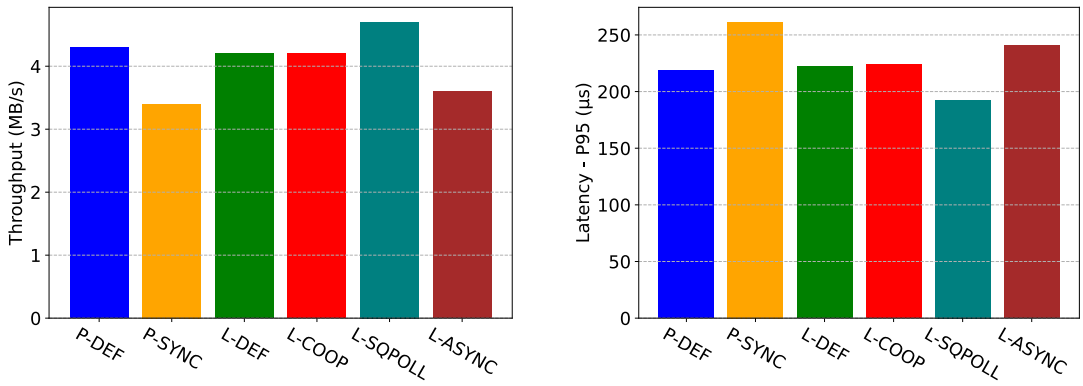
We use eBPF to observe the I/O system calls used by the POSIX backend and by our `liburing` backend. It confirms that `multiget` operations result in `io_uring` usage in the POSIX backend, and that the configuration flags we use in our backend are used, e.g. worker threads are created, and IPIs are prevented.

5.3 Evaluation Results

For our evaluation, we create an `ext4` file system on an Intel Optane device. We then create a RocksDB database with the `fillrandom` benchmark, creating a database of 10 million key-value pairs, with keys being 100 bytes, and values 100 bytes. We then run the `multireadrandom` benchmark for each backend configuration. We disable compression, enable direct reads, and always run benchmarks with the same random seed.

In addition, we pin the RocksDB process to a single CPU, and we configure the NVMe driver so that interrupts do not come in on that core. As we saw in chapter 4, running on a non-interrupt CPU core is needed to see improved performance with `io_uring`'s `COOP_TASKRUN` flag. For experiments with a kernel polling thread, we also pin the kernel thread to a non-interrupt core, different from where RocksDB runs.

We run experiments with 6 backend configurations, 2 with RocksDB's default `io_posix` backend, and 4 with our custom `liburing` backend. For the default backend, we run using default configuration (**P-DEF**), and with `io_uring` disabled (**P-SYNC**). For our backend, we run with a ring in the default configuration (**L-DEF**), with cooperative task running enabled (**L-COOP**), with SQ polling enabled (**L-SQPOLL**), and with forced async submission enabled (**L-ASYNC**). We were unable to get registered files working in our backend, so we skip that experiment.



(a) Throughput

(b) P95 Latency

Figure 5.2: Throughput (MB/s) and p95 latency (μs) for each RocksDB I/O backend configuration, running the `multireadrandom` benchmark

Figure 5.2 shows the results for each backend configuration. The left figure shows throughput in megabytes per second and the right figure shows 95th percentile latency in microseconds. First, we observe that the default backend performs better with `io_uring` enabled (which it is by default), with 26.5% higher throughput and 16.5% lower latency. Second, we see that our backend gives the same performance as the default backend, both with the default configuration, and with the `COOP_TASKRUN` flag. Using submission queue (SQ) polling gives the best performance, with 11.9% higher throughput and 13.1% lower latency compared to the default configuration. However, using forced asynchronous submission in our backend gives a worse performance.

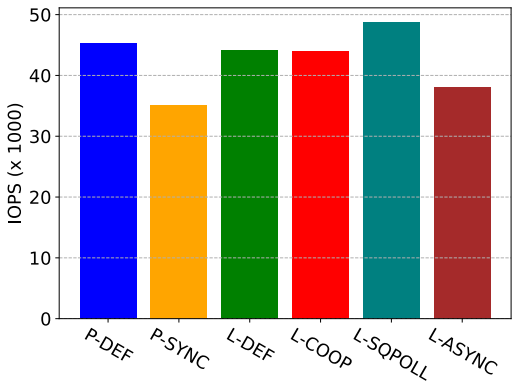


Figure 5.3: Throughput in IOPS for each RocksDB I/O backend configuration

Figure 5.3 shows the measured IOPS on the NVMe device during each experiment. This

shows that the RocksDB configuration that we used results in disk throughput between 35 and 49 thousand IOPS. In addition, we measure that 95.3% of requests are between 8k to 16k in size, and 4.7% are between 4k and 8k. In chapter 4 we have shown that this device is capable of over 200 thousand IOPS on a single core. From this, we conclude that RocksDB is more CPU intensive than our microbenchmarks, which can explain why the same `io_uring` configurations result in less performance difference here compared to in the previous chapter.

5.4 Summary

In this chapter, we successfully implement a `liburing` storage backend for RocksDB and evaluate it with different `io_uring` configurations. We run experiments on both the default RocksSB storage backend, and our custom liburing backend. First, we show that having `io_uring` enabled in the default backend improves throughput by 26.5% and lowers P95 latency by 16.5%. Second, we show that by changing the `io_uring` configuration to use SQ polling, we can improve throughput by 11.9% and lower latency by 13.1% compared to a ring with the default configuration. The default RocksDB storage backend always creates an `io_uring` ring with default configuration. Finally, we conclude that because RocksDB is more CPU intensive than the previous microbenchmarks, the performance improvements here are not as pronounced as in the previous chapter.

6

Related Work

In this thesis, we have concentrated on a single I/O interface, `io_uring`, that the Linux kernel provides to applications. There is a wide range of research in storage systems that relates to our work. For example in OS block layer design, I/O scheduling, interfaces for storage hardware, and the design of flash-based storage. In this section, we describe several areas of related work.

Linux Block Layer

Bjorling et al. (14) explain how the Linux block layer had become a bottleneck for high-performance storage, and how they redesigned the block layer to SSD scale, with multi-queue (MQ) support and separate software and dispatch queues, to decrease lock contention. Zang et al. (26) explain SPDK, a userspace NVMe storage stack that bypasses the OS block layer, providing higher performance with lower overhead. Didona et al. (8) compare `io_uring` with AIO and SPDK, and evaluate `io_uring`'s polling features. Ren et al. (9) compare the different I/O interfaces and also the performance impact of different Linux I/O schedulers. Joshi et al. (27) describe I/O Passthu, a new Linux mechanism that is not limited to block devices, and extends `io_uring` for NVMe commands.

I/O Schedulers

Most older studies on Linux I/O scheduling are centered around HDDs, where the focus was mainly on ordering requests to maintain sequential access. After the MQ block layer was introduced, running without a scheduler (`nosched`) was the only option for several years, and is still the default option on many systems today. With high-performance SSDs, the CPU overhead of I/O schedulers has become significant. Ren et al. (9) show that the available schedulers on Linux significantly hurt performance. However, the `nosched`

option does not account for fairness, and some processes can become starved of resources. Several fair I/O schedulers have been introduced, including MQFQ (28), D2FQ (29), and blk-switch (30), which demonstrate that fair I/O scheduling is possible with little CPU overhead.

Interfaces for SSDs

In addition to software interfaces, there is also existing research on hardware (device) interfaces, and proposals for new interfaces. Bjorling et al. (31) argue that the traditional block interface is not suitable for the characteristics of flash storage, and introduce ZNS, a zoned block-interface, where zones can only be written sequentially, matching with the append-only nature of flash storage. This eliminates the need for a complex FTL (Flash Translation Layer) on the device to expose a block interface. Previously, Bjorling et al. (31) introduced Open-Channel SSDs, an even less abstracted interface that exposes all internal flash topology to the host. Kang et al. (32) introduced multi-streamed SSDs, an interface that separates hot and cold data, depending on how frequently the data will be updated, allowing the SSD to make better decisions about flash allocation. Ioannou et al. (33) introduce SALSA, an FTL-like software middleware that converts I/O from the host into large sequential blocks on the device.

SSD Performance Characteristics

As described in chapter 2, SSDs contain complex FTLs (Flash Translation Layer) to map block addresses from the host into locations on flash chips, while also running garbage collection, which merges blocks containing stale (overwritten) data, and wear leveling to ensure equal lifetime of flash chips. These internal mechanisms can cause unpredictable performance, such as high tail latencies. He et al. (34) come up with several guidelines for developing programs for SSDs, and explore how well existing applications and file systems conform to these rules. Li et al. (35) come up with a system to learn key internal properties of SSDs, that are normally hidden behind the block interface, and how to use these properties to optimize applications.

7

Conclusion

In this thesis, we have explored the `io_uring` interface, and have made several observations about how its configuration can affect performance. First, we explored the available `io_uring` configuration options, and came up with an evaluation plan. We extended the `fio` benchmarking tool to support the options that we planned to study in detail. We also used `eBPF` and other profiling tools to understand the internals of `io_uring` and its internal implementation of the configuration options. Finally, we wrote a storage backend for RocksDB, to evaluate whether our guidelines could help improve the performance of an I/O-intensive application.

In this section, we start by answering the research questions that we presented in the beginning, along with the guidelines that we have come up with for application developers interested in using `io_uring` for storage. We then discuss the limitations of our work, and finally, describe several ideas for future work in this area of research.

7.1 Research Questions

RQ1 - What methods are available for evaluating and measuring io_uring performance under different configurations?

In chapter 3, we came up with an evaluation plan and discussed what tools could help in our exploration. We found that `fio` is easy to extend and provides detailed results, and `eBPF` can give insights into almost every aspect of internal kernel behavior.

RQ2 - What io_uring features can affect the throughput and latency of application I/O and under what circumstances?

In chapter 4, we found that several `io_uring` features can significantly affect performance, but the results can depend on specific host or application conditions, such as IRQ placement or CPU utilization. In the following subsection, we provide the guidelines that our evaluation has resulted in.

7.2 Guidelines

From our evaluation of `io_uring`, we have come up with the following guidelines based on the features that we have explored.

- Using the `COOP_TASKRUN` flag can result in significantly higher throughput and lower latency if doing I/O on a device that has fewer NVMe hardware queues than the number of host CPU cores. In other cases, including the flag is also harmless, unless doing ring completions on a different thread from submissions, in which case completions can stall for a long time

- In general, registered files should always be used. Using registered files results in a slight increase in throughput and a decrease in latency. We could not find a case where registered files impacted performance negatively.

- Using `FORCE_ASYNC` to submit requests asynchronously on a kernel worker thread may improve throughput, e.g. if your application is not utilizing all CPU cores. If your application is limited to a single thread, forcing asynchronous submission can work as offloading I/O to other cores. The flag is also beneficial if SQEs are expected to fail in non-blocking submission, e.g. due to lock contention. The flag can degrade performance in CPU-bound applications, as spawning and communicating with worker threads has a higher overhead compared to in-line submission.

- SQ (Submission Queue) polling can provide significantly lower latency, and comparable throughput, but requires dedicating a CPU core to a kernel polling thread. In addition, to optimally utilize I/O and CPU resources, it is recommended to look into polling efficiency, i.e. whether each application and kernel thread are close to fully utilizing a CPU core.

7.3 Limitations

The first limitation of our study is that we only evaluated 4 configuration options of `io_uring`. In chapter 3 we discussed that `io_uring` offers 24 configuration flags and we categorized 7 of them as likely to affect performance positively. There may also be more flags that impact performance negatively. In addition, all benchmarks in this study use simple read operations, however `io_uring` supports many other operations, for example, passthrough of NVMe commands.

Another limitation is that our evaluation was only done on a single machine, with a single Intel Optane device. Ideally, we would run our evaluations on other types of SSDs, and systems with different NUMA characteristics. In future work, this could be done with the help of an emulator for example, as there exist emulators for flash storage, and `qemu` can emulate different NUMA topologies.

Finally, there were some results that we could not fully explain during our evaluations with `fio`. For example, when exploring registered files, we followed a hint about threads with children showing a higher benefit when using registered files, because of the file table being shared. However, we did not find a configuration that could show a higher performance difference.

7.4 Future Work

During this thesis, many ideas for future work have come to mind. First, a recurring theme in modern storage is that both hardware interrupts and OS context switches, have a high overhead and should be avoided, especially if they occur for each I/O submission or completion. The `io_uring` API supports both polling (i.e. interrupt-free) and zero system call (i.e. context switch-free) operation. However, `io_uring` is only one recent example of a successful asynchronous I/O interface. We think that there are opportunities to further explore the design space of storage interfaces.

Second, during this thesis, we were only able to explore 4 specific features of `io_uring`. Further research could explore other potential configuration options, such as I/O polling, as we discussed in chapter 3. In addition, comparing the performance of `io_uring` with other available I/O interfaces, such as POSIX system calls, AIO, and SPDK, and exploring what features they provide. A future study could come up with guidelines about storage programming with respect to all available interfaces.

Finally, during our evaluation of the `COOP_TASKRUN` flag, we found that disk I/O performance can depend on what CPU core the workload runs on, e.g. due to IRQ or NUMA asymmetry. On such systems, applications should be manually pinned to their optimal CPU cores, as the Linux process scheduler does not account for storage resources. Designing an I/O aware process scheduler could provide performance benefits and is a possible area of future research.

References

[1] **Cloud Computing Market Size, Share Industry Analysis**. https://www.fortunebusinessinsights.com/cloud-computing-market-102697. Accessed: 2024-05-31. 1

[2] **Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025**. https://www.statista.com/statistics/871513/worldwide-data-created/. Accessed: 2024-05-31. 1

[3] Randal E Bryant and David Richard O'Hallaron. *Computer systems: a programmer's perspective*. Prentice Hall, 2011. 2

[4] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D Davis, Mark Manasse, and Rina Panigrahy. **Design tradeoffs for SSD performance**. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008. 2, 7, 8, 9

[5] **Seagate Exos X10**. https://www.seagate.com/files/www-content/datasheets/pdfs/exos-x-10DS1948-1-1709-GB-en_GB.pdf. Accessed: 2024-05-31. 2

[6] **Intel Optane P5800X**. https://www.intel.com/content/www/us/en/products/sku/201860/intel-optane-ssd-dc-p5800x-series-800gb-2-5in-pcie-x4-3d-xpoint/specifications.html. Accessed: 2024-05-31. 2

[7] Marco Cesati Daniel P. Bovet. *Understanding the Linux Kernel, Third Edition*. O'Reilly Media, 2005. 2, 10, 11

[8] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh 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*, pages 120–127, 2022. 3, 4, 16, 17, 69

[9] Zebin Ren and Animesh Trivedi. **Performance characterization of modern storage stacks: Posix i/o, libaio, spdk, and io_uring**. In *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*, pages 35–45, 2023. 4, 10, 16, 17, 69

[10] **Linux man pages - io_uring_setup**. `https://man.archlinux.org/man/io_uring.7.en`. 4, 16, 17

[11] **Linux man pages - io_uring_enter**. `https://man.archlinux.org/man/io_uring_enter.2.en`. 4, 16

[12] Rajkumar Buyya, Shashikant Ilager, and Patricia Arroba. **Energy-efficiency and sustainability in new generation cloud computing: A vision and directions for integrated management of data centre resources and workloads**. *Software: Practice and Experience*, **54**(1):24–38, 2024. 5

[13] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. **Operational Characteristics of SSDs in Enterprise Storage Systems: A Large-Scale Field Study**. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 165–180, 2022. 9

[14] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. **Linux block IO: introducing multi-queue SSD access on multi-core systems**. In *Proceedings of the 6th international systems and storage conference*, pages 1–10, 2013. 10, 69

[15] Rob Von Behren, Jeremy Condit, and Eric Brewer. **Why Events Are a Bad Idea (for High-Concurrency Servers)**. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, 2003. 12

[16] John Ousterhout. **Why threads are a bad idea (for most purposes)**. In *Presentation given at the 1996 Usenix Annual Technical Conference*, **5**, pages 33–131. San Diego, CA, USA, 1996. 12

[17] Jens Axboe. **Efficient IO with io_uring**. `https://kernel.dk/io_uring.pdf`. Accessed: 2024-05-30. 12, 14

[18] Tamás Koczka. **Learnings from kCTF VRP's 42 Linux kernel exploits submissions**. `https://security.googleblog.com/2023/06/learnings-from-kctf-vrps-42-linux.html`. Accessed: 2024-05-30. 13

[19] Jens Axboe. **Lord of the io_uring**. https://unixism.net/loti/index.html. Accessed: 2024-05-30. 13, 14

[20] **Linux man pages - io_uring_register**. https://man.archlinux.org/man/io_uring_register.2.en. 16

[21] Jonathan Corbet. **Descriptorless files for io_uring**. https://lwn.net/Articles/863071/. 2021-07-19. 16

[22] **Linux man pages - io_uring_register_files**. https://man.archlinux.org/man/io_uring_register_files.3.en. 54

[23] **SQPOLL: io_uring_submit return value greater than what's newly submitted 88**. https://github.com/axboe/liburing/issues/88. Accessed: 2024-05-31. 57

[24] **RocksDB Wiki**. https://github.com/facebook/rocksdb/wiki. 63, 64

[25] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. **Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications**. *ACM Transactions on Storage (TOS)*, **17**(4):1–32, 2021. 63, 64

[26] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. **SPDK: A development kit to build high performance storage applications**. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017. 69

[27] Kanchan Joshi, Anuj Gupta, Javier González, Ankit Kumar, Krishna Kanth Reddy, Arun George, Simon Lund, and Jens Axboe. **I/O Passthru: Upstreaming a flexible and efficient I/O Path in Linux**. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, pages 107–121, 2024. 69

[28] Mohammad Hedayati, Kai Shen, Michael L Scott, and Mike Marty. **Multi-Queue Fair Queuing**. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 301–314, 2019. 70

[29] Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong. **D2FQ:Device-Direct Fair Queueing for NVMeSSDs**. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 403–415, 2021. 70

[30] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. **Rearchitecting linux storage stack for μs latency and high throughput**. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 113–128, 2021. 70

[31] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. **ZNS: Avoiding the block interface tax for flash-based SSDs**. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703, 2021. 70

[32] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. **The multi-streamed Solid-State drive**. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014. 70

[33] Nikolas Ioannou, Kornilios Kourtis, and Ioannis Koltsidas. **Elevating commodity storage with the SALSA host translation layer**. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 277–292. IEEE, 2018. 70

[34] Jun He, Sudarsun Kannan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. **The unwritten contract of solid state drives**. In *Proceedings of the twelfth European conference on computer systems*, pages 127–144, 2017. 70

[35] Nanqinqin Li, Mingzhe Hao, Huaicheng Li, Xing Lin, Tim Emami, and Haryadi S Gunawi. **Fantastic SSD internals and how to learn and use them**. In *Proceedings of the 15th ACM International Conference on Systems and Storage*, pages 72–84, 2022. 70

8

Appendix

8.1 Artifacts

In this section, we explain how to reproduce the results of this thesis. The artifacts are available online on GitHub at `https://github.com/Ingimarsson/iouring-perf-analysis`. The repository contains a `README` file with details about each experiment, and each experiment folder contains the shell script to run the experiment, its full output, parsed output, and plots. To run the experiments, the custom versions of fio and RocksDB have to be compiled.

fio

To compile fio with our `liburing` engine, run the following commands. First, clone fio and checkout the specified version.

```
git clone git@github.com:axboe/fio.git
git checkout 06812a4f
```

Then apply our patch, which includes the liburing engine. Then fio can be compiled.

```
git apply this/fio/liburing.patch
./configure
make
```

RocksDB

To compile RocksDB with our `liburing` storage backend, run the following commands. First, clone RocksDB and checkout the specified version.

```
git clone git@github.com:facebook/rocksdb.git
git checkout 9d64ca55
```

Then, copy our liburing plugin, and compile.

```
cp this/rocksdb/ plugin/liburing/
DEBUG_LEVEL=0 ROCKSDB_PLUGINS="liburing" make -j48 db_bench
```

8.2 Using liburing

In chapter 2, we explained how `io_uring` works from an application programmers perspective. As we saw, `io_uring` is a low-level interface. There are many steps involved, and it requires a fair bit of boilerplate code. To make programs shorter and simpler to write, the `liburing` library provides helper functions that make the code shorter and more readable. In this section, we give an example of a `liburing` program, with explanations of each step. The full program can be found under `tools/io-uring-simple` in the repository, we only show the most important lines here.

First, we create a ring. This makes two calls to `mmap` to allocate buffers, and then calls `io_uring_setup`.

```
struct io_uring ring;
ret = io_uring_queue_init(QUEUE_DEPTH, &ring, 0);
```

Then, we get a free SQE, put a read request into it, and submit it. Submitting involves a `io_uring_enter` system call with the `to_submit` argument set to 1 or higher.

```
sqe = io_uring_get_sqe(&ring);
io_uring_prep_readv(sqe, fd, iovec, 1, 0);
ret = io_uring_submit(&ring);
```

After submitting, the request is in flight and the application can move on to other tasks. To receive the SQEs result, i.e. the CQE, we can wait for it. This `liburing` function is smart, and first checks if the CQE has been put into the completion queue. In that case, no system call is required. Otherwise, it calls `io_uring_enter` with the `GET_EVENTS` flag, and `min_complete` to 1 or higher. This call will block until that number of events are available. The CQE must also be marked as seen after its result is not needed, to free up space in the ring buffer.

```
ret = io_uring_wait_cqe(&ring, &cqe);
io_uring_cqe_seen(&ring, cqe);
```

Finally, we close the ring. This makes a `close` system call to the ring's file descriptor, and then two `munmap` calls to free the two ring buffers.

```
io_uring_queue_exit(&ring);
```

8.3 Using eBPF

eBPF is a Linux kernel feature that makes it possible to run small programs in the kernel without having to modify the kernel source code or build a kernel module. eBPF programs can be attached to various hooks that the kernel provides, including system calls and most kernel functions. The flexible nature of eBPF results in many use cases, including debugging and performance investigations, security monitoring, network packet filtering, and many more.

An eBPF program is loaded into the kernel through the `bpf` system call. The kernel runs the eBPF verifier that makes sure that the program won't hurt the kernel, run for too long, or use too much memory. After that, the program is JIT compiled into native machine code, which will run in kernel mode with full privileges. This makes eBPF programs very fast. The program can read or write values into key-value maps that have been created with the `bpf` system call. User processes with the right privileges can also interact with eBPF maps. This allows the programs to pass e.g. collected statistics to a user process for printing or further handling.

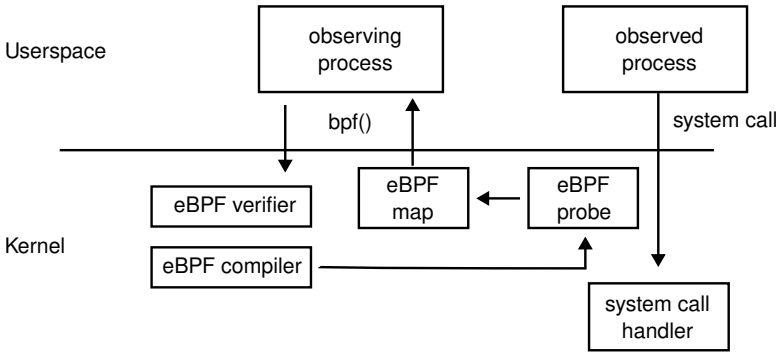


Figure 8.1: The components of eBPF and flow when one process adds a probe to monitor another process

After being loaded into the kernel, an eBPF program needs to be attached to one or more hooks. The Linux kernel provides multiple classes of hooks, including tracepoints, kprobes, and kretprobes. The hooks differ in what information will be available to the program, for example register values and function call arguments. For networking, there are eBPF hooks that allow the program to make decisions, e.g. whether to accept packets or not. eBPF network programs can even be offloaded to some network cards using the XDP standard (eXpress Data Path). Figure 8.1 shows the main components involved in eBPF for observability.

Programs are most commonly written in a C-like syntax that can be compiled into eBPF byte code. In addition, there are several libraries for loading and interacting with eBPF programs from userspace. Two popular libraries are `ebpf-go` for Go and `BCC` for Python. In addition, the `bpftrace` tool is convenient for debugging and tracing with one liner commands. It provides a compact syntax for writing scripts and prints out map contents after running.

During the experiments of this thesis, eBPF was extremely invaluable for gaining insights into the kernel. For example, understanding what system calls are used, what their arguments are, and where certain kernel functions are called from, by printing their stack traces. For most of our experiments, we used the `bpftrace` tool. We will now explain the main features of `bpftrace` and show several useful examples.

bpftrace

A `bpftrace` script typically contains one or more blocks of the following format.

```
probe[,probes...] { action }
```

To list all available probes, we can run `bpftrace -l`. For example, if we want to inspect `readv` system calls, we can use the `tracepoint:syscalls:sys_enter_readv` probe. The action is a program, consisting of e.g. if statements, while statements, and variable assignments. Output can be written to maps, which are prefixed with `@`, for example `@counter = 0`, or `@map[key] = value`.

Several reduction functions are available for maps, such as `count()` and `hist()`, and several built-in variables contain the current context, e.g. `arg0, ..., argN` (function arguments), `kstack` (current call stack), `cpu` (current CPU core), and `pid`. We now demonstrate a few examples of `bpftrace` scripts that we have used in this thesis.

Counting the number of IPIs

```
kprobe:native_smp_send_reschedule { @counter = count(); }
```

Counting the number of NVMe interrupts by CPU

```
kprobe:nvme_irq { @irq[cpu] = count(); }
```

Most frequent stack traces of function

```
kprobe:kick_process { @stack[kstack()] = count(); }
```

Flags on I/O requests passed to `ext4` read iterator

```
kprobe:ext4_file_read_iter {
    @[((struct kiocb*)arg0)->ki_flags] = count();
}
```

Disk I/O size histogram per process

```
tracepoint:block:block_rq_issue { @[comm] = hist(args->bytes); }
```