Accelerating an HTTP Benchmark Tool with `io_uring`

Nithya Vupparapalli

Master of Science

School of Informatics

University of Edinburgh

2023

Abstract

This thesis focuses on the enhancement of `wrk` benchmarking tool by integrating `io_uring` asynchronous I/O framework, aimed at optimizing its performance. Traditionally, `wrk` employs the `epoll` I/O mechanism, which, while efficient, encounters performance bottlenecks due to the increasing overhead of system calls as concurrent connections rise. By harnessing the power of `io_uring`'s shared space between user and kernel, we significantly reduce system calls, using the benefits of operation batching and asynchronous I/O.

Two implementations were introduced: one utilizing batching (`io_uring`) and another without batching (`io_uring-nb`). Experimental results demonstrated that both `io_uring` implementations outshine the traditional `epoll` mechanism in terms of performance, with `io_uring` showcasing an upsurge of up to 33% in throughput, and `io_uring-nb` showing an improvement of 12%. Notably, while `io_uring` excelled in high-load scenarios, its batching mechanism was less effective under limited connections, highlighting the nuances of different implementation approaches.

Furthermore, other factors such as varying response buffer sizes, `QUEUE_DEPTH)` of submission and completion rings, connections were evaluated, with the `io_uring` implementations consistently outperforming `epoll` across all configurations. This research underscores the potential of `io_uring` as a transformative I/O mechanism, paving the way for future optimizations in web benchmarking tools. The findings also advocate for an in-depth understanding of system requirements, emphasizing the necessity of choosing the right I/O mechanism tailored to specific scenarios for optimal performance. Future avenues include multi-core support, extended protocol and scripting support, and benchmarking across diverse hardware platforms.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Nithya Vupparapalli*)

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Michio Honda, for his unwavering guidance, invaluable feedback, and relentless support throughout the course of this research. Their expertise and insights have been instrumental in shaping this thesis.

I wish to extend my appreciation to my friends, Sartaj, Zain, and Aman, who have been a constant source of inspiration throughout this research.

To my parents, whose constant encouragement and belief in my capabilities have been my anchor, I am eternally grateful for their support and it helped me in not giving up.

Lastly, I would like to acknowledge the countless researchers and authors whose works have laid the groundwork for this study. Their contributions to the field have been indispensable in guiding my inquiries and shaping my understanding.

Table of Contents

Chapter 1

Introduction

The Hypertext Transfer Protocol (HTTP)[11], operating at the application layer, is the primary protocol for data communication on the World Wide Web. It has become the standard mechanism for transmitting information across the vast ecosystem of web browsers, servers, proxies, and related networking tools. With the widespread use and growth of the internet, it is crucial to ensure that the efficiency and security of these HTTP protocols are not compromised. It operates as a request-response protocol within a client-server architecture, facilitating effective communication. Servers often handle simultaneous requests from numerous clients. As such, it's crucial for these servers to efficiently manage high volumes of incoming requests while ensuring they respond quickly, accurately, and securely to each client.

As the complexity and demands of web servers and applications have escalated, the tools designed to evaluate their performance must evolve in tandem. It is important for these tools to generate realistic, high-concurrency loads to assess the capabilities of HTTP servers genuinely. Benchmarking tools[4] play a crucial role in this scenario, offering rigorous testing and measuring web server performance. They produce quantifiable metrics such as response time, throughput, and concurrent connection handling, thus, providing insight into a server's strengths and potential bottlenecks. Among the available HTTP benchmarking tools, `wrk` stands out as a representative of modern-day benchmarking utilities. Its proficiency in generating significant traffic, even under stressed resources, makes it a widely used tool for understanding and optimizing an HTTP web server's performance. When `wrk` is used to benchmark a server, it opens multiple concurrent connections to simulate high number of GET requests to the HTTP server and then read the response received from it.

In this dissertation, we focus on the `wrk` benchmarking tool, aiming to understand

its operations and potential enhancement. `wrk` employs the `epoll` system on Linux for I/O event alerts. `epoll` is an advanced I/O event notification system offered by the Linux kernel, which helps in tracking multiple file descriptors, such as sockets, for specific events, including read or write readiness. Within `wrk`, this `epoll` framework is critical in effectively overseeing the numerous active connections made to the targeted HTTP server. This capability is crucial for a benchmarking utility like `wrk`, given its primary function of exerting significant stress on servers through the use of concurrent connections.

During the benchmarking process, `wrk` establishes numerous connections to simulate high traffic or load. Once these connections are made, it waits for the kernel to send an event notification when there is any activity on them, such as when a socket is ready to send more data or when there is incoming data to be read. When `wrk` receives a notification event, it acts immediately, reading the data or sending more requests, as appropriate. However, each time `wrk` needs to wait for events or register interest in events, it makes system calls[22] that involves a transition between user space (where `wrk` operates) and kernel space (the core of the operating system). These transitions come with overhead that is time and resource-consuming. In situations where ultra-high performance is essential, and every microsecond counts, the overhead from these frequent system calls can become a bottleneck.

In order to help mitigate the issues caused by the current I/O mechanism in `wrk` and help improve its performance, we will utilise the new I/O interface called `io_uring`[8, 5, 6]. `io_uring` is a modern and advanced interface for asynchronous I/O operations in the Linux kernel. It is designed to offer efficient, scalable, and fast I/O without the complexities and limitations of traditional I/O mechanisms. Unlike previous methods of I/O handling in Linux, `io_uring` has enhancement features such as operation batching, asynchronous I/O, shared memory space between user & kernel. These qualities are extremely beneficial in improving the performance of the benchmarking tools. Our goal is to use `io_uring` in `wrk` and prove that it indeed plays a significant role in overall performance improvement.

1.1 Problem Statement

In this section, we will establish a clear baseline to highlight the issues observed when the `wrk` benchmark is done on an HTTP server. So, an experiment is conducted between a single-core CPU client containing the `wrk` benchmark tool and a powerful

multiple-core HTTP server to demonstrate the latency and throughput achieved in the benchmarking process. This is because we want to show the potential problems that may arise even in the most simplest configuration. To do so, we will use 2 cloud machines in a small-lan profile provided by the CloudLab research infrastructure tool. These machines are xl170 nodes, that have deafult OS as UBUNTU, x64_86 architecture and use the Two Dual-port Mellanox ConnectX-4 25 GB NIC[1] for effective communication between them. . By default, the machines have 20 active CPU cores on them. For our experiment and to outline the problem, we will activate only 1 CPU core in the client and 8 CPU cores in the server by turning off the remaining CPU cores using CPU hotplugging. The `wrk` benchmarking tool will be set up on the client machine. Meanwhile, the server machine will host an operational `nginx` web server, guaranteeing that port 80 is open and active. Now, we are ready to perform the benchmarking test on the server. Keeping in mind that the client has only 1 active CPU core, we need to execute the test on a single thread.

As part of the thesis, we will focus primarily on the single-core CPU performance, determine the underlying problem and aim to improve the performance. The `wrk` benchmarking tool lets a user customize the total number of threads, connections, and duration of the test before sending requests to HTTP servers. Keeping in mind that the client has only 1 active CPU, the number of threads for this experiment will be 1 and the duration of the test will be 5 seconds. We will, however, change the number of concurrent connections by increasing them gradually. Below is a sample benchmark run on the server and the description of the output:

```
User@node0:~/folder/wrk$ ./wrk -t1 -c10 -d5 http://10.10.1.2:80
Running 5s test @ http://10.10.1.2:80
  1 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   113.95us   46.23us 820.00us   82.88%
    Req/Sec    84.07k     4.78k   90.72k    88.24%
  426578 requests in 5.10s, 349.44MB read
Requests/sec:  83650.01
Transfer/sec:     68.52MB
```

From the benchmarking output, we can discern that during the interval of 5 sec-

[1]Network Interface Card

onds, the tool generated a total of 426,578 requests, with a cumulative data transfer of 349.44MB. The specific metrics for the thread, such as latency and requests per second, offer insights into the server's responsiveness and throughput during the test. In particular, the latency metrics provide the average, standard deviation, and maximum time taken between sending a request and receiving a response. Concurrently, the requests per second (throughput) indicate the processing rate for each thread. To determine the problem statement, we increment the number of concurrent connections in our benchmarking test on the server and monitor the resulting performance patterns.

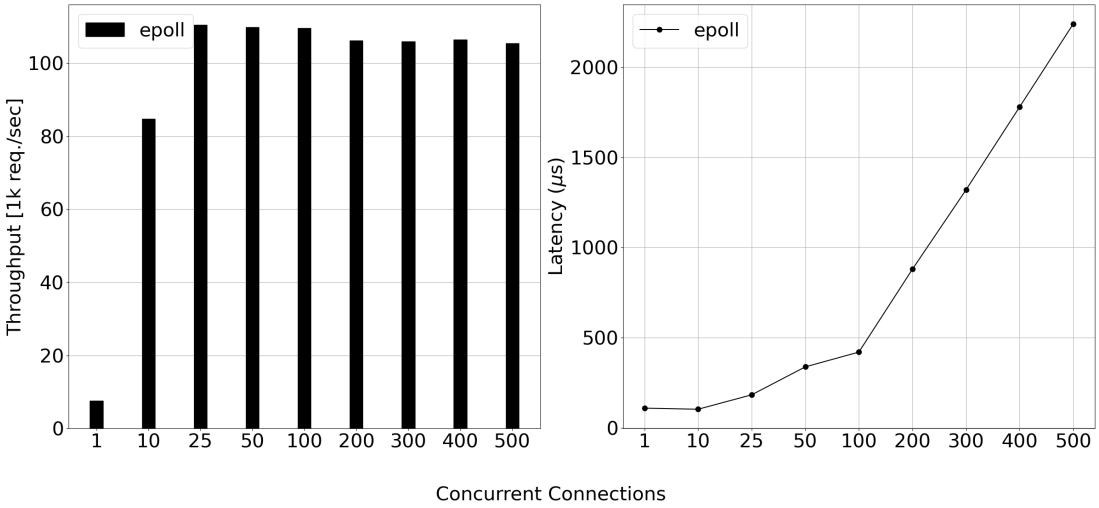

Figure 1.1: Thread Statistics during benchmark with 1 thread for 5 seconds

We conducted a benchmarking test on the `nginx` server, adjusting the concurrent connections in the range of (1, 10, 25, 50, 100 - 500). The derived thread statistics (latency and throughput at the rate of 1000 requests / second) are presented in Figure 1.1. During the experiment, the CPU utilization of both client and server was observed. Initiating a single connection to the server led to the processing of approximately 9k requests per second, with an even distribution of load across all eight CPU cores on the server. Meanwhile, the client, with its single-core, used up to 25% of its CPU capacity. As the concurrent connections increased, the server maintained efficient use of its eight cores, ensuring optimal request processing. In contrast, the client's CPU usage displayed a divergent trend. For 10 concurrent connections, the client's CPU was already burdened, reaching around 87% utilization. When the concurrent connections escalated to 25 or beyond, the client's CPU was fully saturated, operating at its 100% capacity.

This complete utilization of the client's CPU has a direct impact on the through-

put. As the CPU becomes a bottleneck, it's unable to manage additional connections efficiently, leading to stagnancy or even a decline in throughput. The server is still capable of handling more requests, but the client's limitation becomes a restricting factor, preventing further scaling and introducing increased latency. The second graph illustrates an exponential growth in latency, signifying a longer time taken to process requests as the number of connections escalates. This surge in latency can be attributed to the mounting backlog of requests, which accumulates further with the increase in concurrent connections. This indicates that the client is emerging as the limiting factor. Its inability to sustain efficiency becomes evident when the graphs prove the throughput becoming stagnant, indicating a saturation point where the client struggles to manage the increasing demands efficiently.

A key aspect to consider is the `epoll` mechanism that `wrk` employs for I/O event notification. `epoll` is designed for scalable I/O, but its efficiency can be compromised under extreme loads as seen above. As the number of connections increases, the `epoll` system makes frequent calls, like `epoll_ctl()` to manage file descriptors, and `epoll_wait()` to block and wait for notifications. Each of these system calls introduces the overhead of transitioning between user space and kernel space, known as context switches. This overhead can lead to increased CPU usage and subsequently higher latency. Furthermore, with a growing number of connections, the `epoll` mechanism will be continuously triggered by incoming events. The time to process these events to determine the operations, and then to act upon them becomes CPU-intensive, thereby contributing to the increasing latency. Moreover, the mounting backlog of requests, a direct consequence of the rising connections, can cause data to be buffered longer before being processed. This 'queueing delay' in the system increases as the requests grow.

Drawing a connection between the latency trend and the stagnation in throughput, it becomes evident that the client, despite its use of the `epoll` mechanism, is reaching a saturation point. The client's resources and its event-driven mechanism are being fully taxed, leading to a plateau in throughput and an uptick in latency. This suggests that while `epoll` provides scalability advantages, it's not devoid of challenges when pushed to its limits, making the client the bottleneck in this scenario.

1.2 Aim

The primary aim of this research is to explore the potential advantages of the io_uring I/O interface in addressing the identified bottleneck and inefficiencies associated with the epoll mechanism in the wrk benchmarking tool. io_uring is a modern and advanced I/O interface introduced in the Linux kernel to overcome the limitations of previous I/O systems, including epoll and aio. Unlike epoll, which often handles multiple system calls to set up, modify, and retrieve events, io_uring[19, 20] streamlines the process by allowing asynchronous I/O, batched submissions and completions, thus, significantly reducing system call overhead.

The main objective is to integrate the io_uring mechanism into the wrk benchmark tool as an alternative I/O mechanism. We will conduct a series of benchmark tests, similar to the one shown in the problem statement, using wrk with io_uring, comparing its performance metrics—especially latency and throughput—with the existing epoll mechanism. The completion criteria of this thesis are to implement io_uring successfully into the wrk benchmarking tool and justify that it brings improvement in the single thread performance.

1.3 Dissertation Structure

This dissertation contains 5 chapters in total. Chapter 1 introduces foundational concepts related to HTTP and underscores the importance of benchmarking in assessing web server performance. Within the scope of this thesis, the benchmarking tool of focus is wrk. The chapter provides insights into various I/O mechanisms, with a particular emphasis on epoll and io_uring. It delves into the identified problem statement, highlighting the challenges posed by the epoll mechanism, which results in the client becoming a bottleneck during benchmarking processes.

Chapter 2 provides the necessary background information by discussing networking concepts, emphasizing I/O mechanisms, the nuances of benchmarking, and the potential of io_uring. Previous work is also highlighted to the reader, to understand how the dissertation stands out from the prior work.

Chapter 3 offers a deep dive into the io_uring mechanism, detailing its functionality and benefits. It explains the various strategies and methodologies employed to achieve the aim outlined in the first chapter.

Chapter 4 focuses on the empirical aspect, detailing the experiments carried out

based on the methodologies presented in the preceding chapter. It offers a comprehensive evaluation of the results, assessing how they align with the set completion criteria.

Concluding the dissertation, Chapter 5 reflects on the findings, discussing possible avenues for future enhancements. It underscores the success of the thesis in realizing its primary aim.

Chapter 2

Background and Related Works

The internet, with over 5.1 billion users, is a testament to the importance of robust digital infrastructure. At the heart of this vast network is the HyperText Transfer Protocol (HTTP), which ensures smooth digital interactions. The efficiency of HTTP owes a lot to various benchmarking tools developed over the years.

In the late 1990s, the `ab` tool marked its presence. Born from the Apache Software Foundation, this Apache HTTP benchmarking tool was tailored to assess the prowess of servers, especially gauging the number of requests they could handle per second. However, its scope has been limited due to constraints like its single-threaded nature and limited protocol compatibility.

Fast forward to 2015, the landscape witnessed the introduction of `ink`[18] by Peter Kehl. Distinguishing itself, `ink` simulated user traffic and meticulously recorded event traces for each emulated client. Its innovative approach found validation in a study by Virginia Tech, emphasizing its capabilities in offering profound insights into server behaviors.

Meanwhile, other tools like `siege` and `JMeter` were shaping the contours of the benchmarking domain. `siege`, recognized for its adaptability, emerged as a go-to HTTP load testing utility. It garnered acclaim for its capability to emulate a spectrum of user behaviors, thereby enabling developers to anticipate and prepare for varying server loads. This level of granularity in testing scenarios made `siege` indispensable for comprehensive server performance evaluations.

In contrast, `JMeter` brought a different flavor to the table. Another brainchild of the Apache Software Foundation, akin to `ab`, `JMeter` appealed to a broader audience with its user-friendly graphical interface. But beneath its intuitive exterior, it was a powerhouse. Introduced initially for web application testing, over time it evolved,

supporting a wide array of protocols beyond just HTTP. Its versatility, combined with detailed performance metrics, ensured that `JMeter` became the preferred choice for many aiming for in-depth load testing and performance analysis.

The recent entrant to this arena is `wrk`. This open-source HTTP benchmarking tool distinguishes itself with its multithreaded design, poised to measure server performance, especially in high-concurrency scenarios. For the project in question, `wrk` is the primary tool of interest, with extensive documentation supporting its use [10, 17, 9].

However, tools alone don't weave the entire narrative. The smoothness of interactions between servers and clients hinges on efficient I/O operations. The traditional UNIX I/O approaches often hit roadblocks when it came to scalability. In response, Linux championed the introduction of io_uring, an ingenious framework tailored for asynchronous I/O. Animesh Trivedi's research underscored its edge over existing I/O frameworks, highlighting its potential in revolutionizing server performance [24]. The ambition now is to meld `io_uring`'s prowess with `wrk` to drive superior performance on Linux servers.

2.1 `wrk`

`wrk` is an open-source HTTP benchmarking tool, known for its ability to generate significant load on a server, even with limited resources. The tool's popularity can be attributed to its simplicity in design and its capability to offer detailed insights into server performance under stress.

`wrk` uses an event-driven model for its I/O operations, which is a stark contrast to traditional threaded models where each connection might have a dedicated thread or process. In an event-driven model, events (like data arriving on a socket or a socket becoming writable) drive the program's execution.

- **Non-blocking Sockets**: All sockets in `wrk` are set to non-blocking mode. This ensures that I/O operations, such as reading from or writing to a socket, return immediately instead of waiting (or blocking) for the operation to complete. If the operation can't be completed immediately, it's deferred, and `wrk` continues processing other events.

- **Event Multiplexing**: `wrk` uses an event multiplexer (often epoll on Linux systems) to efficiently monitor multiple sockets for events. The multiplexer notifies `wrk` when there's an event on a socket, such as incoming data or readiness to

accept more data for sending. This mechanism allows `wrk` to handle thousands of concurrent connections with minimal overhead.

- **Event Loop**: At the core of `wrk's` event-driven architecture is the event loop. It continually checks for and processes socket events. When an event occurs, such as a server response arriving for a request, the event loop dispatches it to the appropriate handler in `wrk`.

It provides a comprehensive set of statistics that offer insights into the server's performance under the load generated by the tool. Upon completing a benchmarking session, `wrk` displays a summary of the test results. These statistics are designed to give users a clear picture of how the server performed under the simulated load. The primary statistics provided by `wrk` include:

- **Throughput**: Throughput, often represented as requests per second (RPS or req/sec), is a measure of the server's capacity to handle and serve requests. In the context of `wrk`, it represents the average number of requests the server was able to process every second during the benchmarking duration. High throughput values suggest that the server can handle a larger load, whereas low values might indicate potential bottlenecks or performance issues. Factors that can influence throughput include server hardware, server software configuration, network bandwidth, and application logic.

- **Latency**: Latency represents the time taken to process an HTTP request, from the moment it's sent until the response is fully received. `wrk` provides a detailed breakdown of latency, including average, maximum, and various percentile values. The percentile values are particularly informative as they shed light on the distribution of latencies:

  - Average Latency: The mean time taken to serve all requests.
  - 50th Percentile: The median latency value, meaning 50% of the requests were served faster than this value, and 50% were slower.
  - 99th Percentile: Only 1% of the requests had a latency higher than this value. It's a crucial metric to identify long-tail latency issues, which might affect a small percentage of users but can be indicative of deeper system or application problems.

2.2 io_uring

io_uring[2] is a Linux kernel interface introduced in version 5.1 to facilitate high-performance asynchronous I/O operations. Traditional Linux I/O interfaces, such as select, poll, and epoll, have limitations when handling a large number of file descriptors or when performing multiple I/O operations at once. io_uring was introduced to overcome these limitations and offer a more scalable and efficient asynchronous I/O mechanism. [12, 25, 26]

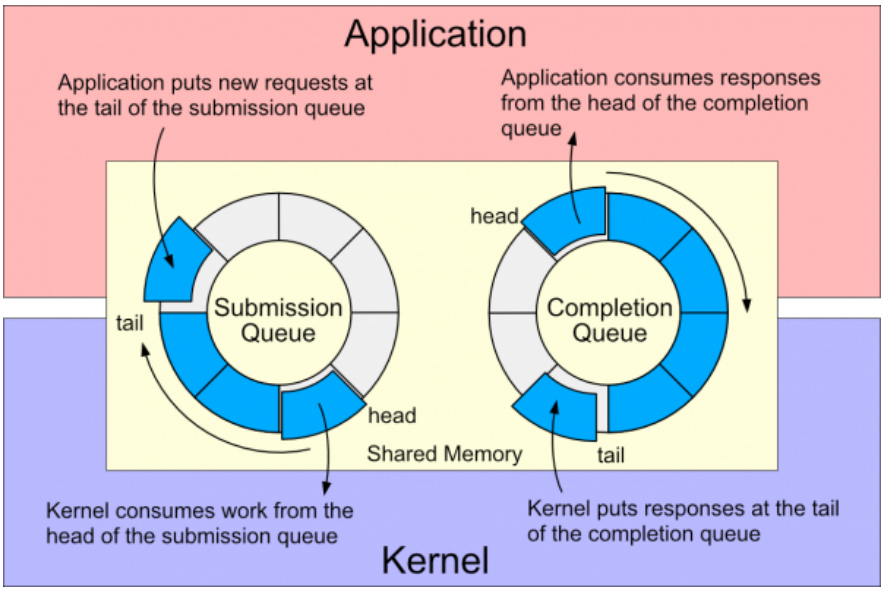

Figure 2.1: Asynchronous interface provided by io_uring between user space & Linux kernel

The io_uring interface has two fundamental operations associated with its mechanism: the submission of a request, and the event that is associated with the completion of said request. For submission of an I/O operation, the application is the producer and the kernel is the consumer. However, regarding the completion of these operations, the kernel produces the completion events and the application consumes them. Hence, it uses a pair of ring buffers that provide an effective communication channel between the application and kernel. These pair of rings are the core of io_uring interface as seen in Figure 2.1. It consists of 2 shared memory ring buffers at its core: submission queue (SQ) and completion queue (CQ). They are mapped into both the application (user) and kernel space, allowing for efficient communication and minimization of data copying. Each queue consists of an array of entries that are maintained using head and tail pointers to effectively manage I/O operations.

- **Submission Queue (SQ)**:
  - **Submission Queue Entry (SQE)**: Each I/O request that a user-space application intends to submit to the kernel is packaged into an SQE. The SQE contains all the necessary details about the I/O operation, such as the type of operation (read, write, etc.), the file descriptor, buffer pointers, and other relevant data.
  - After populating one or more SQEs, the application pushes these entries into the SQ. This can be done in batches, allowing multiple I/O requests to be enqueued together, which reduces the overhead of frequent system calls.
- **Completion Queue (CQ)**:
  - **Completion Queue Entry (CQE)**: Once the kernel has processed an I/O request from an SQE, it wraps up the results and status of the operation into a CQE. The CQE provides feedback about the completed I/O operation, including any potential errors or the number of bytes read/written.
  - User-space applications then inspect the CQ to retrieve these CQEs, allowing them to ascertain the outcomes of their I/O requests and take appropriate actions based on the results.

The introduction of `io_uring` with its submission and completion queue events offers several advantages over the traditional `epoll` mechanism, especially in the context of a benchmarking tool like `wrk`. While `epoll` requires system calls to submit and retrieve events, `io_uring` allows batching of operations, often without needing any system calls due to the shared memory rings. The application can enqueue multiple I/O requests in the SQ at once and can read multiple completions from the CQ in a single operation. This minimizes the overhead associated with frequent context switching and repeated system call invocation.

2.3 Related Works

The integration of `io_uring` into various software applications signifies its growing importance in the realm of high-performance I/O operations. This asynchronous I/O interface, introduced in the Linux kernel 5.1, has been heralded for its efficiency and scalability, especially in environments demanding intensive I/O operations. As

a testament to its potential, several software projects have endeavored to incorporate `io_uring` to enhance their performance and responsiveness. Highlighted below are a few notable software applications that have embarked on this integration journey:

- QEMU: QEMU (short for "Quick Emulator") is an open-source software intended to be buildable on all modern Linux platforms, OS-X, Win32 (via the Mingw64 toolchain) and a variety of other UNIX targets. It operates as a hypervisor, enabling full virtualization of guest systems on host systems. QEMU can work in conjunction with the Linux KVM to run virtual machines at near-native speeds by executing the guest code directly on the host CPU. KVM is a virtualization module in the Linux kernel that allows the kernel to function as a hypervisor. It currently supports `LINUX AIO API`, and Aarushi Mehta [23, 16] has tried to implement `io_uring` interface to improve performance of the QEMU hypervisor. They have implemented it in 2 ways, with basic `io_uring` and Submission Queue polling with file descriptor registration. With the help of her implementations, she could see better throughput than the traditional Linux `AIO` as well as threads. This is one of the successful implementations of `io_uring` which proved that it usually improves the speed of processing requests against the traditional `I/O` systems.

- Frodo: Frodo is a POC, that was developed to experiment with the `io_uring` APIs in Go [21, 1]. In this experiment, Agniva de Sarker states that he wanted to implement `io_uring` to test it's performance for simple operations like read and write to files. With the help of `fio`, they have performed few benchmark tests by varying block sizes (the response size) as well as the QUEUE_DEPTH of `io_uring` ring to perform random read and write operations. The results of their experiments show that against the traditional `libaio`, `io uring` was able to perform write operations at thrice the speed and read operations at almost 5x the speed without polling. He suggests that implementation of `io_uring` with polling could potentially improve this performance even higher and there are future works to do so. This is a second case that shows how powerful `io_uring` can be and how fast it can process requests regardless of the block size and ring depth.

- Echo Server: An echo server was a bare minimum server that was developed to implement `io_uring` by Hielke de Vries [7] to conduct benchmark tests against the traditional `epoll` system. It is a simple server that listens for incoming

connections and data. When data is received from a client, the server immediately sends the same data back to the client without processing it. In his implementation, he has used `IO_URING_FAST_POLL` and `IORING_OP_PROVIDE_BUFFERS` requiring Linux 5.7 version or higher to perform benchmark tests. The server is set up to read data from a client connection and then immediately write the same data back to the client, effectively 'echoing' the client's message. These operations are handled using an event loop to perform operations like accepting connections, sending requests and reading responses. The same server is implemented with `epoll` system as well and bench marking tests are performed on various response buffer sizes. `io_uring` showed the best performance in all cases of response buffers and number of clients compared to `epoll`. This showed the efficient usage of batching and asynchronous `I/O` against the traditional `epoll` mechanism that sends system calls to kernel repeatedly, creating an overhead. Since wrk also employs `epoll` mechanism and makes use of ae event loops, the inspiration for implementation of `io_uring` in it was taken from this work. Detailed explanation of how `io_uring` was used in a similar manner is discussed in Chapter 3.

- Haskell I/O Manager: The Haskell I/O manager is a component of the Haskell runtime system, particularly in the Glasgow Haskell Compiler (GHC), which handles asynchronous I/O operations. The I/O manager is crucial for the efficiency of concurrent Haskell programs that perform I/O operations, such as network communication or file access. A new event manager for backend was designed using `io_uring` and evaluated against the `epoll` mechanism. [13] Upon testing and plotting the results, it was claimed that `io_uring` gave 7% of improvement at 5 connections and gradually increased to 9% at 50 connections, 10.8% at 500 connections compared to the original `epoll` mechanism during benchmarking. The author claims that with increase in connections `epoll` was 'overloaded with more garbage collection', meaning that the amount of retained objects was higher than that of `io_uring` (it can drain the request queue at a faster rate, hence being able to take in next batch of requests immediately). So far, all the research has consistently shown that io_uring is very fast and can generate better throughput than most I/O systems. The aim of this paper is also to prove the same in the further chapters.

- ZeroHTTPd: It is a simple web server that was created by Shuveb Hussain [14, 15] and implemented using the `io_uring` interface to accept requests from

multiple clients and return appropriate responses at a quicker rate than `epoll`. This server can send files in the response, and he implemented `io_uring` to read these responses and send them back to the client. The implementation of `io_uring` was largely inspired by this method where an event loop is used to transition between states among the clients (in our case, connections). However, a similar approach used in the Echo server showed how the direction of choosing an event loop is beneficial for its implementation. The detailed analysis of the event loop used in accordance to the wrk benchmarking tool is done in Section 3.2.

Chapter 3

Methodology

This chapter details the approaches taken to incorporate `io_uring` into the `wrk` benchmarking tool. Our strategy involves two distinct methods of integrating `io_uring` into `wrk`. The differentiation between these methods depends on their implementation of batching operations to the ring buffers.

`io_uring` is a complicated interface that has in-depth code implementation even for its most basic use cases. To mitigate some of the complexities, we use `<liburing.h>`, a C library developed to provide a more accessible interface to the `io_uring` asynchronous I/O framework in the Linux kernel. It provides a set of utility functions to simplify common tasks, such as setup, teardown, retrieval of submission and completion of requests.

3.1 General Flow

With the help of `<liburing.h>` library we follow the generic steps to initialise **io_uring** in the `wrk` repository[1]:

1. We'll leverage the existing data structures provided by the `wrk` framework, specifically the structs designated for threads and connections. While `wrk` currently employs `epoll` to manage connection operation states, we'll enhance them by introducing three additional states to be managed by `io_uring`: `CONNECT`, `READ`, and `WRITE`.

2. We will introduce a new command-line option, `-i`, to the `wrk` tool to integrate `io_uring` mechanism during benchmark tests. This provides flexibility to the

[1] https://github.com/wg/wrk/

user in choosing the desired I/O mechanism to benchmark the servers. If the user chooses not to provide the `-i` option, the benchmarking will proceed with the regular `wrk` workflow.

3. Threads are spawned using the established wrk code, with each thread typically allocated its own set of connections. However, in our endeavor to enhance single-thread performance, we've configured it so that only one thread manages all the connections. We employ the `pthread_create()` method to channel these connections into the `thread_io_uring`() function, where the `io_uring` mechanism is present. This ensures that no conflicts or disruptions occur with the existing mechanism.

4. To initiate the `io_uring` flow, we first declare a ring buffer using the `struct io_uring ring;`. All the operations during the benchmark test happen within this `ring`. After declaration, we initialize it using the `io_uring_queue_init(QUEUE_DEPTH,\&ring,0)` function where `QUEUE_DEPTH` specifies the maximum number of entries that the ring buffer can hold. This effectively sets a limit on the number of outstanding I/O requests that can be managed by `io_uring` at any given time.

5. Next, each connection in the thread is assigned a unique socket file descriptor using the socket() function. Each connection's state is initially set to `CONNECT` and other attributes are assigned in the usual `wrk` process. Now, we create a submission queue entry and call `prep_connect()` method. This method handles connections using `io_uring_prep_connect();` function. This function prepares a submission queue entry (SQE) for a non-blocking connect operation.

6. With the SQE now set up for the connect operation, it has to be communicated with the kernel. By submitting this SQE, a signal is sent to the kernel that a request is queued and ready for execution. Once the operation completes, the kernel populates a completion queue entry (CQE). This CQE holds the outcome of the connection request that is ready to be seen by the application.

7. After retrieval of the completion queue entry (CQE), the connection is ready to handle the next operation. Drawing inspiration from the `wrk`'s `epoll` mechanism, I/O operations in our `io_uring` are structured to be implemented in the sequential order of: `CONNECT` → `WRITE` → `READ`. This sequence ensures that we first establish a connection, send a request to the server, and then read the server's response. This

flow is handled through an event loop that runs for the given duration specified by the user in the `wrk` command.

8. **Event Loop**: We have two approaches to implement **io_uring**: with and without batching of operations. For both scenarios, the foundation remains the same: using an event loop. This loop continuously checks for completed operations and initiates new ones based on the updated connection state. The detailed implementation is explained in Section 3.2.

9. **Distinguishing the Approaches**: While the core event loop remains consistent, the two `io_uring` methods differ majorly on how they perceive the completion queue sent by the kernel. The specifics of these differences are elaborated in Section 3.3 for '`io_uring-nb`' (without batching) and Section 3.4 for '`io_uring`' (with batching).

10. The benchmarking occurs when the event loop is being executed. While operations are running concurrently in an asynchronous manner, the in-built HTTP parser validates each response received from the server. It records and stores thread statistics like total requests parsed in a given period of time and latency of each request, i.e., the time taken to send a request and read the corresponding response. After the event loop terminates, the ring buffer is closed using `io_uring_queue_exit()` function.

3.2 Event Loop and State Machine

In `wrk` framework, an event loop is instrumental in the successful deployment of the `io_uring` model. It acts as a central point for I/O coordination and connection lifecycle management. The loop must handle the flow between asynchronous I/O operations and connection state progression elegantly. When a connection is created successfully, the kernel acknowledges this by registering a completion event in the Completion Queue Entry (CQE).

Now the application must continuously check for the Completion Queue Entry ring to keep track of completed operations for each connection. Regardless of the `io_uring` model chosen by the application for benchmarking, the detection of a completion event must be passed to a new connection object. This object fetches details from the completion event using `io_uring_cqe_get_data(cqe)` function. It is important to note

that the first state of each connection will always be set to `CONNECT` state. To manage subsequent stages `(READ, WRITE)` within the event loop, we've chosen to deploy a state machine that operates using a switch loop as below:

- **`CONNECT`**:
  - **Acknowledgment**: The completion of the connection is recognized and acknowledged using the `io_uring_cqe_seen(&ring, cqe)` function. This function effectively informs the `io_uring` interface that the application has seen the completion event and the connection is ready to move to the next state.
  - **Parser Initialization**: The `http_parser_init()` function initializes the HTTP parser for new connection object to prepare it for parsing the response received from the server in `READ` state later on.
  - **Transition to `WRITE` State**: The connection state is updated to `WRITE`, signaling that the next step is to send a simple GET request to the server.
  - **SQE Preparation to Send Request**: A Submission Queue Entry (SQE) is created to indicate that the connection wants to transition to `WRITE` state and passed as an argument in the `prep_send(sqe, connection)` function.
- **`WRITE`**:
  - **Acknowledgment:** The `io_uring_cqe_seen(&ring, cqe)` function tells the application when kernel has successfully sent the GET request to the server.
  - **SQE Preparation for Reading Response**: Now that a request has been sent to the server, the connection is prepared to receive a response. The connection state is now changed to `READ` and is sent through a Submission Queue Entry (SQE) to the `prep_read(sqe, connection)` function.
- **`READ`**:
  - **Acknowledgment:** The `io_uring_cqe_seen(&ring, cqe)` function indicates to the application when kernel has received an HTTP response from the server. It is stored in a Connection Queue Entry pointer `*cqe`.
  - **Data Processing:** Function `http_parser_execute()` checks the incoming HTTP response from the server. If the response is valid, it means the request

was handled completely and `response_complete()` method is called. In this method, the total completed requests (used in overall throughput statistics) and total requests (used in the calculation of per-thread throughput at regular intervals of 100 μs) are incremented each. This method is also responsible for logging latency for each request by calculating the difference between the time at which the request was sent and the corresponding response received by the application.

– **Error and State Handling:** Here, we retrieve the value of `cqe->res`, which is the size of response received from the server in bytes. Depending on its value, further handling is done as below:

  * **Error Detection:** If `cqe->res` returns a value less than 0, it indicates an error during data reception. The precise error is documented, and the error count is incremented. The executing of event loop is terminated.
  * **Connection Closure:** If `cqe->res` is 0, it suggests that the server has processed a request and closed the connection gracefully. This means that it can be re-opened. So, the state of the connection is reset to `CONNECT` and sent to a function called `initialize_connection()` where new socket fd is created and then sent to the `prep_connect()` function, indicating that it is a fresh connection and the event loop executes from the beginning.
  * **Buffer Capacity Reached:** If `cqe->res` is equal to maximum buffer threshold (`char[RECVBUF] = 8192 bytes` by default), and the connection is still active, we can re-use the connection to continuing to read the response.
  * **Default Action:** In the absence of the aforementioned conditions, the connection state reverts to `WRITE` state, readying itself to send another request to the server.

  The cumulative byte count for the thread is incremented by the total bytes received in the current operation (`cqe->res`).

3.3 Method 1: io_uring Implementation without Batching

Batching involves sending multiple requests to the kernel at once, allowing the system to handle other tasks without waiting for responses. However, using basic functions provided by the <liburing.h> interface, we can set up io_uring in the wrk benchmarking tool without batching any operations. This means each request is processed individually and cannot proceed to the next state until a Completion Queue Entry (CQE) is made. As io_uring is meant to be very fast due to asynchronous I/O and shared buffer space with the kernel, we expect that this approach will still contribute towards performance enhancement against the traditional epoll handling.

3.3.1 Implementation

The difference between batching and non-batching comes with how the application perceives the Completion Queue Entry (CQE). In this approach, we make use of an in-built <liburing.h> function io_uring_wait_cqe(). It is a blocking function. It means that, if application has submitted a request and it hasn't been completed yet, invoking this function will pause the connection until that specific request has been processed and updated in the CQE by the Kernel. However, the underlying mechanism of io_uring is still asynchronous, meaning that even though the application is waiting for a Completion Queue Entry (CQE), the kernel will continue to process other queued tasks concurrently. To use this method, we follow the general flow mentioned in Section 3.1. However, all the connections can be submitted to the Submission Queue Entry in a single io_uring_submit() call. Note that this is the **first** submission call in this approach.

After all the connection requests have been queued, the kernel retrieves and processes these requests, subsequently populating the Completion Queue (CQE) with the respective completion events. It's crucial to note that each event in the CQE corresponds to a specific connection, uniquely identified by its file descriptor. To access the connection queue, we **loop through each connection** within the event cycle. This implies that we'll pause and wait for the CQE to be ready for every individual connection. This sequential progression through the (CONNECT → WRITE → READ) lifecycle ensures that each connection is handled with dedicated attention, maintaining a clear order of operations. The absence of batching inherently means that the system does not group multiple

connections or their respective stages to be processed concurrently. Instead, after each state transition, the system **reinvokes** the `io_uring_submit()` function that is written before the event loop closes. This repetitive invocation, for every individual connection during the event cycle, guarantees that the kernel is kept busy and consistently updated with the latest state of each connection.

One primary motivation for adopting this approach is to demonstrate the potential of `io_uring` in its most basic form within the `wrk` benchmarking tool. By using `io_uring` in such a straightforward manner, we aim to highlight that even without harnessing its batching features, it can potentially outperform the traditional `epoll` mechanism. This will be tested and evaluated in the next chapter.

3.4 Method 2: `io_uring` implementation with Batching

In the second method, we harness the true potential of `io_uring` by leveraging one of its most powerful features: batching. Batching not only amplifies the performance by reducing the system call overhead but also optimizes the handling of multiple I/O tasks concurrently. This approach is particularly beneficial for the `wrk` benchmarking tool, where a high volume of requests are executed in rapid succession.

3.4.1 Implementation

The event loop, in this scenario, is executed in a different manner in comparison to the first method. As usual, the first batch of operations before entering the event loop will be the connection operations. In contrast to the **first approach** where connections are submitted **before** entering the event loop, in **this approach**, we make a solitary call to `io_uring_submit()` **after** entering the event loop. Instead of waiting for individual Completion Queue Entries (CQEs) for each connection, we use `io_uring_peek_batch_cqe()` function that allows the application to inspect multiple CQEs in the completion ring at once without consuming them. It gives a ‘peek‘ into the completion queue and fetches a batch of completed tasks. When using this function, the returned CQEs are stored in the `cqes` backlog array of size 4096 bytes. Once the `cqes` backlog is populated, the application **iterates over each CQE in the array** and then calls the **same** `io_uring_submit()` operation that was written at the beginning of the event loop. In contrast to the previous approach where submission was made for every state iteration of each connection, this method submits all the operations just once each

time the event loop is entered. This batch-processing mechanism significantly reduces the overhead associated with checking and processing CQEs, leading to an optimized flow especially when handling a large number of simultaneous connections.

The objective of this approach is to demonstrate enhanced single-thread performance. By comparing it with the baseline model and the first method (which implements `io_uring` without batching), we aim to highlight its capability to manage greater throughput. Given its optimized nature, this method is anticipated to exhibit a significant improvement on a per-thread basis. The subsequent chapter involves implementing both methods within a controlled client-server setting. By capturing and analyzing the results, we can undertake a comprehensive evaluation of their performance.

Chapter 4

Experiments and Evaluation

This chapter reports experimental results with the integration of the two `io_uring` methods into the `wrk` benchmarking tool, as outlined in the previous chapter. We first describe experiment methodology, and then report results and thorough analysis.

4.1 Experiments

4.1.1 Setup

Similar to the experiment in Chapter 1.1, we use a single-core CPU in the client that runs `wrk` and multiple cores in the server that runs the `nginx` web server. We use CloudLab, a research testbed that allows academic users to access bare metal machines for networking experiments. We book two nodes of type `xl170` on the `small-lan` profile, acting as a client and server, respectively. The nodes are equipped with Intel Xeon E5-2640 v4 processor, 64 GB RAM and two Mellanox ConnectX-4 25 GbE NICs, and install Ubuntu Linux 22.04.2 LTS. To ensure that the client and server use one or eight CPU cores, respectively, we disable other cores at the system level using the Linux CPU hotplugging feature; in particular we use the command:
`echo 0 > /sys/devices/system/cpu/cpu*/online`. We also ensure `wrk` uses only one thread. [3]

4.1.2 Methodology

To measure what techniques improve the performance, we compare three systems: unmodified `wrk` that employs `epoll`, modified one to use `io_uring` but without batching,

and the other custom one that exploits batching with io_uring. In the rest of this chapter, we denote those systems as epoll, io_uring-nb and io_uring, respectively.

The first performance metric is throughput. We use requests per second instead of bytes per second because we are interested in how io_uring reduces per-request processing time. When the throughput is higher, it means that the system is able to process more requests each second, hence reducing the per-request processing time. With the help of throughput, we will be able to analyze the performance improvement among the three systems used in benchmarking tests. The second performance metric is latency. We measure the application-level round trip time for the request and response, and report the median (P50) and 99%-ile (P99) values.

To emulate different numbers of clients that send requests to the same server, we vary the number of concurrent TCP connections. When the connection count is high and thus more requests are sent in parallel, more concurrent responses arrive at the client, forming a queue. Therefore, when wrk consumes the responses faster, the next requests can be sent earlier, resulting in higher throughput or requests per second. This translates into io_uring would achieve highest throughput, while epoll would exhibit the lowest. When the number of connections is small, there would be little or no difference between io_uring-nb and io_uring because of an insufficient number of requests or connections processed in a batch, mainly for system calls.

io_uring would increase the latency at a relatively small number of connections where the queuing delay (of responses to be processed) is negligible due to batching, but other benefits, such as asynchronous I/O, would outweigh. We experimentally demonstrate this later in this chapter.

4.2 Results

Figure 4.1 plots the throughput and latency over 1–50 concurrent connections. Throughput peaks at 30 connections in all the systems, where the latency starts increasing at a higher degree due to the increasing response backlog (queue). io_uring-nb always achieves higher throughput than epoll because of asynchronous I/O and zero copy, and io_uring outperforms io_uring-nb because of batching by approximately 13.76%. Lower latency of io_uring-nb than epoll at 1 connection comes from the same reason, and higher latency of those is due to higher throughput where more requests are sent in parallel. Interestingly, io_uring-nb exhibits the lowest latency at 50 connections. This is likely due to the eliminated batching delay while taking advantage of asynchronous

I/O and zero copy.

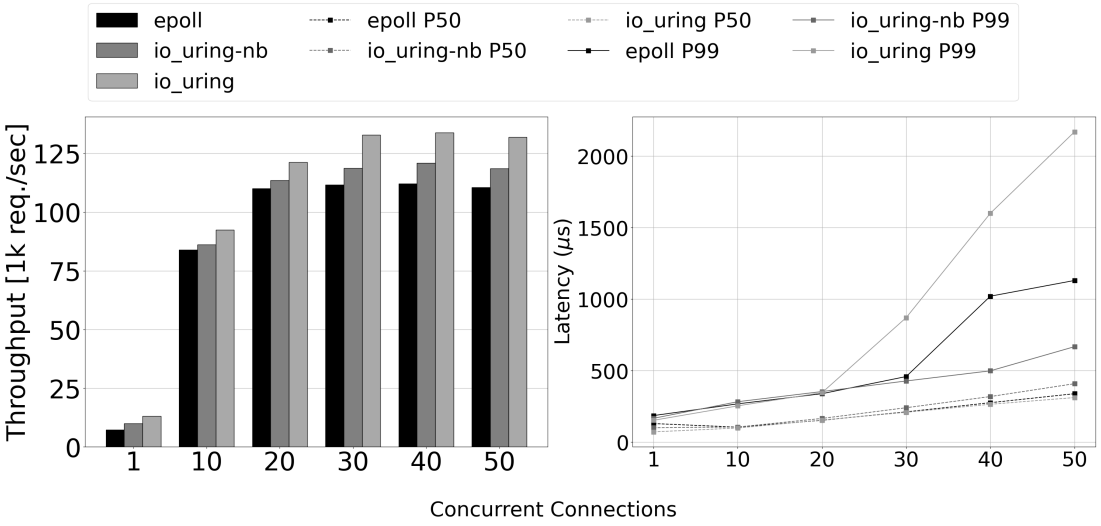

Figure 4.1: Throughput & Latency @ P50, P99 over low concurrent connections across `epoll`, `io_uring-nb` & `io_uring`

Figure 4.2 plots the throughput and latency over 100-600 concurrent connections. As before, throughput is always higher in the order of `io_uring`, `io_uring-nb` and `epoll`. Due to batching, the degree of improvement with `io_uring` is larger when more concurrent connections are used. As in 50 connection cases, `io_uring-nb` achieves the lowest latency. Latency keeps accelerating for `epoll` with an increase in connections, thus proving the decline in throughput observed after 100 connections count. It has the lowest latency at 100 connections, where it was able to process the highest throughput, after which it becomes stagnant.

Unlike `epoll` and `io_uring-nb` that start with low latency, `io_uring` starts with highest latency at 100 connections, as it can process more requests at a faster rate. However, as the connections grow, `io_uring` shows the most stability with its latency where as `epoll` and `io_uring-nb` increase gradually. This in turn reflects `io_uring` being the fastest mechanism, showing how well it is able to efficiently use its batching mechanism with increasing load. The difference between the throughput of `epoll` and io_uring is much more than that in the 1-50 connections range. This indicates that with higher load, `epoll` is unable to utilize the CPU core as efficiently as `io_uring` can with batching of highly concurrent requests.

We detail the absolute throughput and improvement in Table 4.1. For connections 1-50, `io_uring-nb` shows 10.05% of average improvement in throughput than `epoll`. On the other hand, `io_uring` can process requests per second at 25.45% faster rate

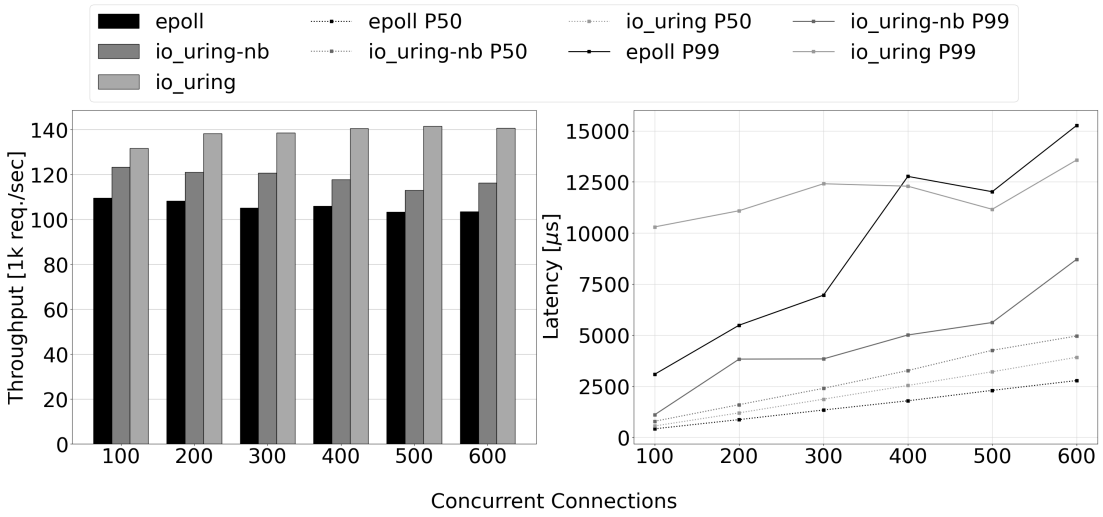

Figure 4.2: Throughput & Latency @ P50, P99 over high concurrent connections across `epoll`, `io_uring-nb` & `io_uring`

than `epoll` on average. For 100-600 connections, a similar trend is observed where both implementations perform much better than `epoll`. `io_uring-nb` improves it's throughput at higher connections by 12.08%, leveraging the underlying asynchronous architecture. By maximizing the use of its batching capabilities, `io_uring` demonstrates superior performance under heavy loads, achieving a 31% enhancement over epoll.

On comparing the performances of both the `io_uring` implementations in low vs high concurrent connections, `io_uring-nb` processes requests at 25% faster rate with high connections, indicating that the asynchronous nature of `io_uring` can significantly improve throughput over epoll event without batching. `io_uring` shows upto 20% improvement in processing requests at high load than the lower, proving that batching capabilities are more significant with an increase in connections. Overall, it can be established that `io_uring` has the fastest throughput rate, followed by `io_uring-nb` and then `epoll` at all configurations.

Another experiment was done to study the effect on performance metrics for low connections (1-50) by turning off `adaptive-rx`, `adaptive-tx` and setting `rx-usecs` to 0μs. This ensures that packet transmission/reception rates remain constant and the NIC interrupts are not delayed for low-traffic scenarios respectively. It means that the CPU processes all the requests immediately without waiting, which could potentially improve throughput and reduce latency. However, upon analyzing the statistics in both scenarios (with `adaptive-rx`, `adaptive-tx` on, and by resetting `rx-usecs`), there was little to no change observed in the thread statistics for all 3 mechanisms. The

Connections	Requests/Second			% Increase	
	epoll	**io_uring-nb**	**io_uring**	**io_uring-nb**	**io_uring**
1	9165.46	10297.80	13975.44	12.35	52.48
10	82650.93	89819.30	91812.70	8.19	11.12
20	106413.40	115722.21	124675.15	8.73	17.29
30	107756.67	116569.16	132979.22	8.18	23.43
40	107058.54	119447.13	133550.76	11.60	24.74
50	106929.44	118973.17	132065.93	11.26	23.65
100	108595.67	122302.34	130188.56	12.62	19.88
200	106732.59	121868.95	137093.16	14.18	28.45
300	105850.69	118880.10	138388.17	12.31	30.74
400	105422.16	117217.22	139991.57	11.19	32.79
500	103639.75	116844.19	140219.73	12.74	35.30
600	105330.36	115283.92	141253.61	9.45	34.11

Table 4.1: Percentage Improvement in Throughput between `epoll` against `io_uring-nb`, `io_uring`

throughput was almost the same and there was negligible change in latency. This is because the latency observed in the experiment was not due to the interrupts but could be due to network stack processing or context switches at low connections where batching is not completely effective. Eventually, interrupt delays and transmission/reception rates did not play a significant role for low connections (1-50), rendering performance of `epoll`, `io_uring-nb` and `io_uring` unaffected.

In the `wrk` benchmarking tool, the default response buffer size is 8192 bytes. Performance of `epoll`, `io_uring-nb`, and `io_uring` was evaluated over buffer sizes of 512, 1024, 2048, and 4096 bytes by conducting similar experiments as above. Upon evaluation, a recurring pattern was evident: `io_uring` consistently outperformed, trailed by `io_uring-nb` and then `epoll`. This indicates the efficiency of these mechanisms and their ability to handle varied buffer sizes seamlessly. Additionally, the effective buffer overflow management in `epoll` and `io_uring` ensures optimal performance irrespective of buffer dimensions. The negligible performance variations might also be attributed to consistent network conditions, server response times, and potential optimizations within the `wrk` benchmarking tool.

Chapter 5

Conclusion and Future Works

5.1 Conclusion

The primary objective of this research was to explore the potential of enhancing the performance of the `wrk` benchmarking tool by integrating it with the `io_uring` asynchronous I/O framework. `wrk` employs the `epoll` I/O mechanism, which relies on system calls to manage communication between the user and kernel spaces. As the number of concurrent connections rises, the frequency of these system calls increases, causing a significant overhead. Due to this, the rate at which requests are processed with the increase in connections becomes slower. This causes an increase in the requests queue backlog, eventually leading to inefficient CPU usage.

In contrast to `epoll`, `io_uring` interface uses a shared space between the user and kernel, resulting in reduced system calls due to batching of operations and asynchronous I/O. We believe that adding `io_uring` to the `wrk` benchmarking tool would make it perform better and handle more requests than before. To check this, we tried two methods: one using batching (`io_uring`) and one without (`io_uring-nb`). The main difference between these two is the batching of operations while sending requests to the submission queue. `io_uring-nb` repeatedly submits operations to the queue for each connection and waits until a response is received. On the other hand, `io_uring`, can send multiple requests to the submission queue at once without waiting for any response. However, the underlying architecture is asynchronous in both cases, which contributes to enhanced performance.

After conducting the benchmarking experiments, it was evident that both the implementations of `io_uring` outperformed `epoll` mechanism. `io_uring` showed the best performance with up to 33% improvement than the traditional model. `io_uring-nb`

was also better than epoll by 12%. This improvement suggests that both the mechanisms were able to process requests at a faster rate than `epoll`, thus resulting in accelerating our benchmarking tool, `wrk`. This observation remained unchanged even when there were varying concurrent connections, response buffer sizes, submission and completion ring `QUEUE_DEPTH`. In every test we conducted, both new implementations outperformed the traditional `epoll` mechanism. However, while the batching approach excelled under high loads, it was less effective with a limited number of connections. This is because batching achieves its peak efficiency when there are ample requests to group together. When faced with fewer connections, the batching mechanism doesn't fully utilize its resources, leading to suboptimal performance.

In conclusion, the thesis was deemed successful with the implementations of `io_uring` in the benchmarking tool, achieving significant improvement over `wrk's` existing I/O mechanism. This accomplishment not only validates the potential of `io_uring` as a robust alternative to traditional I/O mechanisms but also sets a precedent for future research and optimization in this domain. The findings highlight the importance of choosing the right I/O mechanism based on specific use-cases and workloads. While `io_uring` showed immense improvement, particularly in high-load scenarios, the nuances observed in different implementation approaches underscore the need for a thorough understanding of the system's requirements. Going forward, these insights can serve as a foundation for further refining benchmarking tools, ensuring they are both scalable and efficient across diverse scenarios.

5.2 Future Works

Though the thesis is deemed successful, there are scopes for improvement in the following areas:

- **Multi-core support:** As mentioned throughout the thesis, the current implementation is confined to a single-threaded core client. Modern systems often have multi-core architectures, and many real-world applications leverage multiple cores for enhanced parallelism and performance. Investigating how `io_uring` performs in a multi-core environment will provide insights into its scalability and efficiency on modern hardware. By exploring thread-safe data structures and synchronization mechanisms tailored for `io_uring`, `wrk` benchmarking tool can effectively make use of it in major situations.

- **Extended Protocol and Scripting Support:** `wrk` is renowned for its extensibility, chiefly due to its support for `LuaJIT scripts`. These scripts empower users to define custom HTTP request patterns and simulate diverse client behaviors. With `LuaJIT scripts` defining custom concurrency and request patterns, `io_uring` can be further optimized to work in tandem with these patterns. While io_uring offers a robust foundation for asynchronous I/O in `wrk`, its true potential can be unlocked by closely integrating it with the tool's `LuaJIT` scripting capabilities. By doing so, not only can we ensure that `wrk` remains versatile in benchmarking a plethora of web scenarios, but we also ensure that each of these scenarios is handled with optimal efficiency and speed. Future research and development in this direction could redefine the boundaries of web benchmarking, making `wrk` an even more indispensable tool for performance engineers worldwide.

- **Benchmarking on Diverse Hardware:** Broadening `wrk's` scope to operate seamlessly across varied CPU architectures, storage devices, and network configurations is essential. With `io_uring` integrated, testing on diverse hardware will provide richer insights into its asynchronous I/O benefits and ensure consistent benchmarking outcomes irrespective of the underlying hardware platform.

Bibliography

[1] Boran Car Agniva De Sarker. Demo api to play with io_uring in go. 2020. Accessed: 2023-08-02.

[2] Jens Axboe. Efficient io with io_uring. 2021. Accessed: 2023-08-23.

[3] Jens Axboe. That's it. 10m iops, one physical core. `https://twitter.com/axboe/status/1452689372395053062`, 2021. Accessed: 2023-08-23.

[4] H. Chen, P. Luszczek, and R. Brandenburg. Benchmarking filesystems and storage systems: A survey of current and future trends. *ACM Computing Surveys (CSUR)*, 53(4):1–36, 2020.

[5] Jonathan Corbet. Ringing in a new asynchronous i/o api. 2019. Accessed: 2023-08-23.

[6] Jonathan Corbet. The rapid growth of io_uring. 2020. Accessed: 2023-08-23.

[7] Hielke de Vries. io_uring echo server benchmarks. 2021. Accessed: 2023-08-02.

[8] Rust docs. Crate io_uring. 2021. Accessed: 2023-08-23.

[9] Rust Docs. wrk - a http benchmarking tool. 2023. Accessed: 2023-08-02.

[10] Felipe Dutra Tine e Silva. Intelligent benchmark with wrk. 2023. Accessed: 2023-08-02.

[11] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Number RFC 2616, June 1999. Obsoletes: RFC 2068.

[12] Gabriel Haas, Michael Haubenschild, and Viktor Leis. Exploiting directly-attached nvme arrays in dbms. In *10th Conference on Innovative Data Systems Research (CIDR 20)*. www.cidrdb.org, 2020. Online Proceedings.

[13] Wander Hillen. Preliminary benchmarking results for a haskell i/o manager backend based on io uring. *Journal Name or Conference Proceedings*, Volume Number(Issue Number):Page Range, 2020. Any additional information you'd like to add.

[14] Shuveb Hussain. io_uring by example: An article series. *https://unixism.net/2020/04/io-uring-by-example-article-series/*, Volume Number(Issue Number):Page Range, 2020. Any additional information you'd like to add.

[15] Shuveb Hussain. A web server with liburing. *https://unixism.net/loti/tutorial/webserver_liburing.html*, Volume Number(Issue Number):Page Range, 2020. Any additional information you'd like to add.

[16] Aarushi Mehta. block/io_uring: enable kernel submission polling. 2019. Accessed: 2023-08-02.

[17] openresty. Accurate performance testing with 'wrk'. 2023. Accessed: 2023-08-02.

[18] Andrew J. Phelps. ink - an http benchmarking tool. 2020. Accessed: 2023-08-02.

[19] F. Quaglia, M. D. Santambrogio, L. Abeni, R. Giorgio, and M. Paolieri. Boosting user-level networking with io_uring. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 1609–1619, Jul 2021.

[20] D. Salim and A. D. Giorgio. Evaluating the performance of io_uring for high-speed networking applications. *IEEE Access*, 9:143487–143497, 2021.

[21] Agniva De Sarker. Getting hands on with io_uring using go. 2020. Accessed: 2023-08-02.

[22] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, page 33–46. USENIX Association, 2010.

[23] Julia Suvorova. io_uring in qemu: high-performance disk io for linux. 2020. Accessed: 2023-08-02.

[24] Animesh Trivedi. Understanding modern storage apis: A systematic study of libaio, spdk, and io uring. 2021.

[25] Vishal Verma and John Kariuki. Improved storage performance using the new linux kernel i/o interface. https://www.snia.org/educational-library/improved-storageperformance-using-new-linux-kernel-io-interface-2019, 2021. Accessed: 2023-08-23.

[26] WiredTiger. Implement asynchronous io using io_uring api. https://jira.mongodb.org/browse/WT-6833, 2021. Accessed: 2023-08-23.

Appendix A

First appendix

Steps to test the `wrk` benchmarking with `io_uring` and `io_uring-nb`

A.1 Environment Setup

Step 1: There have to be at least 2 systems, one client (node0) and one server (node1), that are able to communicate over a common NIC(ens1f1np1). Below are the systems that have been mentioned in Section 4.1.1 :

- **Client (node0):** Check the public IP address over ens1f1np1 NIC:

```
User@node0:~$ ip addr show
 .
 .
 .
5: ens1f1np1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
   qdisc mq state UP group default qlen 1000
    link/ether 9c:dc:71:49:a8:c1 brd ff:ff:ff:ff:ff:ff
    altname enp3s0f1np1
    inet 10.10.1.1/24 brd 10.10.1.255 scope global ens1f1np1
       valid_lft forever preferred_lft forever
    inet6 fe80::9edc:71ff:fe49:a8c1/64 scope link
       valid_lft forever preferred_lft forever
```

node0 IP: 10.10.1.1

- **Server (node1):**

  Similarly, check the public IP address over ens1f1np1 NIC:

```
User@node1:~$ ip addr show
.
.
.
5: ens1f1np1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
 qdisc mq state UP group default qlen 1000
    link/ether 9c:dc:71:4b:a3:71 brd ff:ff:ff:ff:ff:ff
    altname enp3s0f1np1
    inet 10.10.1.2/24 brd 10.10.1.255 scope global ens1f1np1
       valid_lft forever preferred_lft forever
    inet6 fe80::9edc:71ff:fe4b:a371/64 scope link
       valid_lft forever preferred_lft forever
```

node1 IP: 10.10.1.2

Step 2: Ensure that both the machines can communicate with each other:

- Client(node0) to Server(node1):

```
User@node0:~$ ping 10.10.1.2
PING 10.10.1.2 (10.10.1.2) 56(84) bytes of data.
64 bytes from 10.10.1.2: icmp_seq=1 ttl=64 time=0.300 ms
64 bytes from 10.10.1.2: icmp_seq=2 ttl=64 time=0.127 ms
64 bytes from 10.10.1.2: icmp_seq=3 ttl=64 time=0.129 ms
64 bytes from 10.10.1.2: icmp_seq=4 ttl=64 time=0.127 ms
^Z
[1]+  Stopped                 ping 10.10.1.2
```

- Server(node1) to Client(node0):

```
User@node1:~$ ping 10.10.1.1
PING 10.10.1.1 (10.10.1.1) 56(84) bytes of data.
64 bytes from 10.10.1.1: icmp_seq=1 ttl=64 time=0.127 ms
64 bytes from 10.10.1.1: icmp_seq=2 ttl=64 time=0.128 ms
64 bytes from 10.10.1.1: icmp_seq=3 ttl=64 time=0.129 ms
64 bytes from 10.10.1.1: icmp_seq=4 ttl=64 time=0.224 ms
64 bytes from 10.10.1.1: icmp_seq=5 ttl=64 time=0.101 ms
^Z
[1]+  Stopped                 ping 10.10.1.1
```

Step 3: Check number of online CPUs in node0 and node1. Set them to 1 (CPU 0) in client and 8 (CPUs 0 - 7) in server using CPU hotplugging[1]

- Client

```
User@node0:~$ nproc
20
```

Execute below shell script in root:

```
#!/bin/bash
for i in {1..19}; do
    echo 0 > /sys/devices/system/cpu/cpu$i/online
done
```

Verify the online CPUs:

```
User@node0:~$  cat /sys/devices/system/cpu/online
0
```

- Server:

[1] https://www.cyberciti.biz/faq/debian-rhel-centos-redhat-suse-hotplug-cpu/

```
\begin{verbatim}
User@node1:~$ nproc
20
```

Execute below shell script in root:

```
#!/bin/bash
for i in {8..19}; do
    echo 0 > /sys/devices/system/cpu/cpu$i/online
done
```

Verify the online CPUs:

```
User@node1:~$  cat /sys/devices/system/cpu/online
0-7
```

A.2 Server Setup (node1)

Steps to install `nginx`: **Step 1:** Setting up `nginx` server on node1

Execute below commands in the server:

```
sudo apt update
sudo apt install nginx
sudo systemctl start nginx
sudo systemctl enable nginx
```

Step 2: Check if `nginx` server is running

```
User@node1:~$ sudo systemctl status nginx
 nginx.service - A high performance web
 server and a reverse proxy server
Loaded: loaded (/lib/systemd/system/nginx.service;
 enabled; vendor preset: enabled)
Active: active (running) since Thu 2023-08-24 00:59:00 MDT; 19s ago
```

```
Docs: man:nginx(8)
Main PID: 4716 (nginx)
Tasks: 9 (limit: 76935)
Memory: 11.1M
CPU: 46ms
```

Step 3: Verify that port 80 is listening using

```
netstat -tpln
```

A.3 Client Setup (node0)

Steps to install `wrk` benchmarking tool

Step 1: Install `liburing` to access `io_uring` interface using below commands:

```
sudo apt update
sudo apt install -y make gcc libssl-dev
git clone https://github.com/axboe/liburing.git
cd liburing
make
sudo make install
```

Step 2: Clone `wrk` code with `io_uring` batching[2] or code without `io_uring` batching[3]

```
git clone https://github.com/Nithya-2018/io\_uring\_batching.git
cd io\_uring\_batching
sudo make
```

A.4 Benchmarking

1. Executing with normal wrk:

[2]https://github.com/Nithya-2018/wrk_io_uring_no_batch.git, https://github.com/Nithya-2018/wrk_io_uring_no_batch/tree/master-1

[3]https://github.com/Nithya-2018/wrk_io_uring_no_batch.git, https://github.com/Nithya-2018/wrk_io_uring_no_batch/tree/master-2

```
User@node0:~io_uring_batching/$ ./wrk -t1 -c100 -d5
http://10.10.1.2:80
Running 5s test @ http://10.10.1.2:80
  1 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   551.63us  311.23us   4.28ms   84.23%
    Req/Sec   108.32k     2.33k  109.79k    98.00%
  538669 requests in 5.00s, 441.26MB read
Requests/sec: 107674.92
Transfer/sec:     88.20MB
```

2. Executing with io_uring batching:

```
User@node0:~io_uring_batching/$ ./wrk -t1 -c100 -d5
http://10.10.1.2:80 -i
Using io_uring!
Running 5s test @ http://10.10.1.2:80
  1 threads and 100 connections
Connection created!
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   662.26us  179.46us   9.58ms   93.90%
    Req/Sec   150.28k     5.77k  153.51k    96.08%
  773414 requests in 5.16s, 637.54MB read
Requests/sec: 149870.20
Transfer/sec:    123.54MB
```

3. Executing with io_uring no batching:

```
User@node0:~/io_uring_nb/io_uring_nb$ ./wrk -t1 -c100 -d5
http://10.10.1.2:80 -i
Using io_uring!
Running 5s test @ http://10.10.1.2:80
  1 threads and 100 connections
Connection created!
```

```
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   833.98us   68.05us   4.28ms   88.60%
    Req/Sec   118.22k     3.55k  121.02k    96.00%
  600976 requests in 5.09s, 494.59MB read
Requests/sec: 117963.22
Transfer/sec:     97.08MB
```