SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Efficient Networking for Transactional Cloud Database Systems

Emil Suleymanov

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Efficient Networking for Transactional Cloud Database Systems

Effiziente Netzwerkkommunikation für transaktionale Cloud-Datenbanksysteme

Author:	Emil Suleymanov
Supervisor:	Viktor Leis
Advisor:	Gabriel Haas
Submission Date:	15.10.2024

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

Munich, 15.10.2024 Emil Suleymanov

Abstract

This thesis investigates efficient networking techniques for transactional cloud database systems, with a focus on the modern Linux I/O interface `io_uring`. Motivated by the increasing network bandwidth offered by cloud platforms like Amazon Web Services (AWS) Elastic Compute Cloud (EC2), we explore how `io_uring` can be leveraged to optimize network utilization in cloud database architectures.

Through our experiments on AWS EC2 instances, we compare the performance of `io_uring` implementations against traditional `POSIX` networking approaches. Our results demonstrate that `io_uring` consistently outperforms `POSIX` in terms of throughput, CPU efficiency, and handling of concurrent connections. We observe up to 22% higher throughput and significantly reduced context switches with `io_uring`.

We also examine the challenges of network performance variability in cloud environments and explore alternative approaches like DPDK and `tokio-uring`-based implementations. While `io_uring` shows clear benefits, we highlight the complexities of optimizing network performance in cloud settings.

This research provides insights into the potential of `io_uring` for improving network efficiency in cloud database systems and lays the groundwork for future investigations into integrating advanced networking techniques with specific cloud database architectures.

Contents

1. Introduction

The rise of cloud computing, including cloud databases, has transformed the way organizations manage their data and infrastructure. As businesses increasingly rely on cloud-based services, the need for efficient networking solutions has become paramount, particularly in the context of transactional cloud database systems.

In this context, the AWS EC2 platform, with its affordable high-speed 200 Gbit/s networking capabilities announced in 2023 as part of the new `c7gn` instances launch, offers a compelling solution for optimizing the performance and scalability of cloud-based transactional databases, making the management of network resources one of the key challenges in cloud-based transactional databases. And as the offered network bandwidth of AWS EC2 instances and the likes grows and the cost shrinks, the question of utilization of this newfound network bandwidth becomes more crucial.

One such cloud-native architecture following the Database as a Service (DBaaS) paradigm in the cloud is Socrates [2]. By decoupling compute and storage, Socrates aims to achieve better scaling and availability. One of the elements of Socrates architecture is separate page servers, which store and serve database partitions' pages to compute nodes, which serves the goal of decoupling storage and compute, but creates a lot of network traffic as a result.

As such, this thesis evaluates modern networking techniques that can help with compute efficient network utilization given typical patterns of requests-replies seen in cloud-based transactional database systems, namely page services; available in conjunction with the Linux kernel. This thesis presents an evaluation of high-network bandwidth instances on AWS EC2 with iperf2/3, an implementation using `io_uring` via liburing, and also using a co-routine framework, namely, tokio uring; and a version made using Data Plane Development Kit (DPDK). Due to time and budget constraints, the DPDK implementation is not explored fully and is covered only briefly.

With these advanced networking capabilities, the challenges of efficient bandwidth utilization in cloud database environments are addressed, potentially leading to improved performance and cost-effectiveness of cloud-based transactional database systems.

2. Related Work

There do not seem to be any works strongly related to this thesis's topic. However, there are many projects and works addressing the efficiency of networking.

There are many libraries and frameworks built to optimize networking applications. Many of these projects, such as mTCP [18], Seastar [21], F-Stack [8], and many others; focus on bringing the network stack into the user space, bypassing the kernel and its limitations altogether. mTCP, Seastar, and F-Stack are built on top of DPDK, while Seastar also supports a `io_uring` back-end.

The authors of mTCP [16] identified a few issues with the traditional handling of Transmission Control Protocol (TCP) connections via the kernel, particularly handling of the short-lived TCP connections; these include high syscall overhead, resource contention across Central Processing Unit (CPU) cores, inefficient per-packet processing, and limited scalability. The kernel would spend an large portion of the CPU cycles on TCP connection management, which leaves less compute for the application processing. The authors addressed these issues by designing mTCP, a user-level TCP stack, which bypasses the built-in kernel TCP stack completely, minimizing the syscall overhead by batching packet I/O and other socket events. Additionally, the authors implemented per-core connection management, which reduces inter-core resource contention, improving multi-core scalability.

Seastar is designed with the limitations of conventional networking approaches in mind; mainly, issues tied to multi-threaded and kernel-based implementations. One of the issues Seastar addresses is the overhead of locking present in multi-core threaded environments required for coordination. In a more traditional setup, this leads to performance issues caused by wasted CPU cycles and cache contention. Seastar addresses this with a shared-nothing architecture. Each core is handling its own requests, avoiding shared memory and locks, using instead message passing for coordination. Seastar uses futures and promises abstractions to handle asynchronous operations, which enable non-blocking multitasking. A future is a result that will be available later, and a promise will provide the results when it is ready. Using these abstractions, Seastar can chain operations using $.then()$, and execute tasks without waiting, unlike traditional blocking models [21].

In a 2021 study presented at EuroBSDCon, Drew Gallatin demonstrated how Netflix was able to serve video content at 400 Gbit/s using FreeBSD with hardware support

from Mellanox's ConnectX-6 Dx Network Interface Cards (NICs) [9]. Netflix achieved this throughput by utilizing inline hardware Transport Layer Security (TLS) (kTLS) offload, where encryption tasks are handled directly by the NIC, reducing the burden on the CPU. Additionally, Non-Uniform Memory Access (NUMA)-aware architecture was leveraged to minimize memory bandwidth bottlenecks. By aligning network traffic and data storage with specific NUMA nodes, Netflix reduced costly cross-domain memory accesses, which decrease system efficiency in multi-core architectures. The authors talked about how it is critical to offload CPU-intensive tasks like encryption and leverage user-space networking techniques to minimize latency and overhead in high-throughput environments.

In a blog post by Marek Majkowski from Cloudflare [17], the limitations of the traditional Linux kernel in handling large-scale network traffic were explored. The authors demonstrated that while the kernel can handle approximately 1.4 million packets per second on a single core in their setup, its performance degrades significantly when packets are spread across multiple cores. As a solution, kernel bypass techniques such as DPDK, *PF_RING*, and Netmap were discussed. These frameworks enable user-space applications to directly access NICs, bypassing the kernel's network stack and reducing the overhead of packet processing. Cloudflare's use of kernel bypass allows them to handle high-speed traffic more efficiently by dedicating specific NIC queues to user-space processes, enhancing packet processing rates beyond what the kernel could handle. The author points out that at the time of writing, there are no open-source kernel bypass APIs available, and one must use a supported NIC, which is then usually taken over completely.

3. Background

In this chapter, we will explore the background of the problem we are trying to solve, including the challenges and the state of the art in the field. We will also introduce the technologies that we will be using in the experiments, such as AWS EC2, DPDK, and `io_uring`.

3.1. Transactional Cloud Database Systems

Transactional cloud database systems are a type of database systems designed to handle transactions separately from others, and commit them only if they can be fully executed, ensuring Atomicity, Consistency, Isolation, Durability (ACID) properties. These systems are optimized to read and write data entries at high speeds while ensuring data integrity. These databases utilize Online Transaction Processing (OLTP), which enables such database systems to execute a multitude of transactions concurrently [22]. This makes it possible to process data from many users, often accessing and/or modifying the same data [23].

In 2016, Snowflake introduced a novel model of separating systems into three layers: data storage, virtual warehouses, and cloud services; or in other terms: storage, compute, and control layers [7].

Socrates, a novel DBaaS architecture introduced in a 2019 paper from Microsoft [2], builds on the core ideas of cloud data warehousing, while also introducing unique advancements. While the ideas introduced by Snowflake are seen in the Socrates architecture, Socrates emphasizes even further breaking up of storage and compute, enabling new efficiencies in large-scale transactional workloads. The design of Socrates followed several goals, which included separation of storage and compute with further log separation and tiered storage. As a result, we have four tiers of separation: compute tier - which applications connect to; it handles transactions, log tier - it provides a separation of log by implementing XLOG, storage tier - which is provided by page servers, which stores copies of partitions of database, allowing for scaled-out storage, and mainly serve the pages to the compute nodes on-demand, while also managing checkpointing and backups onto XStore, which brings us to the fourth tier - durable and cheap storage - provided by XStore, a storage service by Microsoft [2]. As such,

there is a need for efficient networking for all of the components, with our focus being mainly on the page servers.

3.2. Shared-Nothing Architecture

Shared-nothing architecture is a distributed computing approach, where each node, or in our case each thread, of a system is designed to be independent and as self-sufficient as possible, with its own separate resources such as memory, storage, and processing power. In this architecture, each node or a worker does not share any resources, and instead communicates with the rest of the nodes through asynchronous channels.

3.3. io_uring

Efficient network I/O becomes crucial for maintaining high performance and scalability. Traditional networking APIs in Linux, such as `select`, `poll`, `epoll`, and `aio`, while functional, often struggle to fully utilize modern high-speed networks, especially in scenarios with high concurrency and low latency requirements. This is where modern networking techniques, particularly `io_uring`, come into play. Unlike traditional syscalls, which are blocking, `io_uring` offers a generic asynchronous syscall interface, which can be used to submit I/O requests to the kernel, while the application can continue processing other tasks. As a result, where in the case of traditional syscalls such as `poll` or `epoll`, where we tell the kernel to notify our process when an event occurs, for us to then take action, we can now tell the kernel what we want to happen on that event, and then continue processing other tasks, while the kernel will notify us when the operation is complete.
`io_uring`, initially introduced by Jens Axboe in 2019, represents a significant advancement in Linux's asynchronous I/O capabilities [4]. The original paper [4] claims that using their machine as an example, they could achieve 608K Input/Output Operations per Second (IOPS) with `aio`, while using `io_uring` they could reach 1.7M IOPS with polling and 1.2M without. `io_uring` offers a more efficient and flexible approach to handling I/O operations, which is particularly beneficial for high-performance applications like cloud-based transactional databases. The key features of `io_uring` that make it particularly suitable for this context include:

- Shared Ring Buffers & Batched Submissions: `io_uring` utilizes two shared ring buffers between kernel and user space - a Submission Queue (SQ) for submitting I/O requests and a Completion Queue (CQ) for receiving completion notifications. This design minimizes the need for context switches and system calls, reducing

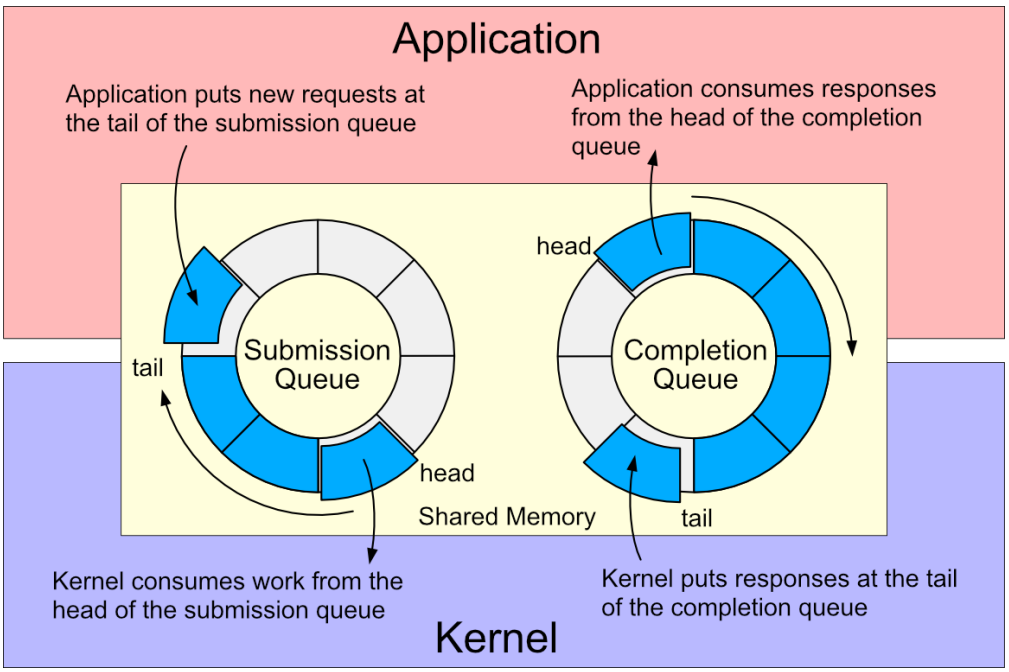

Figure 3.1.: `io_uring` ring buffers diagram. Courtesy - Donald Hunter [13].

computational overhead. See Figure 3.1. Additionally, multiple I/O operations can be submitted in a single system call, further reducing the computational overhead associated with syscalls. This is particularly crucial nowadays since mitigation of CPU exploits such as Spectre and Meltdown resulted in the increase of CPU cycles spent in each syscall [19].

- Polling Mode: In addition to the interrupt-driven mode, `io_uring` offers a polling mode where the kernel actively checks for new submissions. This can significantly reduce latency in high-throughput scenarios but can introduce unnecessary CPU load.

- Zero-copy Design: The shared ring buffer architecture allows for zero-copy operations in many scenarios, improving performance by reducing data movement between kernel and user space.

As pointed out earlier, the `io_uring` communication between the application and the kernel is done using ring buffers. The application writes the requests to the ring buffer, and the kernel reads them and processes them. The kernel also writes the completions to the ring buffer, which the application reads. Both the kernel and the application can

write into the ring memory. When there is a new event from the kernel, it stores that event in the completion queue, and the kernel updates the tail of that ring buffer; and when the application reads the completions, it updates the head of the completion ring buffer. By comparing the head and tail of the ring buffer, the application can determine if there are any new completions.

For cloud-based transactional database systems like Socrates, which involve separate page servers, `io_uring` offers several potential benefits. It can significantly improve the efficiency of network I/O between compute nodes and page servers, reduce CPU overhead, improve latency in page requests and responses, and enhance scalability as the number of concurrent I/O operations increases.

We explore how `io_uring` can be applied to optimize network utilization in these systems, evaluate its performance characteristics, compare it with traditional networking approaches, and assess its potential impact on overall system efficiency and scalability.

3.4. Data Plane Development Kit

The DPDK is a set of libraries and drivers for fast packet processing in data plane applications. It was originally developed by Intel and is now managed by the Linux Foundation. DPDK is particularly well-suited for applications that require high-speed network performance, such as Network Functions Virtualization (NFV), Software-Defined Networking (SDN), and, relevant to our context, high-performance database systems.

Key features and properties of DPDK that can benefit cloud-based transactional database systems include:

- Kernel Bypass: DPDK bypasses the kernel network stack, allowing applications to directly access network interfaces. This significantly reduces the overhead associated with kernel involvement in packet processing, since we can avoid frequent expensive syscalls.
- Poll Mode Drivers (PMDs): Instead of relying on interrupts, DPDK uses a polling mechanism to check for incoming packets.
- Huge Page Support: DPDK uses huge pages for memory allocation. This reduces Translation Lookaside Buffer (TLB) misses, which improves performance.
- NUMA-aware Memory Management: DPDK is designed to work efficiently on NUMA systems, ensuring optimal memory usage across multiple processors. This is particularly relevant for large machines.

- Zero-copy Packet Processing: DPDK allows for zero-copy packet processing, minimizing data movement and CPU cache pollution.
- Batch Packet Processing: DPDK processes packets in batches, avoiding the cost of multiple function calls.

In the context of cloud-based transactional database systems like Socrates, DPDK offers significant potential benefits for the page server component. The high-speed packet processing capabilities of DPDK can improve the efficiency of data transfer between compute nodes and page servers.

3.5. Tokio

Rust is a memory-safe programming language that gained traction in the past few years; however, being a memory-safe language [20], it is not usually loved in the databases community; nevertheless, it is still an interesting case to investigate.

Tokio is a popular asynchronous runtime for Rust [25]. As an event-driven and non-blocking I/O runtime, it provides a solid foundation for building fast, reliable, and scalable concurrent applications, making it a common choice for safety-critical and/or data-intensive applications, such as Cloudflare's Pingora library for network services [6], or services at Discord [12] and Kraken [5]; and was recently promoted by the White House as part of their press release on defense in cyberspace [11]. In this thesis, we take a look at the `io_uring` runtime supported by tokio via an experimental library - tokio-uring [26].

3.6. Seastar

Seastar is a C++ framework built with high performance in mind. Similar to tokio, it is built around an asynchronous programming model, with tasks being executed in a non-blocking fashion by an event loop, allowing for high concurrency. It is also highly optimized for networked applications and is used in ScyllaDB [21], a high-performance NoSQL database.

4. Experiments

In this chapter, we will explore the experimental setup and methodology used to evaluate the performance of the client-server implementations. We will also discuss the results obtained from these experiments and the considerations made during the process.

Unless stated otherwise, all plots have been generated using Python and Matplotlib [24].

4.1. Nuances of AWS EC2 networking

AWS EC2 networking has a few caveats which can affect the networking performance in unexpected ways. One such caveat is that the offered network bandwidth may not always be fully provided. Tests using iperf2/3 on `c6in.32xlarge` instances in Frankfurt showed maximum speed of only ∼176 Gbit/s (see Figure 4.1), while the same type of instances but in the US East region reached ∼196 Gbit/s; both regions promise 200 Gbit/s on this instance type.

Since `c6in.32xlarge` is quite expensive at ∼$7 per hour, further benchmarks use `c7gn.16xlarge`, which also promises a 200 Gbit/s network connection, but costs ∼$4 per hour. Additionally, US East instances usually cost half of Frankfurt instances, and so US East was used for the rest of the experiments. Given the similarity between iperf2 and iperf3, and a slightly better performance with iperf3, only iperf3 is used in the later measurements as to save cost.

Additionally, a single-threaded connection does not fill the whole bandwidth, and one must use multiple streams at the same time. You can see in Figure 4.1 that it reaches the maximum bandwidth before reaching 64 threads. In Figure 4.2 we can see a more granular chart for iperf3 on an `c7gn.16xlarge` instance, which shows that it fills the bandwidth at 32 threads. This behavior is in line with the Amazon EC2 instance network bandwidth documentation page, which says that a single-flow is limited to 5 Gbit/s, and in the same cluster to 10 Gbit/s [1]. To ensure that the machines are in the same cluster, one should request multiple identical machines in a single request.

Furthermore, AWS may throttle instances that are running long-term, high-load network applications. It is important to keep this in mind when running a batch of

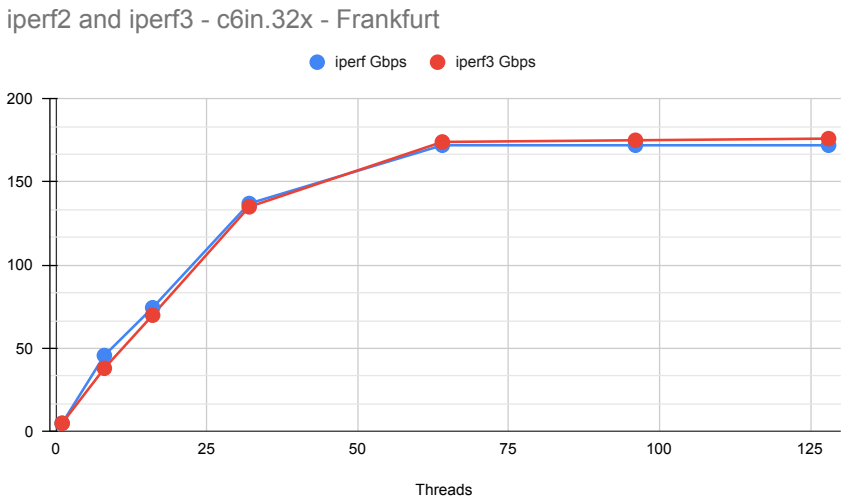

Figure 4.1.: Average bandwidth relative to the number of threads measured using iperf2 and iperf3 between two instances of `c6in.32xlarge` in Frankfurt. Made using Google Sheets.

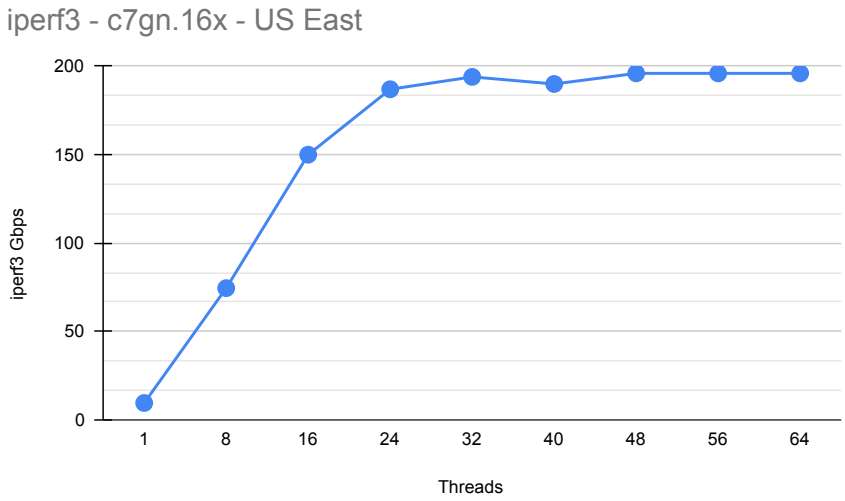

Figure 4.2.: Average bandwidth relative to the number of threads measured using iperf3 between two instances of `c7gn.16xlarge` in the US East region. Made using Google Sheets.

benchmarks, as throttling can impact the consistency and reliability of the performance measurements.

4.2. Hardware

The experiments labeled as local were conducted on a desktop machine equipped with an AMD Ryzen 9 7900X CPU, running Arch Linux with Linux kernel version 6.10.x.

For AWC EC2 experiments, the `c6in` instances were powered by x86-based CPUs, while the `c7gn` instances used ARM CPUs - AWS Graviton3E. Both instance types ran on Ubuntu Server 24.04 LTS, with Linux kernel version 6.8.0.

Both the desktop machine and the `c7gn` instances have a single NUMA node.

4.3. Protocol

We iterated over several protocol ideas, but ultimately settled on a very simple protocol to focus solely on the networking. Initially, the protocol required the client to send a 4-byte value to the server, which was meant to represent a page number being requested. The server would then reply with that value repeatedly copied into a page block of a specific size. Later, this was changed to the following: when the connection is established, the server continuously sends page blocks to the client for the specified duration of the test.

A page size of 4 KiB was chosen as a reasonable page size for most experiments, but other page sizes have also been tested.

4.4. Implementation

The `io_uring` implementation heavily utilized the tutorials and documentation available at unixism.net [14]. There are many variables involved in using `io_uring`. For our protocol, we mainly focus on page size, thread count, and ring size.

Alongside `io_uring` implementation, an efficient POSIX version has been implemented as well. This was done to have a baseline for `io_uring` to be compared to.

Let's explore the details of the implementation.

In both the server and client, `io_uring` is initialized using the `io_uring_queue_init()` function, which sets up a submission and completion queue. The queue size used in most experiments was 512, however, this is quite large for most applications and is not generally advised. Each thread gets a completely separate ring instance.

4.4.1. Memory management

A trivial approach would be to allocate memory using *malloc*, and to set it free once the message is sent. A step further would be pre-allocating a pool of buffers, and cycling through them, as the messages are being sent out. To further improve memory management, memory pinning to prevent swapping was introduced using the *mlock* call; this resulted in a 24% bandwidth increase in local tests.

The server and client both use registered buffers with `io_uring` using `io_uring_register_buffers()` call. This helps avoid copying data between kernel and user space during I/O operations. Registered buffers offer optimization by reducing the overhead of repeated memory mappings and un-mappings with every I/O operation; this is achieved by mapping it into the kernel once. Additionally, using registered buffers avoids page reference counts on each operation, which further reduces overhead.

The server and client handle communication by preparing SQEs (submission queue entries) to send and receive data using `io_uring` prepare send and receive calls respectively, with zero-copy and fixed where possible. Unfortunately, the zero-copy send consistently resulted in much lower throughput, and was not used in the later experiments. These requests are then submitted in batches using `io_uring_submit()` to reduce the frequency of system calls. On the client side, after sending a batch of requests, it waits for completion events using `io_uring_wait_cqe()`. Then, for each completion event, the completion queue entry (CQE) is retrieved, and based on whether or not it succeeds, it is either retried or the send/receive is completed.

4.4.2. Handling Multiple Connections per Thread

Both the server and client are designed to handle multiple connections within each thread, where each thread manages its connections independently. As was mentioned earlier in this chapter, AWS limits single flow speed to 5 Gbit/s in different availability zones, however, we can handle higher throughput per thread, if not for this limitation, hence, handling multiple connections simultaneously in each thread allows us to workaround this limitation.

The server adds connections to a connections list, where each threads has its own list of file descriptors, which are handled by worker threads. Each thread iterates over its assigned connections, submitting and handling I/O requests while checking for connection closures or errors using `io_uring` completion events. To ensure balanced processing of all connections, the requests are submitted in a round-robin manner, avoiding starvation of any single connection. Similarly, the client creates multiple connections in each thread, and I/O operations are managed concurrently.

4.4.3. Misc `io_uring` options

IORING_SETUP_SINGLE_ISSUER

The `IORING_SETUP_SINGLE_ISSUER` flag is an option intended to optimize performance when only a single thread issues I/O requests to the submission queue. In practice, this flag did not provide any measurable benefits and was omitted.

SQPOLL

The `SQPOLL` flag enables kernel-side polling of the submission queue. This will offload submission handling to a dedicated kernel thread. While this can reduce the overhead of user-to-kernel transitions, it was not used in our `io_ring` implementation. The reason is that `SQPOLL` introduces a dedicated CPU thread, which would consume resources constantly; even when there is no actual work to be processed.

IOPOLL

The `IOPOLL` flag enables kernel-side polling of the completion queue, potentially reducing latency in high-performance I/O operations. This option performs busy-waiting for I/O completions instead of relying on asynchronous interrupts, which can provide lower latency at the cost of higher CPU usage. However, `IOPOLL` is currently not supported for network I/O and requires compatible file systems and block devices. In the future, if new versions of `io_uring` extend `IOPOLL` support to network operations, it could offer performance benefits, particularly in low-latency networking scenarios, by enabling faster detection of completed I/O operations, and would remove the overhead of interrupts.

4.4.4. Hyperparameters

In Figure 4.3 and Figure 4.4 we can see plots for the initial grid search for optimal parameters for the `io_uring` client-server implementation running between two instances of `c6in.32xlarge`. Note that this experiment used the public link, which is limited to 100 Gbit/s; with iperf, one can reach 96.3 Gbit/s over this link. From this initial grid search, we can see that targeting a higher bit rate and a higher message throughput is not always aligned. This grid search found the ring size of 256, the page size of 4 KiB, and the thread number of 128 to be the most optimal for maximum bit-rate, getting 91.44 Gbit/s in this experiment. As for the optimizing for the highest messages rate, let us take a look at two cases in Table 4.1. It is evident that the bit rate is quite a bit lower than the maximum possible. The optimal number of threads is the same in this case. This implementation achieved 10 million messages per second, however, this requires a large ring size of 896 or 1024. Given this is using TCP, and we do not have a lot of control over when exactly the packets are sent out, the messages get buffered into packets close to the size of the MTU.

Page Size (bytes)	Ring Size	Message Rate (it/s)	Bit-Rate (Gbit/s)
16	896	10989110.61	5.63
32	1024	10261884.47	10.51

Table 4.1.: Two of the most optimal cases for messages rate for the `io_uring` grid search between two `c6in.32xlarge` instances.

To try mitigating the buffering, using the `TCP_NODELAY` flag was tested. This flag disables Nagle's buffering algorithm, which works by coalescing small outgoing packets. As a result, setting this flag should force the socket to send the data accumulated in the buffer without considering the packet size. However, based on observations conducted using Wireshark [27], this option is not reliable, and it appears that the Linux kernel can ignore it, making it a not very useful option. It was observed that there are still large packets present. The Don't Fragment (DF) IP flag was also tested and had similar results to the `TCP_NODELAY` flag. Additionally, in local tests with a small ring size, both of these options dramatically reduce the overall throughput. In Table 4.2 you can see a locally executed test using `io_uring` client-server with page size of 4096 KiB, and ring size 8. This is not something that was observed during tests on `c7gn.16xlarge` instances, where a substantially larger ring size was used.

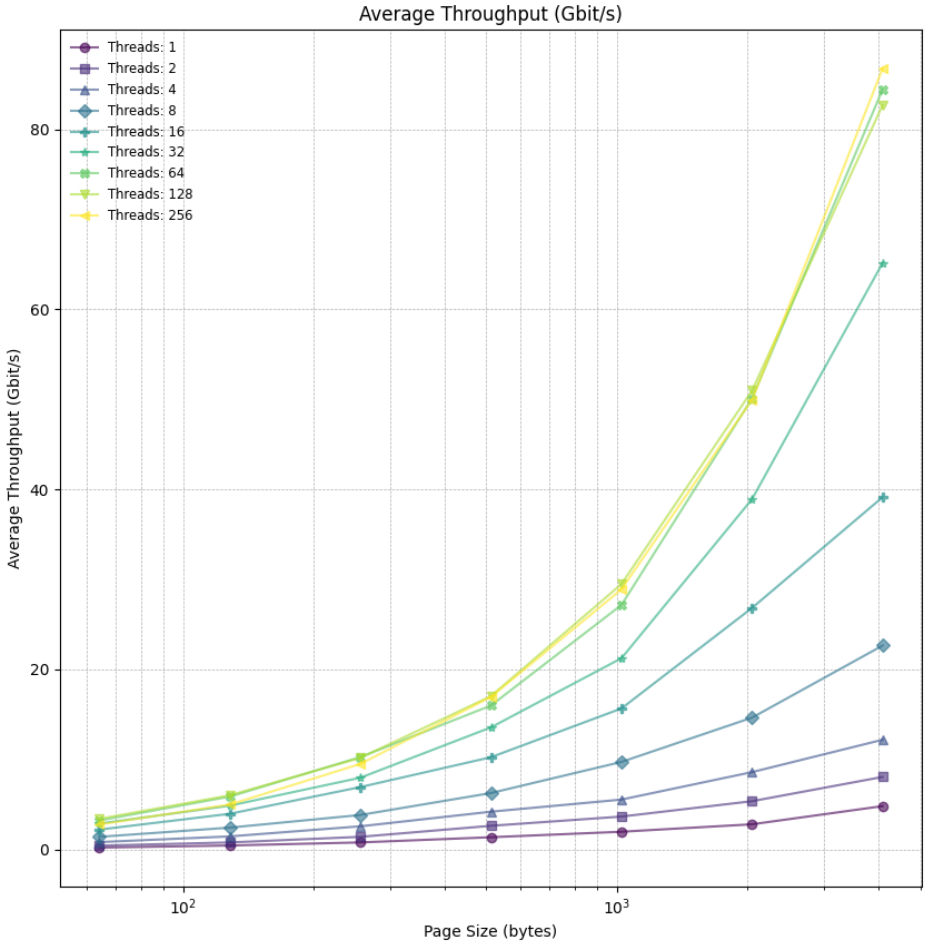

Figure 4.3.: Grid search plot for `io_uring` client-server between two instances of `c6in.32xlarge` over public network measuring the average Gbit/s relative to the ring size, page size, and thread count. The x-axis is logarithmic.

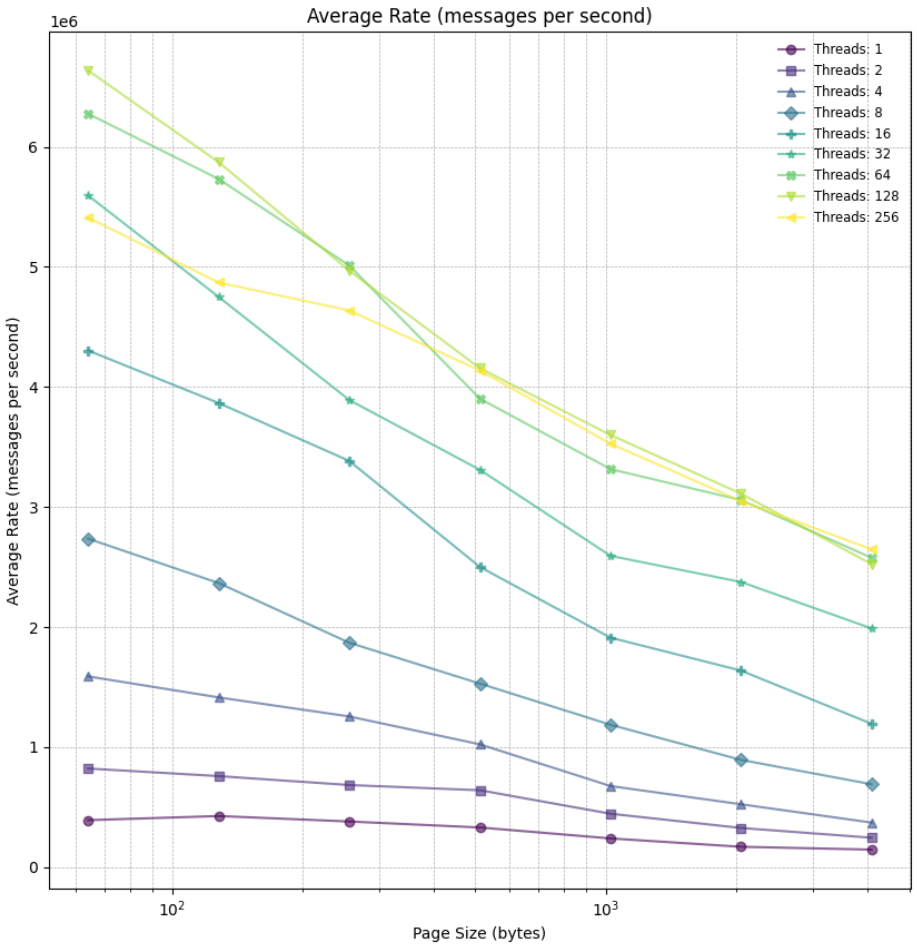

Figure 4.4.: Grid search plot for `io_uring` client-server between two instances of `c6in.32xlarge` over public network measuring the average messages per second relative to the ring size, page size, and thread count. The x-axis is logarithmic.

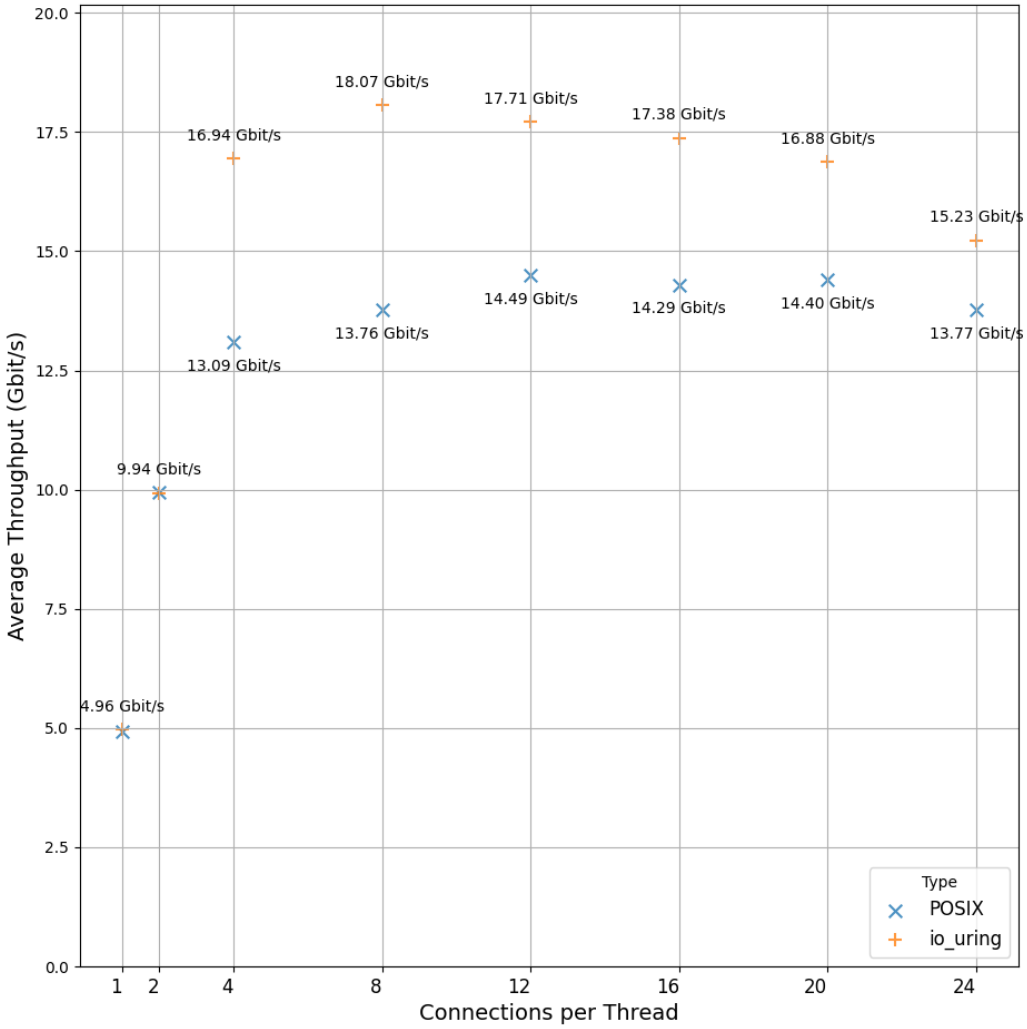

Figure 4.5.: Comparison of `io_uring` and POSIX implementations on a `c7gn.16xlarge` instance. This plot illustrates the average throughput (Gbit/s) achieved with a single thread while varying the number of connections per thread, and using 4 KiB messages.

Configuration	Bit-Rate (Gbit/s)
None	22.76
`TCP_NODELAY` + DF	5.85
`TCP_NODELAY`	6.01
DF	6.03

Table 4.2.: Comparison of TCP configurations and their impact on the bit-rate in the local environment with a single thread using 4KiB page size and ring size of 8.

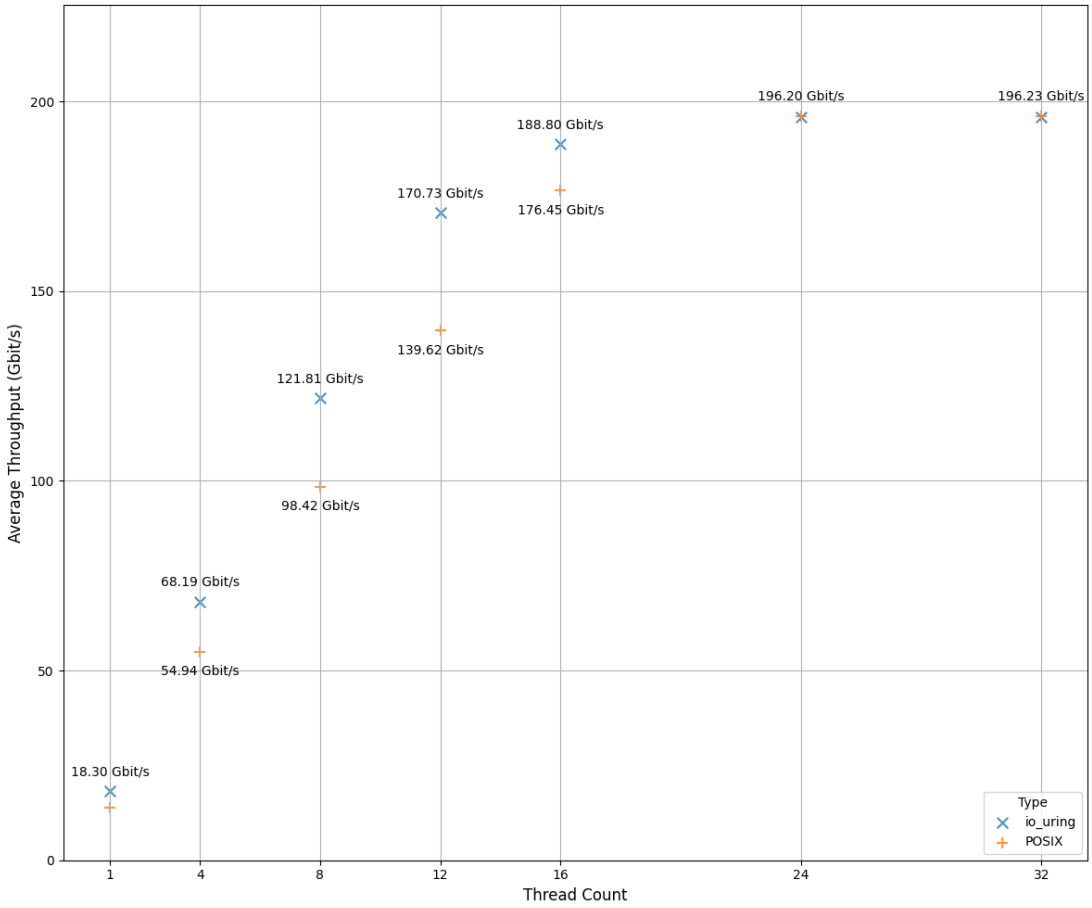

Figure 4.6.: Comparison of `io_uring` and POSIX implementations on a `c7gn.16xlarge` instance. This plot illustrates the average throughput (Gbit/s) achieved with a fixed number of 8 connections per thread while varying the number of threads and using 4 KiB messages.

4.4.5. Effect of Flags on Throughput

In this subsection, we analyze the impact of the four key flags we have implemented in our server-client code on the throughput, namely: `ALLOC_PIN`, `ENABLE_NAGLE`, `INCREASE_SOCKET_BUFFERS`, and `PIN_THREADS`.

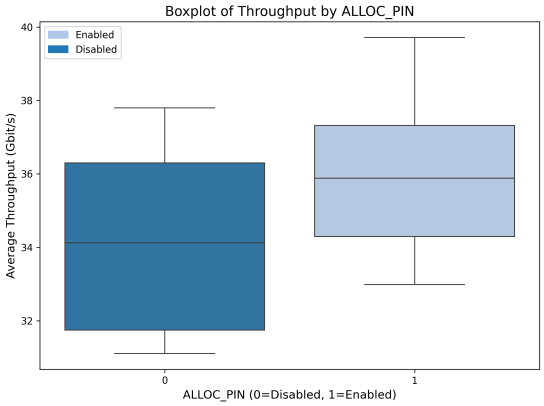

Figure 4.7.: Boxplot of average throughput (Gbit/s) categorized by whether memory was pinned to prevent swapping, illustrating the distribution of throughput in pinned and non-pinned states on a `c7gn.16xlarge` instance using 2 threads and 4 connections per thread, and using 4 KiB messages.

`ALLOC_PIN` **(Memory Pinning):** This flag determines whether memory buffers for socket operations are pinned using the `mlock` call; this means that the system will lock to the specified memory range, and also prevent the memory from getting paged. This helps reducing page faults and swapping overhead, ensuring faster references to that memory range. The boxplot in figure 4.7 shows that enabling `ALLOC_PIN` (set to 1) noticeably increases throughput. The median throughput when memory is pinned is higher, and the overall variance is reduced, indicating more stable performance. This performance improvement can also be observed in the heatmap figure 4.11, where configurations with `ALLOC_PIN` enabled consistently outperform those where it is disabled.

`ENABLE_NAGLE` **(Nagle's Algorithm):** Nagle's buffering algorithm, as described earlier, combines multiple packets into larger ones, before sending them in one transmission. From the boxplot in figure 4.8, we can observe that enabling it can reduce throughput but reduce the overall variance. From the heatmap figure 4.11, we can see that this generally holds true for most of the combinations of flags.

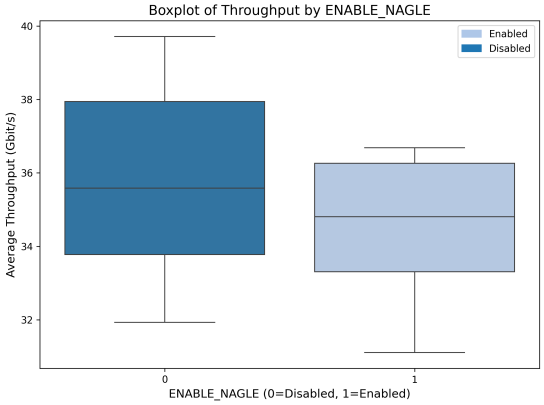

Figure 4.8.: Boxplot of average throughput (Gbit/s) categorized by whether Nagle's algorithm was enabled or disabled, showing the distribution of throughput on a `c7gn.16xlarge` instance using 2 threads and 4 connections per thread, and using 4 KiB messages.

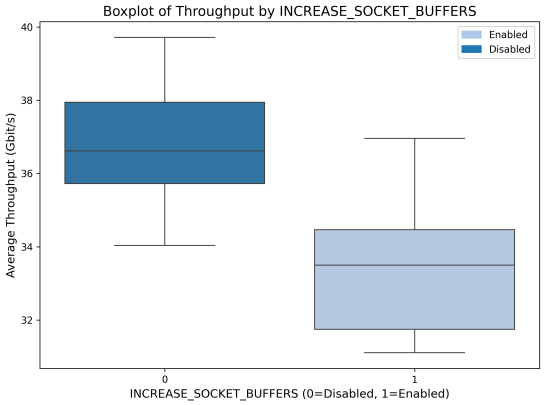

Figure 4.9.: Boxplot of average throughput (Gbit/s) categorized by whether the socket buffer sizes were increased, demonstrating the impact of socket buffer size adjustments on throughput performance on a `c7gn.16xlarge` instance using 2 threads and 4 connections per thread, and using 4 KiB messages.

`INCREASE_SOCKET_BUFFERS` **(Socket Buffers Increase):** This flag increases the socket receive and send buffers' sizes from 16 KiB to 416 KiB each. This is achieved by setting socket options `SO_SNDBUF` and `SO_RCVBUF` using the `setsockopt` call. From the boxplot in figure 4.9 we can see that the throughput is negatively affected by the increased buffers' size. This can also be observed in the heatmap figure 4.11.

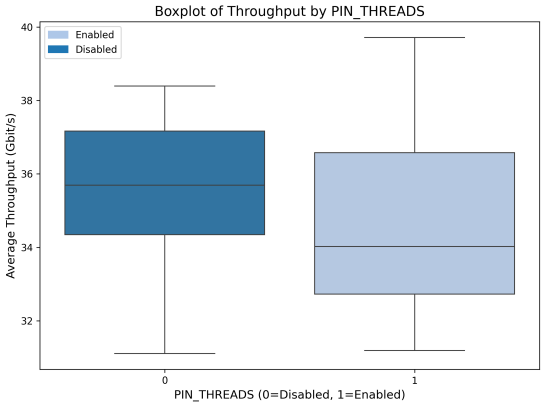

Figure 4.10.: Boxplot of average throughput (Gbit/s) categorized by whether threads were pinned to specific CPU cores, illustrating how thread pinning affects throughput distribution on a `c7gn.16xlarge` instance using 2 threads and 4 connections per thread, and using 4 KiB messages.

`PIN_THREADS` **(Thread Pinning):** With this flag, the thread affinity is being set to a specific CPU core, using the `pthread_setaffinity_np` call. The goal of pinning threads is to reduce context switching and to improve CPU cache locality. It is hard to make a definitive conclusion regarding this option given the spread of the results, however, the most performant run was possible only with this option enabled. This could be attributed to the fact that the used AWS EC2 instances have 64 cores, but we are using only 16 threads, and by pinning them, we are stopping the kernel from using all cores for compute.

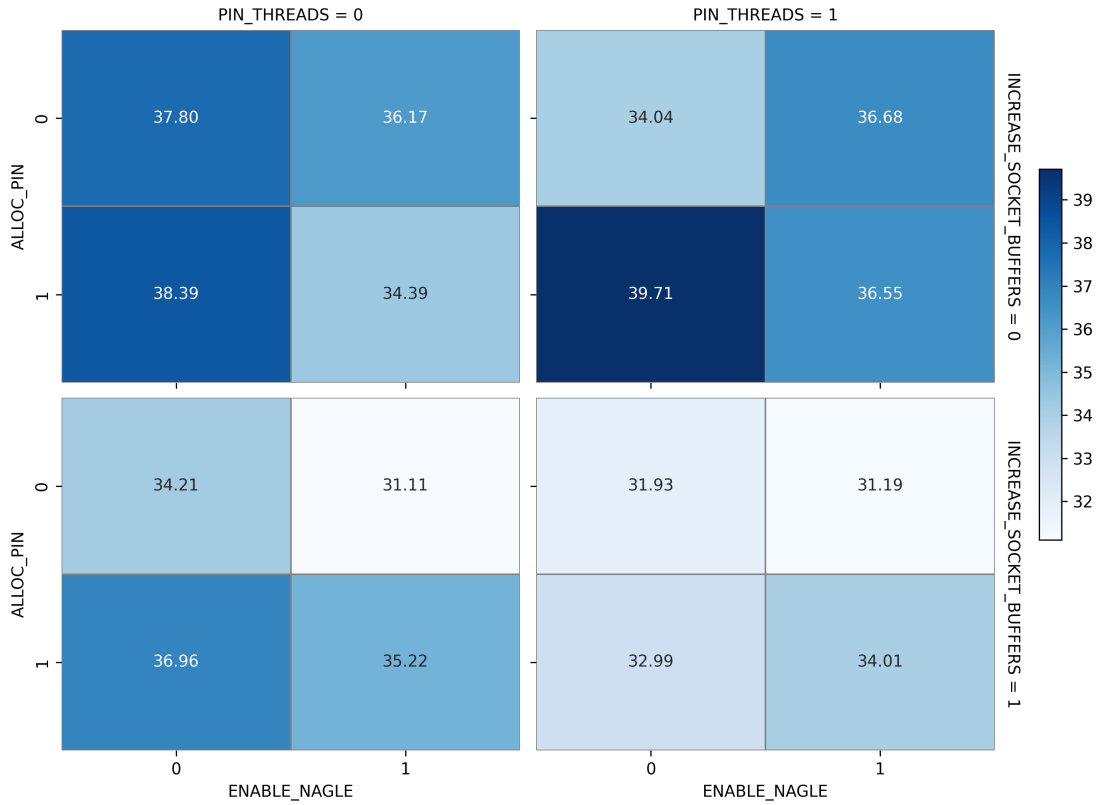

Figure 4.11.: Heatmap displaying the combined average throughput (Gbit/s) across various configurations of `ALLOC_PIN`, `ENABLE_NAGLE`, `INCREASE_SOCKET_BUFFERS`, and `PIN_THREADS` flags for the *io_uring* version executed on a `c7gn.16xlarge` instance using 2 threads and 4 connections per thread, and using 4 KiB messages. Each cell represents the mean throughput for a specific combination of flag settings, using a consistent blue color gradient for intensity.

4.4.6. Stability of AWS EC2 networking

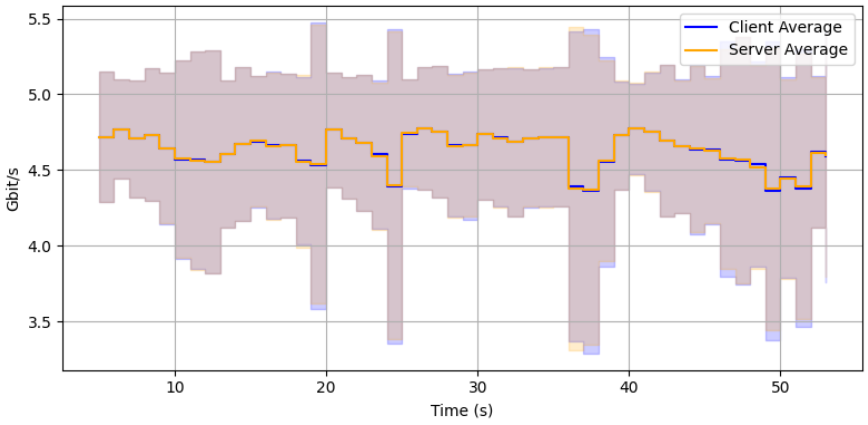

Figure 4.12.: Average throughput (Gbit/s) over time for both client and server combined. The benchmark was configured with 16 threads, each handling 1 connection (total of 16 connections), using the `io_uring` implementation on a `c7gn.16xlarge` instance.

AWS EC2 networking in our experience with the *c7gn* series instances has shown a notable degree of variability, especially while handling multiple TCP streams. As we know, each TCP stream is capped at 10 Gbit/s on the same zone instances, and at 5 Gbit/s otherwise.

In Figure 4.12, the overall throughput fluctuates significantly, with drops from around 5 Gbit/s to as low as 3.5 Gbit/s. This variance points to instability in individual connection performance over time, likely due to the way AWS handles network resources.

Further evidence of this instability can be seen in Figure 4.13, where each individual connection's performance is tracked. Here, we can observe that many of the connections exhibit unstable throughput, while a few manage to approach and maintain the 5 Gbit/s limit.

When we examine the total combined throughput in Figure 4.14, we can observe the instability at a cumulative level. The total throughput oscillates between 70 and 76 Gbit/s, again indicating that the network bandwidth is not consistent.

Figures 4.15, 4.16, and 4.17 show a more stable throughput. Here each thread handles multiple connections; these benchmarks were configured with 4 threads, each managing 4 connections (for a total of 16 connections), on a `c7gn.16xlarge` instance. In this case, the average throughput over time is more stable, and the per-stream throughput shows less fluctuation. This suggests that assigning multiple connections per thread helps

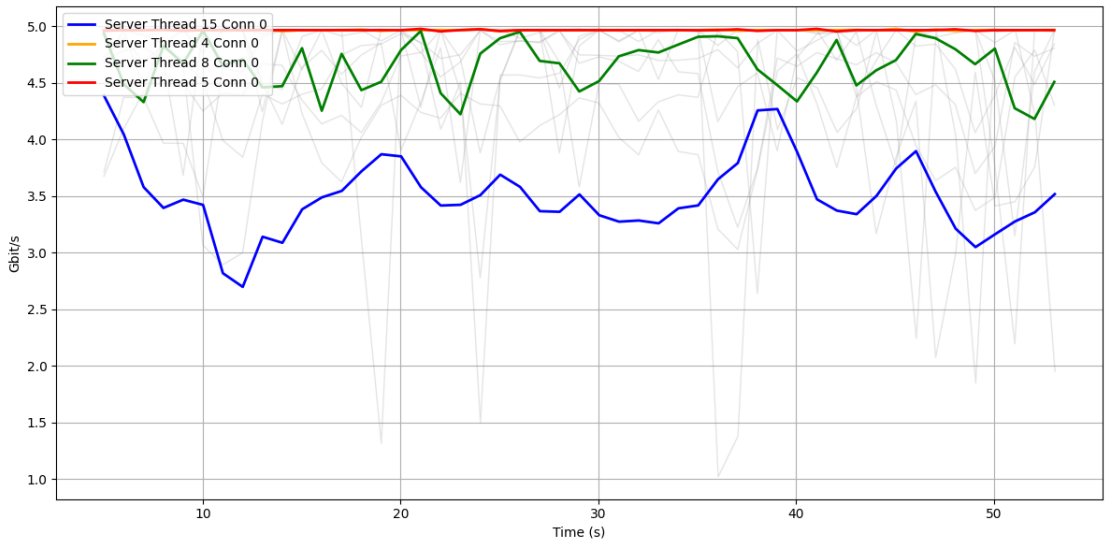

Figure 4.13.: Per-stream (per-thread and per-connection) throughput (Gbit/s) over time for both client and server combined. The benchmark was configured with 16 threads, each handling 1 connection (total of 16 connections), using the `io_uring` implementation on a `c7gn.16xlarge` instance.

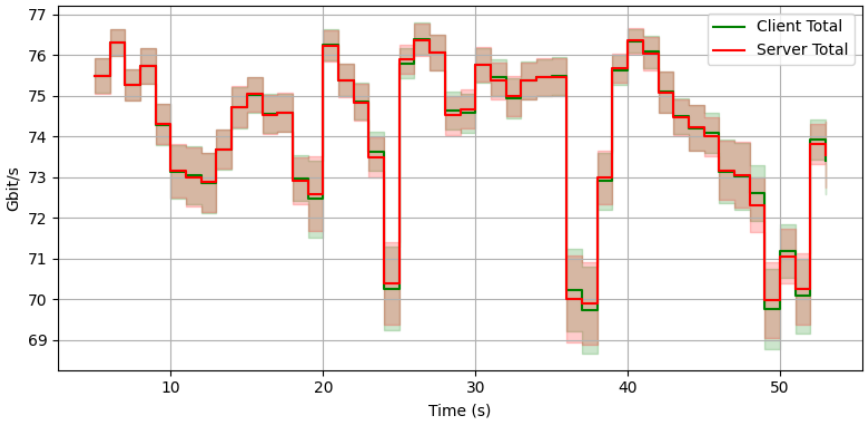

Figure 4.14.: Total throughput (Gbit/s) over time for both client and server combined. The benchmark was configured with 16 threads, each handling 1 connection (total of 16 connections), using the `io_uring` implementation on a `c7gn.16xlarge` instance.

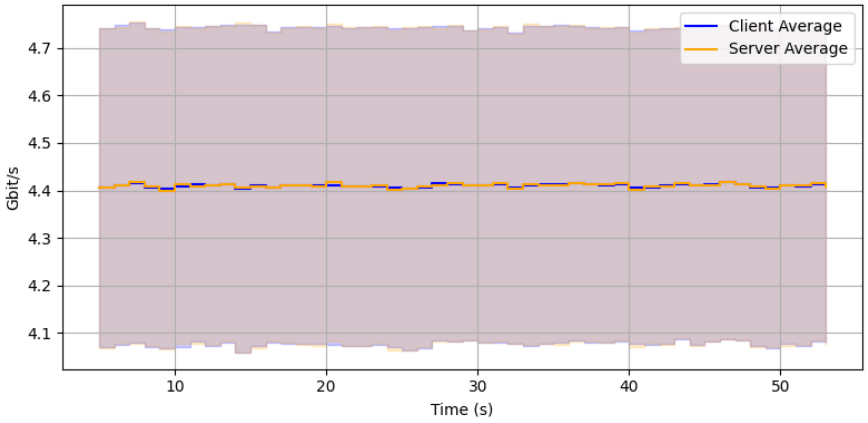

Figure 4.15.: Average throughput (Gbit/s) over time for both client and server combined. The benchmark was configured with 4 threads, each handling 4 connections (total of 16 connections), using the `io_uring` implementation on a `c7gn.16xlarge` instance.

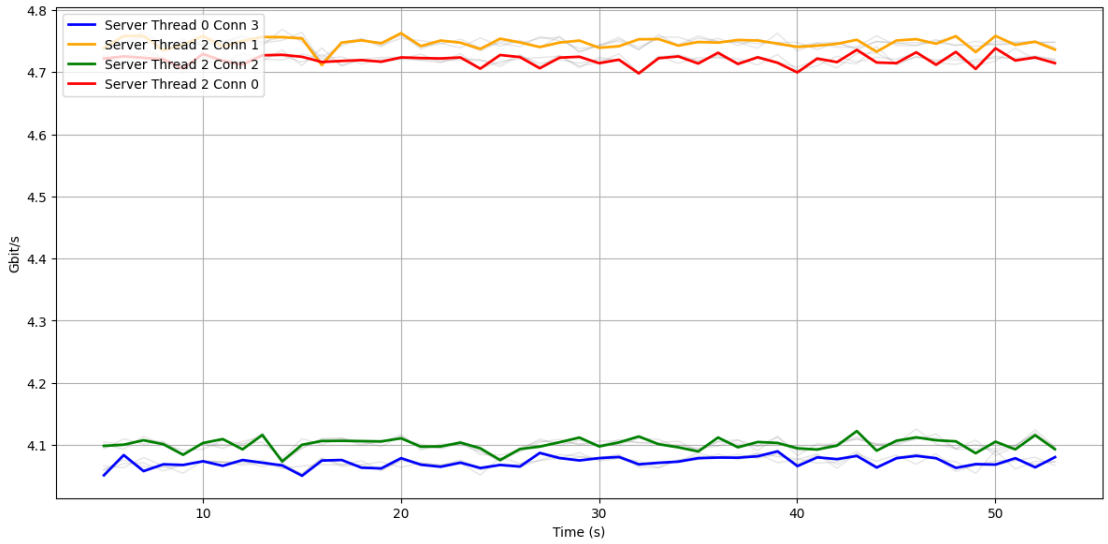

Figure 4.16.: Per-stream (per-thread and per-connection) throughput (Gbit/s) over time for both client and server combined. The benchmark was configured with 4 threads, each handling 4 connections (total of 16 connections), using the `io_uring` implementation on a `c7gn.16xlarge` instance.

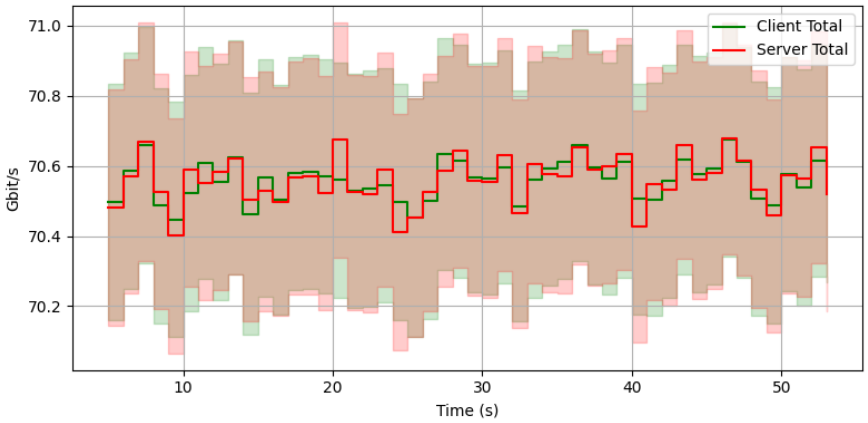

Figure 4.17.: Total throughput (Gbit/s) over time for both client and server combined. The benchmark was configured with 4 threads, each handling 4 connections (total of 16 connections), using the `io_uring` implementation on a `c7gn.16xlarge` instance.

mitigate the variance observed when each thread handles only one connection, allowing for more consistent bandwidth utilization. In figure 4.16 we can observe how each connection settles in a narrow range, with two groups emerging, one at about 4.75 Gbit/s and another at about 4.05 Gbit/s.

Let us consider a larger case, with 16 threads and 4 connections per thread, as depicted in Figures 4.18 and 4.19 for the POSIX implementation, and in Figures 4.20 and 4.21 for the `io_uring` implementation.

In these figures, we can observe a clear pattern where individual connections appear to establish a "stability zone" — a range of throughput where they settle for a certain period of time. These zones are not static, however. Over the course of the 60-second benchmark, we see frequent transitions where connections jump between different throughput ranges. For instance, in Figure 4.20, some connections that start in a higher throughput zone (around 3.5 Gbit/s) drop suddenly to lower zones (around 2.5 Gbit/s or even 1.5 Gbit/s) and remain there for several seconds before potentially returning to higher zones. This behavior suggests that individual connections are dynamically adjusting their throughput, likely influenced by factors such as network congestion, resource allocation, or TCP dynamics. Additionally, in figure 4.20, we can observe how the blue stream almost completely dies out.

These transitions between stability zones are not limited to reductions in throughput; we also observe connections jumping upwards. In Figure 4.18, for example, some connections that initially linger around 1.5 Gbit/s later increase their throughput

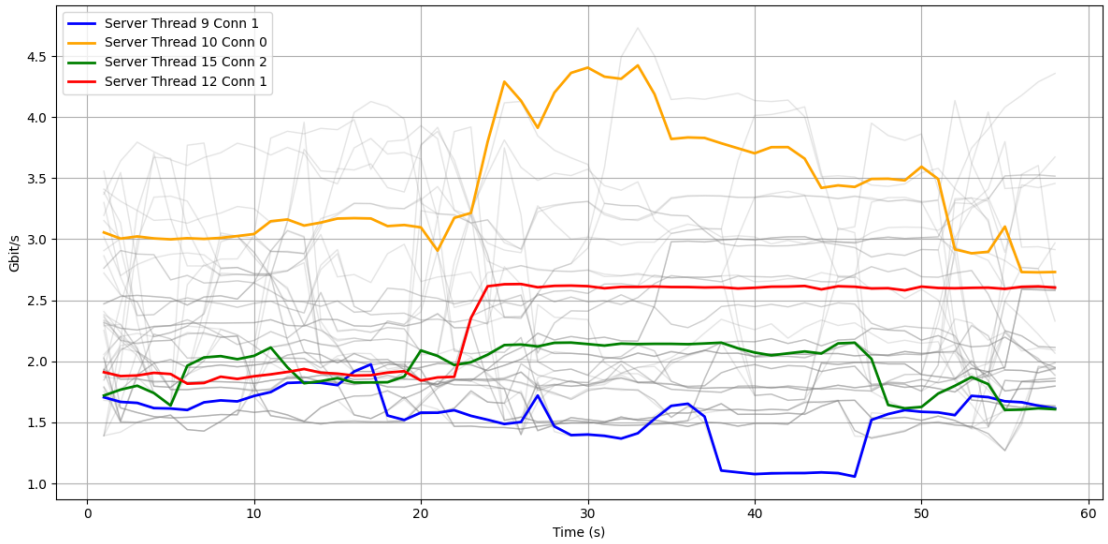

Figure 4.18.: Per-stream (per-thread and per-connection) throughput (Gbit/s) over time for both client and server combined. This figure represents the POSIX implementation.

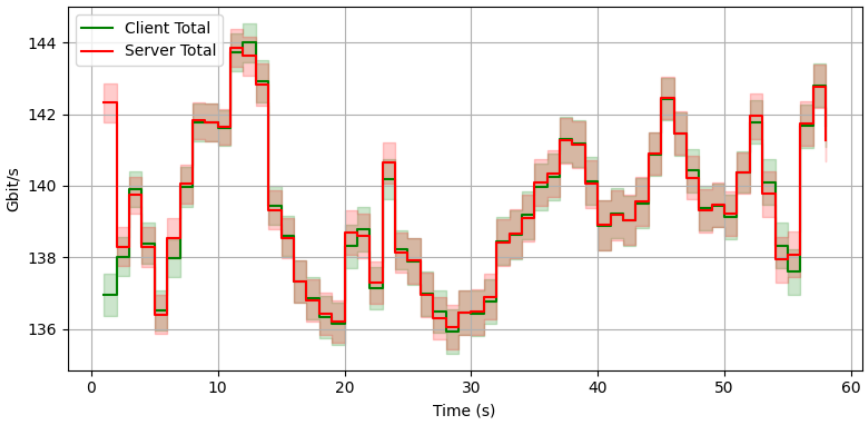

Figure 4.19.: Total throughput (Gbit/s) over time for both client and server combined. This figure represents the POSIX implementation.

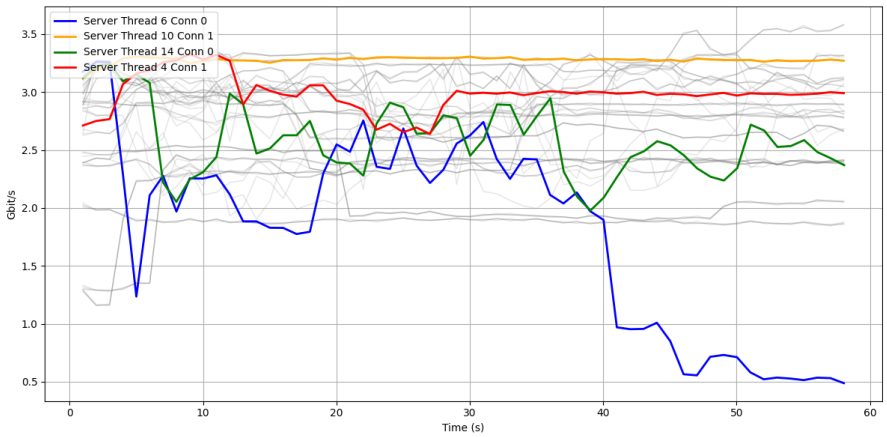

Figure 4.20.: Per-stream (per-thread and per-connection) throughput (Gbit/s) over time for both client and server combined. This figure represents the `io_uring` implementation.

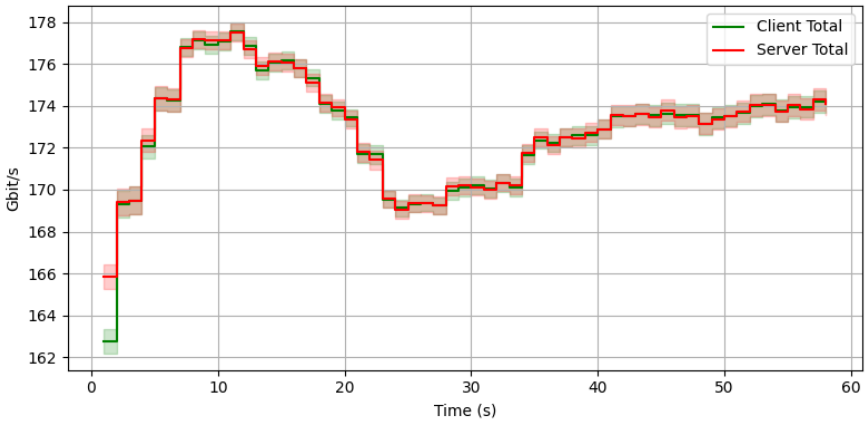

Figure 4.21.: Total throughput (Gbit/s) over time for both client and server combined. This figure represents the `io_uring` implementation.

to approach 3 Gbit/s. This kind of oscillation highlights the unpredictable nature of network resource management in multi-threaded, multi-connection environments, where connections may compete for bandwidth, and resource scheduling may cause sudden shifts in available capacity.

Moreover, when considering the total combined throughput in Figures 4.19 and 4.21, we can still see the effects of these individual connection variances. Although the total throughput appears more stable overall, there are still noticeable dips and peaks, with the bandwidth fluctuating between 170 and 176 Gbit/s for the `io_uring` implementation, and between 164 and 174 Gbit/s for the POSIX implementation. This range of oscillation suggests that while the system manages to maintain high aggregate throughput, individual connections continue to experience fluctuations, contributing to the overall instability seen at the connection level.

This behavior of fluctuating between throughput "zones" could be attributed to the way TCP adjusts its window size in response to varying conditions or how AWS EC2 instances dynamically allocate network resources. It is possible that as congestion control algorithms on each connection react to packet loss or delays, they move between these zones of stability. Furthermore, the way AWS virtualizes its networking infrastructure might also introduce variability as bandwidth is allocated across different machines or across multiple regions and availability zones.

In conclusion, the case of 16 threads with 4 connections per thread offers insight into the complexity of network behavior on AWS EC2. The frequent transitions between bandwidth stability zones highlight the challenge of maintaining consistent throughput, even in high-performance instances. While the total throughput remains relatively high, the individual connections exhibit significant instability, jumping between different throughput ranges throughout the duration of the benchmark.

4.4.7. perf analysis

To further understand the performance of the `io_uring` and `POSIX` implementations, we conducted experiments measuring various system metrics using the `perf` tool. The results are summarized in Table 4.3.

For this experiment, `perf` was configured to save results for each second, matching our stats logging frequency, and the $-a$ parameter was used to include kernel side compute. Here is an example of the `perf` commands used:

```
THREAD_COUNT=16 CONNECTIONS_PER_THREAD=8 sudo -E perf stat -a -o \
    perf_server.log -I 1000 ./server
THREAD_COUNT=16 CONNECTIONS_PER_THREAD=8 sudo -E perf stat -a -o \
    perf_client.log -I 1000 ./client
```

Table 4.3.: Performance metrics comparison between `io_uring` and POSIX implementations. The experiment used instances of `c7gn.16xlarge` with 16 threads and 4 connections per thread, single second slice, and 4 KiB messages.

Metric	**io_uring**		**POSIX**	
	Server	**Client**	**Server**	**Client**
Throughput (Gbit/s)	108.49		88.94	
Messages (Thousand)	52.5		43	
Context Switches	8,023	2,185	28,261	8,155
CPU Migrations	74	65	87	70
CPU Cycles (Billion)	24.61	31.01	28.82	33.14
Instructions (Billion)	34.65	33.44	45.24	45.35
Instructions per Cycle	1.41	1.08	1.57	1.37
Branches (Billion)	5.26	5.12	7.11	7.03
Branch Misses (Million)	24.32	27.18	47.47	52.27
Branch Miss Rate (%)	0.46	0.53	0.67	0.74
Bad Speculation (%)	1.3	1.2	2.2	2.0

We can see from the table 4.1 that the `io_uring` implementation outperforms the `POSIX` implementation. In this test case, the `io_uring` implementation achieves a throughput of 108.49 Gbit/s and processes 52,500 messages per second, which is approximately 22% higher than the `POSIX` server's throughput of 88.94 Gbit/s and 43,000 messages per second.

It is also evident that the `io_uring` implementation endures fewer context switches compared to the `POSIX` version. The `io_uring` server and client have 8,023 and 2,185 context switches respectively, whereas the `POSIX` server and client have 28,261 and 8,155 context switches. This reduction in context switches contributes to the improved performance because the context switches create overhead.

In terms of CPU cycles and instructions executed, the `io_uring` server uses fewer CPU cycles (24.61 billion) and executes fewer instructions (34.65 billion) compared to the POSIX server (28.82 billion cycles and 45.24 billion instructions).

Additionally, the branch prediction metrics also favor the `io_uring` implementation. Both the `io_uring` server and client have a lower branch miss rate compared to the POSIX versions, sitting at 0.46% and 0.53% for `io_uring`, while it is 0.67% and 0.74% respectively for `POSIX`.

To gain deeper insights into the performance characteristics of our implementations, we generated flame graphs using the `perf` tool and the FlameGraph visualization scripts [10]. With flame graphs, we can observe the stack traces graphically, which

allows us to identify where our program spends most of its time. Figures B.1 and B.2 show the flame graphs for the client and server implementations, respectively.

The following is an examople of the command used to profile the server (a similar command was used for the client):

```
./build/server &
SERVER_PID=$!
sleep 30 # We want to record one second in the middle of the run
perf record -F 99 -o perf_server.data -g -p $SERVER_PID -- sleep 1 &
PERF_SERVER_PID=$!
wait $PERF_SERVER_PID
perf script -i perf_server.data > perf_server.script
```

Using this approach, we can observe one second of execution of our programs in the middle of the run, where we avoid the noise of the setup and destruction of the program.

From the generated flame graphs, we observed that `io_uring_submit` takes up the largest portion of the execution time; however, in the server flame graph, we can also see that about 20 per cent of the time is taken up by the `io_uring_wait_cqes` call. You may find the full charts in the appendix - Figure B.1 and Figure B.2.

4.5. tokio uring

Up to this point, all implementations used C/C++ code. In this section, we explore the use of `io_uring` in Rust by leveraging the asynchronous runtime provided by the Tokio framework, specifically tokio-uring, which offers a more abstracted interface to `io_uring` while maintaining the benefits of asynchronous, high-performance I/O operations.

The tokio-uring library simplifies working with `io_uring` in Rust by abstracting away much of the complexity. While this allows using `io_uring` without directly handling lower-level I/O operations, this abstraction comes at a potential performance cost, since it allows for less fine-grained control compared to manually handling `io_uring` operations in C/C++ via liburing [3].

The server is made to handle multiple clients at the same time. For each new client, it spawns an separate task, which then sends 4 KiB messages to the client for the duration of the benchmark. Additionally, preallocated `io_uring` fixed buffers [15] are used to optimize memory management in the Linux kernel.

The client follows the same protocol as previous experiments, where it sends a 4-byte message representing a page number, and in return receives a 4 KiB reply with the same message repeated over and over in the reply.

In a local test over the loopback network interface, the implementation reached an average throughput of 26 Gbit/s, which is lower than the maximum achievable bandwidth we can observe in the C++ `io_uring` implementation, which can reach 75 Gbit/s in local tests over loopback. This could be attributed to the abstraction overhead of the library. Unlike in C++ `io_uring` version, we can't batch multiple write requests, or add multiple read requests in advance, knowing that the client is expected to send a large number of requests.

4.6. DPDK

In this section, we talk about the DPDK implementation of the server and the client. Since DPDK requires supported NICs, all development and testing was done on AWS EC2 instances; and ultimately, due to budget and time constraints, the DPDK version of the server and the client was not fully fleshed out, reaching only about 8 Gbit/s throughput on `c7gn.16xlarge` instances.

Manual ARP Handling in DPDK: Unlike in a typical network setup using the kernel stack, where the kernel handles Address Resolution Protocol (ARP) when using DPDK, it must be handled manually. This is due to the fact that DPDK is bypassing the kernel,

meaning there's no automatic mechanism for IP-to-MAC resolving, which is required for packet forwarding at the Ethernet layer. As such, in order to address this, a custom code was implemented to manage ARP requests and replies.

Spoofing UDP as TCP for AWS Throughput: As was pointed out earlier, AWS EC2 throttles UDP traffic. Using `iperf3` in UDP mode, we observed an approximate limit of 100 Mbit/s for UDP traffic. Given this limitation, the TCP protocol ID was set in the IP header in order to spoof it to resemble TCP packets, which do not get throttled.

5. Discussion

In this thesis, we analysed `io_uring`'s performance in network-intensive operations, specifically within the context of transactional cloud database systems. Our experimental results revealed that `io_uring` implementations surpassed a traditional `POSIX` approach across various key performance indicators. Particularly noteworthy was the observation of up to a 22% boost in throughput and a decrease in context switches when utilizing `io_uring`. These findings strongly indicate that `io_uring` could provide significant advantages for cloud-based database systems, especially for components such as page servers that heavily depend on efficient network communication.

The better performance of `io_uring` can be attributed to its several features. The ability to batch submissions reduces system call overhead, a critical factor in high-performance networking scenarios. The reduction in context switches, as evidenced by our performance metrics, results in higher CPU efficiency. Furthermore, `io_uring`'s design allows for better handling of concurrent connections, which is relevant for cloud database systems that must manage multiple simultaneous client requests and the limitations of the AWS EC2 networking.

However, our thesis also highlighted challenges in conducting precise performance evaluations in cloud environments. Throughout our experiments on AWS EC2 instances, we observed substantial variability in network performance, especially when using TCP streams. This variability manifested as fluctuations in throughput and inconsistent performance across different instance types and regions.

The noise in the AWS EC2 environment poses a significant challenge for accurate benchmarking. While cloud platforms offer convenience and scalability, they introduce numerous uncontrollable variables that can impact network performance. These factors include resource contention with other virtual machines, variations in underlying hardware, and the opaque nature of the virtualized network infrastructure.

In light of these challenges, we propose that future work in this area should consider conducting experiments in a more controlled, local environment. A setup consisting of two physical machines each connected via dedicated NIC would provide a more stable and reproducible environment for precise performance measurements. This approach would allow for better isolation of the factors directly related to `io_uring`'s performance, without the confounding variables introduced by cloud virtualization. Additionally, such a setup can potentially save money in the long run by eliminating

ongoing cloud infrastructure costs associated with extended experimentation periods.

While our primary focus was on `io_uring`, we also briefly explored other approaches such as DPDK and higher-level abstractions like `tokio-uring`. Each of these approaches presents its own set of tradeoffs. DPDK offers the potential for even higher performance through kernel bypass but comes with increased complexity and hardware requirements. On the other hand, `tokio-uring` provides a more developer-friendly interface but at the cost of some performance overhead.

Despite the challenges in benchmarking, our results suggest that `io_uring` has a great potential for improving the performance of networking for cloud-based transactional database systems. The observed improvements in throughput and CPU efficiency could translate to better resource utilization and increased capacity for page servers and other network-intensive components. However, the variability we observed in the cloud environment underscores the need for robust and adaptive networking strategies that can handle fluctuations in available bandwidth and latency.

Moving forward, we see several directions for future research. Conducting more controlled experiments in a local, non-virtualized environment would provide valuable baseline data for comparison with cloud-based results. Further investigation into the integration of `io_uring` with specific cloud database architectures could yield insights into real-world performance gains.

The insights gained from this thesis extend beyond the specific context of transactional cloud databases. The challenges we encountered in cloud-based performance testing are relevant to a wide range of network-intensive applications. Our findings highlight the complexity of performance optimization in cloud environments and the need for careful consideration of environmental factors in benchmarking and system design.

The findings of this thesis have reinforced the importance of environmental considerations in networking benchmarks. While cloud platforms offer many advantages, they also introduce complexities that can obscure the true performance characteristics of systems under test. Nevertheless, our results demonstrate that low-level performance tuning, such as the adoption of `io_uring`, can yield significant benefits even in cloud environments.

It is important to acknowledge the limitations of this thesis. Our focus on specific AWS EC2 instance types may limit the generalizability of our findings to other cloud providers or instance types. Time and budget constraints prevented us from fully exploring some promising avenues, such as a more comprehensive implementation using DPDK. Most significantly, the variability inherent in the cloud environment makes it challenging to draw definitive conclusions about absolute performance metrics.

6. Conclusion

In this thesis, we explored efficient networking techniques relevant to transactional cloud database systems. The particular focus was on `io_uring`, a modern Linux API which can be used for high-performance network programming. The research was motivated by the increasing importance of network stack efficiency and performance in cloud-based database systems, especially in light of the high-speed networking capabilities offered by platforms like AWS EC2.

We began by examining the challenges of traditional networking approaches used in Linux. Our investigation of `io_uring` revealed key features like shared ring buffers, batched submissions, and zero-copy design that can reduce system call overhead and enhance performance.

Our experiments on AWS EC2 instances provided insights into performance characteristics of `io_uring` compared to traditional `POSIX` networking approaches. The results demonstrated that `io_uring` consistently outperformed `POSIX` implementations in terms of throughput, with lower CPU utilization and fewer context switches. This performance advantage was particularly evident in scenarios with multiple connections per thread, where we could see `io_uring`'s efficient handling of concurrent I/O operations, which just fit in with the batch submission design.

However, we also faced some limitations and challenges with the variability in network performance on AWS EC2 instances with TCP streams. This highlights the complexity of optimizing network performance in cloud environments. We observed that individual connections often displayed unstable throughput with fluctuating over time. This shows the importance of designing a robust and adaptive networking strategy in cloud database systems, which can account for these fluctuations.

While our exploration of DPDK was limited due to time and budget constraints, it offered a glimpse into the potential of kernel bypass techniques for further performance improvements. Future work could go deeper into DPDK and other kernel bypass methods to investiage their performance characteristics for usecases relevant to transactional cloud database systems.

We also examined Rust-based implementations using the `tokio-uring` library, which offers an abstraction over `io_uring` within an asynchronous runtime. It was found that while it offers ease of use, the abstraction comes at the cost of performance overhead compared to implementations interfacing directly with `io_uring`. Ultimately, in a

case with more realistic traffic, or with clever batching on the application level, the performance overhead could be lower; however, this would require more work on the application side to do this right.

In conclusion, this thesis demonstrates that `io_uring` represents a significant advancement in Linux's asynchronous I/O capabilities. This offers concrete benefits for network-intensive applications like cloud database systems. Its ability to reduce system call overhead, improve CPU efficiency, and handle concurrent connections effectively makes it a promising technology for optimizing network performance in cloud environments.

Future research could explore more deeply the integration of `io_uring` with specific cloud database architectures, investigate its performance in real-world database workloads, and examine its interplay with other emerging networking technologies. Additionally, it would interesting to see a DPDK implementation of the server and the client, as well as a comparison of performance between DPDK and `io_uring`. As cloud providers continue to enhance their networking capabilities, ongoing study will be crucial to fully leverage these advancements for the benefit of transactional cloud database systems.

A. Appendix: Code and Results

You may find the code and the results from this thesis in the following repository on GitHub - `https://github.com/sssemil/thesis_code`.

B. Appendix: Flame graphs

The following chapter of the appendix contains the full flame graphs for the `io_uring` server and client.

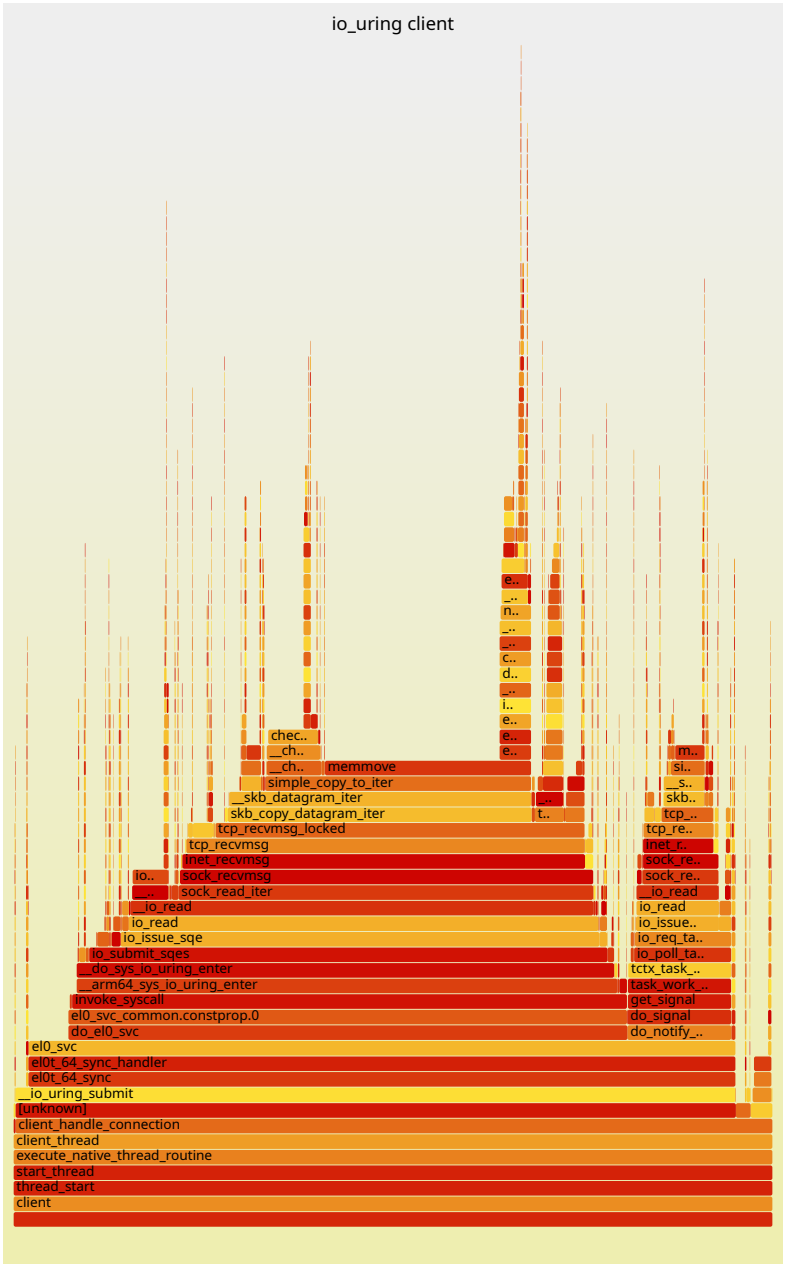

Figure B.1.: Flame graph of the `io_uring` client implementation. The x-axis represents the percentage of total samples, while the y-axis shows the stack depth. Each rectangle represents a function in the stack, with wider rectangles indicating functions where more time is spent.

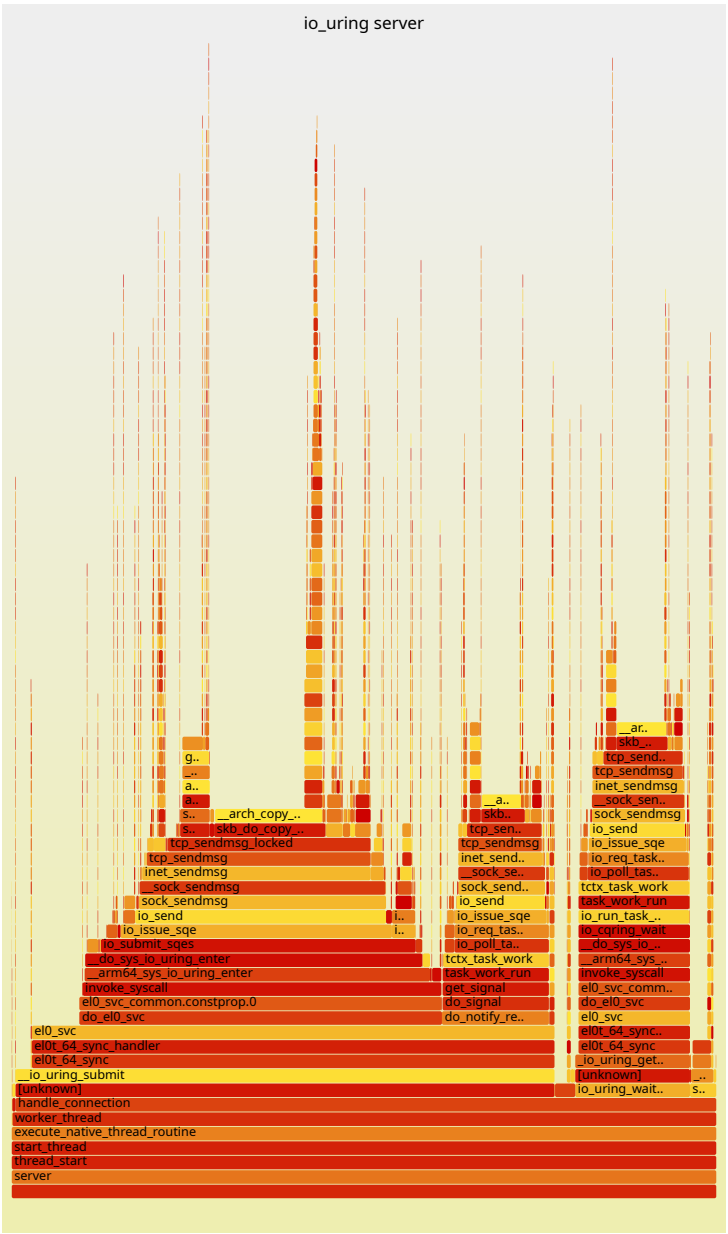

Figure B.2.: Flame graph of the `io_uring` server implementation. The x-axis represents the percentage of total samples, while the y-axis shows the stack depth. Each rectangle represents a function in the stack, with wider rectangles indicating functions where more time is spent.

Abbreviations

DPDK Data Plane Development Kit 1

AWS Amazon Web Services iii

EC2 Elastic Compute Cloud iii

DBaaS Database as a Service 1

ACID Atomicity, Consistency, Isolation, Durability 4

OLTP Online Transaction Processing 4

ARP Address Resolution Protocol 32

TCP Transmission Control Protocol 47

NIC Network Interface Card 3

CPU Central Processing Unit 45

NUMA Non-Uniform Memory Access 3

TLS Transport Layer Security 3

NFV Network Functions Virtualization 7

SDN Software-Defined Networking 7

SQ Submission Queue 5

List of Figures

List of Tables

Bibliography

[1] *Amazon EC2 instance network bandwidth*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-instance-network-bandwidth.html, Accessed: 2024-07-11. 2024.

[2] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, et al. "Socrates: The new sql server in the cloud." In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 1743–1756.

[3] axboe. *liburing - Library providing helpers for the Linux kernel io_uring support*. https://github.com/axboe/liburing, Accessed: 2024-09-01. 2024.

[4] J. Axboe. *Efficient IO with io_uring*. https://kernel.dk/io_uring.pdf, Accessed: 2024-08-31. 2024.

[5] S. Chemouil. *Oxidizing Kraken: Improving Kraken Infrastructure Using Rust*. https://blog.kraken.com/product/engineering/oxidizing-kraken-improving-kraken-infrastructure-using-rust, Accessed: 2024-09-01. 2021.

[6] Cloudflare. *Pingora - A library for building fast, reliable and evolvable network services.* https://github.com/cloudflare/pingora, Accessed: 2024-09-01. 2024.

[7] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, et al. "The snowflake elastic data warehouse." In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 215–226.

[8] *F-Stack: A User-Space Network Development Kit with High Performance*. https://github.com/F-Stack/f-stack. Accessed: 2024-10-05. 2024.

[9] D. Gallatin. "Serving Netflix Video at 400Gb/s on FreeBSD." In: *EuroBSDCon 2021*. Accessed: 2024-10-05. 2021.

[10] B. Gregg. *FlameGraph: Stack trace visualizer*. https://github.com/brendangregg/FlameGraph. Accessed: 2024-10-05. 2024.

[11] T. W. House. *BACK TO THE BUILDING BLOCKS: A PATH TOWARD SECURE AND MEASURABLE SOFTWARE.* https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf, Accessed: 2024-09-01. 2024.

[12] J. Howarth. *Why Discord is switching from Go to Rust.* https://discord.com/blog/why-discord-is-switching-from-go-to-rust, Accessed: 2024-09-01. 2020.

[13] D. Hunter. *Why you should use io_uring for network I/O*. https://developers.redhat.com/articles/2023/04/12/why-you-should-use-iouring-network-io, Accessed: 2024-09-01. 2023.

[14] S. Hussain. *Unixism - What is io_uring?* https://unixism.net/loti/what_is_io_uring.html, Accessed: 2024-09-01. 2024.

[15] S. Hussain. *Unixism - What is io_uring?* https://unixism.net/loti/tutorial/fixed_buffers.html, Accessed: 2024-09-01. 2024.

[16] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. "mTCP: a highly scalable user-level TCP stack for multicore systems." In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 489–502.

[17] M. Majkowski. *Kernel Bypass*. https://blog.cloudflare.com/kernel-bypass/. Accessed: 2024-10-05. 2015.

[18] *mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems*. https://github.com/mtcp-stack/mtcp. Accessed: 2024-10-05. 2024.

[19] Red Hat, Inc. *Speculative Execution Exploit Performance Impacts - Describing the performance impacts to security patches for CVE-2017-5754, CVE-2017-5753, and CVE-2017-5715*. https://access.redhat.com/articles/3307751. Updated March 7, 2019 at 12:20 PM - English. 2019.

[20] rust-lang. *Rust - A language empowering everyone to build reliable and efficient software.* https://www.rust-lang.org, Accessed: 2024-09-01. 2024.

[21] Seastar. *Seastar.* https://seastar.io, Accessed: 2024-09-01. 2024.

[22] Snowflake. *OLAP vs. OLTP: The Differences*. https://www.snowflake.com/guides/olap-vs-oltp, Accessed: 2024-09-01. 2024.

[23] Snowflake. *What Is a Transactional Database?* https://www.snowflake.com/guides/what-transactional-database, Accessed: 2024-09-01. 2024.

[24] T. M. D. Team. *Matplotlib: Python plotting*. https://matplotlib.org/. Accessed: 2024-10-14. 2024.

[25] tokio-rs. *tokio - A runtime for writing reliable asynchronous applications with Rust. Provides I/O, networking, scheduling, timers, ...* `https://github.com/tokio-rs/tokio`, Accessed: 2024-09-01. 2024.

[26] tokio-rs. *tokio-uring - An io_uring backed runtime for Rust.* `https://github.com/tokio-rs/tokio-uring`, Accessed: 2024-09-01. 2024.

[27] *Wireshark: The World's Most Popular Network Protocol Analyzer*. `https://www.wireshark.org/`. Accessed: 2024-10-05. 2024.