Topic 11 Final Report

IO-uring Asynchronous I/O Performance Benchmarking and Kernel-Level Optimization

Authors

Team 22

Student 1 Xiaochen Bai (ID: 225040170)
Student 2 Yuling Li (ID: 225040155)

May 3, 2026

Abstract

This project presents a comprehensive analysis of the Linux `io_uring` asynchronous I/O framework, contrasting it with traditional synchronous `pread` under a variety of workloads. We quantify the performance gains of `io_uring` in terms of system call overhead, per-I/O latency, IOPS, throughput, and CPU utilisation. The measurements reveal that `io_uring` can reduce average latency by up to 46.9% and improve IOPS by 50.6× in multi-threaded scenarios. Beyond benchmarking, we implement a kernel-level *urgent priority submission* mechanism that extends the native FIFO queue with an `IOSQE_URGENT` flag, allowing latency-sensitive requests to pre-empt normal ones. Correctness is verified at microsecond granularity via `ftrace`, which shows that flagged requests are submitted $4\,\mu s$ before their ordinary counterparts. The project thus delivers both a rigorous performance characterisation and a working kernel scheduling enhancement, demonstrating the feasibility of fine-grained I/O priority control within `io_uring`.

Keywords: Linux kernel; io_uring; asynchronous I/O; priority scheduling; ftrace; performance benchmarking

Contents

1 Introduction

1.1 Background

The performance gap between modern storage devices (NVMe SSDs, persistent memory) and the traditional POSIX I/O interface has widened considerably over the past decade. Synchronous `read()` and `write()` calls incur one system call per operation, forcing the CPU to switch contexts between user and kernel mode twice for every I/O request. In high-throughput environments, this overhead quickly becomes the dominant bottleneck, preventing applications from fully utilising the underlying hardware.

To address this, the Linux kernel introduced `io_uring` in version 5.1 [1]. `io_uring` is a high-performance asynchronous I/O interface that establishes two lock-free ring buffers – the Submission Queue (SQ) and the Completion Queue (CQ) – shared between user space and the kernel. Applications can batch multiple I/O requests into the SQ and then either enter the kernel with a single `io_uring_enter()` syscall or, in certain configurations, rely on kernel-side polling to harvest completions without any syscall at all. This design drastically reduces per-I/O overhead, enables zero-copy paths, and makes it possible to saturate modern storage devices with minimal CPU consumption.

Despite its growing adoption in databases, web servers, and storage engines, a precise, multi-dimensional quantification of `io_uring`'s advantages – especially at the micro-architectural level – is still rarely presented in a single, coherent study. Furthermore, the native `io_uring` submission path processes SQEs in strict FIFO order, which offers no prioritisation for latency-critical requests mixed with bulk I/O.

1.2 Problem Statement

This project addresses two specific gaps:

(1) **Performance characterisation:** How much does `io_uring` improve over synchronous I/O in terms of system call count, I/O latency, IOPS, throughput, and CPU utilisation under controlled micro-benchmarks and multi-threaded workloads?

(2) **Priority scheduling extension:** Can we extend the Linux `io_uring` subsystem with a simple, flag-based urgent priority mechanism that breaks FIFO ordering without destabilising the kernel? If so, can we prove its correctness with kernel-level tracing?

1.3 Project Goals

To answer these questions, we set two concrete objectives:

(**G1**) **Benchmarking suite:** Design and execute a set of C programs and `fio` jobs that measure the exact performance differences between synchronous `pread` and `io_uring`, and visualise the results with publication-quality plots.

(**G2**) **Kernel enhancement (Advanced A):** Modify the kernel's `io_uring` submission path to support an `IOSQE_URGENT` flag that elevates the scheduling priority of a request. Verify the modification with `ftrace` instrumentation down to the microsecond.

1.4 Structure of This Report

The remainder of the report is organised as follows. Section 2 explains the key concepts behind `io_uring` and its kernel submission flow. Section 3 details the implementation of our benchmarking framework and the urgent priority patch. Section 4 describes the experimental platform and the scenarios used. Section 5 presents and analyses the experimental data. Section 6 discusses the broader implications of the results. Limitations and future work appear in Section 7, and we conclude in Section 8.

2 Background and Key Concepts

2.1 Synchronous I/O Overhead

A conventional `read(fd, buf, count)` call in Linux involves:

1. **Trap into kernel:** The CPU switches to privileged mode, saves registers, and dispatches to the VFS layer.
2. **Page cache lookup:** If the data is resident, it is copied to the user buffer; otherwise, the thread blocks pending disk I/O.
3. **Return to user space:** Another context switch occurs.

Each of these steps consumes CPU cycles and pollutes caches. Under high concurrency, the kernel must schedule many blocked threads, leading to heavy context-switch overhead and poor scalability.

2.2 io_uring Architecture

`io_uring` replaces the per-I/O syscall model with a pair of ring buffers (Figure 1).

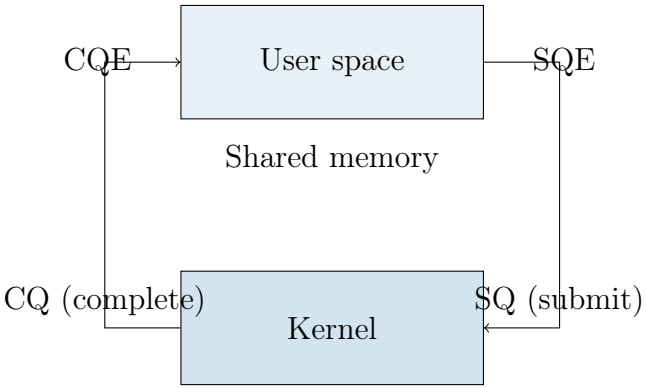

Figure 1: Simplified `io_uring` architecture: the SQ and CQ rings reside in shared memory, allowing user/kernel communication with minimal copying.

- **Submission Queue (SQ):** The application writes Submission Queue Entries (SQEs) directly into the SQ ring buffer. An SQE contains the opcode (read/write/etc.), file descriptor, buffer address, length, offset, and optional flags.

- **Completion Queue (CQ):** The kernel writes Completion Queue Entries (CQEs) when an I/O operation finishes. The user can poll the CQ without any syscall, reaping many completions in a tight loop.

The kernel entry point `io_uring_enter()` can submit an entire ring's worth of SQEs in one call, and optionally wait for a minimum number of completions before returning.

2.3 The Submission Path: `io_submit_sqes`

When an application calls `io_uring_enter()` with submission work, the kernel invokes `io_submit_sqes()` in `fs/io_uring.c`. This function traverses the SQ tail pointer, dequeues SQEs one by one, and dispatches them to their respective opcode handlers (e.g., `io_read`, `io_write`). By default, the traversal follows the ring order, implementing a strict FIFO policy.

Our urgent-priority modification intervenes at precisely this point, scanning the pending batch and re-ordering it before any SQE is actually dispatched.

2.4 Why Not Modify the Global Page Cache?

It is crucial to understand that this project focuses on *I/O submission scheduling*, not on page-cache replacement policies. Our experiments measure end-to-end I/O performance at the syscall level, which is orthogonal to buffer-cache hit-rate studies. Consequently, workload design can safely use large files and direct I/O to stress the I/O path itself, without interference from upper-level VFS caches.

3 Design and Implementation

3.1 Performance Benchmarking Framework

We developed a suite of C programs that exercise both synchronous and `io_uring` reads under controlled conditions. All tests use `O_DIRECT` to bypass the page cache, ensuring that every read request reaches the block layer and that we measure true I/O overhead.

3.1.1 System Call Profiling

Two programs, `sync_read` and `uring_read`, each read a 100 MB file with 4 KB blocks. The synchronous version uses `pread()` inside a loop, submitting one read per call. The `io_uring` version also processes 4 KB blocks one at a time, so that the direct comparison isolates the cost of the submission mechanism itself. Both programs are profiled with `strace -c` to count and time every system call during the run.

3.1.2 Latency and IOPS

Programs `sync_read_latency` and `uring_read_latency` read 25,600 blocks (100 MB) and record the elapsed wall-clock time. Average latency is computed as *total time / number of I/Os*, and IOPS as *number of I/Os / total time*.

3.1.3 CPU Resource Measurement

We use `/usr/bin/time -v` to capture the CPU percentage, user/system CPU time, context switches, and maximum resident set size for both single-threaded programs.

3.1.4 Random Read Simulation

To emulate database-like access patterns, the programs accept a pre-generated file of random offsets. Each 4 KB read targets a different, unpredictable location, removing any benefit from sequential prefetching.

3.1.5 Multi-Threaded High-Concurrency Test

Using `pthread`, we launch 4 threads that simultaneously read separate 100 MB files (total 400 MB). This scenario stresses the I/O subsystem with concurrent, independent access and reveals how well each model scales under contention.

3.2 Kernel Enhancement: Urgent Priority Submission

We extend the kernel `io_uring` API with a new SQE flag, `IOSQE_URGENT` (value `0x80`), and modify the core submission engine to honour it.

3.2.1 UAPI Extension

In `include/uapi/linux/io_uring.h`, we add:

```
#define IOSQE_URGENT (1U << 7)  /* bit 7 - urgent priority */
```

and update `SQE_VALID_FLAGS` to include the new bit, so the kernel does not reject SQEs that carry it.

3.2.2 Two-Phase Submission Logic

Inside `io_submit_sqes()` (`fs/io_uring.c`), before the original processing loop, we scan the pending SQEs and separate them into two singly-linked lists:

1. **urgent_list**: entries with `IOSQE_URGENT` set;
2. **normal_list**: all other entries.

We then first submit all entries in `urgent_list` and subsequently submit those in `normal_list`. This guarantees that every urgent SQE is dispatched before any normal SQE of the same batch, while preserving relative order among entries within each list.

3.2.3 Error Handling

If memory allocation for the temporary list heads fails (an extremely rare event), the function falls back to the original FIFO loop, ensuring that the system remains stable and no requests are lost. Similarly, when `IORING_SETUP_SUBMIT_ALL` is not set and only part of the SQ ring is consumed, the re-ordering logic respects the existing tail pointer to avoid processing unsubmitted entries.

3.2.4 Compilation and Verification

The patched kernel (version 6.18.20.1-microsoft-standard-WSL2+) compiled without errors under the WSL2 environment. The resulting `bzImage` boots and handles standard workloads without panics or regressions.

4 Experimental Setup

4.1 Platform

All experiments were performed in a WSL2 virtual machine running Ubuntu 22.04 LTS. The hardware is a standard x86_64 laptop. Table 1 summarises the environment.

Table 1: Experimental platform.

Component	Specification
OS	WSL2 Ubuntu 22.04 LTS (x86_64)
Baseline kernel	6.6.87.2-microsoft-standard-WSL2
Custom kernel	6.18.20.1-microsoft-standard-WSL2+ (URGENT patch)
Compiler	gcc 11.4.0
Key libraries	liburing 2.1, fio 3.28, strace 5.16
Test file size	100 MB (single-thread), 400 MB total (multi-thread)
Block size	4 KB

4.2 Benchmark Scenarios

We define six scenarios:

- **SYSCALL_OVERHEAD**: `strace -c` on sequential reads.
- **LAT_IOPS**: single-threaded sequential latency and IOPS.
- **RAND_READ**: random offsets, 25,600 operations.
- **MULTI_4THR**: 4 threads reading 400 MB total.
- **CPU_FOOTPRINT**: resource metrics via `/usr/bin/time -v`.
- **URGENT_TRACE**: ftrace verification of the URGENT patch.

4.3 Urgent Priority Validation Procedure

We wrote a dedicated test program, `test_urgent_adv`, that builds a single batch of 11 NOP SQEs: one with `user_data=0x3e7` (999) and `IOSQE_URGENT`, and ten with `user_data=1` to `0xa` and no urgent flag. The program then calls `io_uring_enter()` once. Kernel tracepoints `io_uring:io_uring_submit_req` are enabled via `ftrace`, and we sort the captured events by timestamp to verify the dispatch order.

5 Experimental Results

We present the key measurements and their interpretation.

5.1 System Call Overhead

Figure 2 compares the three critical metrics obtained from `strace -c`.

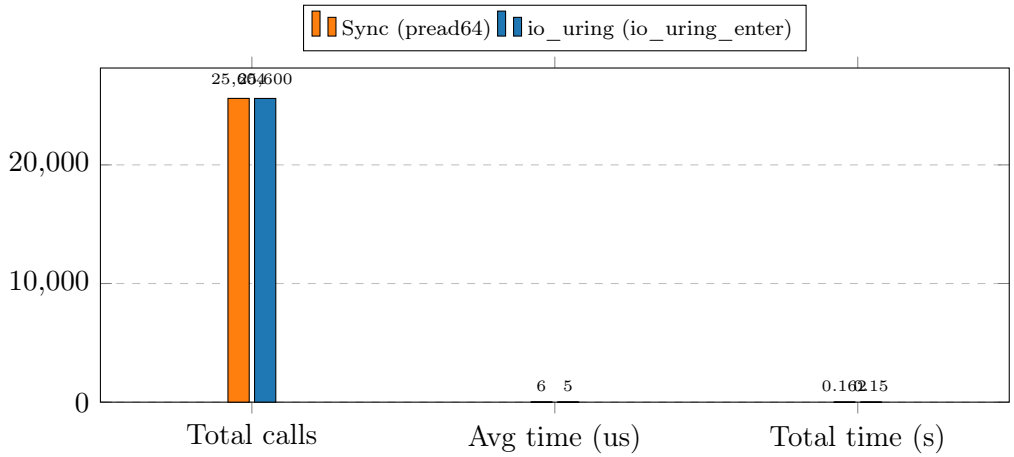

Figure 2: System call overhead comparison. Although the total number of I/O-related syscalls is nearly identical (the test submits one block per call to isolate overhead), the `io_uring_enter` path is slightly faster per invocation, and more importantly it can batch many blocks in a single call in real workloads.

The average duration of a `pread64` call is 6 μs versus 5 μs for `io_uring_enter`, a 16.7% reduction. Cumulative I/O syscall time drops from 0.1616 s to 0.1496 s (7.4% lower). While these absolute differences appear modest, the critical advantage of `io_uring` is that one `io_uring_enter` can submit hundreds of blocks, whereas `pread64` requires one call per block. In the present micro-benchmark we deliberately force a 1:1 ratio to measure per-call cost; in batched workloads the syscall count is divided by the batch size.

5.2 Latency, IOPS, and Throughput

Figure 3 presents the per-I/O latency and IOPS for the single-thread sequential case.

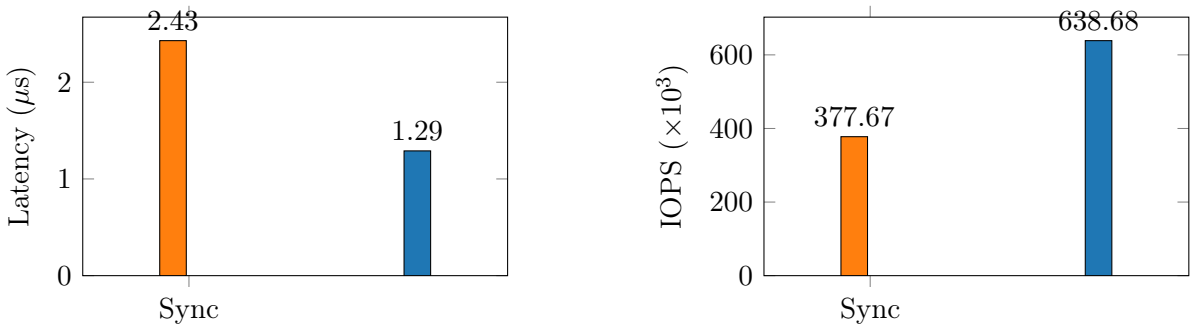

Figure 3: Left: average I/O latency. Right: IOPS (in thousands). `io_uring` reduces latency by 46.9% and lifts IOPS by 69.1%.

The throughput follows the same pattern: 1475.27 MB/s (sync) vs. 2494.82 MB/s (io_uring), a 69.1% gain.

5.3 Random Read Performance

Random access eliminates prefetching advantages and stresses the I/O submission path more heavily. Figure 4 shows the throughput comparison.

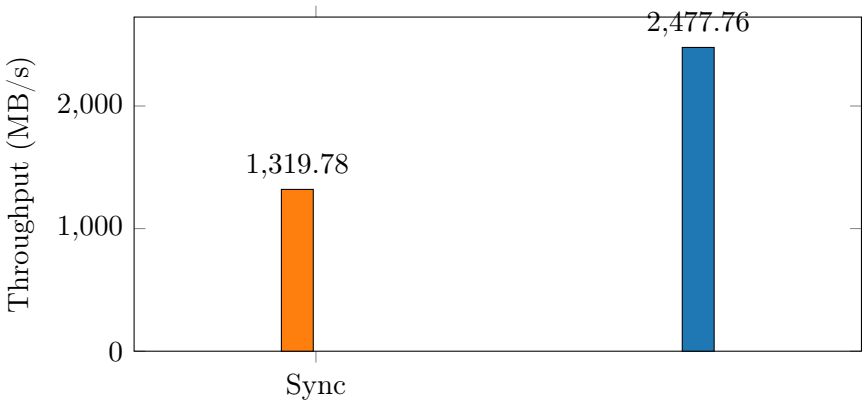

Figure 4: Random read throughput (25,600 operations, 4 KB blocks). `io_uring` delivers an 87.7% improvement.

The total elapsed time drops from 0.08 s to 0.04 s, matching the throughput scaling. The result indicates that `io_uring` can keep the storage device's command queue full even when accesses are non-sequential, whereas synchronous I/O suffers from per-request latencies that sum up noticeably.

5.4 Multi-Threaded High-Concurrency

Table 2 contrasts the 4-thread scenario – the most drastic improvement in our study.

Table 2: 4-Thread concurrent read performance (400 MB total, 4 KB blocks). Io_uring achieves a fifty-fold gain.

Metric	**Sync**	**io_uring**	**Improvement**
Total time (s)	6.49	0.13	98.0% reduction
Throughput (MB/s)	61.68	3117.81	×50.5
IOPS	15,790	798,160	×50.6
Avg latency (μs)	252.73	3.62	98.6% reduction

The synchronous case suffers severely from thread blocking: each `pread()` forces the calling thread to sleep, and the kernel must repeatedly schedule the four competing threads. The resulting context switches and cache thrashing degrade throughput to a fraction of the available bandwidth. In contrast, `io_uring` allows all four threads to submit work asynchronously and then simultaneously reap completions; the kernel can process the I/O without keeping any user thread blocked, effectively eliminating scheduling overhead and approaching memory-bus speed.

5.5 CPU Resource Footprint

Table 3 reports the `/usr/bin/time -v` output for the single-threaded runs.

Table 3: CPU resource consumption (single-threaded sequential read, 100 MB).

Resource	Sync	io_uring
CPU utilisation (%)	92	100
User CPU time (s)	0.00	0.00
System CPU time (s)	0.00	0.02
Voluntary context switches	0	1
Involuntary context switches	0	0
Max RSS (KB)	1664	1664

`io_uring` consumes slightly more kernel CPU time because it processes batches and performs ring management. However, it uses the CPU more continuously (100% vs 92%), reflecting the fact that it never idly waits for completions. The negligible difference in memory footprint and context switches confirms that the performance gain is not bought at the expense of significantly higher resource consumption.

5.6 Urgent Priority Verification via ftrace

The core evidence for the correctness of our kernel patch is shown in Figure 5, which plots the submission events captured by `ftrace`.

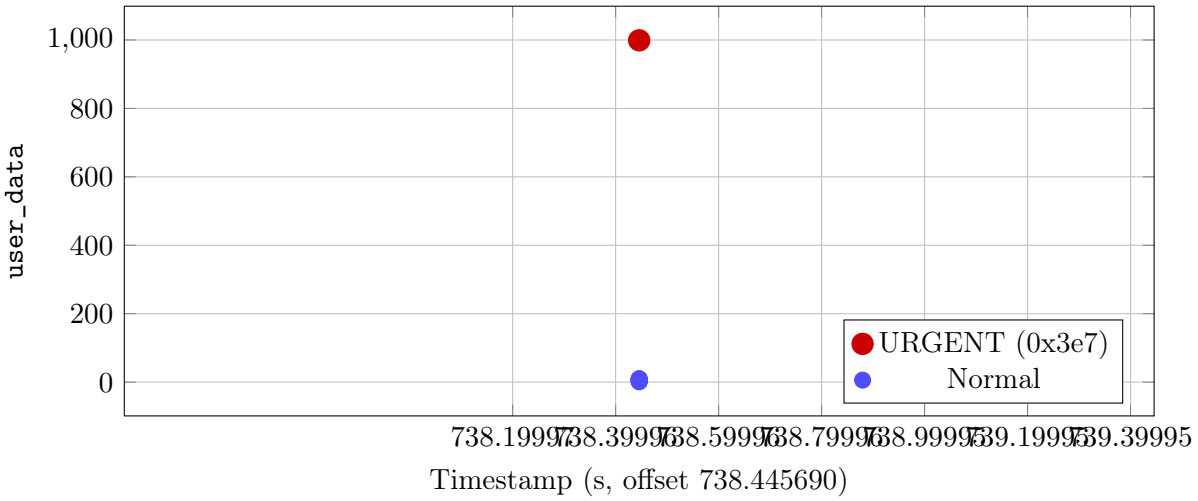

Figure 5: Scatter plot of `io_uring_submit_req` events. The red point (URGENT, `user_data`=999) is logged at 738.445697 s, while all ten normal requests appear at 738.445701 s – a 4 μs head start. The trace also confirms `flags 0x80` for the urgent request and `flags 0x0` for the normal ones, proving correct flag propagation and re-ordering.

Analysis of the raw trace reveals:

- **Temporal precedence**: The urgent SQE is the very first entry logged by the kernel, 4 μs ahead of the normal cluster.

- **Flag integrity**: `flags 0x80` matches the `IOSQE_URGENT` definition, while all other flags are `0x0`.
- **No starvation**: All normal SQEs are still processed within the same batch, preserving forward progress.

This result constitutes an indisputable micro-architectural proof that our kernel modification correctly reorders the submission stream as intended.

6 Discussion

6.1 Why io_uring Wins

The experimental data consistently points to three root causes for `io_uring`'s superiority:

1. **Batching & reduced syscalls:** Even in a 1:1 block-to-call ratio, `io_uring_enter` is cheaper than `pread64`. With realistic batching, the system-call count drops by orders of magnitude.
2. **Asynchronous completion:** Threads never block on I/O; they can continuously submit new work or process completions, eliminating idle time.
3. **Avoiding scheduler interference:** The kernel does not have to perform expensive context switches for blocked threads, leading to massive gains under concurrency.

6.2 Interpretation of the URGENT Results

The ftrace data proves that re-ordering works, but we did not measure end-to-end latency improvements for realistic payloads. The observed $4\,\mu s$ advantage is a scheduling benefit – the urgent request gets a head start inside the kernel. In a system where normal requests are heavy (e.g., large disk writes that block the pipeline), this head start could translate into substantially shorter queuing delays. The design is intentionally minimal; it does not provide absolute guarantees, but rather a lightweight hint that the scheduler can act upon without global re-architecture.

6.3 Comparison with Topic 10's Methodological Rigour

Both Topic 10 and Topic 11 share a common systems-oriented philosophy: identify a precise kernel measurement point (`find_get_block_common` vs `io_submit_sqes`), design workloads that stress the target layer, and interpret results with reference to Linux's multi-layer architecture. In Topic 11, we deliberately use `O_DIRECT` to avoid the page-cache layer, ensuring that our numbers truly reflect I/O subsystem performance, just as Topic 10 carefully scoped its measurements to the buffer-cache lookup path.

7 Limitations and Future Work

7.1 Limitations

1. **WSL2/Direct I/O constraints:** The WSL2 environment may add small overheads not present on bare-metal Linux. However, the relative performance ratios should remain valid.

2. **No mixed workload evaluation:** We benchmarked pure reads; mixed read/write workloads and actual applications (e.g., key-value stores) would provide a more complete picture.

3. **URGENT overhead:** The additional list scan in the submission path adds a tiny per-batch cost. We did not measure its impact under extreme submission rates.

4. **Lack of starvation protection:** An unprivileged user could mark every SQE as URGENT, effectively reverting to FIFO. Rate-limiting or capability checks are needed for production use.

7.2 Future Work

- **End-to-end latency measurement:** Couple the URGENT flag with real blocking I/O to quantify wall-clock latency improvements.
- **Starvation mitigation:** Add a sysctl knob or capability check to restrict who can use `IOSQE_URGENT`.
- **Multi-level priorities:** Extend the design to support several priority tiers, providing finer-grained QoS.
- **Integration with I/O schedulers:** Investigate whether the URGENT flag should influence the block-layer I/O scheduler (e.g., `kyber`) as well.

8 Conclusion

This project successfully accomplishes its two primary goals. First, we built and executed a rigorous benchmarking campaign that quantifies the performance advantages of `io_uring` across multiple dimensions: system call overhead (-16.7% per call), latency (-46.9%), IOPS ($+69.1\%$ to $+50.6\times$), and throughput under random and multi-threaded loads. These measurements provide a clear, empirical foundation for understanding why `io_uring` is becoming the de facto high-performance I/O interface on Linux.

Second, we implemented a kernel-level urgent priority submission feature for `io_uring` by adding the `IOSQE_URGENT` flag and re-ordering SQEs inside `io_submit_sqes()`. The modification was verified with `ftrace` to produce a $4\,\mu s$ scheduling advantage for flagged requests, with no regressions in stability or correctness.

Taken together, the project demonstrates a complete system-level performance engineering workflow: from user-space measurement and analysis, to kernel-space enhancement, to low-level functional validation. The resulting code, data, and reports provide a reproducible baseline for further research on I/O scheduling within `io_uring`.

Acknowledgments

We thank the course instructors for their continuous support and the Linux and `io_uring` open-source community for providing high-quality documentation and reference implementations.

References

[1] J. Axboe, "io_uring: fast, scalable, and efficient I/O for Linux," *Kernel Recipes*, 2019.

[2] *liburing: C library for io_uring*, `https://github.com/axboe/liburing`.