UringCL: A Lightweight io_uring Convergence Layer for Adoption in Legacy Event Loops

by

Armin Afsharian

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2026

© Armin Afsharian 2026

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

High-performance network servers depend on efficient I/O mechanisms to manage thousands of concurrent connections with minimal latency and overhead. While traditional readiness-based interfaces (e.g., `select`, `poll`, `epoll`) notify applications when I/O operations can proceed, they still require synchronous system-calls to execute the operations. This synchronous requirement causes frequent user–kernel transitions, which limits scalability under heavy load. In contrast, the `io_uring` interface offers a fundamentally different approach by providing a completion-based I/O model that minimizes system-call overhead and enables true asynchronous data transfer.

Although the performance benefits of `io_uring` are well established in storage systems, its integration into high-throughput network applications remains limited. This thesis aims to bridge this integration gap by making the adoption of `io_uring` accessible and providing a structured vehicle for evaluating its performance in network-bound environments. To this end, the `io_uring` Convergence Layer (`UringCL`) is presented to transparently map the synchronous I/O calls of readiness-driven applications onto asynchronous `io_uring` operations. The UringCL simplifies initialization, event handling, and data transfer while preserving the existing control flow of legacy applications, allowing for incremental migration toward completion-based I/O without major redesign.

The `UringCL` architecture facilitates the practical integration of `io_uring` into established network architectures and provides a consistent framework for measuring its impact on throughput, latency, and CPU efficiency. Experimental results demonstrate significant performance advantages over traditional models. In bulk-transfer workloads, the system delivers up to 40% higher throughput than `epoll` due to superior batching capabilities. In request-response scenarios involving Memcached, the integration achieves higher peak throughput and maintains significantly lower and more stable tail latency under heavy load. Furthermore, `UringCL` achieves these benefits with negligible overhead, proving that completion-based I/O can be adopted seamlessly to enhance the efficiency of modern network servers.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor Martin Karsten, whose guidance and support made this thesis possible.

I also would like to thank my lab-mates: Mohammadamin Shafiei, Alireza Salamat, Ali Abbasi, Qishen Wu, and Michale Lapshin. I am grateful for the support, friendship and memorable experiences that we shared during this program.

Dedication

This thesis is dedicated to my wife, whose immense support made this journey possible. Her compassion, love, and unwavering care helped me achieve my dreams; without her, this work would not exist.

I also dedicate this thesis to my parents, who have supported me unconditionally throughout my life. Their wisdom and love were fundamental to this achievement.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures ix

List of Tables x

List of Abbreviations xi

1 Introduction 1

2 Background and Related Works 3

2.1 Synchronous I/O . . . 3

2.1.1 Early Multiplexing Interfaces: `select` and `poll` . . . 3

2.1.2 Improvements and Alternative Mechanisms . . . 4

2.1.3 Scalability Characteristics . . . 4

2.1.4 Event Handling on Linux via `epoll` . . . 5

2.2 Asynchronous I/O Interfaces . . . 6

2.2.1 Linux Asynchronous I/O (AIO) . . . 6

2.3 The `io_uring` Interface . . . 7

2.3.1 Ring-Buffer Architecture . . . 7

2.3.2 System Call Interface 8
2.3.3 The `liburing` Abstraction 9
2.3.4 Architectural Comparison to `epoll` 9
2.4 Advanced `io_uring` Mechanisms 9
2.4.1 Task–Work Scheduling 10
2.4.1.1 Default Behavior 10
2.4.1.2 Cooperative Task Running 11
2.4.1.3 Deferred Task Running 12
2.4.1.4 Implications 14
2.4.2 Multishot Operations and Buffer Management 14
2.4.2.1 The Standard Receive Model 14
2.4.2.2 Kernel-Managed Buffer Pools 15
2.4.2.3 Multishot Receive 15
2.4.3 Eventfd Integration 16
2.5 Adoption of `io_uring` 17

3 Design of the io_uring Convergence Layer 18
3.1 Motivation 18
3.2 Receive Operations and Buffer Management 19
3.2.1 Automated Initialization 19
3.2.2 Bridging the Semantic Gap 20
3.2.3 Copy-Free Reads via Pointer Swapping 20
3.3 Flexible Event Retrieval 20
3.4 Integration of the Convergence Layer 21
3.4.1 Integration via `eventfd` 21
3.4.2 Integration via Native `io_uring` 22
3.4.3 The Data Retrieval Interface 24
3.5 Integration with Applications 24
3.5.1 Neper 24
3.5.2 Memcached 25
3.5.2.1 The Role of Libevent 25
3.5.2.2 Architectural Adaptation 25

4 Evaluation **27**
4.1 Experimental Setup . . . 27
4.1.1 Hardware . . . 27
4.1.2 Software Environment . . . 28
4.1.3 Benchmark . . . 28
4.1.3.1 Neper . . . 28
4.1.3.2 Memcached . . . 29
4.2 Data Collection . . . 29
4.2.1 Mutilate Metrics . . . 29
4.2.2 System-Level Metrics . . . 30
4.3 Performance Metrics . . . 30
4.4 Neper Evaluation . . . 31
4.4.1 Evaluation Methodology . . . 31
4.4.2 TCP Stream (tcp_stream) . . . 31
4.4.2.1 Experimental Configuration: Disabling Offloads . . . 32
4.4.2.2 Evaluation of Task-Work Scheduling Modes . . . 32
4.4.2.3 Impact of Copy-Free Reads . . . 34
4.4.2.4 Baseline Performance vs. Epoll . . . 35
4.4.3 TCP Request/Response (tcp_rr) . . . 35
4.4.3.1 Evaluation of Task-Work Scheduling Modes . . . 36
4.4.3.2 Impact of Copy-Free Reads . . . 36
4.4.3.3 Baseline Performance vs. Epoll . . . 40
4.5 Evaluation of Memcached . . . 40
4.5.1 Closed-loop Throughput Evaluation . . . 41
4.5.2 Open-Loop Latency Evaluation . . . 43

5 Conclusion **46**
5.1 Future Work . . . 47
5.1.1 Send-Side Integration . . . 47
5.1.2 Leveraging Evolving `io_uring` Features . . . 47

References **49**

List of Figures

2.1 High-level pseudocode for an `epoll`-based server loop 5

2.2 Shared ring buffers in `io_uring`: Submission Queue (SQ) and Completion Queue (CQ) 8

2.3 Default behavior: The kernel forces an IPI to interrupt user-space processing immediately upon completion arrival. 11

2.4 Cooperative behavior: `task_work` piggybacks on the return from an unrelated system-call, avoiding forced IPIs but still running frequently. 12

2.5 Deferred behavior: Completions are strictly batched and processed only when the application explicitly waits, maximizing efficiency. 13

2.6 The standard single-shot `recv` loop 15

2.7 A multishot `recv` loop using kernel-managed buffers 16

3.1 Hybrid integration: `epoll` triggers the callback dispatcher of `UringCL` . . . 23

3.2 Native integration: `uring_poll` drives the loop and callbacks 23

3.3 The read implementation of `UringCL`: Non-blocking user-space copy 24

4.1 Task-Work Modes: Throughput comparison for Neper 33

4.2 Copy vs. No-Copy: Throughput impact of pointer swapping optimization. 34

4.3 Baseline Comparison: `io_uring` vs. `epoll` throughput. 35

4.4 Performance of `tcp_rr` benchmark (Task-Work Scheduling Modes). 37

4.5 Performance of `tcp_rr` benchmark (Copy vs. No-Copy). 38

4.6 Performance of `tcp_rr` benchmark (Epoll baseline). 39

4.7 Queries per Second (QPS) as a function of the number of client connections for Vanilla Memcached and `io_uring` variants. 42

4.8 Average Read Latency vs. Throughput (Memcached 8 cores). 43

4.9 99th Percentile Latency vs. Throughput (Memcached 8 cores). 44

4.10 95th Percentile Latency vs. Throughput (Memcached 8 cores). 45

List of Tables

4.1 `perf` Metrics for Maximum Throughput Measured in User and Kernel Space 43

List of Abbreviations

AIO Asynchronous I/O 1, 6, 7, 9, 17

API Application Programming Interface 17, 19, 20, 25

CPQ Cycles per Query 30, 31, 42

CPS Cycles per Second 31

CQ Completion Queue ix, 7, 8, 15–17, 22

CQE Completion Queue Entry 7, 10, 14–16

FD File Descriptor 3, 6, 15, 33

Gbps Gigabits Per Second 31, 32, 35

GRO Generic Receive Offload 32

IOPS Input/Output Operations Per Second 17

IPC Instructions per Cycle 30, 31, 42

IPI Inter-Processor Interrupt 10, 11

IPQ Instructions per Query 30, 31

IRQ Interrupt Request 28

LLC Last-Level Cache 27

MTU Maximum Transmission Unit 32

NAPI New API 48

NIC Network Interface Card 28, 32

NUMA Non-Uniform Memory Access 27, 28

PMU Performance Monitoring Unit 30

POSIX Portable Operating System Interface 3

QPS Queries per Second ix, 29–31, 41–43

SPDK Storage Performance Development Kit 17

SQ Submission Queue ix, 7, 8, 15

SQE Submission Queue Entry 7, 9, 14, 16

TSO TCP Segmentation Offload 32

Chapter 1

Introduction

High-performance network servers must efficiently manage thousands of concurrent connections while maintaining minimal latency and processing overhead. The foundation for these servers is their I/O mechanism. Traditional interfaces, such as `select`, `poll`, and `epoll`, operate on a **readiness-based** model [5, 17, 19, 32]. These interfaces notify an application when an I/O operation *can* be performed, but the application must still execute a synchronous system call (e.g., `read` or `write`) to transfer the data. This design, while widely adopted in production systems like NGINX [3] and Memcached [16], causes a significant performance bottleneck: frequent and costly transitions between user space and the kernel, which limit scalability under heavy load.

The `io_uring` interface, introduced in recent Linux kernels, offers a fundamentally different approach [26]. It provides a **completion-based** model, designed to overcome the limitations of the older Linux Asynchronous I/O (AIO) interface [2, 25]. As a flexible and efficient I/O paradigm [37], `io_uring` enables applications to submit I/O requests and retrieve their results asynchronously, often with a single system call for an entire batch of operations, thereby minimizing system-call overhead.

While the performance benefits of `io_uring` are well-established in storage systems [25, 30, 45], its adoption in high-throughput network applications remains limited. The primary challenge is an architectural one. The `io_uring` interface is complex, and it exhibit vastly different performance and CPU usage characteristics depending on the workload [25, 45]. This complexity, combined with the fact that most existing network servers are designed around the synchronous, readiness-driven logic of `epoll`, makes migration to `io_uring`'s asynchronous, completion-based paradigm a fundamental and invasive redesign.

This thesis aims to bridge this critical integration gap. The primary contribution is the `io_uring` Convergence Layer (`UringCL`), a lightweight library that reconciles the asynchronous behavior of `io_uring` with the synchronous abstractions used by readiness-driven software [24]. `UringCL` is designed to make `io_uring` adoption more accessible by simplifying initialization, event handling, and data transfer. This architecture preserves

the existing control flow of legacy applications, which enables incremental migration toward completion-based I/O without requiring a major architectural overhaul.

The `UringCL` design serves a dual purpose. It facilitates the practical integration of `io_uring` into established network architectures, and provides a consistent framework for measuring the real-world impact of `io_uring` on throughput, latency, and CPU efficiency. Experimental results demonstrate that completion-based I/O can be adopted seamlessly in complex systems, underscoring the importance of adaptive integration mechanisms in the evolution of high-performance Linux networking.

The remainder of this thesis is organized as follows. Chapter 2 reviews the background and related work in Linux I/O models, including detailed discussions of readiness-based interfaces like `epoll` and the completion-based `io_uring` interface. Chapter 3 presents the design of `UringCL`. It explores the motivation and core components of the library, focusing on how `UringCL` utilizes `io_uring` features such as **provided buffers** and **multishot receives** to simplify application-level integration. Chapter 4 details the experimental setup and evaluation methodology, presents the performance results, and analyzes findings from both the Neper and Memcached benchmarks. Chapter 5 concludes with a summary of findings and directions for future research.

Chapter 2

Background and Related Works

2.1 Synchronous I/O

Traditional Portable Operating System Interface (POSIX) I/O system calls, such as `read` [18] or `write` [21], are synchronous. It means that the application's thread executes the call and must wait for the kernel to return a result. The behavior of this synchronous call depends on the File Descriptor (FD) mode.

If the FD is in its default blocking mode, a `read` call blocks the entire thread until data is available. This model is inefficient for concurrent servers, as one blocked client prevents the thread from servicing others.

If the FD is set to non-blocking, the `read` call returns immediately with an error (e.g., `EAGAIN`) if no data is ready. This approach avoids blocking, but to receive data, the application must resort to busy-polling and continuously calling `read` in a loop, which wastes CPU cycles.

I/O multiplexing interfaces were developed to solve this problem. These event-notification mechanisms allow an application to efficiently monitor multiple FDs simultaneously. Instead of blocking on a single `read` or resorting to a busy-loop, the application blocks on a single call (like `select` [19] or `poll` [17]). This call returns only when one or more FDs are ready, although it can be configured to return immediately if no events are pending. This approach is fundamental for building scalable event-driven servers [32].

2.1.1 Early Multiplexing Interfaces: `select` and `poll`

Early event-driven servers relied on `select` and `poll` for I/O multiplexing. These system calls help an application to monitor multiple FDs simultaneously, returning when at least one becomes ready for I/O. While effective for small descriptor sets, these approaches scale poorly.

Both `select` and `poll` incur $O(n)$ overhead on every call because the kernel must examine every descriptor in the interest set on each call, even if only a small fraction are active. This inefficiency becomes problematic in high-concurrency environments where thousands of simultaneous client connections are common [5, 14, 28].

2.1.2 Improvements and Alternative Mechanisms

To overcome these limitations, several improved event-notification mechanisms were developed:

- **/dev/poll** (Solaris, patched for Linux) separates the declaration of interest from the retrieval of events and reduces data copying by returning results through a shared memory region [32, 43]. Although this mechanism improves performance over `poll`, it was never merged into the mainline Linux kernel and remains a Solaris-centric solution. Consequently, it is not a viable standard for portable Linux applications.
- **kqueue** (BSD) generalizes event notification beyond sockets and file descriptors to include signals, asynchronous I/O completions, file system changes, and process state changes [32, 40]. While `kqueue` provides a unified and efficient interface for diverse kernel events, it is exclusive to BSD-derived systems (e.g., FreeBSD, macOS) and is not available on Linux.
- **epoll** (Linux) serves as the Linux-specific interface to solve the limitations of `poll`. It introduces a kernel-maintained event structure. With `epoll_create`, the kernel allocates an event data structure. The application then manages the interest set via `epoll_ctl` to add, modify, or remove descriptors as needed, rather than resubmitting the entire set for every poll operation. Consequently, `epoll_wait` returns only the currently ready descriptors, which avoids repeated scans and minimizes unnecessary data movement [5, 32].

2.1.3 Scalability Characteristics

Because `/dev/poll` and `kqueue` are not standard on Linux, `epoll` remains the primary high-performance alternative to legacy interfaces. The scalability of these approaches differs based on how the kernel processes the monitored descriptors:

- `select` and `poll` perform work proportional to the *size of the interest set*. For example, if an application monitors 10,000 descriptors but only a few are active, the kernel still scans all 10,000 on each call [28, 43].
- `epoll` decouples registration from event retrieval, making performance proportional to the *number of ready descriptors*. This reduces overhead for workloads with many inactive connections and few active ones, such as web servers [32].

2.1.4 Event Handling on Linux via `epoll`

On Linux, `epoll` is widely adopted for high-concurrency network services because it maintains an in-kernel interest list and returns only ready descriptors, which "scales well to large numbers of watched file descriptors" [5]. Production systems commonly rely on it. For example, NGINX documents `epoll` as its efficient connection-processing method on Linux 2.6+ [3]. Memcached is another example of application that relies on `epoll` to handle tens of thousands of connections per instance [16]. Figure 2.1 illustrates the logic of the server loop. The process proceeds in four steps:

1. The server initializes the listening socket L and registers it with the `epoll` instance E (lines 1–4).
2. It enters the main loop to wait for batches of ready descriptors (line 6).
3. Upon notification, it iterates through the events to either accept new clients (lines 8–10) or process I/O on existing connections (lines 11–19).
4. Finally, it removes and releases any descriptors that close or encounter errors (line 18).

```
1:  L ← create_listen_socket();
2:  bind_and_listen(L)
3:  E ← epoll_create()
4:  epoll_ctl(E, ADD, L, READ)                    // watch for new connections
5:  while true do
6:    events ← epoll_wait(E)
7:    for all (fd, mask) ∈ events do
8:      if fd = L then
9:        c ← accept(L)
10:       epoll_ctl(E, ADD, c, READ)                          // add connection
11:     else if mask has READ then
12:       n ← read(fd, inbuf)
13:       if n > 0 then
14:         if response_ready(fd) then
15:           m ← write(fd, outbuf)
16:         end if
17:       else
18:         epoll_ctl(E, DEL, fd); close(fd);
19:       end if
20:     end if
21:   end for
22: end while
```

Figure 2.1: High-level pseudocode for an `epoll`-based server loop

It is important to note that `epoll` acts as a readiness notification interface. As a result, `epoll_wait` reports that an operation can proceed without blocking, but it does not transfer the actual data. The application must still perform I/O via synchronous system-calls such as `read` and `write` (lines 12 and 15). On non-blocking sockets, a call that would otherwise block returns `-1` with `EAGAIN/EWOULDBLOCK`, at which point the loop resumes waiting for the next readiness notification [5, 18].

2.2 Asynchronous I/O Interfaces

While readiness-based models like `epoll` improve efficiency, they still require the application to perform the I/O using synchronous system-calls (e.g., `read`, `write`) after being notified. An alternative approach is a completion-based model. In this model, the application submits one or more I/O operations and waits for a notification from the kernel only after the operations are fully complete, including the data transfer.

2.2.1 Linux Asynchronous I/O (AIO)

Linux provides a native completion-based interface known as AIO [2]. The interface involves creating a context (`io_setup`), submitting batches of operations (`io_submit`), and retrieving completed events (`io_getevents`). However, this AIO interface has several severe limitations that prevent its widespread adoption [26]:

- **Direct I/O Only:** The most significant limitation is that AIO only supports asynchronous operations for unbuffered (i.e., avoid kernel page cache), and direct I/O (using the `O_DIRECT` flag). For normal buffered I/O, which most applications rely on, the interface behaves synchronously [26].
- **Limited to Files:** The AIO interface is designed specifically for file I/O. It does not support other types of FD, such as network sockets or pipes [2].
- **Blocking Submission:** The `io_submit` system call can block under certain conditions (e.g., if the kernel requires metadata or if internal request slots are full). This blocking in the submission stage is problematic because the intention of an asynchronous interface is to **immediately delegate** the operation to the kernel and return control to the application, without waiting for the operation to complete [26].
- **High Overhead:** The API is inefficient. Each operation requires significant data copying (over 100 bytes) for submission and completion structures, and requires at least two system calls (one to submit, one to harvest events) [26].

These limitations make Linux AIO unsuitable for most general-purpose applications and relegate it to a niche, primarily for database management systems that use direct I/O [1,26]. To address these shortcomings, the Linux community has developed `io_uring`, which is a modern and feature-rich asynchronous interface designed to overcome the limitations of legacy AIO.

2.3 The `io_uring` Interface

The `io_uring` interface has been designed from the ground up to be a true asynchronous, high-performance replacement that solves the deficiencies of readiness-based models like `epoll` and the restricted completion-based model of Linux AIO [26]. To achieve this, it relies on a shared memory architecture that minimizes copying and locking.

2.3.1 Ring-Buffer Architecture

At its core, `io_uring` is built on two ring buffers shared directly between the application and the kernel.

- **Submission Queue (SQ):** A ring buffer where the application produces requests (Submission Queue Entries (SQEs)) for the kernel to consume.
- **Completion Queue (CQ):** A ring buffer where the kernel produces results (Completion Queue Entries (CQEs)) for the application to consume.

This design implies a single-producer, single-consumer model for each ring. In the SQ, the application functions as the producer and the kernel as the consumer; conversely, in the CQ, the roles are reversed. This explicit separation allows the kernel and application to coordinate access using lightweight memory ordering barriers rather than shared locks (mutexes), which would otherwise introduce significant system-call overhead [26].

As shown in Figure 2.2, the application submits work by advancing the SQ tail, while the kernel consumes requests from the head. Similarly, the kernel posts completions to the CQ tail, which the application consumes from the head.

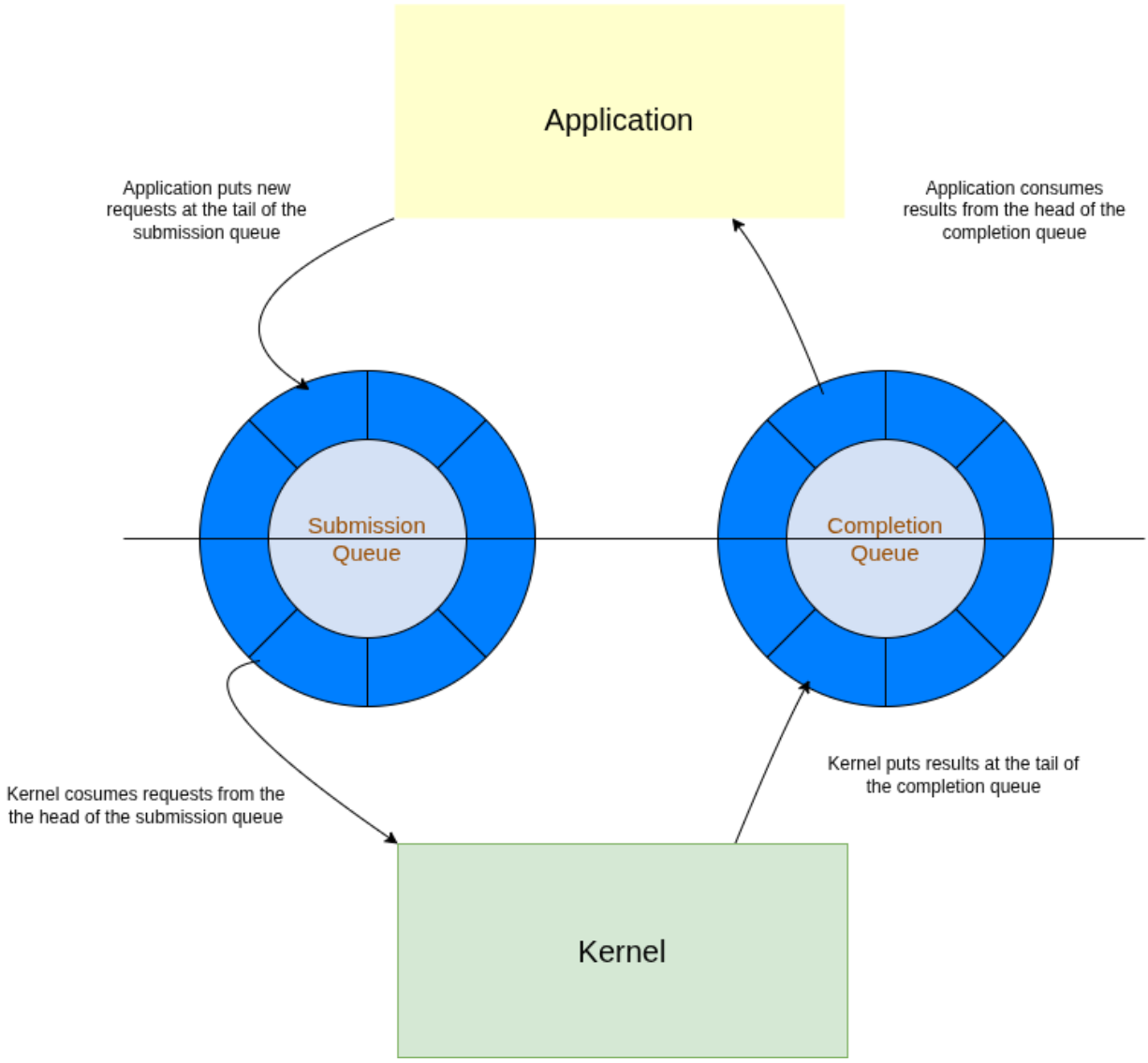


Figure 2.2: Shared ring buffers in `io_uring`: SQ and CQ

2.3.2 System Call Interface

While the data transfer happens via shared memory, the application must still initialize the interface and signal the kernel. This administration is managed through two primary system calls:

1. `io_uring_setup`: Initializes the instance and creates the rings. Crucially, it returns a file descriptor that the application uses to map the shared ring buffers into its own address space using the `mmap` system-call. This mapping is what enables the zero-copy communication channel [26].

2. `io_uring_enter`: This system call is the main communication signal. It informs the kernel that new SQEs are available in the submission queue. It serves a dual purpose: with a single call, it can submit new operations *and* wait for a minimum number of completions, effectively batching submission and completion into one user/kernel transition [26].

2.3.3 The `liburing` Abstraction

Directly manipulating these shared ring pointers, memory barriers, and invoking raw system calls is error-prone and requires significant boilerplate code. To address this, the `liburing` library provides a lightweight user-space abstraction layer.

`liburing` simplifies development by managing the ring setup, teardown, and memory mapping internally. It exposes a simplified API for preparing, submitting, and collecting operations, while maintaining the efficiency of the underlying completion-driven event loop [7, 26].

2.3.4 Architectural Comparison to `epoll`

The shift from `epoll` to `io_uring` represents a fundamental change in control flow:

- **Event Type:** While `epoll` reports *readiness* ("data is available to read"), `io_uring` reports *completion* ("data has been read into this buffer").
- **System Calls:** `epoll` requires separate system-calls to wait for events and then to perform the I/O. `io_uring` can batch submission and completion into a single `io_uring_enter` call.
- **Versatility:** Unlike AIO, which is largely limited to file I/O, and `epoll`, which is primarily used for monitoring readiness events on networking and device file descriptors **but does not support file I/O**, `io_uring` supports a unified interface for a broad range of operations including file I/O, sockets, timers, and signals [26].

2.4 Advanced `io_uring` Mechanisms

Although the basic ring-buffer mechanism provides the foundation for asynchronous I/O, achieving high-throughput networking requires the use of advanced kernel features to effectively manage latency and execution flow. This section examines the specific architectural mechanisms that are most critical for optimizing network applications.

2.4.1 Task–Work Scheduling

When an `io_uring` operation completes, such as when data arrives at a socket, the kernel processes the completion and posts the CQE. Unlike traditional readiness-based models where an application must issue a `recv` system-call after notification, `io_uring` is completion-based. The kernel performs the actual receive operation when data becomes pending on the socket and places that data into the provided buffer.

To achieve this transfer, the kernel triggers an internal callback to handle the data as soon as it arrives. `io_uring` developers refer to this mechanism as `task_work`. The application controls the execution mode of these callbacks, which determines system latency and throughput. `io_uring` provides three distinct models to control when this queued work runs [10].

2.4.1.1 Default Behavior

By default, `io_uring` prioritizes latency, ensuring that completions are processed immediately. The kernel employs two strategies to achieve this immediacy:

- The kernel processes data arrival whenever the application transitions from kernel-space to user-space, such as upon return from any system-call.
- If the application is executing in user-space, the kernel forces an immediate kernel-to-user-space transition. This transition is achieved by sending an interrupt to the application thread. Although an Inter-Processor Interrupt (IPI) is typically employed for alerting another core, the kernel uses it here to alert the application about the kernel routine that needs to run on the same core. This interrupt forces the process into the kernel to handle the pending work [39].

Forceful interruption negatively affects performance, as illustrated in Figure 2.3. For many applications, this default strategy causes degradation due to the overhead of the kernel-user transition, the interruption of the application's processing logic, and reduced batching efficiency [10, 39].

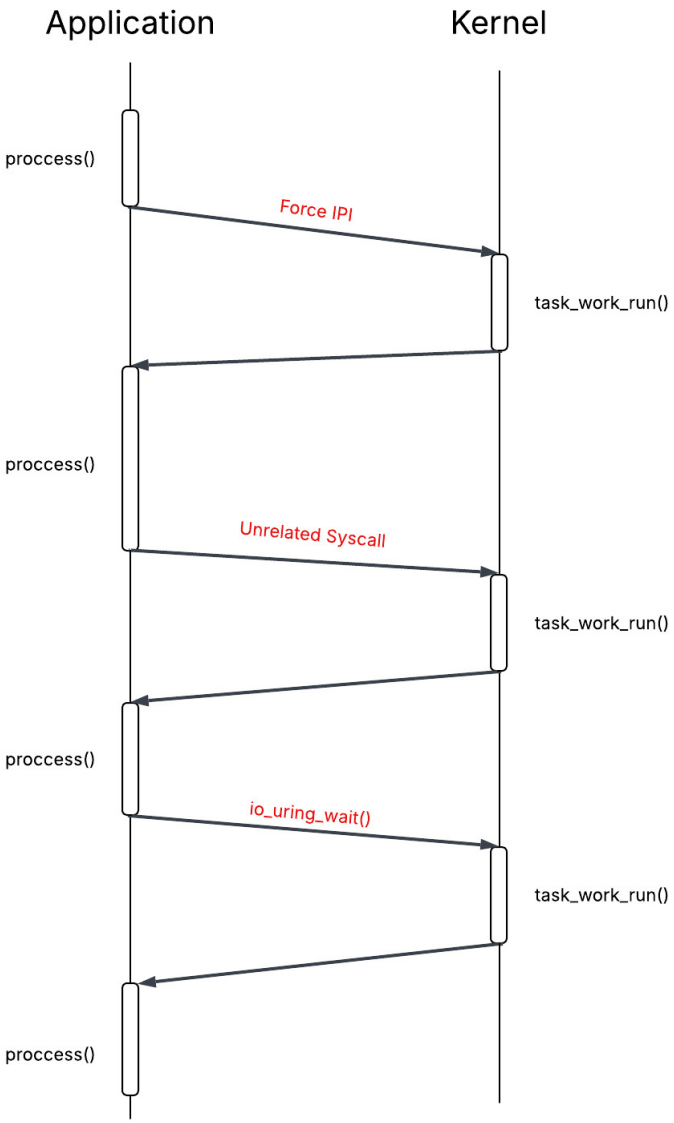


Figure 2.3: Default behavior: The kernel forces an IPI to interrupt user-space processing immediately upon completion arrival.

2.4.1.2 Cooperative Task Running

To mitigate the overhead of interrupts, the `IORING_SETUP_COOP_TASKRUN` flag prevents the kernel from sending forced IPIs. With this flag set, the kernel processes `task_work` only at the task's next natural kernel-to-user transition.

This flow, demonstrated in Figure 2.4, avoids the high cost of forced interruptions but couples completion processing to unrelated system activity. `task_work` runs on the return

path of *any* system call, even those unrelated to `io_uring` (shown in the figure as "Unrelated Syscall"). This coupling makes achieving optimal completion batching difficult [10].

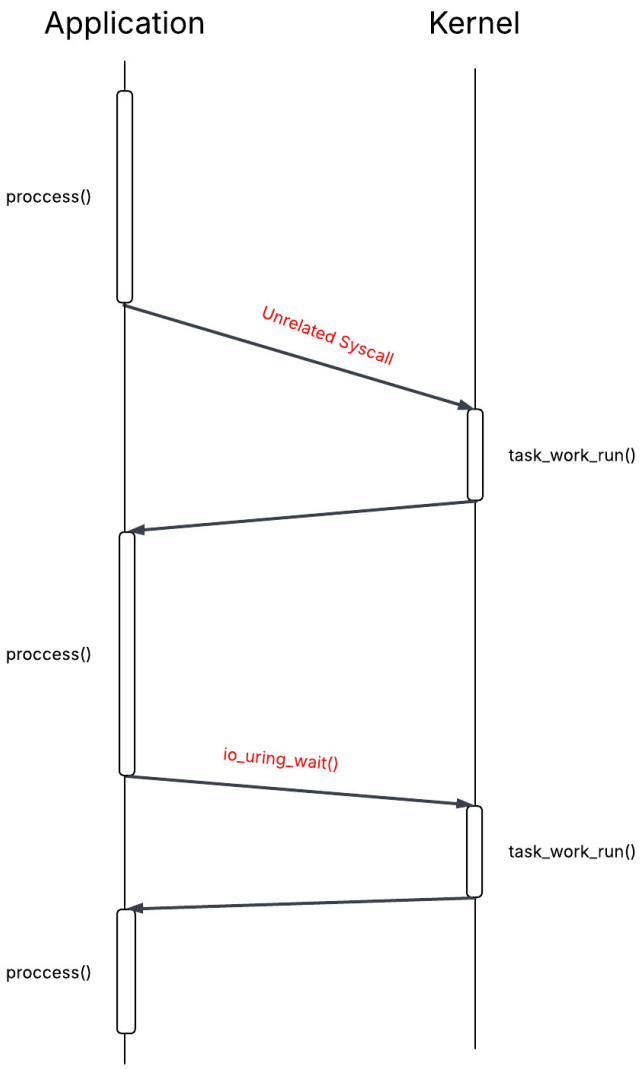


Figure 2.4: Cooperative behavior: `task_work` piggybacks on the return from an unrelated system-call, avoiding forced IPIs but still running frequently.

2.4.1.3 Deferred Task Running

The `IORING_SETUP_DEFER_TASKRUN` mode provides the application with full control over the batching of completions. When this flag is set, the kernel handles `task_work` only when the application explicitly waits for completions (e.g., during a call to `io_uring_submit_and_wait`).

The kernel queues many completion events, while the application runs uninterrupted, as shown in Figure 2.5, processing all events at once in a single, efficient batch. While the core `io_uring` mechanism follows a single-consumer (kernel) to single-producer (application) pattern, the interface technically allows multiple application threads to submit requests (multi-producer), although this configuration is generally not advised due to complexity. The `IORING_SETUP_SINGLE_ISSUER` flag, explicitly restricts submission to a single thread. This single-issuer constraint enables further batching optimizations and avoids unnecessary synchronization overhead in the kernel [10].

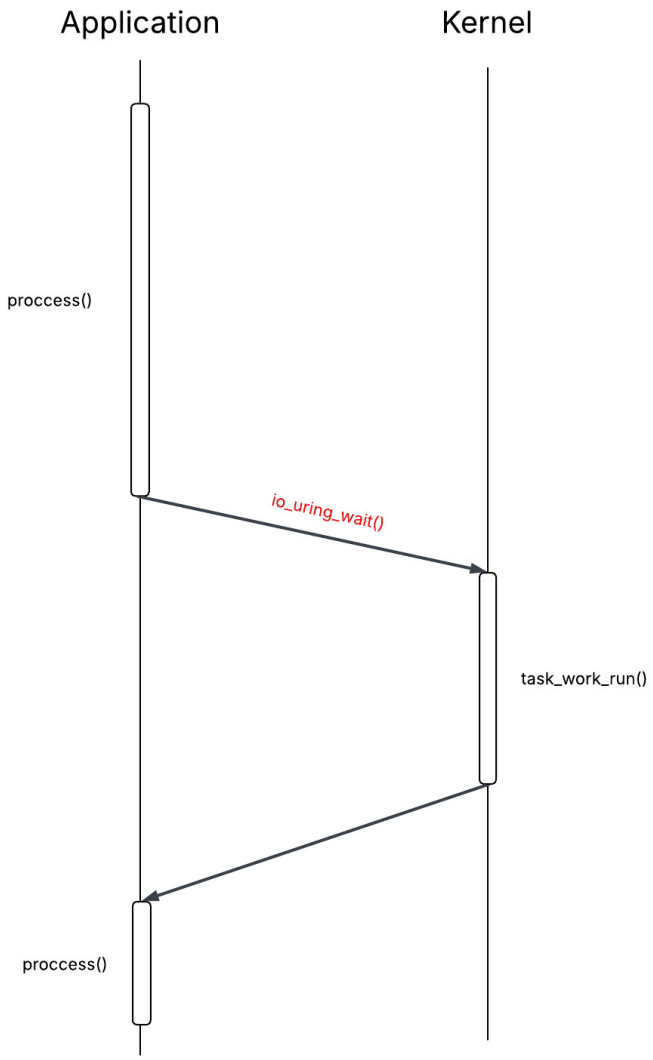


Figure 2.5: Deferred behavior: Completions are strictly batched and processed only when the application explicitly waits, maximizing efficiency.

2.4.1.4 Implications

Configurable completion processing directly affects interrupt traffic, batching efficiency, CPU overhead, and latency. The default mode minimizes delay but increases interrupts and transitions. Cooperative and deferred modes reduce these costs by aligning completion handling with natural or explicit wait points, giving the application fine-grained control to balance throughput and latency [10].

2.4.2 Multishot Operations and Buffer Management

2.4.2.1 The Standard Receive Model

Fundamental to the design of `io_uring` is the lifecycle of an I/O request. In the standard operational model, a strict one-to-one mapping exists between submission and completion: a single SQE yields exactly one CQE. Once the kernel posts the completion event, the request is considered consumed. For recurring operations, such as receiving a continuous stream of data, this single-shot model requires the application to manage a repetitive cycle of submission and completion. This standard approach, illustrated in Figure 2.6, functions as follows:

1. The application allocates a dedicated buffer for the connection (Line 1).
2. Inside the loop, it prepares a standard receive request, explicitly attaching the specific buffer (Line 3).
3. It submits the request and waits for the kernel to fill that specific buffer (Lines 4–5).
4. Once data arrives, the application processes the buffer and must immediately prepare a new SQE to receive the next packet (Lines 3–7).

This architecture introduces two sources of inefficiency. First, the requirement to issue a new system call and submission for every packet creates CPU overhead. Second, memory usage scales poorly; the application requires a dedicated buffer for every open socket to ensure data can be received. To handle thousands of concurrent connections, the application requires thousands of allocated buffers, even if most connections are idle.

```
1: buffer ← malloc(SIZE)
2: while true do
3:   sqe ← io_uring_get_sqe(R)
4:   io_uring_prep_recv(sqe, S, buffer)
5:   io_uring_submit_and_wait(R, 1)
6:   cqe ← io_uring_peek_cqe(R)
7:   if cqe.res > 0 then
8:     process_data(buffer, cqe.res)
9:   end if
10:  io_uring_cqe_seen(R, cqe)
11: end while
```

Figure 2.6: The standard single-shot `recv` loop

`io_uring` addresses these inefficiencies by combining **Multishot Requests** with **Kernel-Managed Buffer Pools**.

2.4.2.2 Kernel-Managed Buffer Pools

To resolve the memory scalability issue, `io_uring` introduces a mechanism, termed **provided buffers**, that allows the application to register a shared pool of memory with the kernel. Instead of attaching a unique buffer to every individual request, the application organizes buffers into a group and registers them with the `io_uring` instance.

When an I/O operation arrives, the kernel dynamically selects a free buffer from this shared pool, writes the data, and returns the specific Buffer ID in the completion event. This delegates memory management to the kernel, ensuring that buffers are consumed only when data actually arrives and eliminating the need to lock memory for idle connections.

2.4.2.3 Multishot Receive

The Multishot Receive operation (`io_uring_prep_recv_multishot`) leverages this buffer pool. A single request is submitted to SQ, which remains active in the kernel, monitoring the socket. Whenever data arrives, the kernel selects a buffer, fills it, and posts a CQE. It is important to note that the kernel *may* terminate the request if an error occurs on the FD or if CQ lacks sufficient space to post completions. Figure 2.7 demonstrates multishot pattern:

1. The application registers a buffer pool with the ring (Line 2).
2. It submits a single multishot receive request linked to that pool (Lines 5–6).

3. In the event loop, it retrieves the Buffer ID from the CQE to locate the data (Line 11).

4. After processing, the buffer is explicitly added back to the pool for reuse (Line 13).

5. Finally, the application checks the request status (Lines 15–17). If the `IORING_CQE_F_MORE` flag is absent (value is 0), the kernel has terminated the multishot request. In this case, the application must explicitly *re-arm* the socket by submitting a new multishot SQE to continue receiving data.

```
1: io_uring_queue_init(depth, R)
2: setup_and_register_buffer_pool(R, bgroup_id)
3: S ← connected_socket()
4: sqe ← io_uring_get_sqe(R)
5: io_uring_prep_recv_multishot(sqe, S, bgroup_id)
6: io_uring_sqe_set_flags(sqe, IOSQE_BUFFER_SELECT)
7: while true do
8:   io_uring_submit_and_wait(R, 1)
9:   while cqe ← io_uring_peek_batch_cqe(R) do
10:    if cqe.res > 0 then
11:      bid ← get_buffer_id(cqe)
12:      process_data(buffer_pool[bid], cqe.res)
13:      recycle_buffer(R, group_id, bid)
14:    end if
15:    if (cqe.flags & IORING_CQE_F_MORE) == 0 then
16:      rearm_recv_multishot(S, group_id)
17:    end if
18:    io_uring_cqe_seen(R, cqe)
19:  end while
20: end while
```

Figure 2.7: A multishot `recv` loop using kernel-managed buffers

2.4.3 Eventfd Integration

`eventfd` provides a simple kernel mechanism for event notification, originally designed for lightweight user–kernel signaling [6]. It exposes a file descriptor that maintains a 64-bit counter. Writing to the descriptor increments the counter, while reading returns the current value and resets the counter if the read is blocking. These capabilities makes it useful as a generic event source that can be polled alongside other descriptors.

In `io_uring`, an application can register an `eventfd` so that completions are signaled through it [11]. The application blocks on the `eventfd` descriptor, each time a new completion is posted to CQ, the kernel writes to the registered `eventfd`, waking up the application.

This feature allows integration of `io_uring` with existing event-driven loops that already rely on readiness interfaces like `epoll`.

Registering the descriptor is done via `io_uring_register_eventfd`, which associates an `eventfd` with the `io_uring` instance. Once active, the application can use normal I/O multiplexing calls (e.g., `epoll_wait`) to wait on the `eventfd`, and then harvest completions from the CQ in batches. This approach can be used to integrate `io_uring` with existing readiness-based architectures while adding the benefits of completion-based I/O.

2.5 Adoption of `io_uring`

The `io_uring` interface has been introduced primarily to solve long-standing inefficiencies with asynchronous storage I/O on Linux, serving as a high-performance replacement for the AIO interface [26]. Consequently, its initial adoption and research context have been overwhelmingly focused on disk-bound applications. High-performance database systems and key-value stores have been among the first major adopters. For example, RocksDB has been shown to benefit from `io_uring` integration [39], and PostgreSQL officially integrated `io_uring` as a new asynchronous I/O method in its PostgreSQL 18 release [36].

This storage-centric view is reinforced by benchmarks comparing `io_uring` against other storage Application Programming Interfaces (APIs), such as `aio` and the user-space Storage Performance Development Kit (SPDK). These analyses primarily measure performance in terms of disk Input/Output Operations Per Second (IOPS) and latency, showing that `io_uring` not only surpasses `aio` but also shows potential to be competitive with kernel-bypass solutions like SPDK for high-throughput storage operations [30].

This storage-oriented trend extends even to in-memory applications; for instance, recent work on Redis utilizes `io_uring` to optimize disk persistence [42]. This focus has created a significant gap in the literature regarding the feasibility of `io_uring` for high-throughput networking. Addressing this gap is challenging because most high-performance servers, such as Memcached, rely heavily on the `epoll` readiness model. Despite the introduction of networking features in `io_uring`, migrating these established architectures requires a fundamental shift to a completion-based paradigm.

Chapter 3

Design of the io_uring Convergence Layer

3.1 Motivation

Chapter 2 discusses traditional readiness-based mechanisms such as `epoll`, which notify applications when a file descriptor is ready. However, even with these notifications, the application must still perform the actual I/O operation synchronously. By contrast, `io_uring` introduces a submission/completion model in which operations are issued asynchronously, and results are retrieved from the completion queue upon their finalization. This model diverges significantly from the synchronous completion semantics around which many production systems are designed.

Applications such as Memcached [15] exemplify this challenge. As described in Chapter 2, Memcached relies on `epoll` to manage tens of thousands of concurrent connections. Its internal abstractions assume synchronous completions and readiness-driven control flow [16]. Directly porting such a system to use `io_uring` would require invasive modifications, including reworking large portions of the application logic, buffer management, redesigning abstractions, and accommodating a fundamentally different event model.

To bridge this gap, the `io_uring` Convergence Layer (`UringCL`) has been developed as part of this work [24]. The library provides an interface layer that reconciles the asynchronous paradigm of `io_uring` with the synchronous completion semantics expected by legacy code. Rather than forcing a complete redesign, `UringCL` exposes interfaces that resemble existing synchronous abstractions, while internally mapping them to `io_uring` operations. This design minimizes intrusive code changes, reduces integration complexity, and makes incremental adoption feasible in large applications.

The following sections describe the most important features of the Convergence Layer and how they address the challenges discussed in Chapter 2.

3.2 Receive Operations and Buffer Management

`UringCL` is explicitly designed to optimize network-heavy applications, such as Memcached, by leveraging `io_uring`'s most efficient features: multishot operations and kernel-managed buffers. As discussed in Section 2.4.2, these features allow a single submission to handle successive data arrivals without constant re-arming, while delegating memory management to the kernel.

Although the standard single-shot `recv` operation initially appears simpler, the approach necessitates complex user-space buffer management and frequent system calls. Consequently, the multishot mechanism aligns more closely with the structure that the application expects. Standard asynchronous `recv` requires the caller to provide a buffer at the moment of submission. However, in a legacy synchronous application, the buffer is only provided later, at the moment of the `read` call. Bridging this gap using standard `recv` requires the library to implement a complex user-space intermediate buffer pool to hold data until the application requests the data. This buffer management duplicates the complexity of memory management in user space.

In contrast, the multishot variant integrates natively with the kernel's buffer management. The kernel holds the data in an internal pool until `UringCL` retrieves the data. This strategy delegates the complexity of memory lifecycle management to the operating system, simplifies the implementation, and ensures that the application remains non-blocking and fully asynchronous.

3.2.1 Automated Initialization

To utilize these features, the application must perform significant boilerplate setup. Establishing the submission and completion rings requires explicit queue sizing, memory mapping, and bookkeeping. Furthermore, using kernel-managed buffers requires creating buffer groups, registering them with the kernel, and ensuring their visibility to the ring. Performing these steps manually is error-prone and demands deep familiarity with the `io_uring` API. These steps are straightforward for kernel experts but represent a barrier for developers unfamiliar with `io_uring`'s low-level interface. Automating them reduces integration time and minimizes the risk of subtle configuration bugs that degrade performance or correctness.

`UringCL` abstracts this complexity through a consolidated initialization routine. Internally, it creates the ring, and automatically allocates and registers the necessary buffer pools. From the application's perspective, the entire asynchronous infrastructure, including the setup of kernel-managed buffers, is established via a single function invocation, preventing the need for direct interaction with low-level kernel interfaces.

3.2.2 Bridging the Semantic Gap

The core challenge in `UringCL` design is reconciling the semantic difference between `epoll` and `io_uring`. In an `epoll`-based model, a notification indicates that a socket is *ready* to be read. In the `io_uring` completion model, a notification indicates that the kernel has *already* read data into a buffer.

`UringCL` hides this distinction to maintain transparency. It effectively "pre-reads" data into the kernel-managed buffers described in Section 2.4.2. When the legacy application issues a standard synchronous `read` call, the library does not invoke a new system call. Instead, it locates the data already captured by the multishot operation and copies it from the internal kernel-managed buffer into the application's supplied buffer.

From the application's perspective, this behavior is indistinguishable from a normal blocking read: the call returns with the requested data. Internally, however, the data path avoids the overhead of repeated system-calls. This design preserves the simplicity of the synchronous API while delivering the efficiency of asynchronous background processing.

3.2.3 Copy-Free Reads via Pointer Swapping

While the default behavior copies data from the internal buffer pool into the application's buffer, this memory movement creates significant overhead in high-throughput workloads. To address this cost, `UringCL` provides a copy-free variant of the read interface that employs pointer swapping. In this mode, the convergence layer transfers ownership of the kernel-provided buffer directly to the application, eliminating the memory copy.

However, this technique lacks universal applicability. A critical issue arises when an application, such as Memcached, does not fully consume the data in the current buffer before issuing the next read request. In such scenarios, the application expects the unconsumed data to remain accessible at the original memory location. If `UringCL` swaps the buffer pointer during a subsequent read, the reference to the previous buffer is lost. Consequently, this disruption leads to the potential loss of client data, making the copy-free approach unsafe for applications that rely on incremental processing or partial buffer consumption.

3.3 Flexible Event Retrieval

Section 2.4.3 describes how `io_uring` can integrate with readiness-based event loops through the `eventfd` mechanism. When an `eventfd` is registered with a ring, the kernel signals it whenever new completions are posted to the completion queue. This allows applications to monitor the `eventfd` descriptor using interfaces such as `epoll_wait`, thereby combining `io_uring`-driven completions with existing readiness-based loops.

UringCL makes this integration straightforward by exposing an interface that allows applications to choose how completions are harvested. Depending on the integration scenario, an application can either:

- rely on an eventfd associated with the ring and incorporate it into an existing event loop, or
- use the native io_uring polling interface to wait for completions directly from the ring.

This flexibility is particularly valuable for large, established systems such as Memcached, which already maintain a event loop. Instead of discarding or rewriting this loop, the application can register UringCL's eventfd with its existing epoll interest set. From the application's perspective, the eventfd behaves like any other file descriptor: when signaled, it indicates that new completions are available in the io_uring completion queue. The application then reaps those completions in batches, just as it would handle readiness notifications from sockets.

At the same time, UringCL also provides a straightforward path to migrate away from eventfd entirely. If an application later decides to adopt a pure completion-driven design, it can be moved to a native io_uring event loop without significant structural changes. This capability makes it possible to start with minimal integration effort and evolve toward a deeper adoption of io_uring as needed.

3.4 Integration of the Convergence Layer

The integration of UringCL utilizes a callback-driven architecture to establish a unified interface for both eventfd and Native modes. The application registers a specific handler function, on_data_ready, with UringCL. Regardless of the underlying notification mechanism, the convergence layer invokes this callback when data arrives. This design mirrors the event-loop pattern employed by many high-performance applications, such as Memcached, which already utilize callback mechanisms with epoll. Consequently, this structural similarity significantly simplifies the integration process, as the application can map existing logic directly to the UringCL's interface without major architectural changes.

3.4.1 Integration via eventfd

In this configuration, UringCL uses an eventfd mechanism to signal the existing epoll loop of the application. This mode allows legacy applications to adopt the layer convergence interface without restructuring their core event loop. Figure 3.1 illustrates this process:

- **Lines 1–5 (Callback Definition):** The `on_data_ready` function is defined as the central handler. It calls `uring_read` to retrieve data (Line 3) and subsequently passes it to the application logic (Line 4).
- **Lines 7–8 (Initialization):** `UringCL` is initialized in **eventfd** mode (`eventfd=True`). The associated `eventfd` is explicitly added to the existing `epoll` instance so the kernel notifies the application of`UringCL` activity.
- **Lines 10–12 (Registration):** The application iterates through client sockets and registers the `on_data_ready` callback for each file descriptor, which maps the socket to the handler within `UringCL`.
- **Lines 14–22 (Event Loop):** The main loop uses `epoll_wait` to monitor events.
- **Lines 17–19 (Dispatching):** When an event triggers on the `uring.eventfd`, the application calls `uring_process_events`. This function drains the CQ and invokes the `on_data_ready` callback for all relevant sockets.
- **Lines 20–21 (Legacy Handling):** If the event belongs to a standard file descriptor, the application proceeds with its legacy event handling path.

3.4.2 Integration via Native `io_uring`

In native mode, the application loop is simplified significantly. The `uring_poll` function acts as a wrapper around `io_uring_submit_and_wait`, which indicates to `io_uring` that the application has finished the previous run and is waiting for new events. The primary difference is that the loop is handled directly through `io_uring`, which eliminates the need for the `eventfd` mechanism and the `epoll` system call. Figure 3.2 shows the native integration flow:

- **Lines 6–10 (Setup):** The initialization mirrors the `eventfd` mode, except that `uring_init` is called with `eventfd=False`, and no `epoll` registration is required.
- **Lines 12–15 (The Loop):** Instead of calling `epoll_wait`, the application enters a loop driven by `uring_poll` (Line 15).
- **Line 14 (Blocking and Dispatch):** The `uring_poll` function blocks until I/O completions occur. Internally, it automatically calls `uring_process_events`, which triggers the registered `on_data_ready` callbacks. This architecture provides a direct path from kernel completion to application logic.

```
function on_data_ready(fd)
    {Callback invoked when UringCL has buffered data}
    data ← uring_read(fd) {Non-blocking retrieval}
    process_application_logic(data)
end function

uring ← uring_init(eventfd=True)
epoll_ctl(epoll_fd, ADD, uring.eventfd, EPOLLIN)

for all fd in client_sockets do
  {Register FD with the callback handler}
  uring_add_event(uring, fd, RECV_MULTISHOT, on_data_ready)
end for

while true do
  events ← epoll_wait(epoll_fd)
  for all ev in events do
    if ev.data.fd == uring.eventfd then
      {Drains CQ and fires 'on_data_ready' callbacks}
      uring_process_events(uring)
    else
      handle_legacy_event(ev)
    end if
  end for
end while
```

Figure 3.1: Hybrid integration: `epoll` triggers the callback dispatcher of `UringCL`

```
function on_data_ready(fd)
    data ← uring_read(fd)
    process_application_logic(data)
end function

uring ← uring_init(eventfd=False)

for all fd in client_sockets do
  uring_add_event(uring, fd, RECV_MULTISHOT, on_data_ready)
end for

while true do
  {Waits for IO and internally triggers callbacks}
  uring_poll(uring, timeout)
end while
```

Figure 3.2: Native integration: `uring_poll` drives the loop and callbacks

1: **function** `uring_read`(fd, $buffer$, len)
2: {Retrieve pre-buffered data from user space list}
3: $internal_buf \leftarrow$ `get_next_buffer`(fd)
4: `copy`($buffer$, $internal_buf$, len)
5: `recycle_buffer`($internal_buf$)
6: **return** *bytes_copied*
7: **end function**

Figure 3.3: The read implementation of `UringCL`: Non-blocking user-space copy

3.4.3 The Data Retrieval Interface

The `uring_read` function, which is called within the callbacks, creates the illusion of a synchronous read operation. However, no system call is performed at this stage. The function merely dequeues data from user-space buffers populated by `uring_process_events` moments earlier. This design avoids the overhead of context switches during the read operation. Figure 3.3 details this logic:

- **Line 3 (Buffer Retrieval):** The function retrieves the next available buffer associated with the file descriptor from an internal list.
- **Line 4 (Data Copy):** The data is copied from the internal buffer to the buffer provided by the application.
- **Line 5 (Recycling):** The internal buffer is recycled for future use.
- **Line 6 (Return):** The function returns the total number of bytes copied to the application.

3.5 Integration with Applications

This section details the integration of `UringCL` into two distinct software environments to validate its design and performance. The selection includes a benchmark tool and a production-grade application. These case studies represent opposite ends of the complexity spectrum, which allows for an assessment of the library's adaptability to different event loop architectures and workload requirements.

3.5.1 Neper

The first application integrated with `UringCL` is Neper [8], a Linux network performance tool designed for benchmarking communication latency and throughput. Neper is a simple

user-space application, which makes it ideal for measuring the raw performance impact of `io_uring` without interference from complex computation or application-level logic.

A modified version of Neper [23] incorporates `UringCL` directly into the I/O path. The simplicity of Neper allows the impact of `UringCL`'s design to be isolated and observed clearly, since the application performs no additional computation beyond measuring, sending, and receiving data. Furthermore, Neper relies on `epoll` as the main event notification mechanism. This reliance makes Neper an effective environment to validate how well the layer bridges readiness-based and completion-based event models, allowing for a direct comparison of performance characteristics across both `eventfd` and Native integration modes.

3.5.2 Memcached

To evaluate `UringCL` in a realistic, production-level environment, Memcached serves as the primary test subject. This version supports both integration modes of `UringCL`: the `eventfd`-based approach and the native `io_uring` event loop. The purpose of this experiment is to demonstrate that asynchronous I/O allows for integration into a mature, event-driven server architecture with minimal disruption.

3.5.2.1 The Role of Libevent

Memcached relies on `libevent` to manage its central event loop. `libevent` acts as an abstraction layer for asynchronous event notification, providing a uniform API to execute callback functions when specific events occur on file descriptors or when timeouts expire. Internally, `libevent` supports multiple backend mechanisms, including `kqueue`, `select`, and, most importantly for Linux, `epoll` [41].

By default, Memcached utilizes `libevent` to interface with the Linux `epoll` subsystem, enabling the efficient handling of thousands of concurrent connections. A significant advantage of `UringCL`'s design is that it requires no modifications to the `libevent` source code. Since it operates cleanly alongside the existing event loop structure, the third-party dependency remains untouched, preserving the portability and maintainability of the broader system.

3.5.2.2 Architectural Adaptation

Memcached is structured around this central event loop and a state machine that manages the lifecycle of client connections. In the library integration, the state machine requires no modification. The primary changes occur within the event loop logic where Memcached interacts with the network. Since `libevent` provides a readiness-driven model, certain parts of Memcached are inherently coupled to its abstractions. These specific components are identified and replaced with thin compatibility functions that match the layer abstractions.

This integration preserves Memcached's original logic for handling client requests and responses, while transparently substituting `io_uring`-based asynchronous operations beneath the existing control flow. As a result, Memcached retains the same architecture yet benefits from the efficiency of completion-driven I/O. This demonstrates that a complex and widely deployed application can adopt `io_uring` with minimal structural changes, enabling incremental migration without a full redesign [22].

Chapter 4

Evaluation

The focus of this chapter is twofold. First, as discussed in Section 2.5, the performance of `io_uring` is well-understood in storage contexts, but a research gap exists for high-throughput network applications. `UringCL` provides a vehicle to address this gap. Therefore, the first objective is to use the convergence layer to evaluate the performance of `io_uring` itself in a demanding, network-bound environment.

The second objective is to evaluate the practical value of the `UringCL`. The layer is designed to simplify the integration of `io_uring` into existing applications, particularly those built on readiness-based models like `epoll`. If `UringCL` can achieve a performance improvement, while simplifying `io_uring` integration, it can be easily used in many applications that currently use readiness-based models.

4.1 Experimental Setup

4.1.1 Hardware

The evaluation is performed on a dual-socket Non-Uniform Memory Access (NUMA) system equipped with two Intel Xeon E5-2670 processors. Each processor provides eight physical cores, resulting in a total of 16 cores throughout the system. The machine contains 128 GiB of memory, evenly distributed between the two NUMA nodes, with 64 GiB allocated to each socket. Network connectivity is provided by a Mellanox ConnectX-4 Lx Ethernet adapter operating at 25 Gbps. Hyperthreading is disabled by binding threads exclusively to the first hardware thread of each physical core. The cache hierarchy per core consists of a 32 KB Level-1 data cache, a 32 KB Level-1 instruction cache, a 256 KB Level-2 cache, and a shared 20 MB Last-Level Cache (LLC). Seven similar client machines are used to generate workload for the server during evaluation.

4.1.2 Software Environment

All servers used in the experiments run Ubuntu 24.04. Because `io_uring` is under active development and new capabilities are frequently introduced, the evaluation is conducted using a recent version of the Linux kernel (v6.15.0), compiled with the generic configuration [13]. This version provides access to the most recent features required by `UringCL`. For the same reason, `liburing` version 2.11 is employed, ensuring that the user-space interface matches the latest kernel functionality.

For each test configuration, interrupt handling is controlled via static Interrupt Request (IRQ) routing. The Network Interface Card (NIC) is configured with distinct hardware queues, where interrupts for each queue is routed to a dedicated CPU core. This mechanism establishes a strict one-to-one mapping between hardware interrupts and processing cores. In addition, all experiments are restricted to a single NUMA node.

4.1.3 Benchmark

4.1.3.1 Neper

As discussed in Section 3.5.1, Neper is an effective tool for benchmarking `UringCL`. It includes a built-in load generator that can operate in client mode, allowing direct measurement of throughput and latency under various workloads. For evaluation, the unmodified (vanilla) version of Neper is compared against multiple variants of Neper, which integrate `io_uring` through `UringCL`.

Neper provides several types of benchmarks that emulate different communication patterns. Two representative benchmarks are selected to capture distinct workload characteristics and to evaluate how `io_uring` performs under each:

- **tcp_rr:** Generates a request/response workload over TCP, representative of short, latency-sensitive transactional exchanges typical of interactive network services.
- **tcp_stream:** Generates a bulk data transfer workload over TCP, similar to high-throughput applications such as file transfer.

These two benchmarks highlight different aspects of system behavior. The `tcp_rr` test stresses the responsiveness and system-call efficiency of the I/O path, whereas `tcp_stream` emphasizes sustained throughput and data transfer performance. Together, they provide a comprehensive view of how `io_uring` behaves under both latency-sensitive and throughput-intensive conditions.

4.1.3.2 Memcached

Memcached is a widely used in-memory key–value store that has become a standard benchmark for evaluating network stacks and systems software. Its lightweight design exposes the performance characteristics of the underlying runtime, while maintaining realism through practical workloads. For this evaluation, Memcached version 1.6.23 is used, and results are compared against the modified version that integrates `io_uring` with the help of `UringCL`.

Workloads are generated using Mutilate [9], a well-established Memcached benchmarking tool. Each of the seven client machines runs eight threads (one per core), and operating with a pipeline depth of one. The experiments simulate the synthetic "ETC" workload described in prior studies [29, 46], using a dataset containing one million keys.

4.2 Data Collection

The output of multiple tools is used to analyze the behavior and performance of each test case. The primary tools used in this evaluation are Mutilate [9], Neper [8], and Perf [34].

4.2.1 Mutilate Metrics

After each test run, Mutilate produces two key metrics:

- **QPS:** the number of client requests processed by the Memcached server per second, serving as a measure of overall throughput.
- **Latency:** the time required to fetch data, measured from the moment a `GET` request is issued until the response is received by the client. Mutilate reports several latency statistics, including minimum, maximum, average, and the 95th and 99th percentiles. The analysis focuses on the 95th and 99th percentiles to evaluate tail latency and response-time stability.

Peak throughput is measured using a *closed-loop* configuration. In this setup, each Mutilate agent sends a pipelined request up to a specified depth, then waits for the response before issuing a new one. The pipeline depth is set to one, meaning that each agent sends a single request and blocks until a response arrives. With a sufficient number of concurrent connections, the server reaches full capacity, resulting in 100% CPU utilization. Under these conditions, the measured QPS represents the maximum sustainable request-handling rate of the server in this configuration [46].

Latency is evaluated under an *open-loop* configuration, where the request rate is explicitly controlled by the load generator. This approach enables precise monitoring of response times by maintaining a fixed rate below maximum capacity. For latency analysis, the 95th and 99th percentile values are emphasized, as they represent thresholds below which 95% and 99% of observed requests complete [46].

4.2.2 System-Level Metrics

`Perf` is a performance analysis framework that interfaces with the CPU's Performance Monitoring Unit (PMU). It provides access to hardware and software performance counters with low overhead. Crucially, `Perf` easily distinguishes between execution contexts, allowing measurements to be separated into user-space and kernel-space components. In this study, these split metrics are utilized to collect the number of CPU cycles and instructions executed, forming the basis for computing derived quantities such as Instructions per Cycle (IPC) and Instructions per Query (IPQ).

Because modern CPUs support only a limited number of hardware counters, monitoring all events simultaneously without multiplexing is impossible. Counter multiplexing introduces sampling overhead and reduces measurement precision. To mitigate this issue, related events are grouped and collected in separate runs. Following this approach, the evaluation proceeds in two stages: the first measures overall CPU cycles and instruction counts, while the second focuses on workload throughput and latency [46].

4.3 Performance Metrics

Performance differences across test cases are evaluated by examining the total number of executed instructions and CPU cycles. Analyzing variations in instruction count and cycle usage makes it possible to identify the specific factors that contribute to efficiency gains.

As highlighted in previous work [29, 47], IPC is a useful measure of how effectively the CPU execution pipeline processes a workload. In this context, IPC correlates directly with overall throughput, often reported as QPS. The relationship is expressed by the following equations:

$$IPC = \frac{\text{instructions}}{\text{cycles}} \qquad IPQ = \frac{\text{instructions}}{\text{queries}} \tag{4.1}$$

$$CPS = \frac{\text{cycles}}{\text{unit time (1s)}} \qquad CPQ = \frac{\text{cycles}}{\text{queries}} \tag{4.2}$$

$$QPS = \frac{CPS \times IPC}{IPQ} \quad \Leftrightarrow \quad \frac{QPS}{CPS} = \frac{IPC}{IPQ} \tag{4.3}$$

These formulas indicate that higher QPS is achieved when the IPQ decreases or when IPC increases. A lower IPQ shows that fewer instructions are required to process each query, reflecting more efficient execution. Similarly, a higher IPC suggests fewer pipeline stalls, enabling the CPU to retire more instructions per cycle. Cycles per Query (CPQ) serves as the aggregate measure of these efficiencies, representing the total processing cost

(in cycles) for a single query. Consequently, reducing CPQ, either by executing fewer instructions or by executing them faster, directly drives throughput gains.

Cycles per Second (CPS) also serves as a supporting measure, as a lower CPS indicates that the CPU consumed fewer cycles to complete the same instruction workload [46]. However, the interpretation of CPS depends on the workload model. In closed-loop experiments, where the CPU is consistently saturated, low CPS and IPQ together reflect a more efficient execution path, which results in improved IPC and QPS. In open-loop experiments, on the contrary, a lower CPS may simply reflect reduced CPU utilization rather than improved efficiency, and thus is less meaningful for performance evaluation [46].

4.4 Neper Evaluation

As a micro-benchmark, the Neper evaluation is intentionally restricted to a single server core and thread. This approach reduces experimental complexity, placing a direct focus on the core features of `UringCL` and `io_uring`. By isolating the benchmark in this manner, the analysis aims to identify fundamental performance patterns that reveal how `io_uring` behaves with different kinds of workloads.

4.4.1 Evaluation Methodology

The evaluation for both Neper benchmarks, `tcp_stream` and `tcp_rr`, is structured around three primary comparisons.

- **Task-Work Scheduling:** The impact of the `io_uring` task-work scheduling modes (described in Section 2.4.1) is analyzed. This includes comparisons between the default behavior, cooperative scheduling (`co-op`), and deferred scheduling (`defer`).
- **Copy-Free Reads:** The performance of the copy-free (`no-copy`) read optimization (Section 3.2.3) is compared against the default memory copy (`copy`) behavior.
- **Baseline Comparison:** Finally, after identifying the optimal settings, the best-performing `io_uring` configurations (both `native` and `eventfd`) are benchmarked against the traditional `epoll` implementation to determine the overall performance impact.

4.4.2 TCP Stream (tcp_stream)

The `tcp_stream` benchmark evaluates performance using a bulk data transfer workload, where throughput (measured in Gigabits Per Second (Gbps)) serves as the primary metric. Unlike request-response patterns, this workload involves a continuous flow of data, which

is highly conducive to batching. This characteristic allows `io_uring` to aggregate many operations, thereby highlighting the performance impact of efficient batch processing. To ensure statistical reliability, ten iterations were performed for each connection count. Since the standard deviation for throughput consistently remained below 5%, the plotted data points represent the average of these trials.

4.4.2.1 Experimental Configuration: Disabling Offloads

Preliminary experiments indicate that the 10 Gbps network link of the server acts as a bottleneck, which prevents meaningful analysis of CPU and software efficiency. This limitation stems from hardware-level network accelerations:

- **TCP Segmentation Offload (TSO):** Allows the kernel to send a single, large data buffer (up to 64 KB) to the NIC, which then segments it into standard Maximum Transmission Unit (MTU) packets [48].
- **Generic Receive Offload (GRO):** Performs the reverse operation in software. The kernel reassembles a stream of incoming packets into a single large buffer before passing the data up the network stack, thereby reducing the per-packet processing overhead [48].

While TSO and GRO benefit production systems, they mask the performance of the underlying software stack by offloading the CPU-intensive work of packet segmentation and reassembly to the hardware. To create a CPU-bound scenario in which the benefits of software-level batching are measurable, both TSO and GRO are disabled for all `tcp_stream` experiments. These restrictions force the CPU to process every packet individually, making system efficiency directly dependent on the overhead of the I/O mechanism.

4.4.2.2 Evaluation of Task-Work Scheduling Modes

This experiment examines how different task-work scheduling configurations affect batching efficiency in a high-volume packet environment. The evaluation compares the default behavior against cooperative and deferred task-work modes. Crucially, all configurations in this section utilize the copy-free mechanism. Since integrating the copy-free variant of receive operations into Neper is straightforward and eliminates memory copy overhead, this approach is selected to focus the analysis strictly on scheduling efficiency.

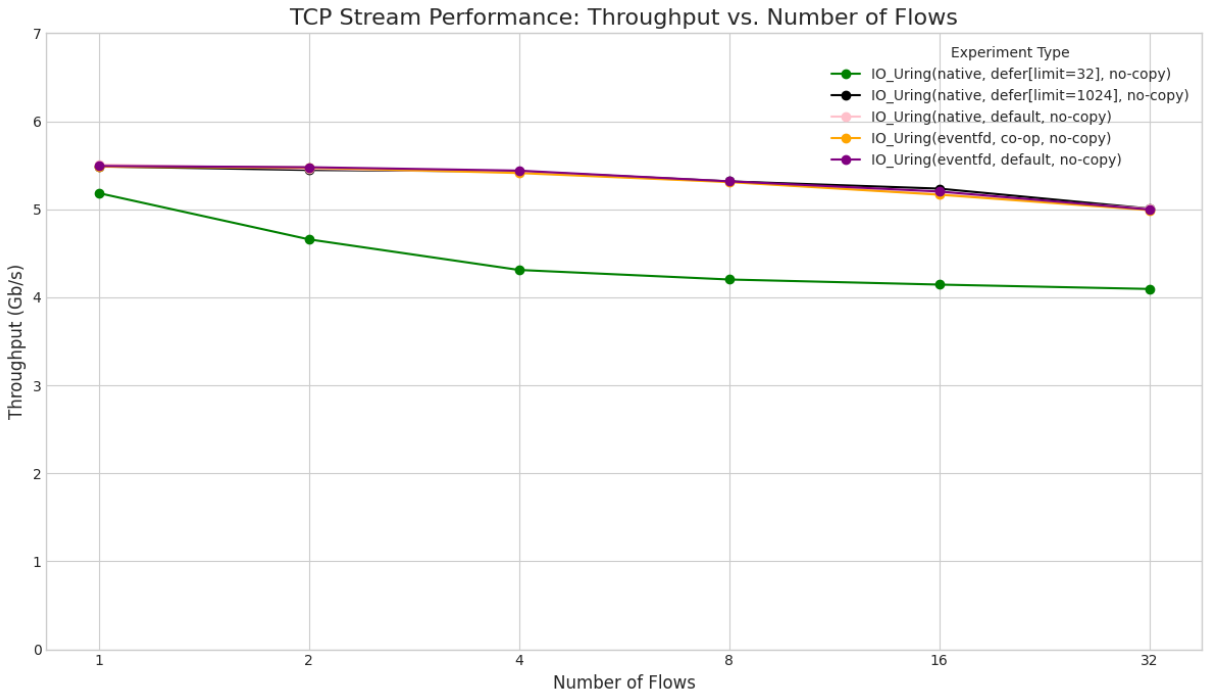


Figure 4.1: Task-Work Modes: Throughput comparison for Neper

Figure 4.1 presents the throughput results. A distinct anomaly is observed in the `(native, defer[limit=32])` configuration, which exhibits a dramatic drop in performance compared to all other modes. However, when this limit is raised to 1024, the deferred mode's performance recovers completely, aligning with the high throughput of the default and cooperative modes.

This performance drop is explained by the kernel's internal multishot **fairness** mechanism. By default, the kernel limits the number of completions processed in a single cycle to 32 (`MULTISHOT_MAX_RETRY`) per FD to prevent I/O operations from starving the application logic. While this design ensures responsiveness for mixed workloads, it becomes a bottleneck for bulk streaming.

In a pure throughput scenario like `tcp_stream`, the application's goal is to ingest data as fast as possible. Limiting processing to 32 packets forces the kernel to interrupt the task-work loop and return control to the application frequently. This premature interruption destroys the efficiency of the batch, increasing the overhead of context switches. Increasing the limit via changing the constant value in the kernel to 1024 allows the kernel to process significantly larger batches in a single pass, amortizing the overhead across more packets and restoring the expected throughput.

4.4.2.3 Impact of Copy-Free Reads

This experiment evaluates the Copy-Free read optimization described in Section 3.2.3. It compares the standard data copy mode against the `no-copy` (pointer swapping) variant for both `eventfd` and native `io_uring` integrations.

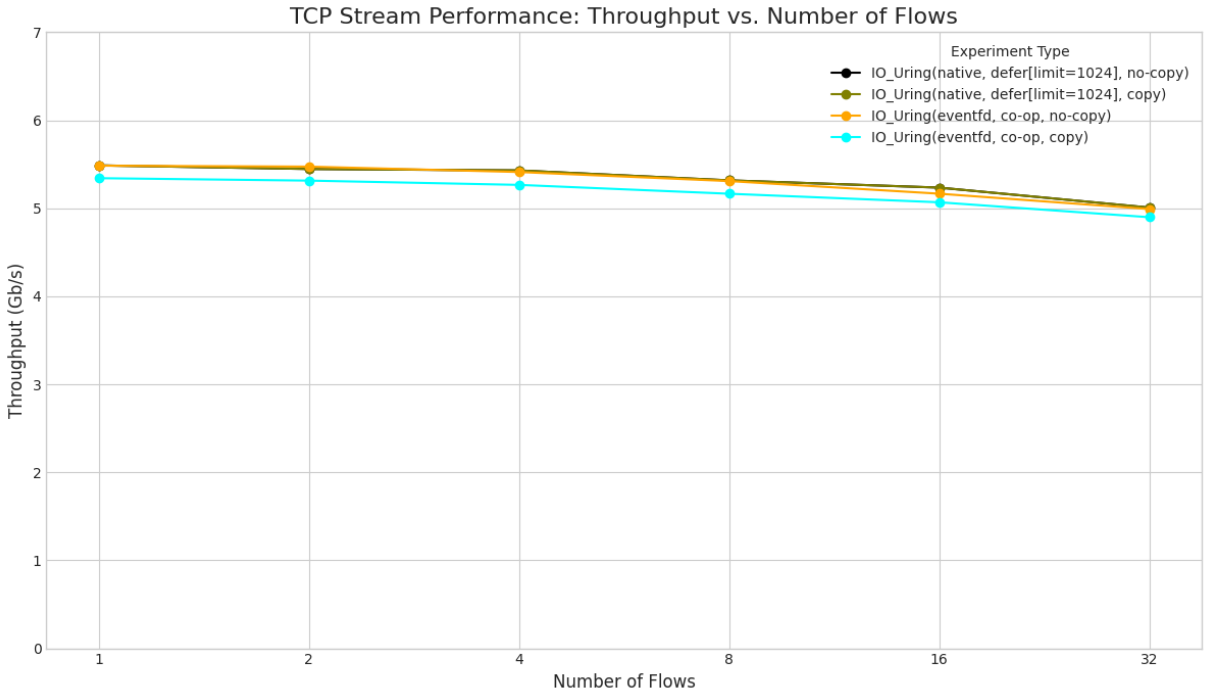


Figure 4.2: Copy vs. No-Copy: Throughput impact of pointer swapping optimization.

Figure 4.2 demonstrates that while the `no-copy` configurations achieve a measurable throughput advantage, the performance gap is relatively narrow. For the native integration, the throughput of the copy-based implementation is nearly identical to that of the optimization.

This minimal difference indicates that the overhead of a single memory copy, while small, is not a dominant bottleneck on this hardware. This finding is significant for the broader applicability of `UringCL`, particularly for the Memcached evaluation. As noted in Section 3.2.3, Memcached's internal buffer management is incompatible with the pointer-swapping technique used by the `no-copy` mode. The fact that the standard copy mechanism performs competitively implies that `UringCL` can employ the copy-based approach for complex applications without incurring a substantial performance penalty.

4.4.2.4 Baseline Performance vs. Epoll

The final comparison benchmarks the optimized `io_uring` configurations (using the best settings identified above) against the standard `epoll` baseline.

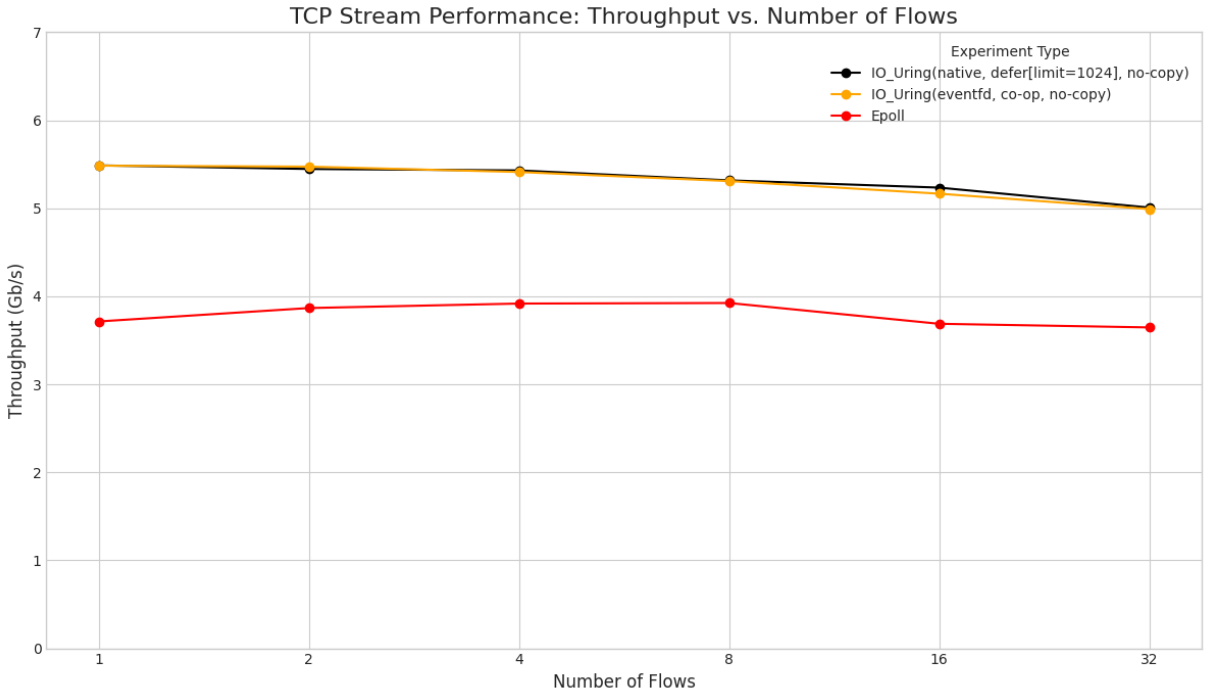


Figure 4.3: Baseline Comparison: `io_uring` vs. `epoll` throughput.

As shown in Figure 4.3, the performance gap is substantial. Both native and `eventfd` `io_uring` modes achieve a throughput of approximately 5.0–5.5 Gbps. In contrast, the `epoll` implementation saturates at roughly 3.7–3.9 Gbps, representing a performance difference of approximately 40%.

This improvement is a direct result of the superior batching capabilities of `io_uring`. The `epoll` model, which essentially notifies readiness for each operation, incurs significant system-call and context-switch overhead per packet. `io_uring` minimizes this overhead by submitting and completing large batches of operations in a single cycle, allowing the CPU to spend more time processing network traffic and less time on kernel transitions.

placeins

4.4.3 TCP Request/Response (tcp_rr)

The `tcp_rr` benchmark evaluates the performance of `io_uring` under a latency-sensitive, request-response workload. This test measures both maximum throughput (transactions

per second) and 99th-percentile (P99) tail-latency. Unlike `tcp_stream`, clients in this workload wait for a server-response before sending the next request. Consequently, there is limited potential for batching within a single-connection, resulting in smaller performance differences between I/O mechanisms. Consistent with the previous benchmark, each data-point represents the average of ten iterations, as the standard-deviation for throughput remained below 5% across all trials. To characterize the latency-distribution, the median of the 99th-percentile (P99) is presented.

4.4.3.1 Evaluation of Task-Work Scheduling Modes

This experiment evaluates the `io_uring` task-work scheduling modes (Section 2.4.1), comparing the default behavior against cooperative task-running (`co-op`) and deferred task-running (`defer`).

Notably, the `limit=1024` tuning, which was critical for `tcp_stream`, is not applied here. The `tcp_rr` benchmark simulates a request-response workload, meaning each flow has only one request in flight at a time. This pattern prevents large-scale batching within a single-connection. Therefore, the analysis maintains the default kernel-fairness limit (32), as any potential for batching primarily occurs across different client-flows.

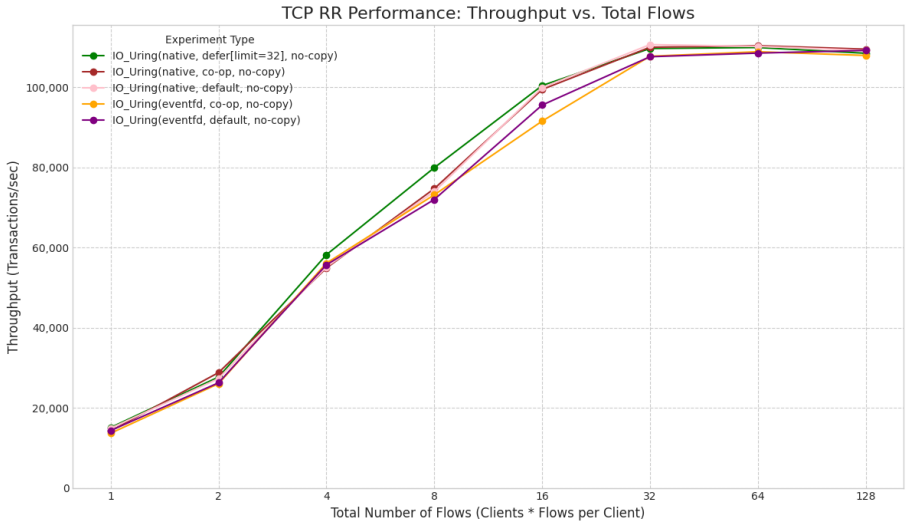
As shown in Figure 4.4a, all configurations demonstrate similar throughput-scaling. However, the `(native, defer)` and `(native, co-op)` variants achieve the highest peak-throughput, slightly edging out the default native implementation. This performance suggests that these optimized scheduling modes provide a marginal improvement in throughput under saturation.

The latency-plot (Figure 4.4b) reveals distinct behavior. While all configurations maintain low latency at moderate loads, the `(native, defer)` implementation shows a small improvement in tail-latency at the highest throughput-levels, sustaining low latency longer than the other configurations. The `co-op` variants also perform well, aligning closely with the `defer` mode. This trend indicates that these kernel-side optimizations, which reduce forced interruptions, are effective at managing high request-rates and reducing P99 tail-latency.

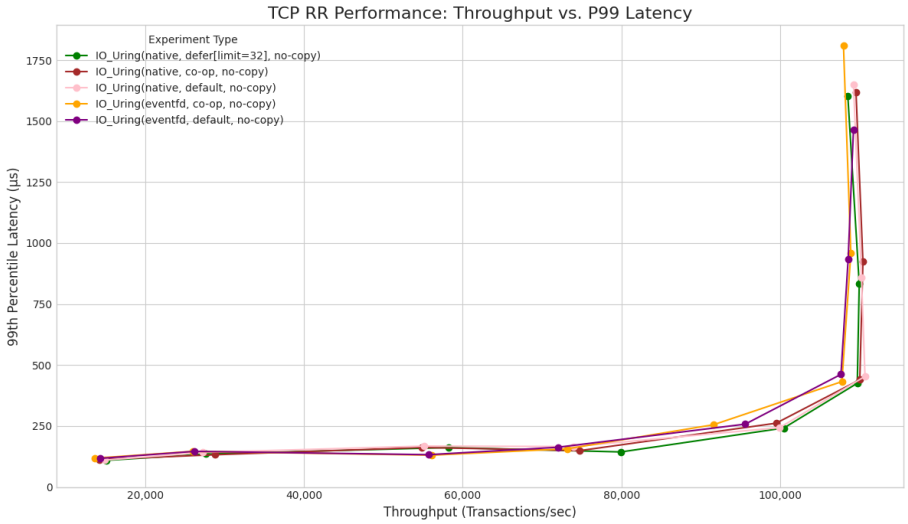
4.4.3.2 Impact of Copy-Free Reads

As described in Section 3.2.3, `UringCL` supports a copy-free read variant that avoids an extra memory-copy by swapping buffer-pointers. This experiment compares the performance of `eventfd` and native modes, each with the standard data-copy enabled (`copy`) and with the copy-free mechanism enabled (`no-copy`).

Figure 4.5a illustrates the throughput-results. The lines for the `no-copy` and `copy` variants overlap almost perfectly across the entire range of flow-counts. Unlike the bulk-transfer workload, where a small separation was visible, the request-response workload



(a) Throughput vs. Total Flows

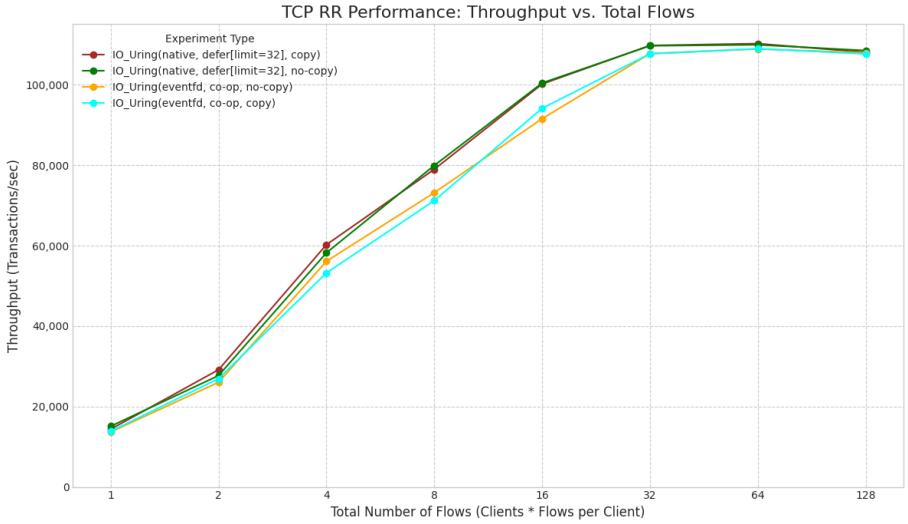


(b) P99 Latency vs. Throughput

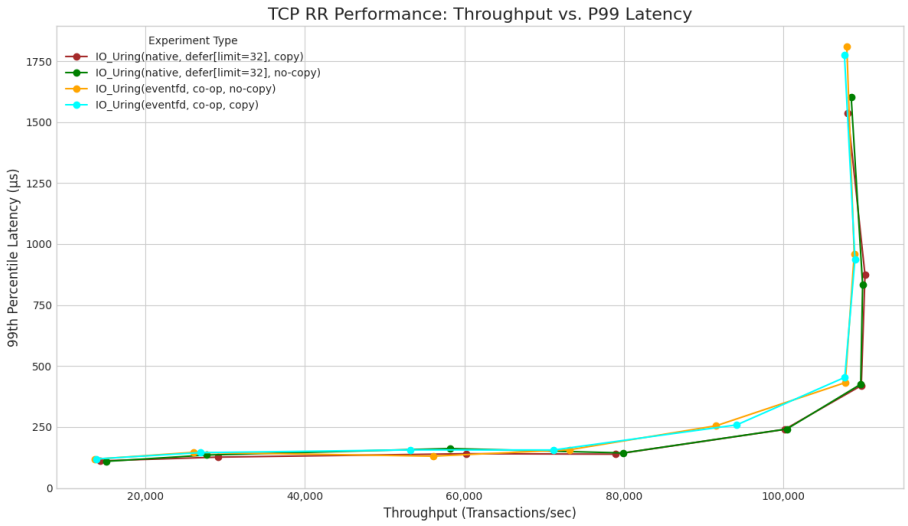
Figure 4.4: Performance of `tcp_rr` benchmark (Task-Work Scheduling Modes).

shows virtually no throughput-difference between the two approaches. The latency-plot in Figure 4.5b confirms this behavior. Both configurations exhibit almost identical latency-characteristics, maintaining low response-times until they reach the same saturation-point. The `no-copy` optimization does not delay the tail-latency spikes.

These results reinforce the findings from the `tcp_stream` evaluation in the previous section. The overhead of a memory-copy is negligible compared to the costs of network-transmission and event-processing. Consequently, this optimization yields no significant performance benefit in this scenario.

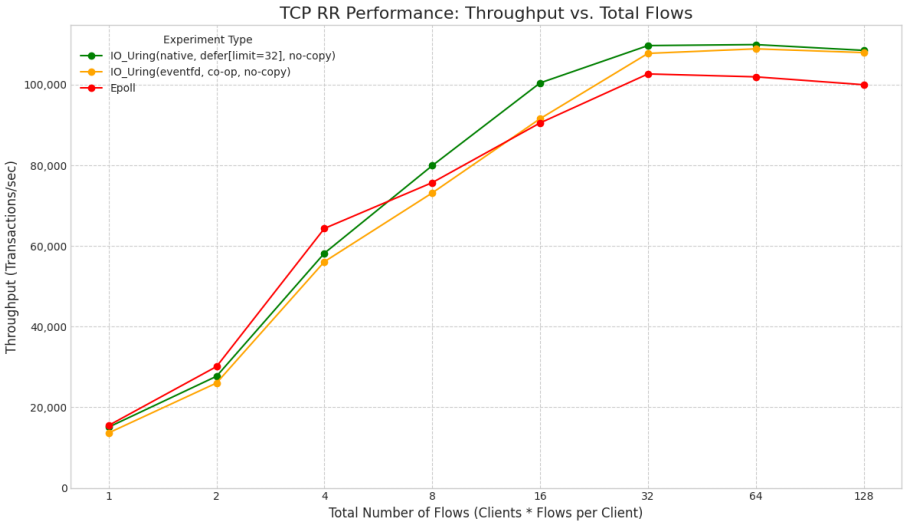


(a) Throughput vs. Total Flows

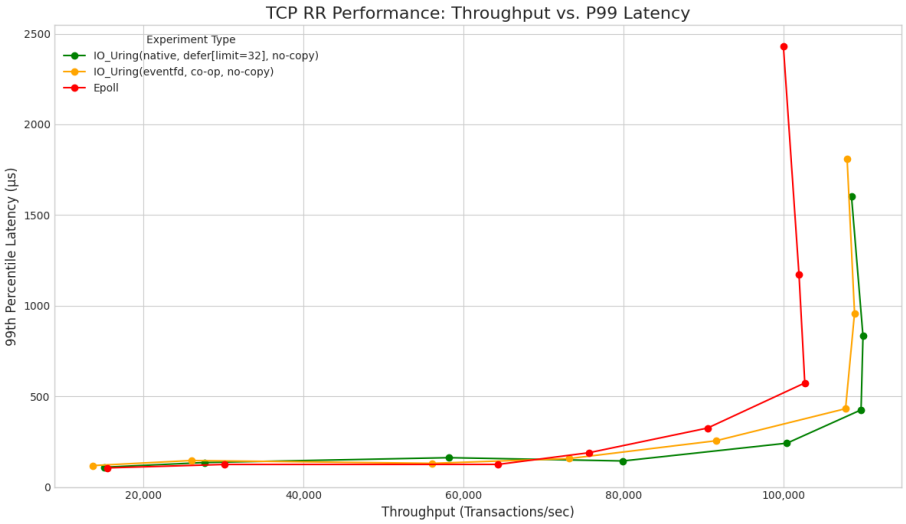


(b) P99 Latency vs. Throughput

Figure 4.5: Performance of `tcp_rr` benchmark (Copy vs. No-Copy).



(a) Throughput vs. Total Flows



(b) P99 Latency vs. Throughput

Figure 4.6: Performance of `tcp_rr` benchmark (Epoll baseline).

4.4.3.3 Baseline Performance vs. Epoll

The final experiment compares the performance of the optimized `io_uring` library-modes against the standard `epoll`-based implementation.

Figure 4.6a shows the throughput as a function of the total number of client-flows. The performance of all implementations scales as the number of flows increases, with the performance-gap widening at higher flow-counts. This difference suggests that even with limited batching-potential, `io_uring`'s ability to aggregate operations provides a performance-advantage. The native `io_uring` implementation achieves the highest peak-throughput, closely followed by the `eventfd` integration. Both `io_uring` variants demonstrate a measurable improvement over the baseline `epoll` implementation, which saturates at a lower throughput-level. A notable characteristic of the `tcp_rr` workload is that the number of packets per connection is significantly lower compared to `tcp_stream`. This reduction decreases batching capacity, which represents the primary strength of `io_uring`. Consequently, when the flow-count is low, the overhead of the convergence layer and `io_uring` becomes more prominent; both `io_uring` versions perform worse than the baseline in scenarios with four flows or fewer.

Figure 4.6b presents the 99th-percentile latency as a function of throughput. At lower throughput-levels, all three implementations maintain similarly low latency. However, as the load increases, the `epoll` implementation exhibits a sharp increase in tail-latency at a much lower throughput-threshold than the `io_uring` variants. The native `io_uring` mode sustains the highest throughput before latency increases, demonstrating more stable performance under high load. The `eventfd` mode also provides a clear improvement over `epoll`, sustaining higher throughput before latency increases.

4.5 Evaluation of Memcached

This section reports the results of performance experiments on Memcached, a workload that closely resembles the `tcp_rr` benchmark due to its request-response nature. The objective is to establish a performance baseline between the traditional `epoll` readiness model and the `io_uring` completion model.

It is essential to note that all Memcached configurations in this evaluation utilize the copy-based interface of `UringCL`. As discussed in Section 3.2.3, implementing copy-free reads requires extensive modifications to Memcached's internal memory management logic. Furthermore, prior microbenchmarks indicate that the copy-based approach incurs minimal overhead compared to the copy-free variant, yielding comparable performance without requiring major architectural changes.

To ensure statistical reliability in this real-world scenario, 20 iterations were performed for each connection count. Throughput results are reported as the average of these trials, as the standard deviation remained consistently below 5%. For tail latency measurements,

the median value across iterations is reported to provide a robust representation of the distribution and minimize the impact of transient outliers. The three evaluated configurations are:

- **Vanilla Memcached (`epoll`):** This configuration represents the standard Memcached implementation, which relies on `epoll` for readiness event handling. Each socket is monitored for I/O readiness, and upon notification, the application performs synchronous `read` and `write` operations. This model serves as the baseline for evaluating the impact of integrating `io_uring`.
- **`io_uring` with Eventfd Integration:** This variant incorporates `io_uring` in combination with the `eventfd` mechanism. This configuration uses the optimized settings (e.g., cooperative task-work) identified in the Neper evaluation. In this setup, `io_uring` completions are signaled through an `eventfd`, allowing the existing event loop to monitor both traditional file descriptors and completion notifications.
- **`io_uring` with Native Loop:** This configuration uses `UringCL`'s native event loop mode, which operates entirely within the `io_uring` framework. This variant also uses the optimal settings (e.g., deferred task-work) from the Neper evaluation. Instead of relying on `epoll` or `eventfd`, the application polls the completion queue directly, eliminating intermediate signaling and system-call overhead.

4.5.1 Closed-loop Throughput Evaluation

Figure 4.7 shows the QPS as the number of client connections increases. The experiment begins with a single connection per thread and gradually raises the load to push the system toward its maximum throughput. As the number of concurrent connections grows, the throughput initially increases sharply and eventually plateaus as the system approaches saturation. A clear improvement is observable in both `io_uring` configurations once the system reaches its operational limit.

This behavior highlights the impact of batching. At low connection counts, the overhead of any I/O model is minimal. However, as the number of connections increases, the `epoll` model must individually manage more file descriptors. In contrast, `io_uring` can batch operations from many connections, reducing the per-operation overhead. When the system is saturated, `io_uring` extends the throughput ceiling, allowing the server to process a higher number of queries per second compared to the baseline vanilla Memcached. This improvement indicates that the asynchronous completion model of `io_uring` can better utilize available CPU resources under high load.

Another important observation from Figure 4.7 is that the performance difference between the two `io_uring` variants is marginal. This outcome suggests that even partial integration of `io_uring` through `eventfd` achieves nearly the same throughput as the fully

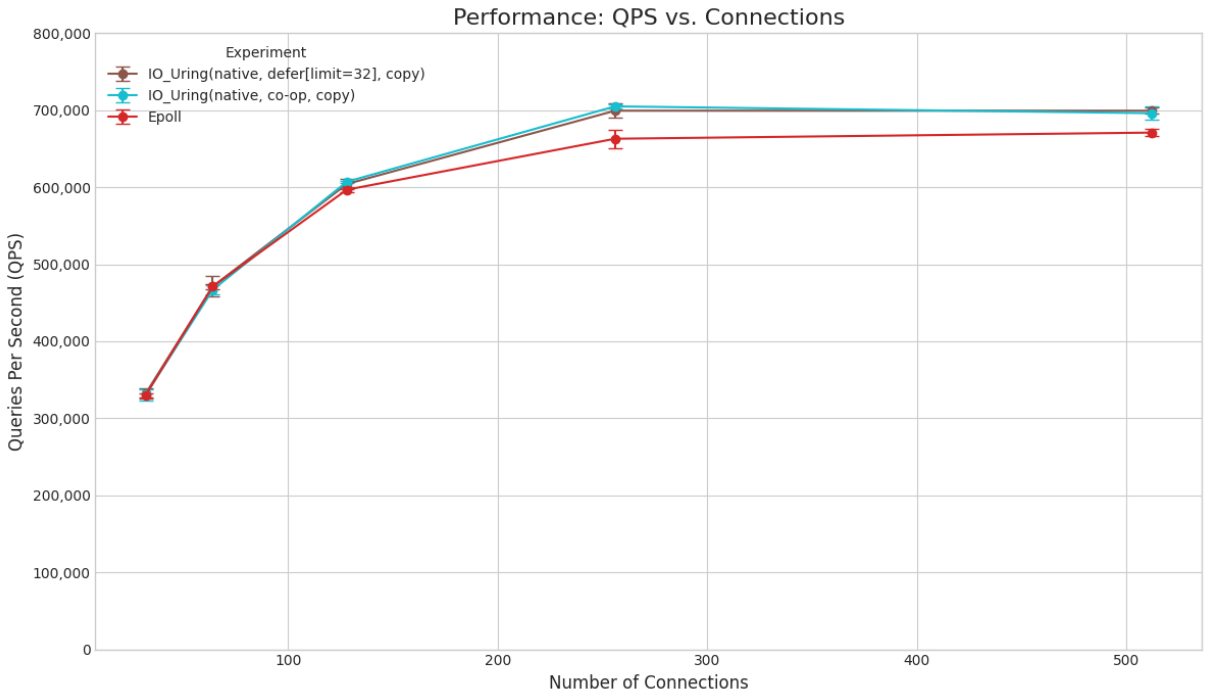


Figure 4.7: QPS as a function of the number of client connections for Vanilla Memcached and `io_uring` variants.

native implementation. Because `eventfd` integration requires minimal modification to existing event-driven systems, it offers a practical path for improving performance with less architectural change.

Table 4.1 summarizes the performance metrics for the configuration that achieves the highest throughput (Figure 4.7). The IPC and CPQ values reinforce the observation from Figure 4.7. Kernel-side CPQ values are significantly higher than user-space CPQ values across all configurations, indicating that most of the processing time occurs within the kernel. However, both `io_uring` configurations demonstrate a slightly higher kernel-side IPC (**K-IPC**) and a lower kernel-side CPQ (**K-CPQ**) compared to the vanilla version. This improvement implies that the kernel executes instructions more efficiently, completing a greater number of useful operations per CPU cycle and requiring fewer cycles per query.

This efficiency can be attributed to the fundamental difference in event handling between the `epoll` model and `io_uring`. In the `epoll` model, a readiness notification triggers a transition from user space to kernel space and back again for each receive operation. In contrast, the `io_uring` model eliminates this additional transition by delivering completions directly to the application context as soon as the data become available. Hence, the system performs fewer transitions and maintains a more streamlined execution path.

Testcase	QPS	U-IPC	K-IPC	U-CPQ	K-CPQ
`Epoll`	671,925	0.6	0.9	4,675	28,802
`IO_Uring(Eventfd)`	703,959	0.62	0.94	4,809	27,074
`IO_Uring(Native)`	702,188	0.62	0.94	4,853	27,126

Table 4.1: `perf` Metrics for Maximum Throughput Measured in User and Kernel Space

4.5.2 Open-Loop Latency Evaluation

In addition to throughput, latency, particularly tail latency, is a critical metric for evaluating the performance of high-load server applications. To measure this, the experiments are executed in an *open-loop* mode using the load generation setup described in Section 4.1.3. In this mode, the request rate is controlled externally, enabling precise observation of how response times vary as throughput increases.

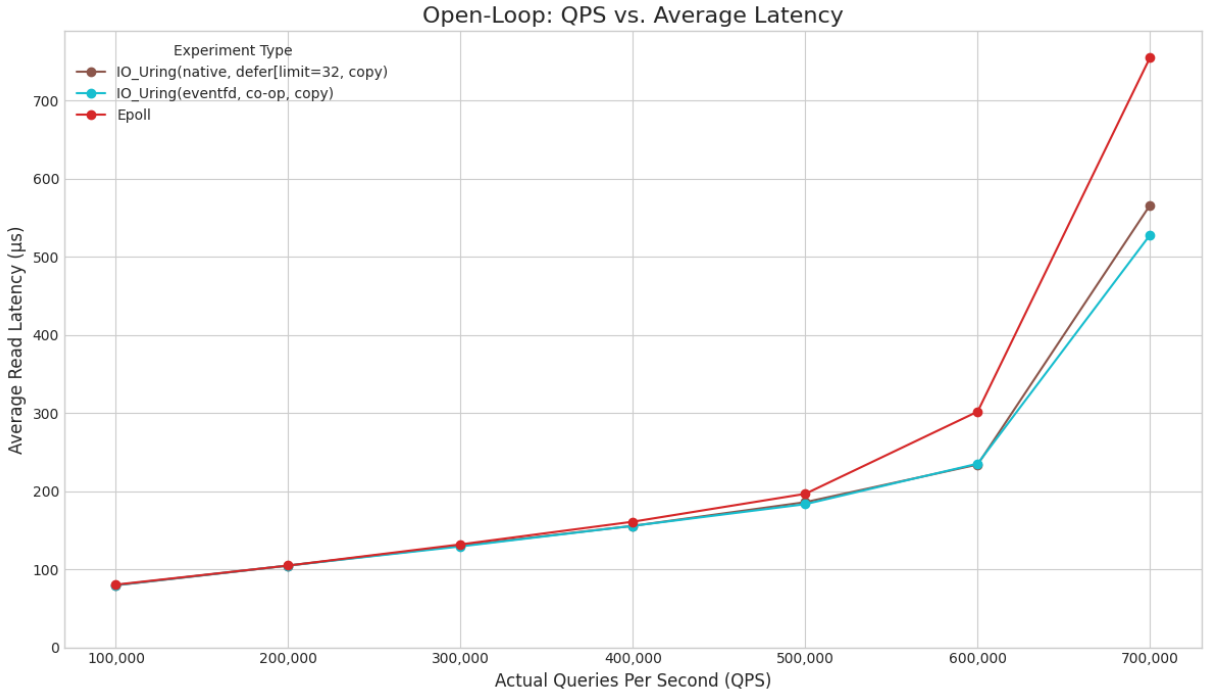


Figure 4.8: Average Read Latency vs. Throughput (Memcached 8 cores).

Figure 4.8 presents the average read latency as a function of the achieved QPS. At lower request rates, all three configurations, Vanilla Memcached, `io_uring` with Eventfd, and `io_uring` with the native loop, exhibit similar latency behavior. However, as the offered load approaches the system's saturation point, the difference becomes evident.

Both `io_uring` configurations sustain significantly lower average latencies than the baseline implementation, particularly beyond 600,000 queries per second.

The advantages of `io_uring` are even more pronounced in tail latency metrics. Figure 4.9 and Figure 4.10 illustrate the 99th and 95th percentile trends, respectively. These results indicate that `io_uring` not only improves average response times but also stabilizes latency under heavy load.

This stability is a direct result of efficient batching, which allows the kernel to process many completions in aggregate, maintaining high throughput without the latency spikes seen in the readiness model. The marginal difference between the `eventfd` and native versions further demonstrates that partial adoption of `io_uring` is sufficient to achieve most of the latency benefits.

These results reinforce the observations made in the closed-loop experiments. In the `io_uring` configurations, the kernel processes incoming data as soon as it becomes available, eliminating the additional transition to kernel space required in traditional synchronous system-call mechanisms. When the application is awakened in an `io_uring` setup, the data are already prepared and can be retrieved directly without a separate `recv` system call. This design significantly reduces context switching overhead and improves responsiveness by keeping the data path within the kernel until completion. Consequently, end-to-end latency is reduced, and the system achieves a more efficient interaction between the user and kernel space.

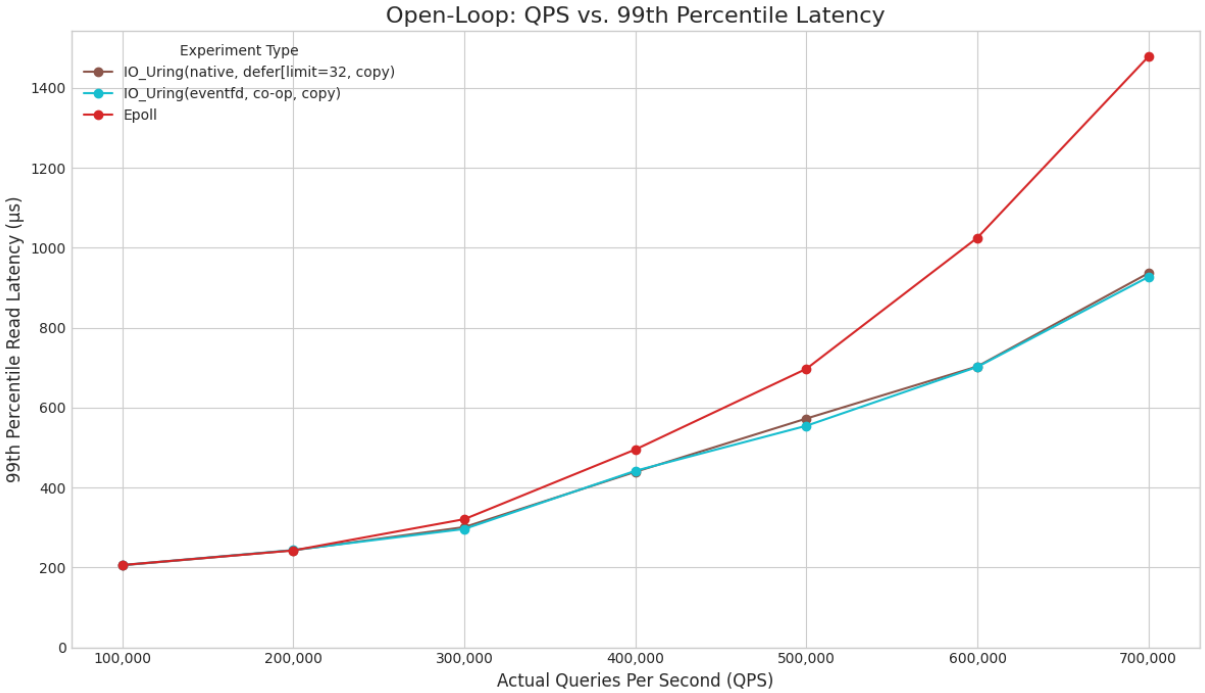


Figure 4.9: 99th Percentile Latency vs. Throughput (Memcached 8 cores).

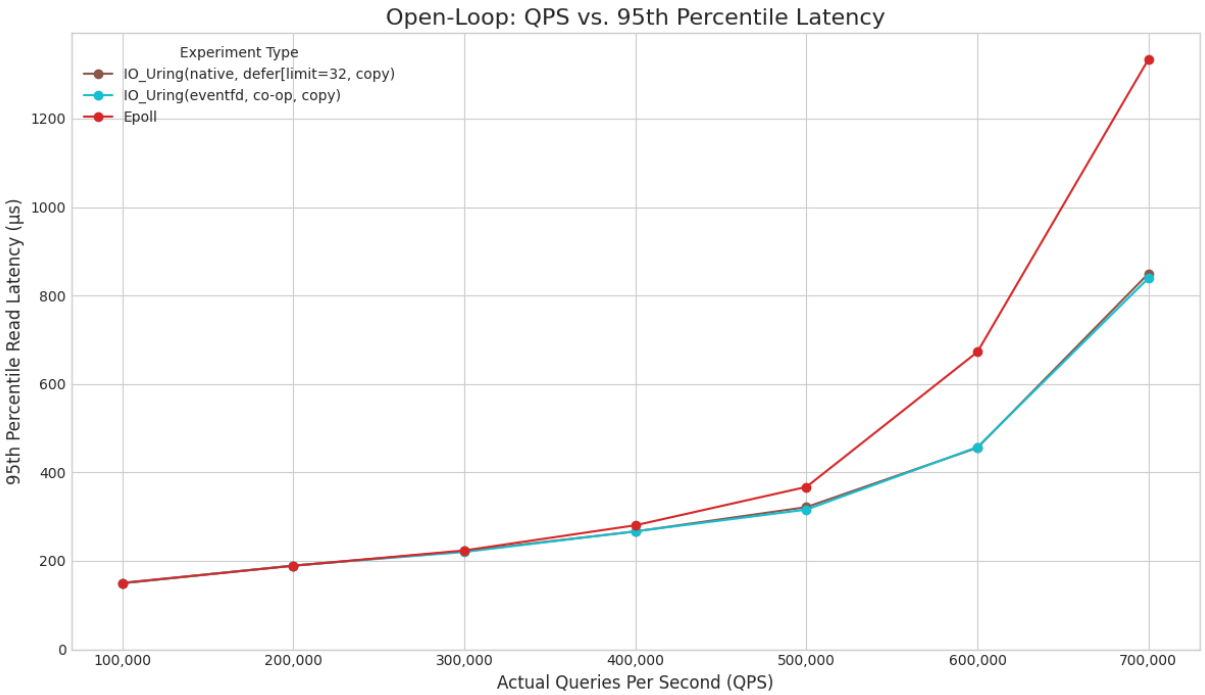


Figure 4.10: 95th Percentile Latency vs. Throughput (Memcached 8 cores).

Chapter 5

Conclusion

High-performance network servers traditionally depend on readiness I/O interfaces, such as `epoll`, to manage thousands of concurrent connections. While this model is well-established, it still requires synchronous system calls for data transfer, which introduces user-kernel transition overhead. The `io_uring` interface, introduced in recent Linux kernels, offers a more efficient completion model that minimizes system calls and enables true asynchronous I/O [26]. However, its adoption in network applications remains limited, primarily because migrating from a synchronous, readiness-driven architecture to an asynchronous, completion-driven one often requires a fundamental and invasive redesign.

This thesis addresses this critical integration gap. The primary contribution of this work is the `io_uring` Convergence Layer (`UringCL`), a lightweight library that reconciles the asynchronous paradigm of `io_uring` with the synchronous abstractions expected by legacy, readiness-driven software [24]. `UringCL` is designed to make the adoption of `io_uring` more accessible and to facilitate incremental migration, allowing complex applications to benefit from `io_uring` without a complete architectural overhaul.

To evaluate this approach, `UringCL` has been integrated into both a simple network benchmark (Neper) [8] and a complex, production-level application (Memcached) [22]. The evaluation has two objectives: to measure the performance of `io_uring` in network-bound environments and to validate the practical value of `UringCL` as an integration tool.

The experimental results confirm that `io_uring` provides significant performance advantages over `epoll` in networking workloads. In bulk transfer tests (`tcp_stream`), `io_uring`'s superior batching capabilities deliver up to 40% higher throughput. In request-response evaluations, which are representative of applications like Memcached, the `io_uring` integrations consistently achieve higher peak throughput (QPS) and, critically, demonstrate significantly lower tail latency (P95 and P99) under higher load.

Furthermore, the evaluation validates the effectiveness of `UringCL`. A key finding is that the partial-integration mode, which uses `eventfd` to connect `io_uring` completions to an existing `epoll` loop, delivers performance nearly identical to that of a fully native

`io_uring` event loop. This result is significant, as it proves that applications can gain the vast majority of `io_uring`'s benefits with minimal, non-invasive code changes.

This work demonstrates that completion-based I/O can be adopted seamlessly in complex systems. `UringCL` provides a practical and effective path for evolving existing high-performance network applications, enabling them to leverage modern Linux I/O capabilities without sacrificing their established architectural foundations.

5.1 Future Work

This work successfully demonstrates a minimally invasive method for integrating `io_uring`'s completion-based receive model into readiness-driven applications. This provides a foundation for future investigation, which can be extended in two primary directions.

5.1.1 Send-Side Integration

This thesis focuses exclusively on optimizing the I/O **receive path**. The **send path** (i.e., `write` and `send` operations) is not currently integrated into `UringCL`. This is a deliberate choice, as the send path presents a different integration challenge.

Unlike the receive path, where a pool of provided buffers can be pre-allocated and managed by the kernel, application responses are often created dynamically. In a server like Memcached, a response buffer is populated only after a request is parsed and processed. Submitting these dynamic buffers to `io_uring` asynchronously would require a more invasive approach. The application would need to manage the buffer's lifetime, ensuring it remains valid until the kernel signals the send completion. This introduces complexity that conflicts with `UringCL`'s goal of minimal integration effort. Future work could explore strategies for an asynchronous send-side. This might involve:

- Developing new buffer management abstractions that simplify lifetime tracking for dynamic response data.
- Identifying application-level design patterns that are more naturally compatible with fully asynchronous send operations.

5.1.2 Leveraging Evolving `io_uring` Features

The `io_uring` interface is continuously evolving, with new features added in nearly every kernel release [10]. This rapid development opens several opportunities to extend and improve `UringCL` capabilities.

- **True Zero-Copy Receive:** `UringCL` currently implements a "copy-free" read optimization using pointer swapping. While this is effective, future work could integrate *true* zero-copy mechanisms. As new kernel features emerge, such as the zero-copy receive data plane [49], it may become possible for the kernel to place incoming data directly into a user-space buffer that can be used by the application without any copies *or* pointer swaps, further enhancing throughput.

- **New API (NAPI) Integration:** Recent kernel versions, including the version used in this evaluation, offer direct integration between `io_uring` and the NAPI framework [12]. This integration allows an `io_uring` ring to poll the network device's NAPI context directly [10]. An investigation into adding NAPI-driven polling to `UringCL`'s native event loop would be a valuable next step.

References

[1] AIO — PostgreSQL Wiki — Wiki.PostgreSQL.org. https://wiki.postgresql.org/wiki/AIO. [Accessed 23-10-2025].

[2] Aio(7) — Linux Manual Page — Man7.org. https://man7.org/linux/man-pages/man7/aio.7.html. [Accessed 23-10-2025].

[3] Connection Processing Methods — Nginx.org. https://nginx.org/en/docs/events.html. [Accessed 09-09-2025].

[4] Creating an Event_Base — Libevent.org. https://libevent.org/libevent-book/Ref2_eventbase.html. [Accessed 09-09-2025].

[5] Epoll(7) — Linux Manual Page — Man7.org. https://man7.org/linux/man-pages/man7/epoll.7.html. [Accessed 08-09-2025].

[6] Eventfd(2) — Linux Manual Page — Man7.org. https://man7.org/linux/man-pages/man2/eventfd.2.html. [Accessed 15-09-2025].

[7] GitHub — axboe/liburing: Library Providing Helpers for the Linux Kernel io_uring Support — GitHub.com. https://github.com/axboe/liburing. [Accessed 10-09-2025].

[8] GitHub — google/neper: Neper Is a Linux Networking Performance Tool — GitHub.com. https://github.com/google/neper. [Accessed 20-10-2025].

[9] GitHub — leverich/mutilate: Mutilate — High-Performance Memcached Load Generator — GitHub.com. https://github.com/leverich/mutilate. [Accessed 26-09-2025].

[10] io_uring and Networking in 2023 — GitHub.com. https://github.com/axboe/liburing/wiki/io_uring-and-networking-in-2023. [Accessed 10-09-2025].

[11] io_uring_Register_Eventfd(3) — Linux Manual Page — Man7.org. https://man7.org/linux/man-pages/man3/io_uring_register_eventfd.3.html. [Accessed 15-09-2025].

[12] io_uring_Register_Napi(3) — Linux Manual Page — Man7.org. https://man7.org/linux/man-pages/man3/io_uring_register_napi.3.html. [Accessed 12-11-2025].

[13] KernelBuild — Linux Kernel Newbies — KernelNewbies.org. https://kernelnewbies.org/KernelBuild. [Accessed 26-09-2025].

[14] Man7.org. https://man7.org/training/download/ipc_epoll_slides-mkerrisk-man7.org.pdf. [Accessed 08-09-2025].

[15] Memcached — A Distributed Memory Object Caching System — Memcached.org. https://memcached.org/. [Accessed 16-09-2025].

[16] Performance and Efficiency — Docs.Memcached.org. https://docs.memcached.org/serverguide/performance/. [Accessed 09-09-2025].

[17] Poll(2) — Linux Manual Page — Man7.org. https://man7.org/linux/man-pages/man2/poll.2.html. [Accessed 08-09-2025].

[18] Read(2) — Linux Manual Page — Man7.org. https://man7.org/linux/man-pages/man2/read.2.html. [Accessed 09-09-2025].

[19] Select(2) — Linux Manual Page — Man7.org. https://man7.org/linux/man-pages/man2/select.2.html. [Accessed 08-09-2025].

[20] Socket(7) — Linux Manual Page — Man7.org. https://man7.org/linux/man-pages/man7/socket.7.html. [Accessed 08-09-2025].

[21] Write(2) — Linux Manual Page — Man7.org. https://man7.org/linux/man-pages/man2/write.2.html. [Accessed 23-10-2025].

[22] Armin Afsharian. Memuring: A Memcached Variant With io_uring Integration. https://github.com/armin-afsharian/memuring, 2025. Accessed: 2025-09-26.

[23] Armin Afsharian. Neper-io_uring: Neper Benchmark Integration With io_uring. https://github.com/armin-afsharian/Neper-Io-Uring, 2025. Accessed: 2025-11-25.

[24] Armin Afsharian. UringCL: A Convergence Layer for Simplified io_uring Integration. https://github.com/armin-afsharian/UringCL, 2025. Accessed: 2025-09-26.

[25] Byoungchul An, Hanul Sung, and Hyeonsang Eom. A2L: User-Transparent Workload-Adaptive Asynchronous I/O Layer (Industry Track). In *Proceedings of the 25th International Middleware Conference Industrial Track (MIDDLEWARE '24)*, pages 36–42, New York, NY, USA, 2024. ACM.

[26] Jens Axboe. Efficient IO With io_uring. https://kernel.dk/io_uring.pdf, 2019. Version 0.4, 2019-10-15.

[27] Maurice J Bach. *Design of the UNIX Operating System*. Prentice Hall, Philadelphia, PA, May 1986.

[28] Gaurav Banga and Jeffrey C. Mogul. Scalable Kernel Performance for Internet Servers Under Realistic Loads. In *1998 USENIX Annual Technical Conference (USENIX ATC 98)*, New Orleans, LA, June 1998. USENIX Association.

[29] Peter Cai and Martin Karsten. Kernel vs. User-Level Networking: Do Not Throw Out the Stack With the Interrupts. *Proc. ACM Meas. Anal. Comput. Syst.*, 7(3), December 2023.

[30] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. Understanding Modern Storage APIs: A Systematic Study of libaio, SPDK, and io_uring. In *Proceedings of the 15th ACM International Conference on Systems and Storage (SYSTOR '22)*, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Facebook. RocksDB: A Persistent Key-Value Store for Fast Storage. GitHub Repository, 2024. [Accessed October 30, 2025].

[32] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. Comparing and Evaluating Epoll, Select, and Poll Event Mechanisms. *Proceedings of the 6th Annual Ottawa Linux Symposium*, 01 2004.

[33] Sebastien Godard. Sar Linux User's Manual. https://linux.die.net/man/1/sar, 2025. Accessed: September 26, 2025.

[34] Brendan Gregg. Linux Perf Examples. https://www.brendangregg.com/perf.html, 2024. Accessed: September 26, 2025.

[35] The PostgreSQL Global Development Group. PostgreSQL: Source Code. GitHub Mirror, 2024. [Accessed October 30, 2025].

[36] The PostgreSQL Global Development Group. PostgreSQL 18 Released! Web Page, September 2025. [Accessed October 30, 2025].

[37] Wanning He, Hongyi Lu, Fengwei Zhang, and Shuai Wang. RingGuard: Guard io_uring With eBPF. In *Proceedings of the Workshop on eBPF and Kernel Extensions (SIGCOMM '23)*, pages 56–62, New York, NY, USA, 2023. ACM.

[38] Donald Hunter. Why You Should Use io_uring for Network I/O — Red Hat Developer — Developers.RedHat.com. https://developers.redhat.com/articles/2023/04/12/why-you-should-use-iouring-network-io. [Accessed 09-09-2025].

[39] Brynjar Ingimarsson. Exploring the Performance of the io_uring Kernel I/O Interface. Master's thesis, Vrije Universiteit Amsterdam and Universiteit van Amsterdam, June 2024.

[40] Jonathan Lemon. KQueue—A Generic and Scalable Event Notification Facility. In *2001 USENIX Annual Technical Conference (USENIX ATC 01)*, Boston, MA, June 2001. USENIX Association.

[41] Nick Mathewson, Azat Khuzhin, and Niels Provos. Libevent — An Event Notification Library. https://libevent.org/. Accessed: 2025-12-08.

[42] Kshitij R. Patel, Dhruv S. Thakkar, and HENDRA. Asynchronous I/O Persistence for In-Memory Database Servers: Leveraging io_uring to Optimize Redis Persistence. In *Proceedings of the 2024 6th International Conference on Software Engineering and Information Management (ICSIM)*, pages 168–175, New York, NY, USA, 2024. Association for Computing Machinery.

[43] Niels Provos and Chuck Lever. Scalable Network I/O in Linux. In *2000 USENIX Annual Technical Conference (USENIX ATC 00)*, San Diego, CA, June 2000. USENIX Association.

[44] Irfan Pyarali, Tim Harrison, and Douglas C. Schmidt. Proactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 1999.

[45] Zebin Ren and Animesh Trivedi. Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io_uring. In *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '23)*, pages 35–45, New York, NY, USA, 2023. ACM.

[46] Mohammadamin Shafiei. Adaptive Interrupt Handling for High-Performance Network I/O in Linux and Virtualized Environments. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, 2025.

[47] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Science/Engineering/Math, 2004.

[48] The Linux Kernel developers. Segmentation Offloads. https://docs.kernel.org/networking/segmentation-offloads.html, 2024. [Online; accessed 6-November-2025].

[49] David Wei and Pavel Begunkov. Fast ZC Rx Data Plane Using io_uring. In *NetDev 0x17*, Vancouver, Canada, Oct 2023.