Introduction to Monoio: A High-Performance Rust Runtime

March 25, 2025 23 min read

Rust Async Runtime io_uring Performance

Background

Several months ago, a potential client approached me with a request to build "the fastest HTTP proxy in Rust using the Monoio runtime." This led me into several days of research to determine if Monoio was mature enough for an HTTP proxy implementation and what such an implementation might look like.

While the client ultimately decided not to move the project forward, I found very limited documentation on Monoio. I believe documenting my findings could benefit others interested in this promising runtime. Now that I've had some time, I'm sharing what I discovered. Spoiler alert: the performance results are breathtaking.

In this post, I'll introduce Monoio and explain the technical advantages it offers over other Rust async runtimes. This will be the first in a series of posts where I'll progressively build and benchmark increasingly complex examples using Monoio.

What is Monoio?

Monoio is an asynchronous runtime for Rust, similar to the popular Tokio runtime. However, it is designed with a different set of priorities and architecture

to Tokio. Created by ByteDance (the company behind TikTok), Monoio is specifically built as a thread-per-core runtime that leverages io_uring on Linux for maximum performance.

Unlike Tokio, which is designed to be a general-purpose runtime with work-stealing schedulers that distribute tasks across threads, Monoio follows a thread-per-core model where tasks are pinned to specific threads. This approach eliminates the need for synchronization between threads, potentially offering significant performance benefits for certain workloads. I think it is important to repeat that Monoio should not be used for everything. It only has performance benefits for specific workloads which I cover below.

What Makes Monoio Different?

There are several key differences that distinguish Monoio from other Rust async runtimes:

1. Thread-per-core Architecture

Monoio is designed with a thread-per-core model, where:

- Each thread executes only its own tasks without work stealing
- The task queue is thread-local, eliminating locks and contention
- Cross-thread operations are minimized, optimizing cache performance
- Tasks on a thread never move to another thread

This approach is similar to how high-performance servers are designed, including:

- NGINX with its `worker_cpu_affinity` directive
- HAProxy through its `cpu-map` option
- Envoy, which has an in-depth explanation of their threading model

While it may not fully utilize CPU resources when workloads are unevenly distributed, it excels at network-heavy workloads with predictable patterns.

2. io_uring Integration

Monoio was designed from the start to leverage io_uring, a relatively new Linux kernel interface introduced in 2019 that provides a more efficient way to perform I/O operations. This matters because io_uring significantly reduces syscall overhead and context switches, resulting in better performance for I/O-heavy workloads.

Instead of using traditional epoll-based I/O, Monoio prioritizes io_uring while maintaining fallback support for epoll (Linux) and kqueue (macOS).

3. Unique I/O Abstraction

To fully utilize io_uring's capabilities, Monoio implements a different I/O abstraction than what is found in Tokio or the standard library. The key difference lies in buffer ownership:

- **In Tokio/std**: You provide a reference to a buffer during I/O operations, and maintain ownership
- **In Monoio**: You give ownership of the buffer to the runtime (known as "rent"), which returns it to you once the operation completes

This ownership model is necessary because when using io_uring, the kernel

needs direct access to your buffers, so the runtime must ensure those buffers remain valid throughout the operation.

What is io_uring and Why Does it Matter?

The io_uring interface was introduced in Linux 5.1 and represents a significant advancement in how applications interact with the kernel for I/O operations.

Traditional I/O vs. io_uring

In traditional asynchronous I/O models like epoll:

1. Applications must first check if an I/O resource is ready (e.g., via epoll_wait)
2. Once ready, they make separate system calls to perform the actual I/O
3. This requires at least two context switches per I/O operation

With io_uring, applications:

1. Submit I/O requests to a submission queue that is shared with the kernel
2. The kernel processes these requests asynchronously after being notified with a single syscall (io_uring_enter)
3. Results appear in a completion queue

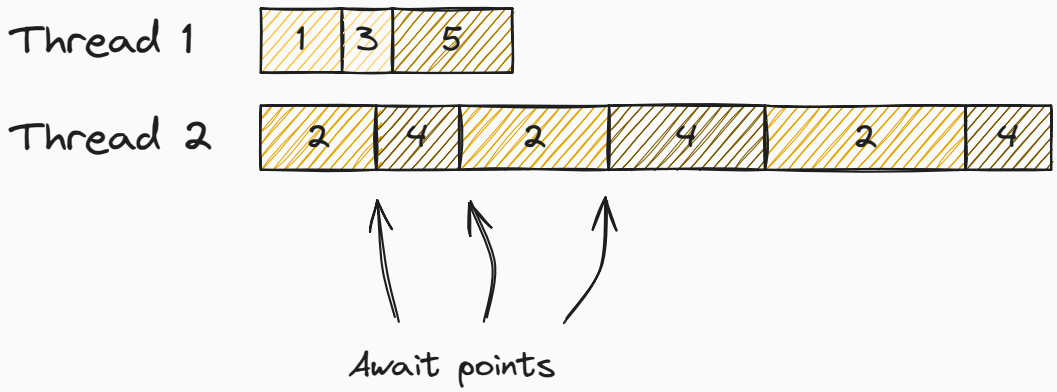
The key advantages include:

- **Reduced syscalls**: Applications can submit multiple I/O operations with a single syscall rather than separate syscalls for each operation

- **Batched operations**: Multiple operations can be processed in batches
- **True asynchronous file I/O**: Unlike traditional interfaces, io_uring enables genuinely asynchronous file operations
- **Zero-copy operation**: The kernel can directly access application buffers

When is Monoio Faster?

Let's visualize the difference between a thread-per-core model and a work-stealing model with a simple example. Imagine we have an application with two threads and five tasks:



In this diagram, we can see how these tasks are distributed:

- Task 1: Which gets allocated to thread one and completed rather quickly.
- Task 2: Which gets allocated to thread two, but takes quite a while to complete and **`awaits`** a few times.
- Task 3: Which goes on thread one when the thread is open and completes quickly too.
- Task 4: Which gets allocated to thread two, but is also quite long with a few **`await`** points.

- Task 5: Which is allocated to thread one and completes quickly.

After these tasks are scheduled, no new tasks arrive for quite a while. Tasks 2 and 4 block at `await` points, which allows the other tasks to make progress if possible. Because Monoio has no work stealing, tasks 2 and 4 will stay on thread two while thread one remains idle until new work comes in.

A work-stealing runtime like Tokio could move task 2 or task 4 to thread one. This would cause the moved task to complete faster, and possibly the other task too since it wouldn't have to wait for an await point to be started again.

This example demonstrates the fundamental tradeoff of the thread-per-core model: while it eliminates synchronization overhead and maximizes cache efficiency, it can lead to unbalanced resource utilization when tasks have varying completion times.

According to benchmarks from the Monoio team, their gateway implementation has outperformed NGINX in optimized benchmarks by up to 20%, and their RPC implementation showed a 26% performance improvement compared to the Tokio-based version - but these impressive results are primarily achieved in workloads where tasks are evenly distributed across cores.

Based on these characteristics, here are the scenarios where Monoio excels:

1. **Network-intensive workloads**: HTTP servers, proxies, gateways
2. **File I/O heavy operations**: Database-like workloads, file processing
3. **Scenarios with predictable task distribution**: Services where load can be evenly distributed across cores
4. **High throughput, low latency services**: When maximizing requests per second is critical

However, Monoio might not be the best choice for:

1. **CPU-bound workloads**: Where task scheduling flexibility matters more than I/O performance
2. **Workloads with unpredictable task distribution**: Where some threads might be idle while others are overloaded
3. **Applications requiring extensive ecosystem compatibility**: Where compatibility with the broader Tokio ecosystem is important

Simple Monoio Applications

Basic Single-Threaded Example

Let's start with a simple Monoio application that demonstrates the async runtime without networking:

```
use std::time::Duration;

use monoio::time::sleep;

// timer_enabled is required for sleep() to work
#[monoio::main(timer_enabled = true)]
async fn main() {
    println!(
        "Starting Monoio application on thread: {:?}",
        std::thread::current().id()
    );

    // Spawn a few tasks
    for i in 1..=3 {
```

```
        let task_id = i;
        monoio::spawn(async move {
            println!("Task {task_id} started");

            // Simulate some async work
            sleep(Duration::from_secs(1)).await;

            println!("Task {task_id} completed after 1s");
        });
    }

    // Main task does its own work
    println!("Main task doing work...");
    sleep(Duration::from_secs(2)).await;
    println!("Main task completed after 2s");
}
```

Output

```
Starting Monoio application on thread: ThreadId(1)
Main task doing work...
Task 1 started
Task 2 started
Task 3 started
Task 1 completed after 1s
Task 2 completed after 1s
Task 3 completed after 1s
Main task completed after 2s
```

This example demonstrates several important concepts:

1. The **`#[monoio::main]`** attribute that sets up the runtime with timer support
2. Spawning multiple tasks that run concurrently
3. Using async sleep to simulate non-blocking work

When you run this program, you'll notice that all tasks execute concurrently within a single thread. This is the most basic usage of Monoio and runs on a single core.

Multi-Core Thread-Per-Core Example (Using macros)

Now let's see how to utilize multiple cores with Monoio's built-in multi-threading support:

```
use std::time::Duration;

use monoio::time::sleep;

#[monoio::main(timer_enabled = true, worker_threads = 2)]
async fn main() {
    let thread_id = std::thread::current().id();

    println!("Starting Monoio application on thread: {thread_id:?}",);

    // Spawn a few tasks
    for i in 1..=3 {
        let task_id = i;
        monoio::spawn(async move {
            println!("Task {task_id} started on thread {thread_id:?}");

            // Simulate some async work
            sleep(Duration::from_secs(1)).await;

            println!(
                "Task {task_id} completed on thread {thread_id:?} after 1
            );
        });
```

```
    }

    // Main task does its own work
    println!("Main task doing work on thread {thread_id:?}...");
    sleep(Duration::from_secs(2)).await;
    println!("Main task completed on thread {thread_id:?} after 2s");
}
```

Output

```
Starting Monoio application on thread: ThreadId(1)
Main task doing work on thread ThreadId(1)...
Task 1 started on thread ThreadId(1)
Task 2 started on thread ThreadId(1)
Task 3 started on thread ThreadId(1)
Starting Monoio application on thread: ThreadId(2)
Main task doing work on thread ThreadId(2)...
Task 1 started on thread ThreadId(2)
Task 2 started on thread ThreadId(2)
Task 3 started on thread ThreadId(2)
Task 1 completed on thread ThreadId(1) after 1s
Task 2 completed on thread ThreadId(1) after 1s
Task 3 completed on thread ThreadId(1) after 1s
Task 1 completed on thread ThreadId(2) after 1s
Task 2 completed on thread ThreadId(2) after 1s
Task 3 completed on thread ThreadId(2) after 1s
Main task completed on thread ThreadId(2) after 2s
Main task completed on thread ThreadId(1) after 2s
```

Notice an important difference from Tokio: the **`worker_threads`** parameter in **`#[monoio::main]`** actually executes the entire program on each worker thread, rather than distributing tasks across threads. This is a key characteristic of Monoio's thread-per-core model. Each thread runs its own complete instance of the application with isolated tasks.

Multi-Core Thread-Per-Core Example (Production-Ready)

For a production environment, we typically want more control over thread

creation and CPU affinity. This example demonstrates how to create a thread for each available CPU core and bind each thread to its specific core:

```
use std::{num::NonZeroUsize, thread::available_parallelism, time::Duratic

use monoio::{IoUringDriver, time::sleep, utils::bind_to_cpu_set};

fn main() -> std::io::Result<()> {
    // Determine how many logical CPU cores are available
    let thread_count = available_parallelism().map_or(1, NonZeroUsize::ge
    println!("Starting with {thread_count} threads (one per logical CPU)'

    // Create threads for each core except the main one
    let threads: Vec<_> = (0..thread_count).map(start_thread_on_core).col

    // Wait for all threads to complete
    for thread in threads {
        thread.join().unwrap();
    }

    println!("All threads completed");

    Ok(())
}

fn start_thread_on_core(core: usize) -> std::thread::JoinHandle<()> {
    std::thread::spawn(move || {
        // Pin this thread to the specific CPU core - this must be done
        // inside the thread but before creating the runtime
        bind_to_cpu_set([core]).unwrap();

        // Create a runtime for this thread
        monoio::RuntimeBuilder::<IoUringDriver>::new()
```

```
            .enable_timer()
            .build()
            .expect("Failed to build runtime")
            .block_on(_main());
    })
}

async fn _main() {
    let thread_id = std::thread::current().id();

    println!("Starting Monoio application on thread: {thread_id:?}",);

    // Spawn a few tasks
    for i in 1..=3 {
        let task_id = i;
        monoio::spawn(async move {
            println!("Task {task_id} started on thread {thread_id:?}");

            // Simulate some async work
            sleep(Duration::from_secs(1)).await;

            println!(
                "Task {task_id} completed on thread {thread_id:?} after 1
            );
        });
    }

    // Main task does its own work
    println!("Main task doing work on thread {thread_id:?}...");
    sleep(Duration::from_secs(2)).await;
    println!("Main task completed on thread {thread_id:?} after 2s");
}
```

Output

```
Starting with 4 threads (one per logical CPU)
Starting Monoio application on thread: ThreadId(2)
Starting Monoio application on thread: ThreadId(3)
Main task doing work on thread ThreadId(2)...
Starting Monoio application on thread: ThreadId(4)
Starting Monoio application on thread: ThreadId(5)
Task 1 started on thread ThreadId(2)
Main task doing work on thread ThreadId(5)...
Main task doing work on thread ThreadId(4)...
Main task doing work on thread ThreadId(3)...
Task 1 started on thread ThreadId(4)
Task 2 started on thread ThreadId(2)
Task 3 started on thread ThreadId(2)
Task 1 started on thread ThreadId(5)
Task 2 started on thread ThreadId(5)
Task 3 started on thread ThreadId(5)
Task 1 started on thread ThreadId(3)
Task 2 started on thread ThreadId(3)
Task 3 started on thread ThreadId(3)
Task 2 started on thread ThreadId(4)
Task 3 started on thread ThreadId(4)
Task 1 completed on thread ThreadId(4) after 1s
Task 1 completed on thread ThreadId(5) after 1s
Task 2 completed on thread ThreadId(4) after 1s
Task 3 completed on thread ThreadId(4) after 1s
Task 2 completed on thread ThreadId(5) after 1s
Task 3 completed on thread ThreadId(5) after 1s
Task 1 completed on thread ThreadId(2) after 1s
Task 2 completed on thread ThreadId(2) after 1s
Task 3 completed on thread ThreadId(2) after 1s
Task 1 completed on thread ThreadId(3) after 1s
Task 2 completed on thread ThreadId(3) after 1s
Task 3 completed on thread ThreadId(3) after 1s
Main task completed on thread ThreadId(5) after 2s
Main task completed on thread ThreadId(3) after 2s
Main task completed on thread ThreadId(4) after 2s
Main task completed on thread ThreadId(2) after 2s
All threads completed
```

This production-ready approach offers several advantages over the simpler macro-based version:

1. **CPU affinity**: Each thread is explicitly bound to a specific logical CPU core with **`bind_to_cpu_set`**, preventing the OS scheduler from moving

threads between cores and maximizing CPU cache utilization. Note that this uses all logical CPUs, including hyperthreaded cores.

2. **Error handling**: The code properly handles potential errors with `Result` returns, making it more robust for production environments.
3. **Full control**: You have complete control over thread creation, runtime configuration, and lifecycle management.
4. **Resource utilization**: It automatically scales to use all available CPU cores in the system.

The multi-core example demonstrates several important patterns for building high-performance applications with Monoio:

1. **Manual thread creation**: Unlike Tokio's multi-threaded runtime which handles thread creation internally, with Monoio you explicitly create threads for each core
2. **Separate runtimes per thread**: Each thread gets its own independent Monoio runtime
3. **Task isolation**: Tasks spawned on one thread never move to another thread
4. **No cross-thread synchronization**: No locks or shared data structures between threads

This approach is exactly what we'll need for our proxy, as it allows us to:

1. Maximize throughput by utilizing all cores
2. Minimize contention by keeping tasks isolated
3. Avoid synchronization overhead between threads
4. Take full advantage of CPU cache locality

Note the key differences from a Tokio application:

1. We use `#[monoio::main]` for simple cases or manually create runtimes for more control
2. With I/O operations, ownership of buffers is transferred (the "rent" pattern)
3. Tasks are always thread-local without any work stealing

Conclusion

Monoio represents an exciting advancement in Rust async runtime technology, particularly for I/O-bound server applications. By leveraging io_uring and a thread-per-core architecture, it offers significant performance benefits for the right use cases.

The thread-per-core model isn't new — it's a proven approach used by high-performance servers like NGINX and Envoy. What makes Monoio special is how it brings this architecture to Rust with native io_uring support, potentially offering best-in-class performance for network services.

As we progress through this blog series, we'll build on these concepts to create increasingly sophisticated components:

1. First, we'll develop a basic HTTP server with Monoio and benchmark it against Hyper/Tokio
2. Next, we'll implement HTTP/2 protocol support
3. Then, we'll add TLS support and evaluate its performance impact
4. Finally, we'll create a complete proxy server with all these features

At each stage, we'll benchmark against both equivalent Tokio implementations and industry-standard proxies like NGINX, Caddy, Envoy, and HAProxy. This will give us a clear picture of Monoio's real-world performance characteristics.

Stay tuned for the next post where we'll build a basic HTTP server with Monoio and benchmark it against an equivalent Hyper/Tokio implementation!