Maximizing Requests per Second on a Single Thread through `io_uring`

Jackson Mowry

4/16/24

I Abstract

This research paper investigates the transformative potential of io_{uring} in scaling HTTP web servers to accommodate the needs of numerous concurrent clients. Despite the extensive exploration of various scaling techniques, little attention has been paid to the potential paradigm shift that `io_uring` may introduce in asynchronous I/O architectures. `io_uring` enables the handling of workloads traditionally confined to highly parallelized systems. This paper aims to explore the implications of `io_uring` on asynchronous I/O systems, shedding light on its promise for enhancing server scalability and performance in the face of ever increasing demands.

II Introduction

Web servers are a piece of software that sit between nearly every interaction a client makes on their device, and access to the files housed on a server. Despite this fact they are rarely inovated on, due to the fact that for most use cases they are "good enough". It is true that even a naive HTTP server can serve thousands of requests per second, which easily surpasses the needs of most users. The same architecture can be transplanted on top of a machine with many CPU cores in order to scale up to meet demands.

But these workloads are still inhrently synchronous at some level (excluding `aio`). `io_uring` brings a true asynchronous io system to the table with its introduction into the Linux kernel (version 5.1).

In this article we will explore "multiplexed io" on linux through the use of `io_uring`. Implementations are compared to the best possible implementation on a single core under other programming models. Along with single core tests we will pit our single core against a threaded server.

III Asynchronous IO

Most if not all system calls used in day to day software are considered "blocking", meaning they will only return once the entire function has finished executing. This model allows for programs to not having to worry about order of execution, as they know their program will run top to bottom without skipping a step. However, large web servers simply cannot afford to spend the time waiting for each system call to execute to completion. To avoid this problem of blocking programmers may choose to design their software around asynchronous IO.

On Linux this has taken the form of `epoll`, which allows for certain operations to be performed without blocking the main thread of execution.

IV `aio`

The `aio` system in Linux has two implementations, POSIX compliant, and the Linux implementation. Both systems suffer the same issue, they are intended to work with regular files, and as such don't play well with sockets. glibc creates a thread pool to perform regular synchronous io off the main thread, giving the illusion of asynchronosity.

For all of these reasons `aio` is generally avoided in application code.

V `epoll`

`epoll` is kernel based implementation of `poll`, with both providing a way to monitor a range of file descriptors, and alerting the user when one or many are ready for IO. It has become a very common architecture for web servers and other asychronous io systems to be built on top of. Most notably to implement the primatives golang's `net/http` package is built upon, and `libuv` which powers the Node.js event loop.

`epoll` expands on the original ideas of poll by sharing a list of file descriptors between the user and kernel, preventing the need for data to be copied back and forth. When any number of file descriptors are ready for IO they will be placed in a separate shared list, which the user can then perform the desired action on.

This architecture allows for a single thread to handle large numbers of active file descriptors, only slowing down to perform the synchronous operations like reading or writing. The actual implementation does not allow for asynchronous sending or recieving of data, merely alerting the user when those operations can be performed without blocking.

One downside of `epoll` is that it does not behave consistently across file descriptors of different types. While it is true that recieving data from a socket can

block, which `epoll` is aware of, regular files do not exhibit the same behavior. On Linux read and write calls to a file should not block, but as we all know this is not true. You can make a write call and expect it to complete instantly, but if the kernels write cache is full, you will have to wait. The same goes for a read call which can blocked if the file is on a slow drive.

VI `io_uring`

`io_uring` is the latest attempt at adding asynchronous operations to Linux. Its design makes it obvious that it the designer learned from many of the shortcomings of `aio`. Not only does `io_uring` provide a common interface across all types of file descriptors, it also implements most systems calls in an asyncronous fashion.

The design can be broken down into two distinct parts, a job submission queue, and a job completion queue, both implemented via ring buffers shared between the kernel and user space. At a basic level each submission is a combination of an op code defining which system call to perform, and the associated arguments. If the user desires to keep track of a job they can associate user data with a submission, which takes the form of an 64-bit integer, commonly used to hold a pointer. The implementation garuntees that user data will not be modified.

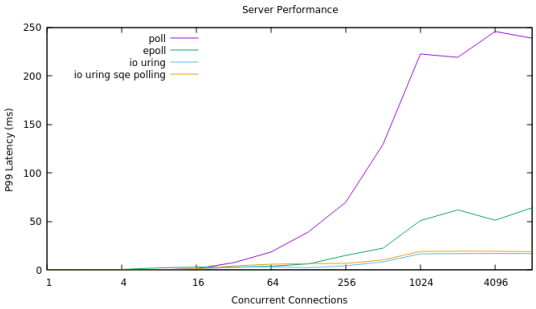
Once a job is complete it is placed in the completion queue, which an application can pull from. Completions have 2 main fields to pay attention to: a result code, and the associated user data. The result code is analogous to the return value from regular blocking system calls with one exception. Due to the concurrent execution of submissions the system cannot garuntee that the `ERRNO` associated with each system call will still be properly set when a user recieves a completion. Instead, `ERRNO` is placed in the completion struct, with its value negated so that it will not be confused with a successful execution.

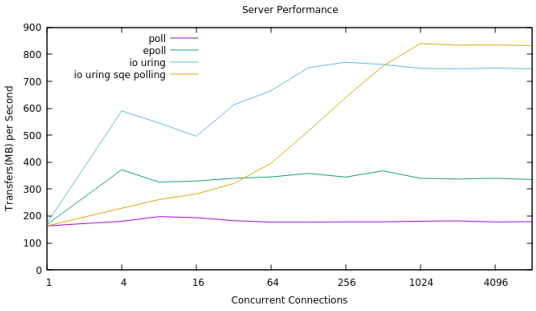
`io_uring` also offers one distinct advantage over the other asynchronous io methods presented here. Jobs can be submitted through a system call, or by having the kernel continuously poll the submission queue using a separate thread. This allows for a program to operate entirely in user space, avoiding system calls which have become even more costly in the age of speculative execution mitigations. We will explore both methods of job submission to see where their advantages lie.

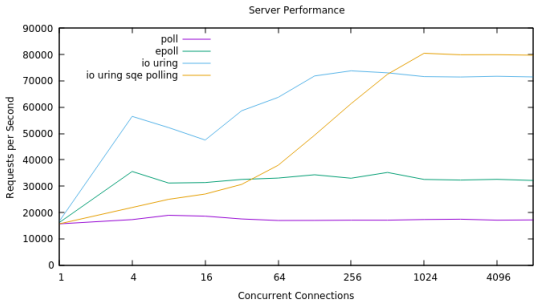
VII Methods

Testing will include an `epoll` server, two distinct `io_uring` servers, and a fourth synchronous server using `poll`. Servers will parse a request, open a file, respond with the appropriate headers, send the file, and finally close the connection. Performance testing using `wrk` at {1, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192} concurrent connections run over 30 seconds, with the mean of 3 runs reported for each metric.

VIII Results







IX Discussion

X Cite

https://www.kernel.org/doc/ols/2003/ols2003-pages-351-366.pdf https://www.landley.net/kdocs/mirror/ols2004v1.pdf#page=215 https://darkcoding.net/software/linux-what-can-you-epoll/ https://darkcoding.net/software/epoll-the-api-that-powers-the-modern-internet/