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

Jackson Mowry
jmowry4@vols.utk.edu

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 class of software that sit between nearly every interaction a client makes on their device, and access to the files hosted on a server. Despite this fact their implementations mainly use software designs of the past, 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, but when it comes time to scale up the same architecture only has two ways to scale. The server can either be rewritten to take advantage of multiple CPU cores, or it can have multiple instances running behind a load balancer.

The root of this issue lies in the fact that these workloads are still inhrently synchronous at some level (excluding `aio`), which leads to large requests stalling the server. `io_uring` brings a true asynchronous io system to the table with its introduction into the Linux kernel (version 5.1). An exciting new space for high performance single core servers now exists, in this paper we will see how `io_uring` can fill that gap. To date `io_uring` has not been implemented in mainstream web server applications, this paper serves to illuminate the potential benefits, to both speed and latency, of the latest in asynchronous io technology.

III Previous Work

TODO

IV 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.

For most applications it may not make sense to perform all system calls asyncronously. We certainly want slower system calls to complete in the background, but the added overhead of an asynchronous system call may make faster calls worse. We expect to see greater differences in throughput for large files as they would stall the server while attempting to read/write data that may not fit in a single block/page.

V `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 to the user-space program.

`aio` is also very limited in scope, only offering real support for read and write calls. This means that without splicing together features from different asynchronous IO frameworks an application will still have to rely heavily on blocking system calls.

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

VI `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 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.

Due to `epoll`'s handling of regular files they will always be placed on the ready list immedietly. 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.

VII `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. In systems where response time is the highest concern, submission queue polling will likely be the best choice due to eliminating a system call per request. We will explore both methods of job submission to see where their advantages lie.

Being the latest in the asyncronous IO space `io_uring` is still lacking some features. Most notably is missing system calls, and event notification on a submission/completion level. Most missing system calls can be implemented without changing the underlying system, with the remaining few not being necessary to implement asynchronously.

VIII System Calls

One of the major mitigations put into place after the speculative execution attacks were discovered (spectre and meltdown) was isolation of kernel and user space memory. This slows system calls as they must switch into a different address space to perform the operation, and switch back once they're done.

To mitigate this, modern software systems have transitioned to a model where system calls are avoided, sometimes entirely. The most obvious implementations of this are systems that manage memory allocations themselves, or those that manage files using shared memory.

`io_uring` allows the program to avoid system calls upon submission, meaning that a system can work entirely in userspace, without the need for costly system calls. This reduces the number of system calls from at a minimum 4 (accept, read, write, close) in a synchronous server, to a potential 0 in an `io_uring` based approach. For a simple file server the only system call that is still required is `pipe`, which does not have an asynchronous implementation yet.

Allowing system calls to proceed asynchronously also allows for a program to handle clients with heavy requests without blocking clients with lighter weight requests. This is a major advantage over synchronous servers where either an entire program is blocking reading/writing a file, or a thread is held up waiting.

This can slow the overall throughput of these systems, whereas an asynchronous implementation would proceed handling other clients while a file is being read.

Many modern web servers also try to combat the cost of system calls by using auxiliary threads to handle either entire requests, or smaller portions. This approach works until a certain point, as the overthread of spawning an operating system thread is non-trivial. The best approach will likely vary application to application, as certain applications may require heavy synchronous computations, which a single thread would not be able to handle(John Ousterhout, 1995).

IX Methods & Implementation

Testing will include an `epoll` server, four distinct `io_uring` servers, and a sixth 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.

For testing four `io_uring` based servers will be compared. For a more realistic general purpose web server (uring simple) example, only the accept, read, and close system calls will be replaced with their asynchronous counterparts, with the rest of the work handled synchronously. The application follows a simple state machine where a connection is either in `ACCEPT`, `READ`, or `WRITE/CLOSE`.

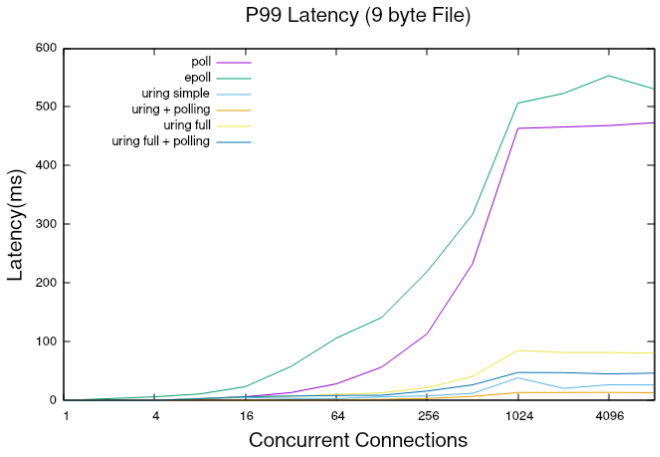
The complete `io_uring` server will replace every system call (except for `pipe` which does not yet exist) with their asynchronous versions (uring full). The state machine follows a similar pattern adding `CLOSE_FILE`, `CLOSE_SOCKET`, `CLOSE_PIPE`, `OPEN`, `SPLICE`, `STATX`, and `SEND`. Each of these servers will also have submission queue polling enabled (uring simple + SQP) (uring full + SQP) as an additional observation point.

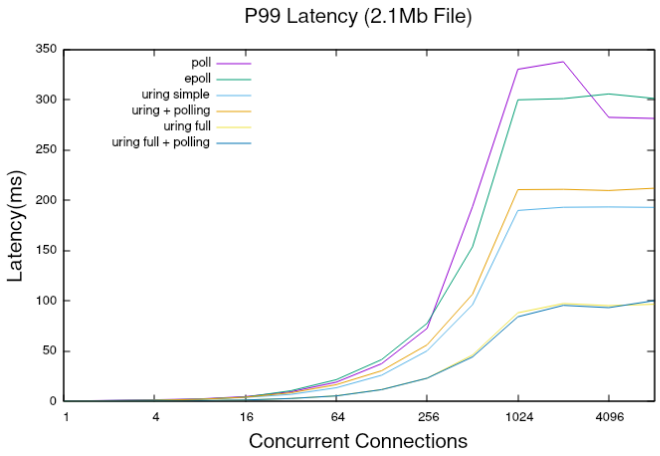
X Results

Latency

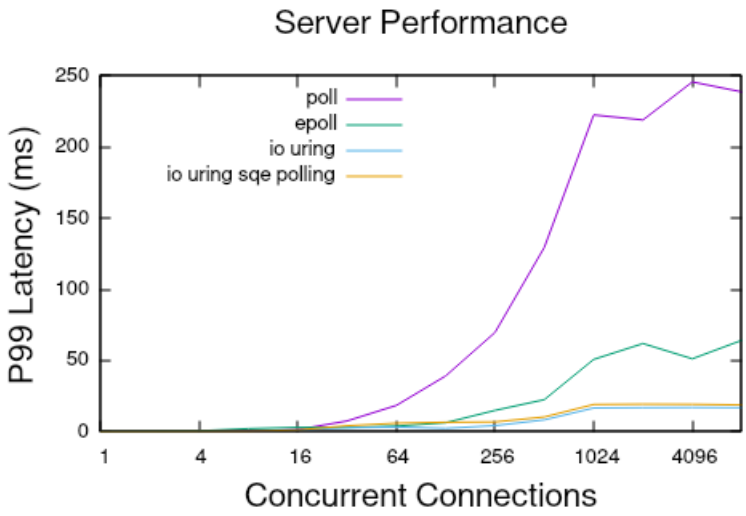
As expected, the synchronous `poll` based server experiences substantial growth in latency as the number of concurrent request in increased. This is due to the fact the scanning the list of watched file descriptors happens in linear time as each is checked for a `POLLIN` event. The `epoll` server is able to maintain a similar latency to either `io_uring` server until 128 concurrent request when it begins to climb rapidly. Latency continues to grow with connections, and would be expected to continue rising.

Both `io_uring` servers exhibit similar P99 latency, quickly reaching a plateau around 25ms from 1024 concurrent requests and beyond. One of the key benefits of `io_uring` is its ability to handle massively concurrent workloads, as observed with either implementation. Submission queue polling increases latency by a fixed amount across the entire test range, the change seems to have the largest impact when serving small files, as was predicted. For the simple and full `io_uring` server enabling polling dropped latency by 67% and 53% respectivly.



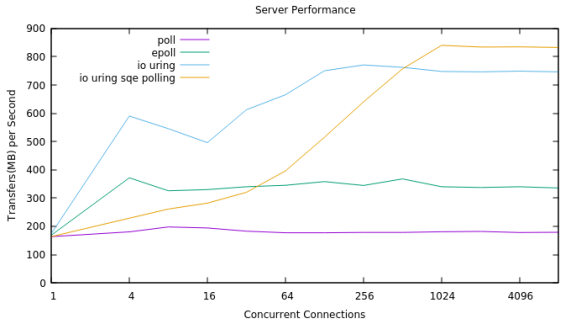


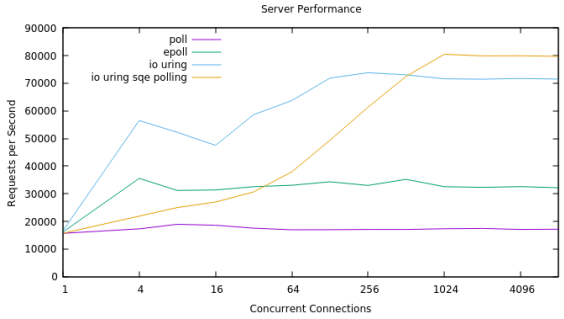
While serving small files still benefits substantially from `io_uring`, the simple `io_uring` server beats a full server by a factor of 2x. A small file tends to be less performance bound by disk IO, as the entire file can easily be read and written in a single page. Moving up to a medium sized file (12Kb) we see that the gap between either `io_uring` remains about the same, just around 2x. It is not until the large file is requested that a full `io_uring` server begins to show its advantages. At a full 8192 connection load the simple `io_uring` server peaks at 200ms of latency, whereas the full `io_uring` server just begins to push 100ms. Either synchronous



server continues to climb in latency up until a peak of 300ms at full load.

Throughput





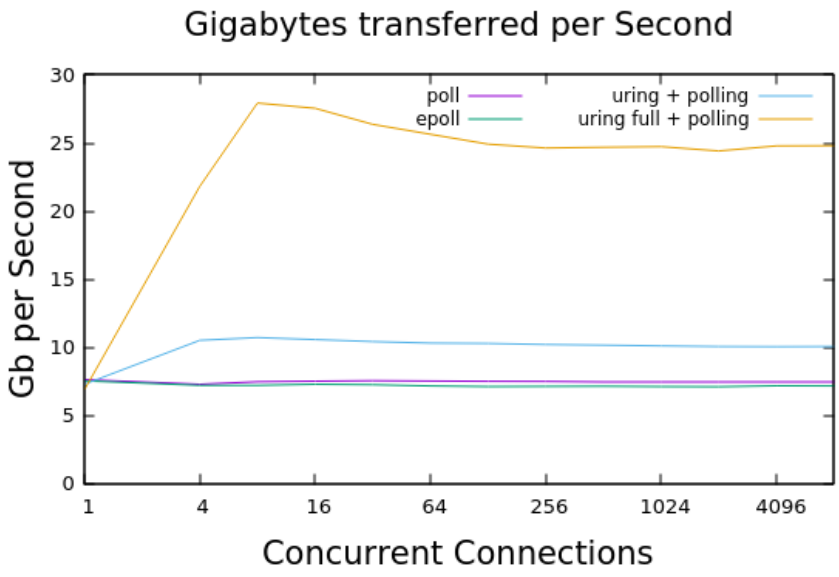
Throughput of the synchronous server follows our expectations, a nearly flat line across the entire testing range. The rate at which we can push data over the wire is entirely limited by the fact that each request is handled one at a time, which is entirely dependent on the time on which we are blocked in system calls. For simple applications this approach may be enough to handle the workload, and it comes with the upside that a `poll` implementation is a much simpler architecture.

`epoll` exhibits nearly the same behavior as the poll based implementation, only at a higher overall throughput. We see no degradation in throughput for either server up to 8192 concurrent request. The advantage for `epoll` comes in the fact that it's "work queue" is only populated with requests that have data ready, meaning that each file descriptor can have work done without having to check its status.

A basic `io_uring` server without submission queue polling quickly reaches its maximum throughput at 128 concurrent requests, which the server is able to maintain all the way up through 8192 concurrent requests. This equates to a maximum throughput of around 750MB/s, or ~6Gbit/s, serving a 12KB text file to each client. When submission queue polling is enabled we see an interesting shift in throughput. A much lower throughput is seen at lower concurrent requests, which quickly jumps past the original implementation at 512 concurrent requests. From 512 requests and beyond a gap of just under 10,000 requests per second is maintained. This equates to a gap of around 100MB/s, or 0.8Gbit/s.

As suspected the fully asynchronous server is still slower for medium sized files. The performance scales in the same manner at half the overall throughput. We observe similar performance impacts from enabling submission queue polling, only overcoming non-polling past 512 concurrent requests as seen in the previous example.

Testing with large file sizes begins to highlight the advantages from the newer `io_uring` system. When clients request a 2.1Mb file we finally begin to see that the full `io_uring` server has major advantages. Without blocking each request on a single synchronous path we greatly increase throughput, and decrease latency. Both synchronous servers, and the simple `io_uring` server spend the majority of their runtime blocked in system calls, meaning that even though we have thousands of connected clients, we cannot concurrently send any information to multiple clients.



The general flow of a request is to first open `stat` the file to gather information, open the file for reading and then using either `sendfile`, or a combination of pipes and splicing to achieve the same effect. When using a single thread in a synchronous context we see a large increase in latency due to this linear pipeline of information transfer. We must wait until the entire file (and corresponding headers) have been passed off to the kernel to send. This results in our latency being enitrely determined by the speed at which a single client can be served.

In a fully asynchronous context, many thousands of clients can connect at the same time, with each of their requests being processed concurrently. The largest contributing factor to decreased throughput (and thus increased latency) is the sending of the large 2.1Mb file, which is entirely mitigated when using asynchronous IO. This results in our fully asynchronous server beating `epoll` by 3.5x, and a simple `io_uring` server by 2.5x. At 8 concurrent clients the fully asynchronous implementation achieves an average throughput of 27.94GB/s, settling in to just around 24.8GB/s for the remaining tests.

We see the opposite results when testing a smaller 9 byte file, as the overhead of processing system calls asynchronously greatly decreases overall throughput. The best performer with small files was the simple `io_uring` implementation with submission queue polling enabled, which managed 8.79MB/s at 8192 concurrent connections. Our fully asynchronous server was able to achieve just under half the throughput (3.84MB/s), still beating either synchronous server (~1.30MB/s).

XI Discussion

Either asynchronous implementation offers obvious advantages over an entirely synchronous server. `epoll` offers an interesting middle ground where the architecture is simpler, at the cost of reduced throughput. However, `epoll` has better portability when if the application needs to run on the family UNIX based systems. Linux has `epoll`, while MacOS and BSD both have `kqueue`, which offers a similar API for performing asynchronous operations on file descriptors. `epoll` has also been around since Linux kernel version 2.5 (2002), meaning resources and documentation are more widely available.

As discussed before, one potential disadvantage with `epoll` is that it only works well with sockets, or files opened in an unbuffered/direct mode. If an application needs to do more than accept an incoming connection on a socket and send back a response, `epoll` will not be useful for other operations. `io_uring` on the other hand offers a uniform interface for the entire range of IO operations. This consistent handling of different file descriptors allows for the programmer to simplify their application to a more robust state machine.

A fully asynchronous server is able to scale in both throughput and latency, which is desireable for many applications. This increase in performance also comes with the simplicity of running on a single user space thread. A single threaded application generally limits the need for concurrency control mechanisms, which can slow the execution of a program.

`io_uring` has shown that with its superior implementation it can out perform the contemporary way we build web servers. Showing a 75.95% improvement (85.01% for SQ polling) in throughput over the `epoll` based server

at 8192 concurrent requests when serving a 12Kb file. The worst case latency of an `io_uring` server also scales better than `epoll`, exhibiting 1/4 the latency at 8192 concurrent requests.

The general performance gap only grows as the response size does as well. When serving a 9 byte file `io_uring` exhibits a 2x increase in throughput, which grows to over 2.5x at 12Kb, and finally 3.5x for a 2.1Mb file. This follows the long known trend that asynchronous operations help IO bound applications scale. As file size increases we would expect to see the same trend of decreased latency and increase throughput.

XII Future Work

More work is needed to explore the different performance impacts of various asynchronous calls through `io_uring`. Theoretically, a server performing very few systems calls should outperform one doing many system calls, but as we have observed, this is not always the case. Once a few more of the key systems calls have been implemented (pipe, and sendfile specifically) performance testing should once again be performed.

XIII 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/