A Faster Copy using Asynchronous I/O

Nalin Mahajan, Vineeth Bandi

August 6, 2022

1 Introduction

For this project, we implemented recursive directory copying with asynchronous I/O. The benefits of such an approach align with the benefits of asynchronous requests as we would not need to block and wait for I/O to finish. This would ideally improve performance for scenarios where we are making large amounts of I/O requests such as a recursive copy as we can also reduce the number of threads required to service all of these requests.

To start this project, we first identified two different methods for asynchronous I/O supported by Linux. The first is `posix aio` and the second is `io_uring`. We describe these methods in further detail later in the report. We then compare this using an implementation of `cp` that matches our framework for `aio` and `io_uring`. We hypothesize that the synchronous `cp` will outperform both `aio` and `io_uring` in most common workloads, but `aio` and `io_uring` can find some performance benefits in workloads with a large amount of files that need to be copied such as a recursive directory copy with unbuffered access. Since the intent of `io_uring` was to fix some of the failures of `aio`, we expect to see further performance gains from `io_uring` for the cases where asynchronous I/O would be optimal. This would primarily come in the form of batching operations or multiple file transfers since we start copying multiple files at once using asynchronous requests.

To test our implementation and identify workloads where asynchronous I/O can be beneficial, we measured the runtime and memory usage of our programs over multiple tests and compared them with each other. The specific results and details of these experiments will be outlined further in the paper. We found that `io_uring` is a suitable replacement for `cp -r` for most workloads when using buffered requests especially when batching read and write operations and performs equally well to `cp -r` in almost all other cases. We see that `cp -r` is still viable in some cases of larger files under single reads and writes requested at once and the best option for small files since it avoids a lot of the overhead of asynchronous methods. Finally, we see that `aio` is a viable alternative to `cp -r` when using it asynchronously with unbuffered I/O requests, but that it is outperformed in almost all cases by `io_uring`. Our implementation code and testing scripts can be found at `https://github.com/nalin29/copy_async`.

2 Background

2.1 cp utility

The linux utility `cp` provides functionality to easily copy files around the user's file system. Originally, this used a series of read and write syscalls to copy a file's contents. Currently, `cp` makes use of the `copy_file_range` syscall. The syscall takes in a source file descriptor and a destination file descriptor along with the size of the copy and returns the amount copied. `cp` applies this repeatedly until a return value of 0 which indicates that there is no more file left to copy.

The `copy_file_range` syscall appeared as part of Linux 4.5 and using glibc `cp` also supports emulation of the syscall in user space if not present. The use of this syscall allows file systems to provide a specific copy implementation that can use copy acceleration techniques. This is useful especially for specialized file system for instance: a copy on write file system such as BTRFS can simply create a reference link between file blocks only writing meta data or certain networked file systems such as NFS can simply request the host to copy the file through a specific API call.

For non-special implementation `copy_file_range` allows copies to be performed from within the kernel. This bypasses the cost of buffer copies between user and kernel space. However, `copy_file_range` is still a synchronous operation requiring the application wait till completion and possibly block for I/O. This is a robust and incredibly simple approach to file copying in linux. The use of a single syscall makes the implementation fairly trivial and is more extensible as file systems may provide their own file copy implementations.

To provide copies of an entire directory, we can specify the use of the recursive flag, `cp -r`, which will iterate through the directory synchronously copying each file before moving onto the next. This is an elegant way of handling directory copies but provides an avenue for optimization. Instead of copying each file one at a time we can possibly increase the speed of larger directory copies by doing simultaneous asynchronous copies to multiple files at the same time. Additionally, since the kernel is still doing reads and writes to files synchronously we could also speed up the single file case by doing multiple asynchronous reads and writes to the same file.

2.2 posix aio

The `aio` library presents a library for posix compliant asynchronous I/O in linux. This was added as part of linux 2.5 and presents an early and limited version of asynchronous I/O. This interface allows programs to leverage asynchronous I/O operations by queuing them and receiving a notification upon completion. Note that the current implementation is in user space which means that there is increased overhead for context switching and signal processing.

The `aio` library also implements an asynchronous state machine for its operations and processes requests using asynchronous waitqueue callbacks. To start requests, we populate the asynchronous I/O control block which is a data structure that defines parameters such as the file descriptor and offset into the file. We then simply call the `aio` version of the syscall we want for example, `aio_read`, which will enqueue the request. To retrieve the status of the request, we call `aio_error` and `aio_return` to get the status and equivalent syscall return value of the requests which would be 0 and the number of bytes read in the case of `aio_read`.

These requests are all run in the context of kernel threads as well. Asynchronous requests only work when the file is opened with the `O_DIRECT` flag. This means that we are required to make unbuffered requests that avoid using the kernel cache. This is a significant limitation to the implementation of asynchronous I/O. Another limitation is that the I/O request can block for many reasons effectively making requests synchronous. For example, we might block if we are waiting for metadata or if the number of request slots on the storage device are full.

2.3 io_uring

The io_uring API is a new way of utilizing asynchronous I/O on linux. This was adopted along with its library liburing in Linux 5.1 with the main goal of replacing aio and providing a highly performant and feature rich solution. The io_uring interface utilizes ring buffers for communication between kernel and user space seeking to minimize the number of system calls made. The interface uses two ring buffers: the submission queue which is denoted as (SQ) and the completion queue (CQ). The rings are set up as shared memory between kernel and user space by generating them in kernel space and utilizing mmap to map them into user space.

To add an instruction you simply create and populate the necessary submission queue entry struct and add it to the tail of the submission queue buffer. The user can then notify the kernel that an operation has been populated which can then add it to the submission queue by reading the buffer. The notification syscall will immediately return providing asynchronous operation to the user. The kernel then processes the submission queue events and then places the results onto the completion queue ring buffer. The kernel can also provide notifications to the user on an event completion and the user may wait on a specified number of events. The user can now reap the Completion Queue Entries from the head of completion queue ring buffer and continuously add Submission Queue Entries and reap completion queue entries.

This design enables a zero-copy system similar to what we saw in cp since the ring buffers are mapped in both kernel and user space. Additionally, the user can register data buffers which get pre-mapped into kernel space for reducing even more data copying. Additionally, we reduce the number of syscalls dramatically as we only need to notify the kernel of new operations to perform which can be batched. For waiting on queue events, io_uring also provides several polling modes such as kernel and user level polling for the selection and completion queue further reducing wait time as well as the number of syscalls needed.

As a result, io_uring has few limitations compared to aio. io_uring works even in buffered scenarios without the use of O_DIRECT. Additionally, through the use of the liburing library the programming interface is sufficiently simplified while still providing extensible functionality for other operations such as network connections or message sending. The result is on paper a much faster system than aio. According to documentation io_uring should be able to outperform aio, with very little optimizations applied, by a factor of almost two even beating certain synchronous interfaces in total throughput.

This new interface is still fairly new and lacks widespread adoption while still being under some development as new features and optimizations get added. However, it seems like a good future proof candidate for asynchronous I/O that can provide good results with even novice implementation. As a result we also design a recursive copy using this interface, to provide a modern implementation of asynchronous I/O.

3 Implementation and Design

3.1 General Implementation

Our general implementation of all of the methods for copying files is to maintain a queue of files and directories we need to process and a working set of files to request I/O operations for. To start, we first perform Breadth-First Search on our source directory. If the next structure is a directory, we will make an equivalent directory at our new destination and start traversing this subdirectory to append files to our queue. If the structure was just a file, we instead find its new name i.e., the name it would have in the destination directory. We then populate the relevant I/O structures which will be discussed in further detail below.

3.2 Copy using aio

To use `aio` we first need to call `aio_init` with a `aioinit` struct which has a number of parameters with most unused. We are interested in `aio_threads` which is the max number of threads we want to use for the implementation, `aio_num` which is the maximum number of requests that we will have enqueued simultaneously, and `aio_idle_time` which is the time before idle threads will terminate. Next to make I/O requests, we must populate a `aiocb` struct. We make two to handle the read and write for copying a file. These require a buffer, file descriptor, number of bytes to read/write, and offset into the file which are referenced with `aio_buf`, `aio_fildes`, `aio_nbytes`, and `aio_offset`, respectively.

Recall that `aio` only works asynchronously when we have the `O_DIRECT` flag set. Our implementations support three different ways of processing files, using single operations per file, multiple operations per file, or batched operations. The specifics of these approaches are outlined below.

3.2.1 Single Operation per File

Here in the main loop, we start queueing files to the working set and start making I/O requests. After populating the `aiocb` struct, we can then start a read of the file in the source directory. We continuously loop until we are done with open requests in our working set. Inside this loop we check if the current file has finished reading and writing and if it has, we add the next file to the queue and start processing that file. If it has only finished reading we start the next write. Since we can only process files based on our buffer size, we read bytes until our buffer is full then write those bytes and repeat until we are done with the file.

3.2.2 Multiple Operations Per File

For supporting multiple operations per file at a time, we now also maintain a queue of currently executing files so that we can interweave reads and writes. We use the queue to update the `aiocb` structs with the next pair of read and write operations that we want to process.

3.2.3 Batched Operations

The idea behind batched operations it that we will be able to split our file into a series of reads and writes and concurrently and asynchronously make requests on these files. This is done similarly to multiple operations per file but we also have a list of read and write `aiocb` structs to help us maintain these files. This leverages `lio_opcode` and `lio_listio` which uses the list of `aiocb` structs to batch the operations based on the opcode. Note that these operations can be executed in any order which does not necessarily matter in this use case as long as we wait for all operations to finish before continuing. We ensure this by passing `LIO_WAIT` as an argument to `lio_listio` which ensures that we block until all I/O operations are complete.

3.3 Copy using io_uring

To utilize `io_uring` we make our life easier by utilizing the API defined in the `liburing` library. Similar to `aio` we need to initialize some of the command structures. The main control structure is the `struct io_uring`. This refers to the combined ring structure which we can pass to API calls to interact with the different ring buffer structures in `io_uring`. Once we create the ring we can call `io_uring_queue_init` which will take in the queue depth and pointer to the ring as well as special flags. This populates the ring data structure very similarly to how we utilize `aio_init`. Then we can begin to start the I/O operations.

To perform a read we can gather a selection queue entry by calling `io_uring_get_sqe`. Then to populate the entry we can call `io_uring_prep_readv` which takes in the entry, the source file, an `iov` struct, the number of operations are supplied and an offset. The `iov` struct is very similar to the `aiocb` structure used in `aio` but with even less constraints. We can then set the data pointer using `io_uring_sqe_set_data` and then to start execution we can place it in the selection queue by calling `io_uring_submit` which will submit all outstanding entries for execution. For write operations we can use the similar `io_uring_prep_writev`.

The `liburing` library also provides a useful feature that allows us to order certain operations. These are called linked entries. After getting the entry and before prepping to the selection queue we can add the `IOSQE_IO_LINK` flag by calling `io_uring_sqe_set_flags`. The next entry that is prepped is then linked to this entry. This means that `io_uring` will ensure it is ordered after the previous operation. We can generate arbitrary size length chains by also adding the `IOSQE_IO_LINK` flag to the next entry. In this way we can easily create read-write pairs. By linking read operations to write operations with the same buffers and offsets.

A special optimization that `liburing` exposes is the use of registered buffers. These are buffers that we allow `io_uring` to know ahead of time. This reduces the cost of having to redo buffer mappings into the kernel on every entry, since we now provide an index representing the already mapped buffer to `io_uring`. We implemented this optimization as an optional feature `-rb`. To implement this, first we must register buffers with `io_uring_register_buffers` which takes a list of pointers to buffers. Then to execute operations using these registered buffers we make use of `io_uring_prep_write_fixed` and `io_uring_prep_read_fixed`. For these operations we must also pass an index referring to the buffer index in the registered buffer list.

3.3.1 Single Operation per File

The first implementation of our copy program using `io_uring` was to have a single read-write pair per file to see if we can make use of executing operations on multiple files at once. This is implemented by generating a read-write pair file using the process mentioned above and executing as many as we can. Then in a structured main loop we keep track of the number of currently executing pairs and collect them using `io_uring_submit_and_wait` this lets us retrieve the completion queue entry that contains the requisite information for this operation. We then check to see that both the read and write operations for that file have checked in and then either add a new read-write pair for the next offset in the file or start a new read-write pair for the next file, submitting them immediately. To ensure that the completion queue entry is popped off the queue we must use `io_uring_cqe_seen` after using the entry allowing for it to be deallocated. This process is repeated until there are no more in-flight operations.

3.3.2 Multiple Operations Per File

The second implementation performs multiple operations per file making use of the asynchronous executions to perform reads and writes at different offsets of each file. This also allows us to perform multiple reads and writes to the same file and different files at the same time. This is implemented in a very similarly way as when only we had only one operation pair per file. The only difference is now we must also track when all operations for a file have checked in so the file may be closed and the current offset in the file for future operations.

3.3.3 Batched Operations

We then implemented batched operations. In this case we do not immediately submit all the entries rather we wait for all entries to first check in. This can be done using `io_uring_submit_and_wait` which submits all entries and then waits for completion. We can then iterate over the entries and prepare new entries as we did in multiple operations per file. The next loop will then submit and wait for completion allowing us to batch operations with ease. We expect this approach to be the fastest as I/O controllers will be able to better reorganize operations if they arrive all at once providing a performance boost.

4 Evaluation

The goals of this project was to explore workloads where we can effectively leverage asynchronous I/O for performance benefits. As such we mainly measure the performance of our implementations with each other through runtime and memory usage. We tested a variety of workloads that work best with single operation, multiple operations, and batched operations. We also test the impact of the optimizations we added to the programs. Finally, we also test what changing the some of the parameters of our implementation does.

Initially, we intended to test what different parameters for our buffer size used for reading and writing files would do since intuitively a larger buffer means we can store more of the file for copying at one time, but we found that using a buffer size of 128 KB was the most optimal. This is because it is the size of the read ahead cache on our system. Note that all files are a multiple of 4096 bytes since that is a requirement of `O_DIRECT`. This likely will not impact any of our comparisons since this is held constant throughout all testing.

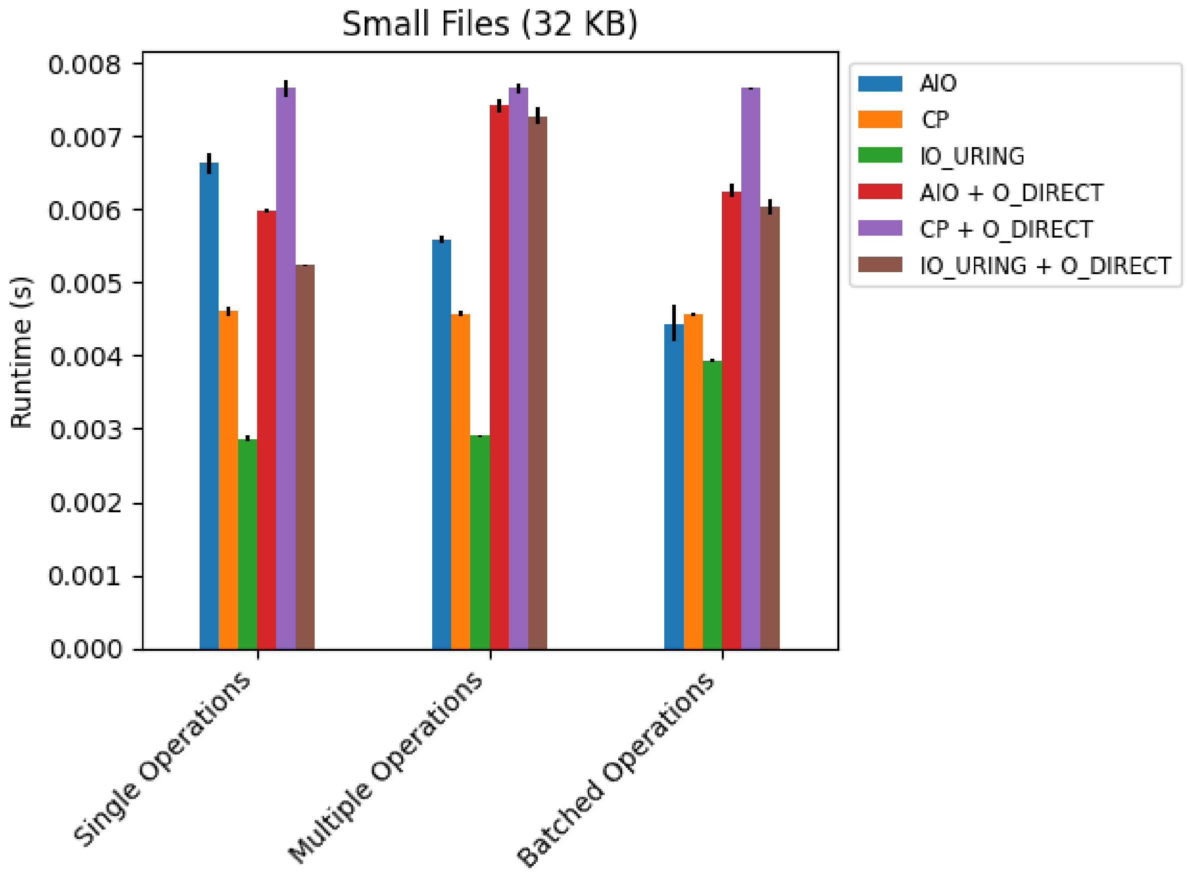
4.1 Test Environment

The operating system used to complete this lab was Ubuntu 22.04 LTS. The device used contains an Intel Core i7-8750H CPU, 16 GB of memory, 256 GB of storage, and a GeForce GTX 1070 Mobile graphics card. Code was compiled with GCC 11.2. Note that the `liburing` package required updating to Ubuntu 22.04 LTS.

4.2 Basic Evaluation of Each Variety of Operation

The first set of tests was made to evaluate how each method of copying works with the three different operation types of single operation per file, multiple operations per file, and batched operations. We varied the file sizes between 32 KB, 1 MB, and 100 MB to represent small, medium, and large files. A total of 128 files were recursively copied for all three operation types. Note that batched operations aren't supported through our implementation of synchronous I/O since this doesn't really make sense. There were also some results where using error bars of two standard deviations lead to overlap suggesting inconclusive results. These results will be called out in the discussion below. Finally, all tests were run with and without the `O_DIRECT` flag to ensure asynchronous behavior from `AIO` and comparable results for all three implementations as well as off to show absolute performance results between different usage contexts.

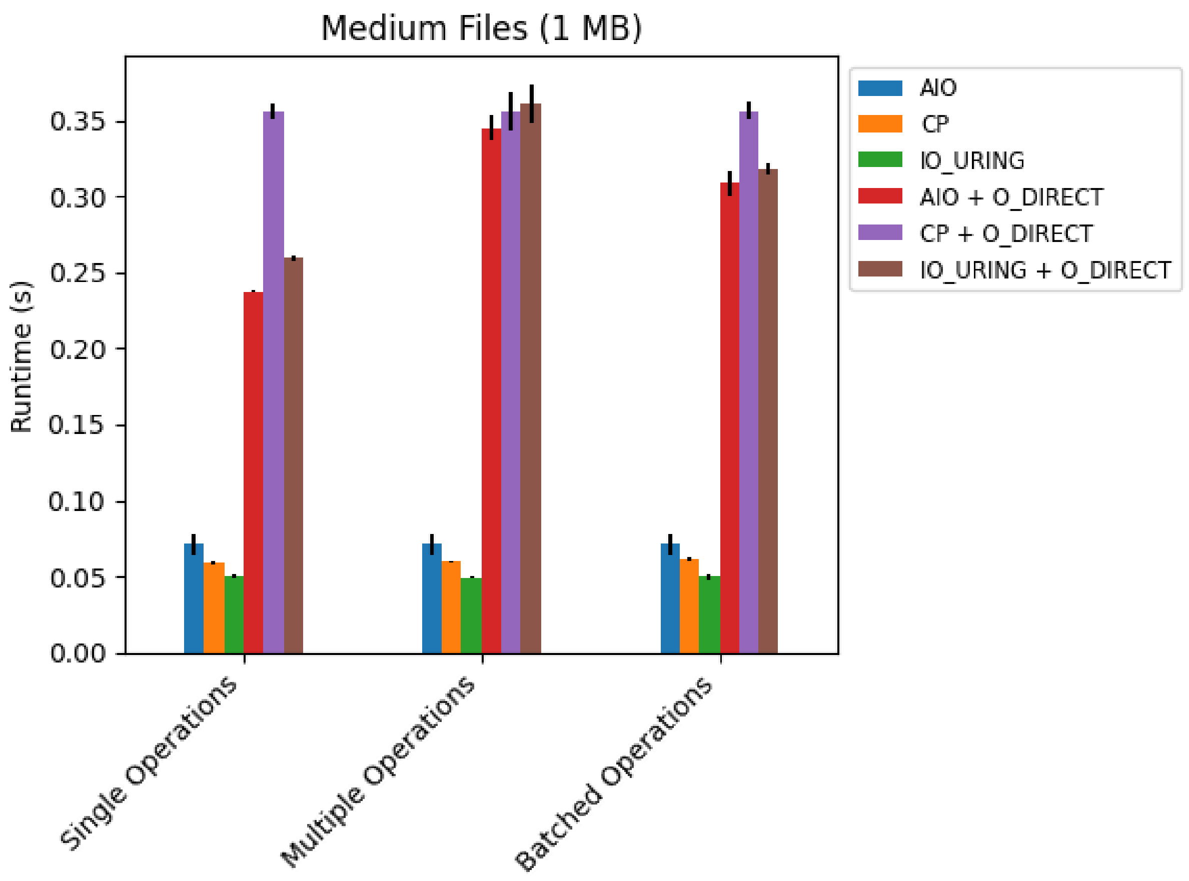
4.2.1 Small Files



For small files under normal conditions, we see that `aio` is the worst performing option for recursively copying files by a large margin, but does make improvements relative to `cp` and `io_uring` when we enable multiple operations per file and batched operations. This makes sense as `aio` does not work well for normal buffered file I/O as a result it defaults to synchronous operation. For `io_uring` we see that the trend is the reverse, it can handily beat `cp` by close to 30%! Allowing multiple operations per file has little performance benefit since each operation has a buffer size of 128KB so a file can easily fit into the entire buffer making it the same as a single operation per file. Although the use of batched operations causes performance regressions, likely due to the added latency of waiting for a batch to finish before preparing the next batch, as buffered file I/O has relatively low latency.

For small files with `O_DIRECT`, we see that overall run-times have gone up significantly as we are no longer using buffered I/O having to address storage directly. For single file operations we can see that `aio` can beat `cp` as now `aio` allows asynchronous operations. Although performance regresses if we attempt multiple operations per file. However, by batching the operations we can get back some of this performance. This is likely due the storage controller being able to better reorder I/O operations. For `io_uring` it follows the same performance characteristics as `aio` and a similar performance gain against normal `cp`. Even in direct file mode we can see that `io_uring` can outperform `aio` displaying that `aio` is not well optimized.

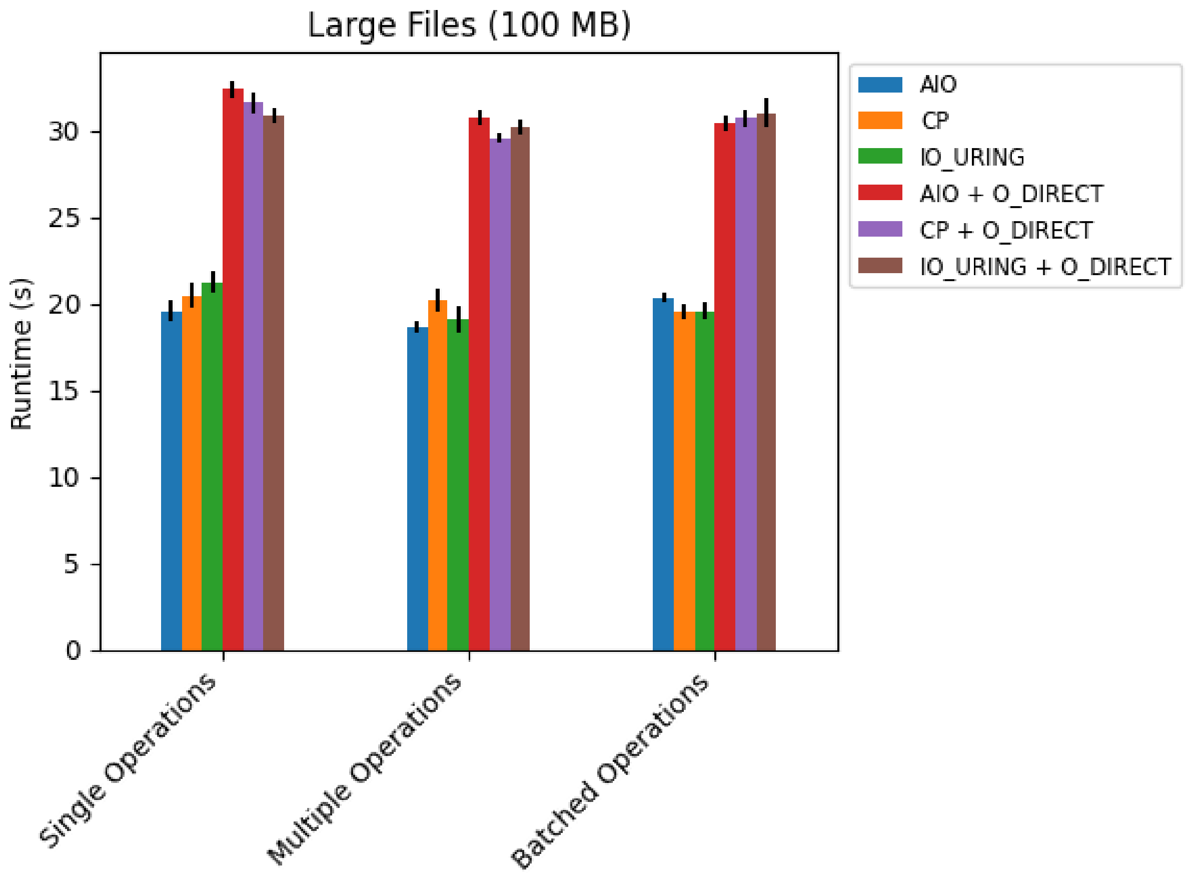
4.2.2 Medium Files



For medium sized files under `O_DIRECT`, we still see that `aio` performs better or within margin of error than the other two methods. Although in general the performance differential has decreased and we see a increase in variance. Enabling multiple operation per file once again regresses performance to the point where the error suggest there is no statistically significant difference in runtime. Once again by enabling batching we are able to improve performance but we still remain with slightly higher run times than performing a single operation per file. In these tests, we see that `io_uring` outperforms `cp` substantially especially for single operations per file, although `aio` does beat it slightly. `io_uring` follows a similar trend to `aio` when allowing multiple operations per file or batching the operations.

For medium files without `O_DIRECT`, we see results similar to small files without `O_DIRECT` enabled. Interestingly, overall the margins have tightened. `aio` still performs the worst as it defaults to synchronous operation and suffers from high variance resulting in it being the worst performing option on all three workloads, but our error bars suggest it may perform comparably to `cp` under these conditions. We now see that `io_uring` is the most performant option for recursive copy of medium files without `O_DIRECT` although the performance benefit on average has dropped to close to just 15% with little variation for each test case.

4.2.3 Large Files



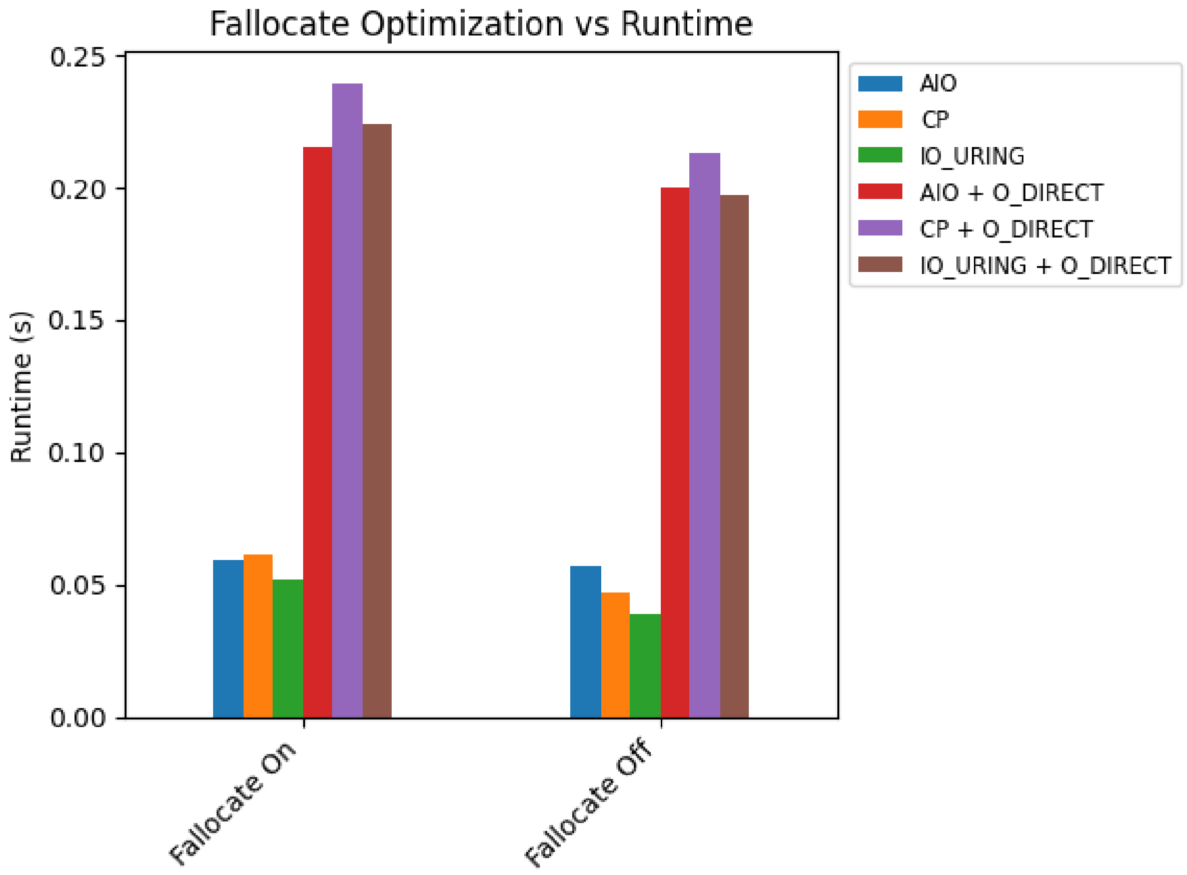
For large files under `O_DIRECT`, we see a very similar trend to medium files. One interesting trend is that we see `io_uring` perform worse than `cp` for single operations, but this is one of the few tests where we had an overlap of error bars, so we think this could be inconclusive. For multiple operations, we also had an overlap of error bars for all three methods which indicates that they may perform relatively similarly under this case. Finally, for batched operations we see a very different trend to medium files where all three methods perform almost equally with no statistically significant differences between runtimes. This was not what we were expecting as we thought multiple operations and batching would favor the asynchronous methods more as file size became larger, but it is possible that there may be more I/O related reasons such as file caching involved which causes medium sized files to be a more performant workload for `io_uring`.

For large files without `O_DIRECT`, we see some very interesting trends. The first is that in comparison to the tests with `O_DIRECT`, we see a much smaller performance gap across the different workloads. In single file operations, `io_uring` on average performs worse than normal `cp` although the high error suggests that they are all performing very similarly. Allowing multiple file operations allows `io_uring` to perform slightly better than `cp` but still within the margin of error. By batching these operations we don't gain much.

4.3 Evaluation of Different Optimization

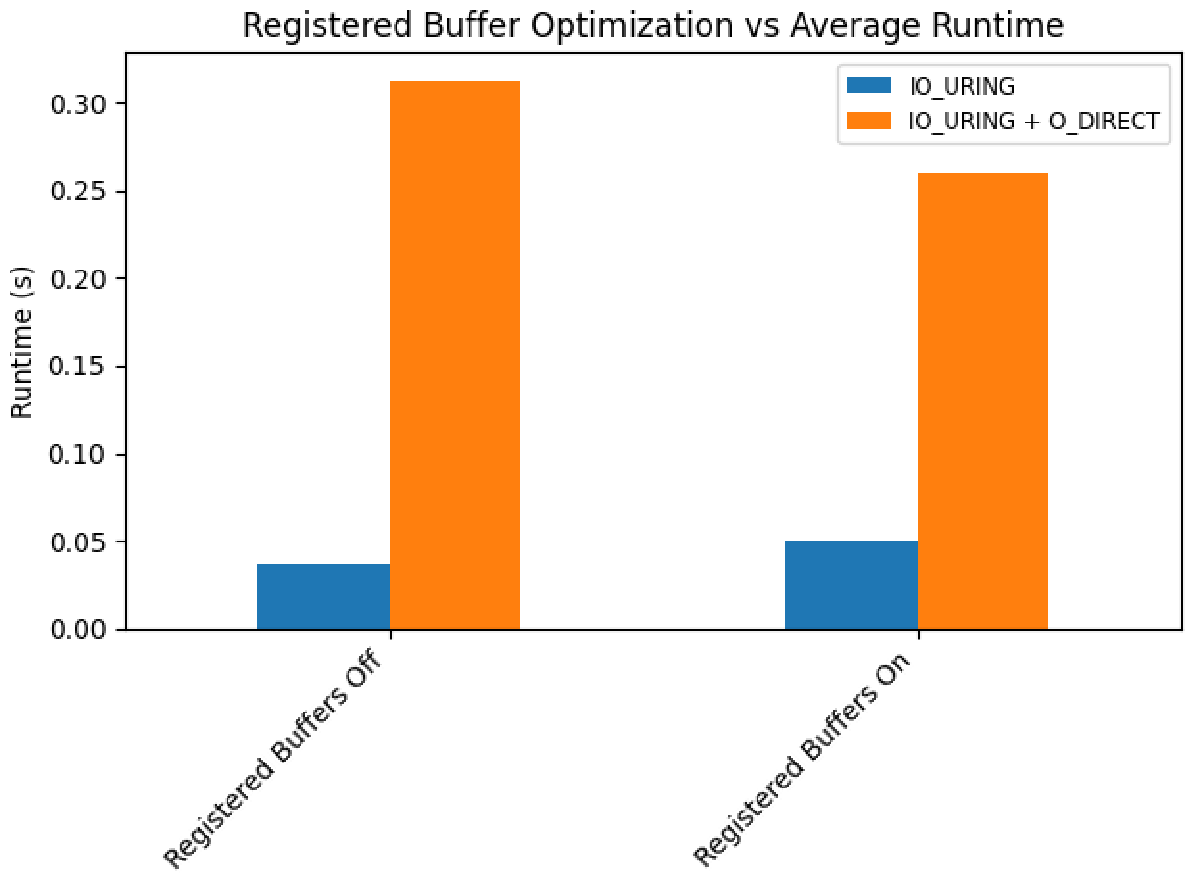
This second set of tests measures the impact of the optimizations we added to our implementations. This includes fallocate for all three implementations, registered buffers for our implementation of `io_uring`, and queue depth for `aio` and `io_uring`. These tests were performed using medium sized files and the `O_DIRECT` flag enabled and disabled to allow for `aio` to be asynchronous and get comparisons between the implementations. The graphs shown were performed with single operations, but the data did not suggest any meaningful trends by using multiple operations or batched operations that were not already discussed.

4.3.1 Fallocate



Our `fallocate` optimization preallocates blocks for the files we end up copying which should speed up the writing of the file itself when we process them inside our main loop. The results from the graph show that in all cases except `aio` under normal operation, fallocate improves performance. Under `O_DIRECT`, for `aio` we measured a 10.1% decrease in average runtime, for `cp` we measured a 11.5% decrease in average runtime, and for `io_uring` we measured a 16.2% decrease in average runtime. These are relatively consistent decreases so we think that `fallocate` benefits all three implementations in a similar way by decreasing the time spent servicing write requests. Under normal operation we see a similar improvement for `cp` and `io_uring` of 13.8% and 17.4% decrease in average runtime, respectively.

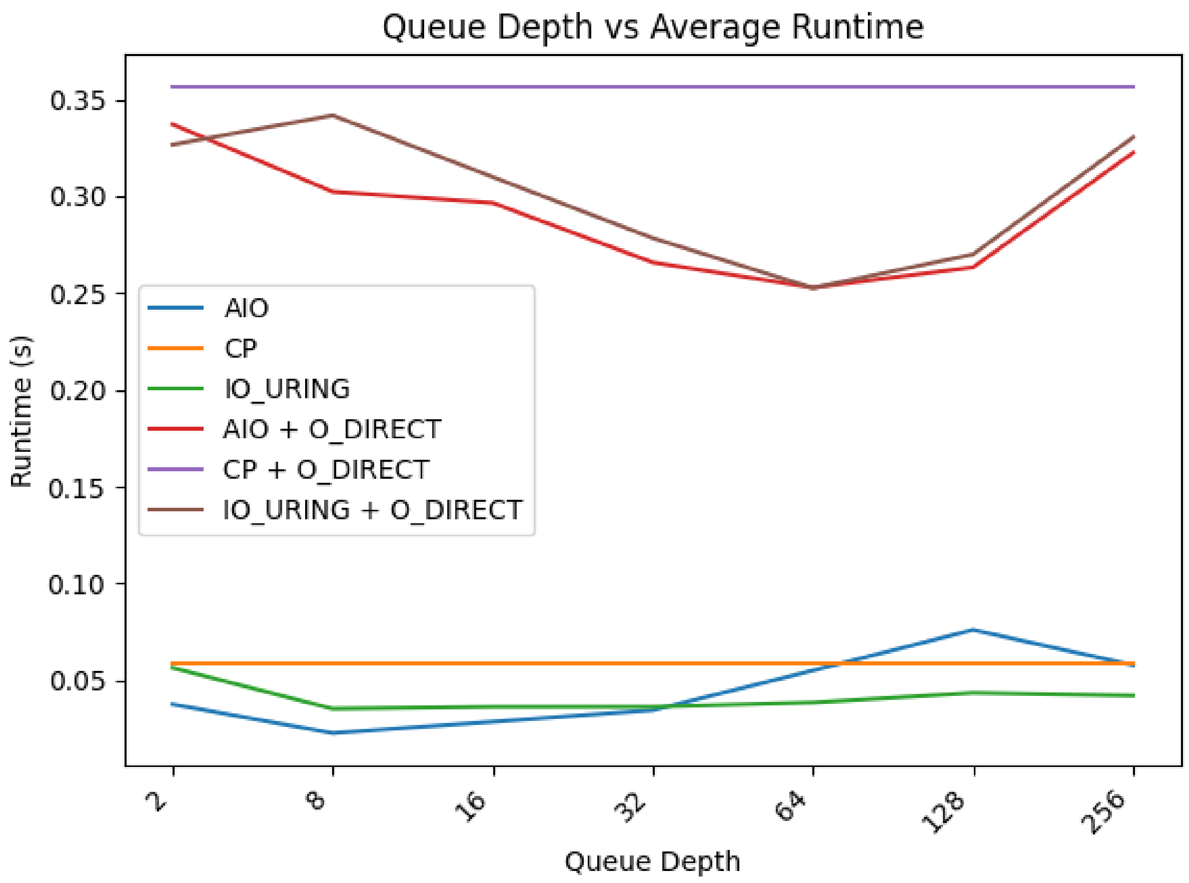
4.3.2 Registered Buffers



With `io_uring` its possible to register a set of fixed I/O buffers. This has a primary advantage when `O_DIRECT` is used. Since the kernel has to map the program's pages into the kernel before we can make I/O requests and also unmap those pages when I/O requests are completed, we can instead use a registered buffer to only have to map and unmap once per page instead of once per operation thus resulting in a 15.4% decrease in runtime. Although we see a slight regression when not using `O_DIRECT` likely due to the fact that pre-registering buffers causes additional latency that is not counteracted by the relatively low latency interactions with file buffers in comparison to directly interacting with a file.

4.4 Scalability of Queue Depth

This set of experiments tests the scalability of changing the queue depth parameter which is the number of files we have at a given time in our queue. Note that `cp` does not have a queue depth parameter since its synchronous and we handle files as they come.

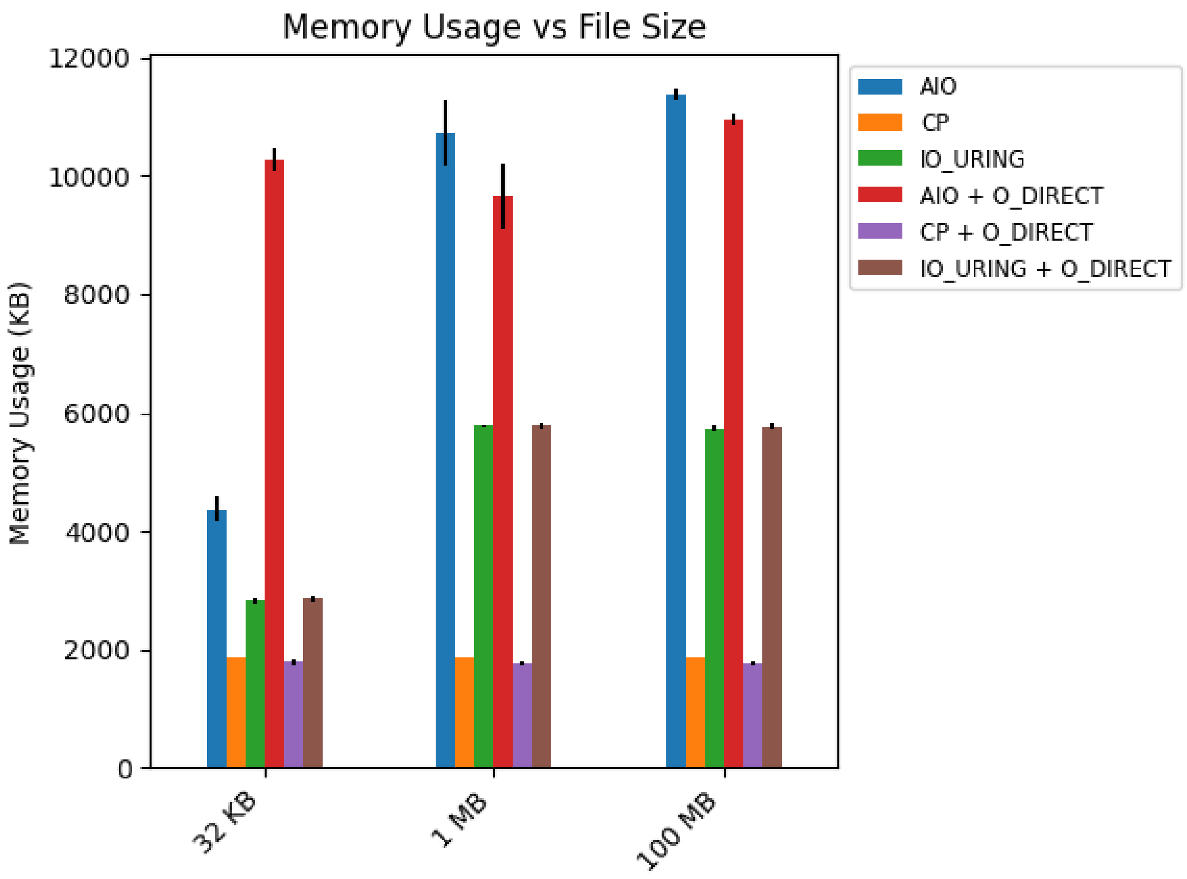


For non-direct file operations we can see that a queue depth of 8 has the best runtime and anything higher slowly tapers up or remains flat providing little benefit. This makes sense since buffered file operations are low latency enough to reuse operation entries relatively quickly and more file operations could result in higher latency reducing performance.

For direct file operations, we can see that both `aio` and `io_uring` scale well reaching a minimum runtime at around a queue depth of 64. This makes sense since the file operations will have higher latency if we can schedule more operations to occur at the same time we can effectively get better performance.

4.5 Memory Usage

Memory usage was measured for all tests using maximum resident set size, but we noticed that none of our optimizations of alternate workloads caused a significant difference in memory usage and the only factor that contributed was file size. The following tests were run under normal operation and `O_DIRECT` enabled under single operations per file for this reason.



This graph shows that `cp` remains consistent in its memory usage over all tests. This makes sense as it simply calls the `copy_file_range` syscall. We see that `aio` under normal operation has a drastic increase in memory usage when moving from small files to medium files and a less significant increase when moving from medium files to large files. This trend is different under `O_DIRECT` where we see a relatively high memory usage regardless of file size. Finally, `io_uring` shows no change whether running under normal operation or with the `O_DIRECT` flag set but does see an increase from small files to medium files. Similarly to `aio` it does not see a major increase between medium files and large files. This makes sense as even as we get into larger file sizes we still have a maximum amount of memory we are using to populate what portion of the files we are using to make requests to I/O. These requests must be filled before we start populating more files which allows us to remove old files from memory.

5 Issues/Future Implementation

Some improvements that can be made with our implementation include error recovery and redundancy for failed I/O requests. In our implementation failed I/O requests cause the file in question to not be present in the destination directory or stop the copy operations. We could add code that retries these failed attempts in a more usable implementation of the system. Another improvement that can be made is for the user experience. Currently there is no way to know the progress of a copy and for some workloads where there is a need to copy a large amount of large files, there is no way to estimate the time left. We could include a progress bar or some other tool that will help with the user experience.

Some additional tests that could be run is testing with different size memory buffers. This could lead to better performance as we can keep more pages in memory while reading and writing the same portions of files. Finally, all tests were performed under `ext4` file system.

Future work could test these results on other file systems to see if there are advantages and disadvantages in the way these methods interact with them. One more test that could be useful is to test more variable amounts of files than 128, as its possible the total amount of files could impact these trends heavily.

6 Conclusion

We found that our hypotheses were mostly correct in that `aio` is a viable alternative for recursively copying files when using `O_DIRECT`, but that `io_uring` does outperform it in every workload we tested. We also see that `cp -r` is equally performant in many workloads to `aio` and `io_uring` and does perform better than `aio` for almost all tests under normal operation. This could be useful in cases where the device is memory constrained since `cp` uses the least amount of memory by a substantial margin, although the total memory usages are relatively small and should be sufficient for most cases. Finally we see that `io_uring` accomplishes its goal of being an improvement of `aio` in both normal operation and with `O_DIRECT` enabled.