Asynchronous I/O for `cp -r`

Vincent Lee
Gualberto A. Guzman

December 8, 2017

1 Goals

Our project proposal is to use Linux's asynchronous IO interfaces (aio) to build an optimized version of `cp -r`. We intend to demonstrate differences in the effects of caching, readahead, and other factors on aio performance, especially as they relate to SSDs vs HDDs. Our implementation takes an existing directory name and a new name and copies the entire directory tree to the location specified. We achieve competitive results within 1.5x the speed of native `cp -r`.

2 Major Decisions

There were two major choices available to us for asynchronous file IO on Linux. The first was the **POSIX AIO** interface, which is standardized as a set of library calls across all POSIX-compliant environments. However, according to the man pages, `glibc` simulates **POSIX AIO** completely in userspace [1]. This means that the kernel is not able to make scheduling and reordering decisions that may affect performance. Instead, we decided to use the nonportable **Linux AIO** system calls.

For the implementation itself, **C++** was chosen as the programming language. In addition to being a fast systems programming language, it also provides useful abstractions and standard libraries, as well as access to the Boost libraries. The Boost Filesystem library was used for its `recursive_directory_iterator`, an iterator which yields all paths along a breadth-first traversal from a given root path. For each path, if the file is small enough, it is copied directly using a Boost call. This is because, below a certain point, the overhead of initializing and submitting an asynchronous IO task dominates the cost of simply using `read` and `write`. This threshold is configurable and the effects of varying it are explored in section 5.

3 Implementation

After parsing all arguments, `acpr` initializes a recursive directory iterator from Boost, which yields a breadth-first traversal of every path in the given source directory. For each path, we run `lstat` to discover if it is a file or directory. If a directory, we create it immediately in the output path with `mkdir`. Since the iterator is breadth-first, we are guaranteed that all parent directories exist at the time of calling. If a file and it is below the AIO threshold, we copy it using a Boost utility function, `boost::filesystem::copy_file`, implemented with `read` and `write`. Otherwise, we use the AIO system.

3.1 AIO Copying

A copy of a single file using AIO is represented by a class `CopyTask`. This class encapsulates the state of each copy, as well as a fixed number of `iocb`'s. An `iocb` is the AIO system's unit of work. Each time `CopyTask::schedule_next_read` is called, we scan to see if there are any free `iocb`'s, and attach to each of them offsets to read, the file descriptor to read from, a buffer to read into, and a callback to invoke when complete. Once enough `iocb`'s have been buffered, the system call `io_submit` is invoked, passing all initialized `iocb`'s to the kernel.

At the end of each loop of the recursive directory iterator, another system call, `io_getevents`, is called to gather any completed reads and writes from the kernel and invoke their callbacks. This call takes a timeout that represents the maximum time the user is willing to wait for the kernel to give it events. Whenever a callback for a write is invoked, it invokes `CopyTask::schedule_next_read`. The `CopyTask` corresponding to each `iocb` is kept in a private lookup table, enabling it to be recovered from the callbacks, which are global function pointers. Whenever a callback for a read is invoked, it simply reinitializes the `iocb` into a write, changes the file descriptor to the output file, and immediately resubmits it. Here, we potentially could have performed batching of writes to reduce the number of system calls, but due to complexity concerns and time limits, decided to leave it as-is although it is a issue for future work.

After the directory iterator finishes, the event queue is then drained by calling `io_getevents` in a loop with unlimited timeout until the `iocb` to `CopyTask` lookup table is empty, meaning all `CopyTask`s have completed.

4 Challenges and Limitations

Our first challenge was dealing with the lack of proper documentation for the Linux AIO API. The user space library was poorly documented in general and has not been updated since 2015; we had to find an Ubuntu man page from 2012 in order to see examples of its usage [2].

Secondly, A function critical to our implementation, `boost::filesystem::relativize`, was missing in the version of Boost installed on the lab computers (1.58). This function takes two absolute paths and makes one into a path relative to the other, so that it can be appended to the destination directory. We backported this function from a later Boost version (1.60) by simply copying the relatively small implementation (160 lines) from the Boost source control [3].

In addition, the userspace header library that wrapped the Linux AIO system also implemented a particular call naïvely. `io_queue_run` is a function that polls the kernel event queues and runs callbacks. However, we looked at the implementation and discovered that it was receiving events one by one, making a separate system call for every single one. We replaced calls to this function with direct system calls to read the event queue ourselves, reducing the amount of system calls needed to process the same number of events.

During early testing, we attempted to copy the Linux 4.4.97 kernel tree on an SSD, a 711 megabyte repository of mostly small files. `acpr` used over 7 minutes before it was killed, while `cp -r` used only 1 second. Some profiling indicated that the AIO subsystem finished very quickly, it was the `fsync` call for each file that slowed everything down. Disabling the `fsync` by default sped the copy up to about 3 seconds. We later verified that `cp` also chose not to call it by observing the system calls it made using `strace`.

Currently, attributes and ownership are not completely copied from the source to the destination. In addition, due to complications arising from relative symbolic links, all symbolic links are skipped when copying.

The event queue is processed with the single calling thread. A potential optimization would have been to have a separate thread driving the event queue while the main thread iterates the directory, but for simplicity, `acpr` is limited to a single-thread implementation.

5 Evaluation

`acpr` was evaluated on two main platforms types, two laptops running Linux with an SSD, and a csres server machine. The test script first runs `cp -r` on the directories under test, then runs `acpr` to another destination directory, and times both. The test script also has the ability to sync and clear the buffer cache by calling `echo 3 > /proc/sys/vm/drop_caches`, which we opt to do for almost all cases below, to test the AIO system's ability to reorder requests effectively for IO. After copying, the source and destination directories were compared using `diff -r`, which performs a deep diff of the entire subtree to find corrupt, absent, or extra files from the copy.

We test two major cases, copying a large file (Lubuntu ISO, 880MB) and copying the Linux 4.4.97 (771MB) source tree. The first exercises `acpr`'s ability to fully utilize the disk's speed, while the second emulates real-world scenarios of copying subtrees filled with files of varying sizes. In all cases, we average completion time in seconds over ten trials on a relative noise-free machine. We note, however, that since the Linux kernel contains some relative symlinks, which we skip, the test script's error-checking will report differences. We have manually verified using `diff -r` that `acpr` is copying the output correctly.

5.1 Thinkpad X1 Carbon & Macbook Pro

The first laptop is a 2013 Thinkpad X1 Carbon, with Arch Linux, updated to the latest packages as of December 5, 2017. The laptop has an Intel i7-4600U processor clocked at 2.1 GHz, 8 GB of RAM, and 4 GB of swap space configured. The tests were run on the owner's home directory, which is on a local ext4 partition on a Samsung MZNTD256 SSD. The second laptop in a Mid-2015 Macbook Pro (version 11,5) running Ubuntu Linux 16.04.2. It has a 2.5 GHz Intel Core i7-4870HQ, 16GB of RAM, and 4GB of swap space. As in the first case, all tests were run on the owner's home directory, on a local ex4 partition on an Apple SM0512G SSD.

5.1.1 Copying a Single Large File

First, we compare a run of `acpr` with all default options with `cp -r` in Figure 1 and Figure 2. For the Thinkpad, `cp -r` copies the ISO in about 3.5 seconds on average, coming to an average of 251 MB/s throughput. `acpr` is weaker, copying the ISO in about 9 seconds, coming to an average of 96.9 MB/s throughput. However, if `acpr` calls `fallocate` to preallocate the destination file and `readahead` on the source file, then it can copy the ISO in 3.75 seconds, a throughput of 234.6 MB/s. For the Macbook Pro, `cp -r` copies the ISO in about 600 ms, coming to an average of 1478 MB/s throughput. In this case, calling `fallocate` and `readahead` does not improve performance much beyond 1.1s with a throughput of 833 MB/s. In comparison to the Thinkpad, the Macbook Pro benefits greatly from the increased bus speed as shown by the tighter errors bars in each case. However, in both cases we cannot improve beyond native `cp -r` performance. This is likely because the AIO system is more indirect, and the system is perhaps unable to detect that a sequential read is occurring, unlike `cp -r` where the repeated calls to `read` and `write` make it very clear that a sequential copy is occurring.

Next, we run the copy with larger block sizes. By default, `acpr` copies 64K per AIO operation. Increasing the size may not make much difference if the files are smaller than 64K anyway, but

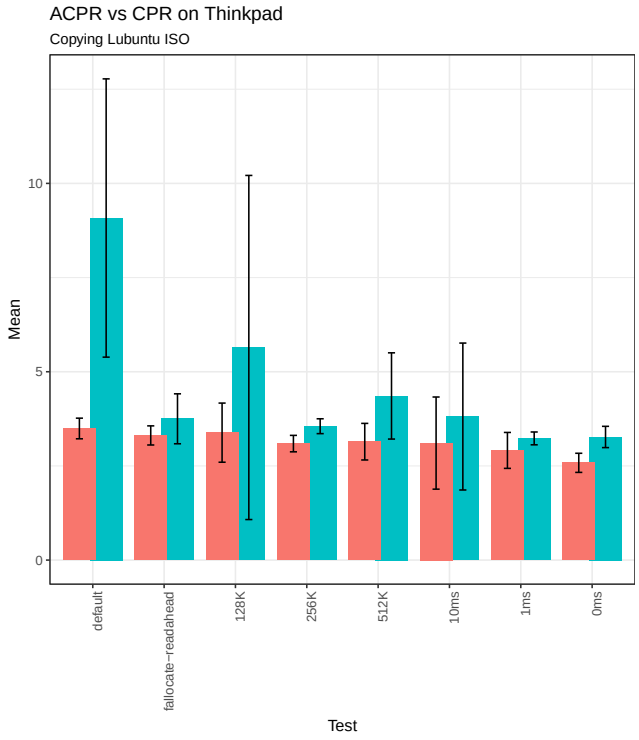

Figure 1: Copying Lubuntu on Thinkpad

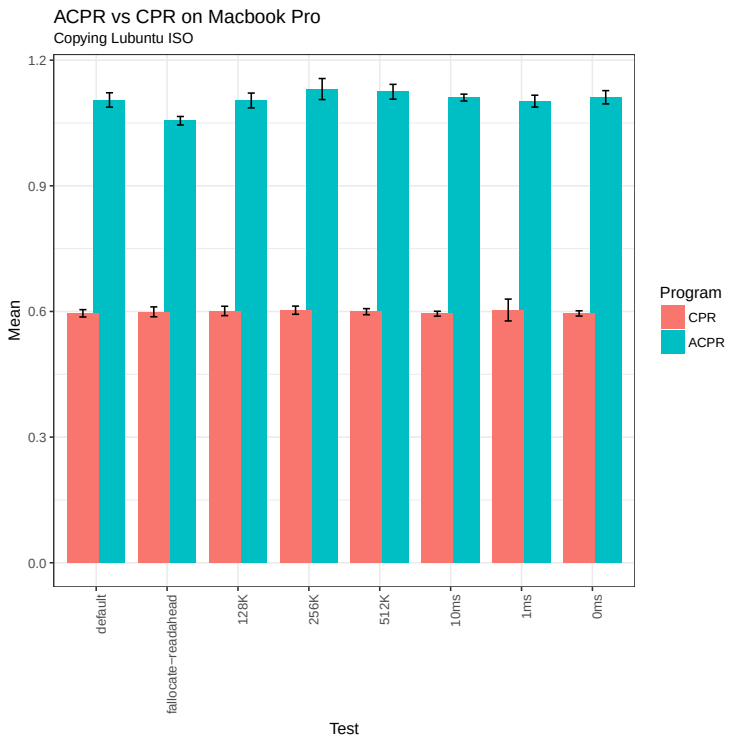

Figure 2: Copying Lubuntu on Macbook Pro

should decrease the number of system calls and improve locality of the buffers when copying a single large file. When increasing the block size to 128K on the Thinkpad, runtime decreases from the baseline of 9 seconds to 5.64 seconds and a throughput of 155.9 MB/s. Further increasing the block size to 256K again improves the performance to 3.55s and 247.6 MB/s. However, increasing again to 512K yields a performance drop to 4.35s and 201.9 MB/s as seen in Figure 1. On the Macbook Pro, increasing the block size beyond the default of 1.1s and 769 MB/s throughput results in a performance decrease of 1.13 s and 782 MB/s throughput. As shown in Figure 2, the small variation in results from varying the block size do not indicate a concrete performance gain. This is probably due to overhead allocating and copying such large buffers across the user-kernel boundary.

Finally, we test what happens if we vary the timeout when gathering events. By default, `acpr` allows the kernel 5 ms to respond with events, yielding the baseline measurement of 9 seconds and 96.9 MB/s. Increasing this timeout to 10 ms improves performance to 3.81 seconds and 230.9 MB/s, while decreasing it to 1 ms also improves performance in a similar manner to 3.23 seconds and 272.4 MB/s for the Thinkpad. Eliminating the timeout (passing 0 to the kernel) also gives a similar boost to 3.26s and 269.2 MB/s. The effect is similarly negligible on the Macbook Pro, from 1.1s and 796 MB/s throughput to 1.11s and 792 MB/s. It is not clear from Figures 1 and 2 that varying the timeout had much of a direct effect on `acpr`'s speed, since there is no clear pattern of change. A possible source of error could be background system noise, though only the terminal emulator and LaTeX editor were running during the experiments on both machines.

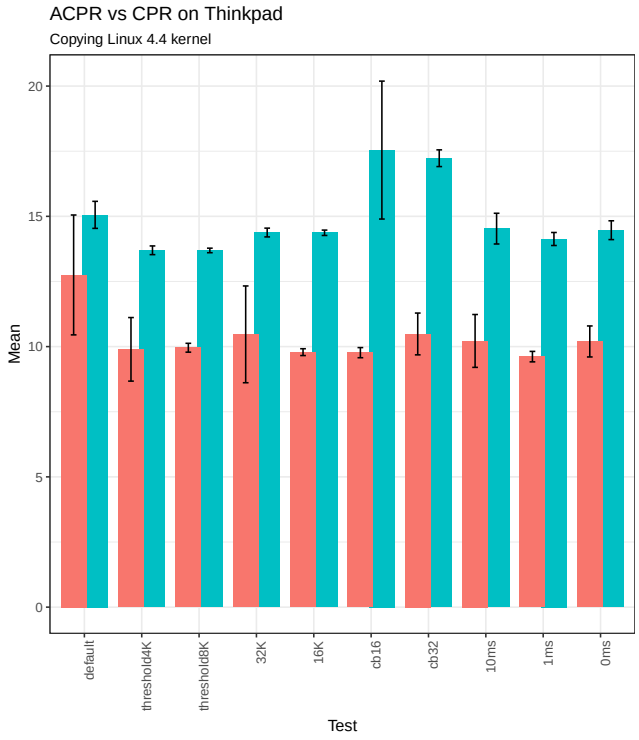

Figure 3: Copying Linux 4.4 on Thinkpad

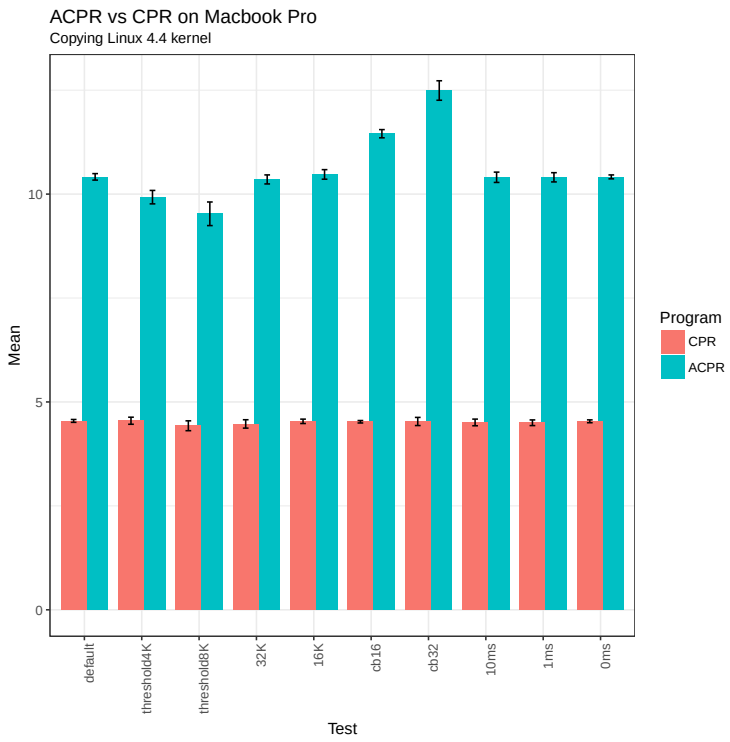

Figure 4: Copying Linux 4.4 on Macbook Pro

5.1.2 Copying Linux Source Tree

First, we compare a run of `acpr` with all default options vs. `cp -r` on the Thinkpad. `cp -r` copies the 711 MB source tree in 12.75s for 46.48 MB/s throughput, while `acpr` manages 15.05s for 39.3 MB/s throughput as seen in Figure 3. On the Macbook Pro, we have that `cp -r` copies the source tree in 4.5s with a throughput of 156 MB/s as shown in Figure 4. However, `acpr` copies the source tree in 10.4s and throughput of 68 MB/s. Note how much lower this is than copying a single file, since there is much more filesystem overhead to `open`, `stat`, and `close` each individual file as opposed to just one.

The Linux source tree is composed of many different sizes of file. The default threshold for `acpr` to use AIO is 1K, which may have too high of an overhead. We instead tried raising the threshold for using AIO to 4K, which would still copy 25984 files in the source tree using AIO. Doing this sped up execution to 13.69s and 43.2 MB/s on the Thinkpad (Figure 3). Further raising the threshold to 8K does not give a similar boost, remaining at 13.69s and 43.3 MB/s. For the Macbook Pro, changing the threshold to 4k slightly increases performance to 9.9s and 71 MB/s throughput while doubling the threshold to 8k results in another performance increase to 9.5s and 74 MB/s. However, these numbers are still twice as slow as the `cp -r` performance of 4.5s and 159 MB/s (Figure 4). This is as expected, since the number of files larger than the threshold decreases drastically as the threshold increases.

The second experiment was to use a *smaller* block size. Since many of the files are source files that are a handful of KB large, the default block size of 64K may have resulted in unnecessary overhead allocating space that ends up going unused. Reducing the block size to 32K yields a small speedup to 14.37s and 41.22 MB/s; reducing further to 16K does not produce further improvements,

staying at 14.37s and 41.25 MB/s, likely because the difference in overhead becomes negligible on the Thinkpad (Figure 3). On the Macbook Pro, reducing the block size to 32k yields a similar negligible speedup to 10.35s and 68 MB/s. Going further to 16K actually decreases performance to 10.47s and 67 MB/s as seen in Figure 4.

The third experiment was to vary the number of `iocb`'s used in copying each file. The default is 5, and `acpr` waits until it has gathered 5 initialized `iocb`'s before it submits them to the kernel. So theoretically, raising the number used should reduce the number of system calls made, as they are batched together more. However, we saw reduced performance on the Thinkpad - for 16 `iocb`'s per file, we got 17.54s and 33.79 MB/s; for 32, we got 17.23s and 34.4 MB/s (Figure 3). Similarly, Figure 4 exhibits the reduced performance on the Macbook Pro for 16 `iocb`'s (11.45s and 62 MB/s) as well as for 32 (12.49s and 56.9 MB/s). This is due to the large (64k or larger) buffers that must be allocated and freed for each `iocb` of each copy task.

Our final experiment for the laptops was to vary the timeout when gathering events. Changing the default of 5 ms to 10 ms on the Thinkpad gives a negligible performance boost to 14.52s and 40.79 MB/s; lowering it to 1 ms does the same: 14.13s and 41.949 MB/s, as does eliminating the timeout completely: 14.46s and 40.97 MB/s (Figure 3). By constrast, Figure 4 demonstrates no significant change to the default performance of 10.4s and 68 MB/s. Combined with the results from copying the Lubuntu ISO, it seems that in the cases we have set up, the timeout passed to the kernel when retrieving events does not matter, meaning that the kernel is almost always ready with events for us to process. This potentially could matter if we had extended `acpr` to use multiple threads for event processing.

5.2 csres

The csres server machines run Ubuntu LTS 16.04.3, and have 32 Intel Xeon E5-2620 v4 processors clocked at 2.1 GHz, 126 GB of RAM, and 128 GB of swap space configured. The tests were run on the `/var/tmp` directory, which is on a local ext4 partition, instead of the NFS home directories. The partition is on a Seagate 7200RPM enterprise-grade hard disk.

5.2.1 Copying a Single Large File

The same test cases as above were performed on `zerberus.csres.utexas.edu`. Overall throughput was lower, as expected, with the default configuration taking 5.86s and 150.18 MB/s. `cp -r` barely did better at 5.82s and 151.15 MB/s (Figure 5).

Employing `fallocate` and `readahead` actually slowed performance down to 5.93s and 148.40 MB/s, though the difference is well within deviation. For the rest of the experiments (adjusting the block size and timeout wait), the results remained very close to the default configuration seen in Figure 5. We attribute this to a bottleneck that is external to both `cp` and `acpr` - disk latency.

To test this, we ran the baseline and `fallocate` cases again, without the flag that calls `sync` and `echo 3 > /proc/sys/vm/drop_caches` every iteration. As expected, throughput was much higher as our calls were now being serviced straight from memory. What was more surprising, however, was that `acpr` had a clear speedup over `cp -r` in both cases. In the default configuration with no syncing, `cp -r` manages 1.49s and 589.59 MB/s while `acpr` finishes in 0.9s and 976 MB/s. With `fallocate` and `readahead`, `cp -r` finishes in 0.85s and 1033.97 MB/s and `acpr` in 0.82s and 1068 MB/s as seen in Figure 5. We attribute these results to the fact that the AIO system now has the flexibility to schedule the disk reads and writes for efficiency as well as the relatively large buffer cache on the csres machines. We were unable to replicate similar results on our laptops, possibly due to their smaller memories and the nature of SSD accesses.

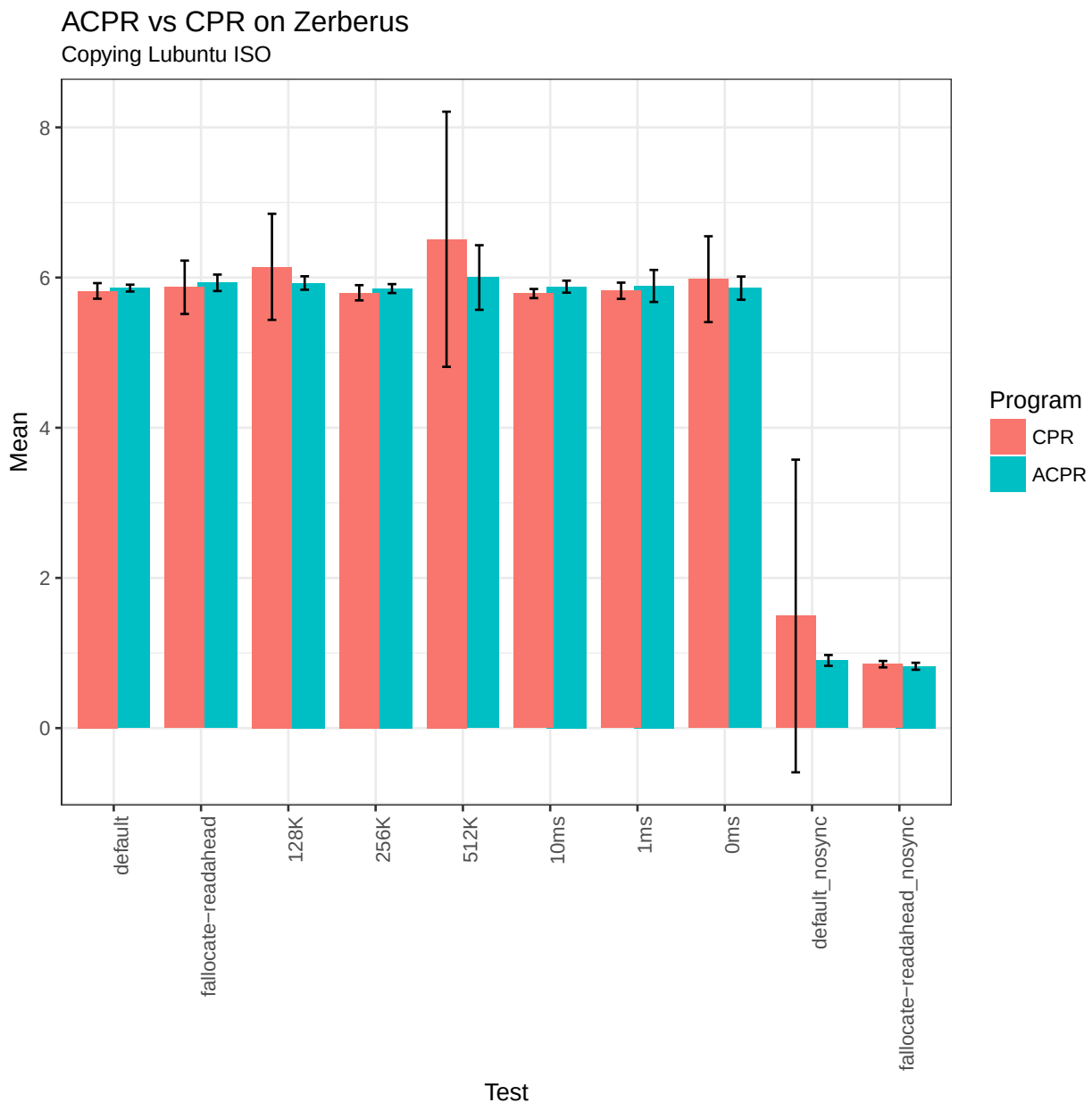

Figure 5: Copying Lubuntu on Zerberus

5.2.2 Copying Linux Source Tree

Both `cp -r` and `acpr` performed quite poorly copying the Linux 4.4.97 kernel source tree, with `cp -r` taking 14.19s and 41.77 MB/s, and `acpr` taking 34.6s and 17.11 MB/s. While increasing the threshold of using AIO did improve performance, the change was miniscule. Likewise, for all other cases (adjusting block size, adjusting `iocb` count, and adjusting timeout), there was no significant change from the default configuration as shown in Figure 6. Again, we attribute this to factors beyond both programs' control. `acpr` is probably slower because the kernel contains many small files, so the overhead of setting up AIO, allocating buffers, etc. dominates the time to actually transfer the data.

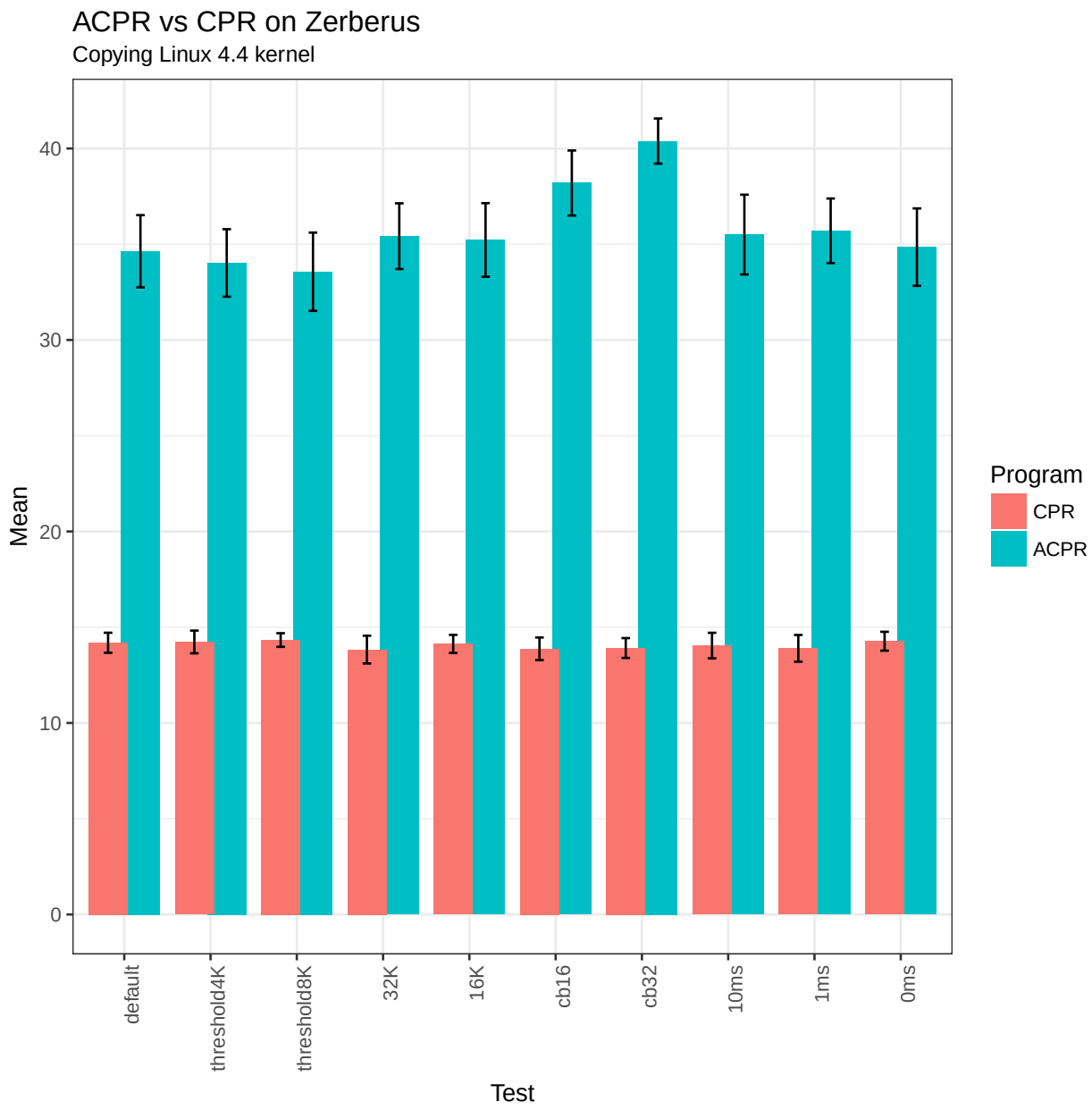

Figure 6: Copying Linux 4.4 on Zerberus

6 Conclusion

We have produced a prototype of a `cp -r` clone that uses Linux's Asynchronous IO interface to perform its data transfers. While it does not outperform `cp -r` except in two cases, it remains within acceptable performance bounds, especially on SSD hardware. Further improvements would include a more robust event handling system with multiple threads, support for symbolic links, support for attribute copying, batching of write task submissions, and a customized `copy_file` implementation.

7 Time Spent

About 1.5 hours were spent researching the state of asynchronous IO, finding documentation for the chosen libraries, and setting up the build system of the project. About 8 hours were spent overall creating the acpr program itself, including planning, programming, and optimization. About 10 hours were spent creating the test harness, test cases, executing test cases, and collecting the data. About 6 hours were spent creating the plots, report, and presentation. In total, that is 25.5 hours spent on this project.

References

[1] aio (7), http://man7.org/linux/man-pages/man7/aio.7.html

[2] Asynchornous IO, http://manpages.ubuntu.com/manpages/precise/en/man3/io.3.html

[3] Boost Source Control, https://github.com/boostorg/boost