CS380L Final Project: Asynchronous cp -r

Aaron Chang

9 May 2019

Time spent on lab: 61 hours

1 Background

I attempted to build an optimized recursive copying program. The Linux utility `cp` has an `-r` flag which causes it to recursively copy a directory tree to a new directory. I set out to emulate this behavior using Linux's aio (asynchronous I/O) library to improve performance over the single-threaded version. The hypothesis here is that issuing multiple writes asynchronously may yield greater I/O performance than issuing and waiting on single reads and writes at a time. The baseline functionality required of my test program is that it correctly copies all required files from the source directory. Success will be measured in time taken to copy the files compared to `cp`.

I chose to implement recursive copy twice, once with Linux aio, and once with POSIX aio. POSIX aio is a user-level implementation that executes regular blocking I/O using multiple threads, giving the illusion that I/O is asynchronous to the main program. My intuition is that this library may not yield much performance gain because it is still using the buffer cache (similar to `cp`).

Linux aio is a kernel-level library that enqueues I/O requests in the kernel and directly accesses disk, bypassing the cache and writing/reading straight to the user-space buffer. This may yield better performance in some cases, but comes with many restrictions on which types of calls are possible, how the file is opened, etc.

2 Design Overview

For both Linux aio and POSIX aio, the design is similar. The copying program needs to traverse the source directory, creating subdirectories as needed and issuing asynchronous reads for files. Once all of the writes are issued, the program either polls or is notified of completion of I/O requests. It can then issue the corresponding write for each finished read. Lastly, it checks the status of all I/O requests until they complete, which indicates that all files were copied successfully.

3 Implementation

3.1 `acp`

`acp.c` contains an asynchronous copy program written using the POSIX (user-level) aio library. The program contains code to abort on receipt of SIGQUIT. To set up and track I/O requests, it creates ioRequest and aiocb (aio control block) arrays. It then calls `nftw()` (passing a flag to avoid following symbolic links) which performs a file tree walk beginning from the source directory, and calls `dispatch_read()` (a

provided function pointer) for every file encountered. Inside `dispatch_read()`, if the file is a directory, it is created using a modified path to the source directory with full permissions using `mkdir()`. The permissions will be masked with the calling processes' permissions, so the end result is the correct permissions on the new directory. If `dispatch_write()` encounters a regular file, it sets up an ioRequest to monitor the index and status of that file, then fills in the `aiocb` structure with the correct file descriptor, buffer to read into, size, and offset. `aio_read()` then dispatches the I/O request, `dispatch_read()` then increments the number of open read requests, and the directory traversal can continue.

In the main function, after all reads have been issues by the directory traversal, it enter into a monitoring loop that periodically (every 1/4 second) checks the status and/or errors of the dispatched reads. If it finds a read ioRequest that has finished, it calls `dispatch_write()` to spawn a corresponding write to the copy location. `dispatch_write()` performs analogous work to `dispatch_read()`. Once the main loop detects that all writes have been issued, it can enter into the final monitoring loop to check completion of all the writes. Once that loop finishes, all files have been copied.

3.2 lacp

The structure for the Linux aio version is similar. `nftw` traverses the directory structure and dispatches read requests. One difference is that Linux aio requires the `O_DIRECT` flag to perform direct disk accesses. `open()` with `O_DIRECT` fails unless the I/O accesses to the resulting file descriptor are aligned to block boundaries, 4096 bytes in this case. The `malloc_aligned` function returns a memory-aligned buffer for `iocb` (I/O control block) purposes. Because the `iocb` does not contain the path for the targeted file, the program records both the original full path and the to-be-copied full path in their respective arrays.

The monitoring loop is also modified. `io_getevents` does not indicate which of the read requests it is returning, only that one request has completed. In `dispatch_write()` the program uses the file descriptor in the returned `iocb` and calls `readlink()` on `/proc/self/fd/NUM` where `NUM` is the desired file descriptor. `realpath()` then yields the full original path, which is used to compare and index the previously stored copy paths. Again, once all the read requests have been processed with corresponding writes, the main function can check for all writes to finish, completing the copy.

4 Roadblocks

Memory was a huge issue in trying to scale up my tests. Because each read request requires a malloc'ed buffer, my computer can quickly run out of memory on the larger tests, dragging everything to a crawl. POSIX aio is also thread-based, so at large numbers of files my computer slows down. The POSIX aio version did not exhibit any true performance gain over `cp`, as expected. There is precious little documentation other than the man pages, and much time was spent trying to debug pointers, pointers to structs, garbage memory, and struct pointers with pointers to structs. The asynchronous functions also did not usually return useful error codes, so debugging was a painstaking ordeal slogging through valgrind and gdb. Running on a machine with more cores and much more memory would have been ideal for testing more extreme edge cases.

5 Evaluation

5.1 Setup

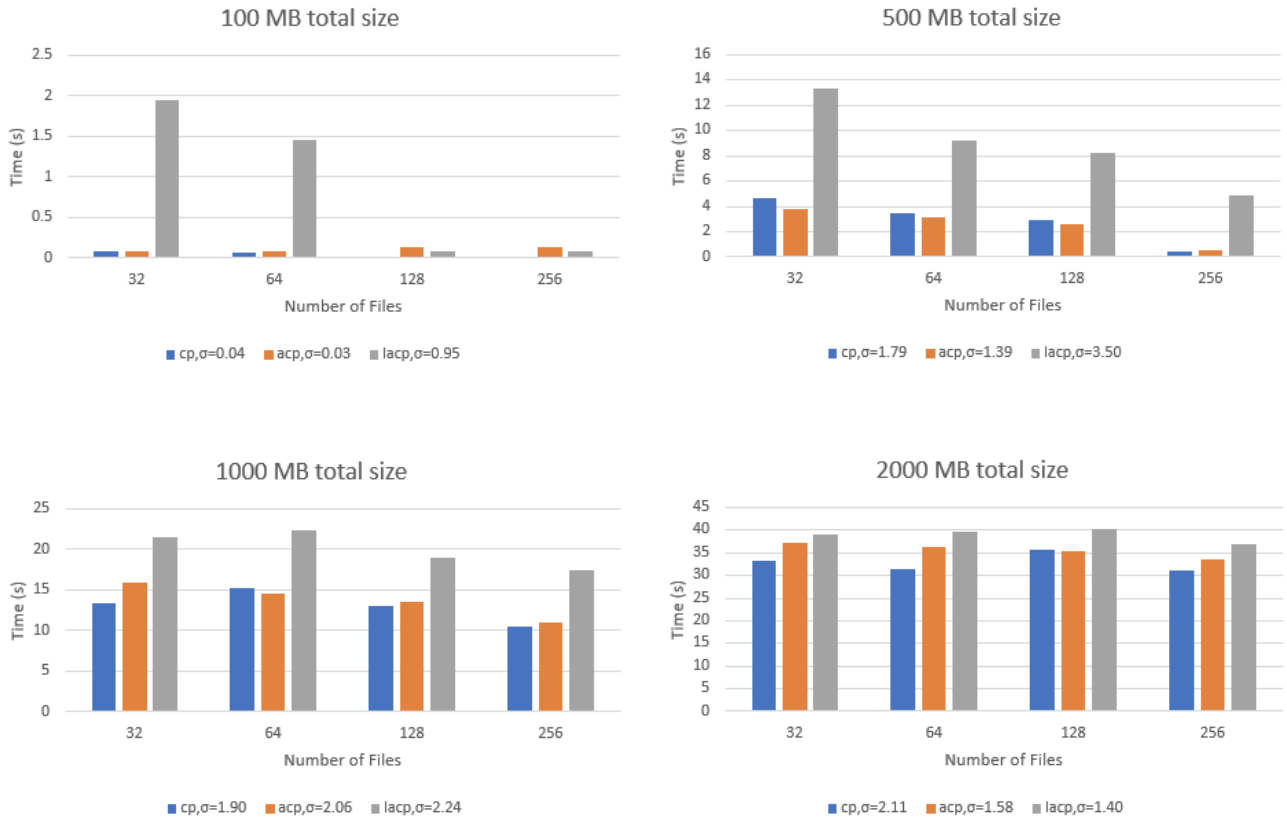
`create_dir_tree.py` contains a script for creating and writing a source directory with one subdirectory, specifying total combined size of files (in MB) and total number of files desired. `*.sh` files contain bash

scripts to run each configuration and display each execution time. The makefile contains commands to clean, build, and run the tests. All caches are flushed before every run.

5.2 Baseline

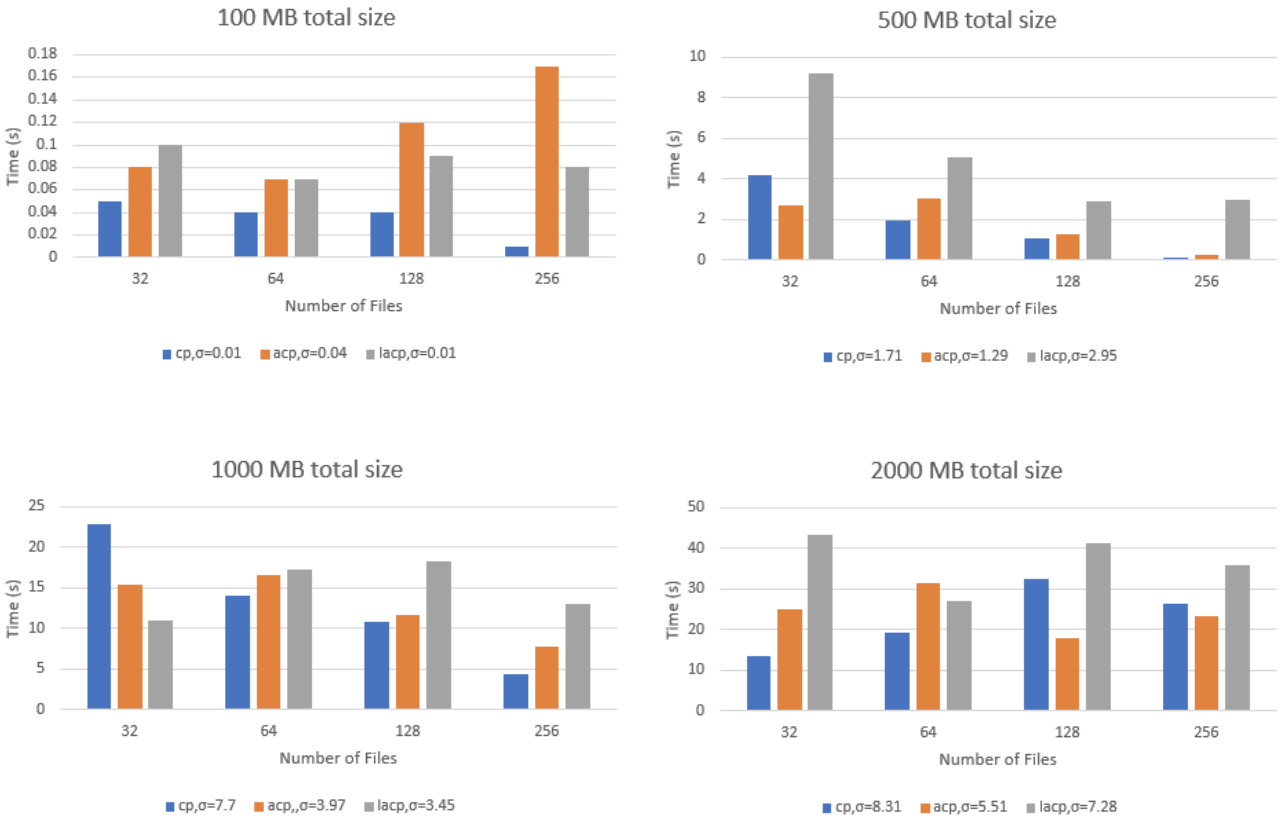
These results are the baseline for comparison. I run `cp`, `acp`, and `lacp` on total file size from 100MB to 2GB and number of files from 32 to 256. Each of the graphs below correspond to a total size. Lower bars are better (less time taken to copy).

Spinning disk results:



For spinning disk, `cp` and `acp` are pretty equivalent. `lacp` is the more interesting case. At very small total file sizes, the overhead far outweighs any potential benefit of kernel asynchronous I/O. As the file sizes get larger, though, there is a very noticeable trend of `lacp` getting better compared to the other two. It would have been interesting to see whether this trend eventually makes `lacp` superior at huge file sizes.

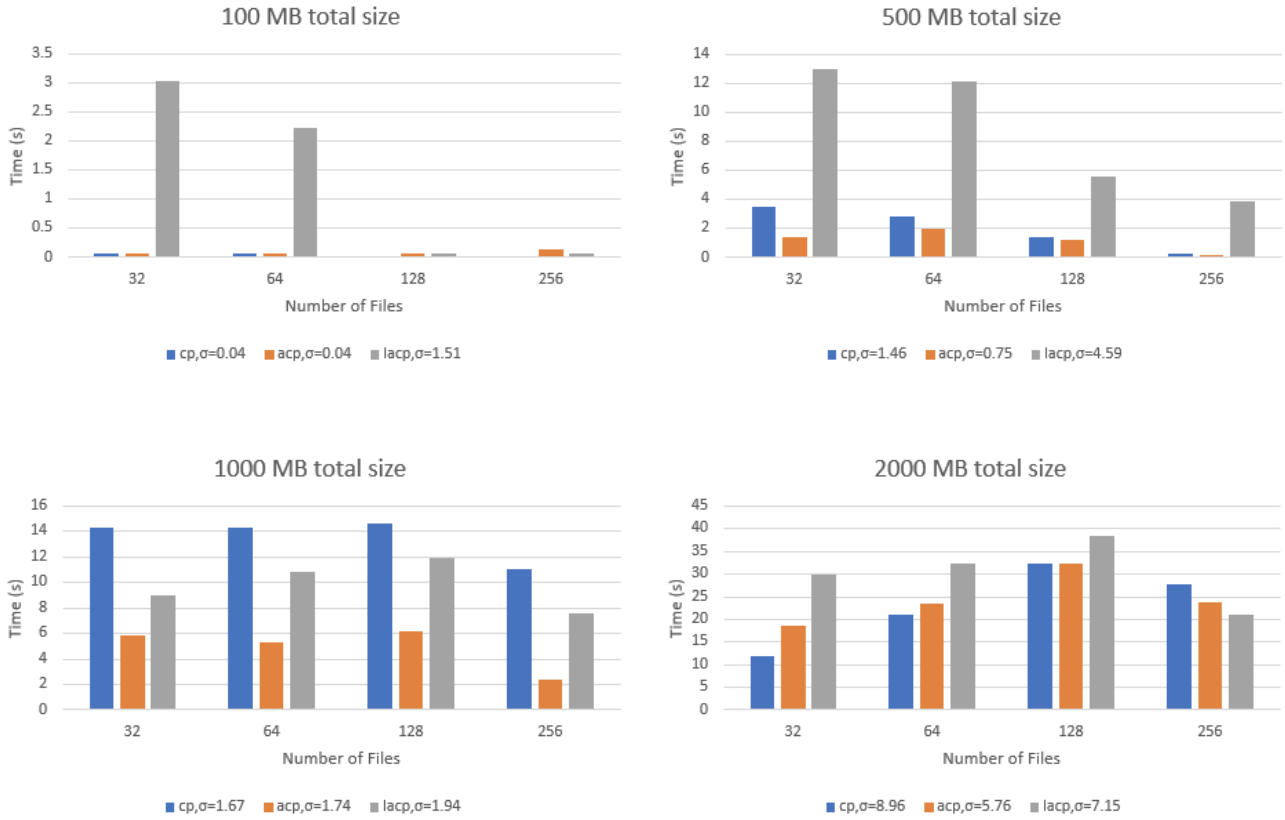
SSD results:



As can be seen from the SSD figures, rarely do the basic versions of asynchronous copy outperform cp. The extra bookkeeping required for setting up contexts, dispatching separate threads (in the case of acp), and perhaps double buffering hurt the performance of the copy. For smaller file sizes, the benefit of asynchronous I/O is overshadowed by the overhead of dealing with many files, as shown in the 100 MB graph. `cp` get faster as the file size gets smaller, but the asynchronous versions typically get slower. For 500 MB and 1000 MB total sizes, the POSIX aio version follows the same trend as `cp`, getting faster as the number of files increase. Both the POSIX and Linux aio versions outperform `cp` on the 1000 MB, 32 files test, as overlapping large file I/O is quite beneficial compared to single-threading the entire operation. At the largest file size, though, it seems that all bets are off. The standard deviation is quite large, indicating that transient effects on memory or disk by the operating system or other process may be strongly affecting the results.

5.3 `readahead()`

Linux provides a `readahead()` function that initiates reading data from a file descriptor into the page cache. It scheduled this read in the background and attempts to return immediately (more asynchronous I/O).

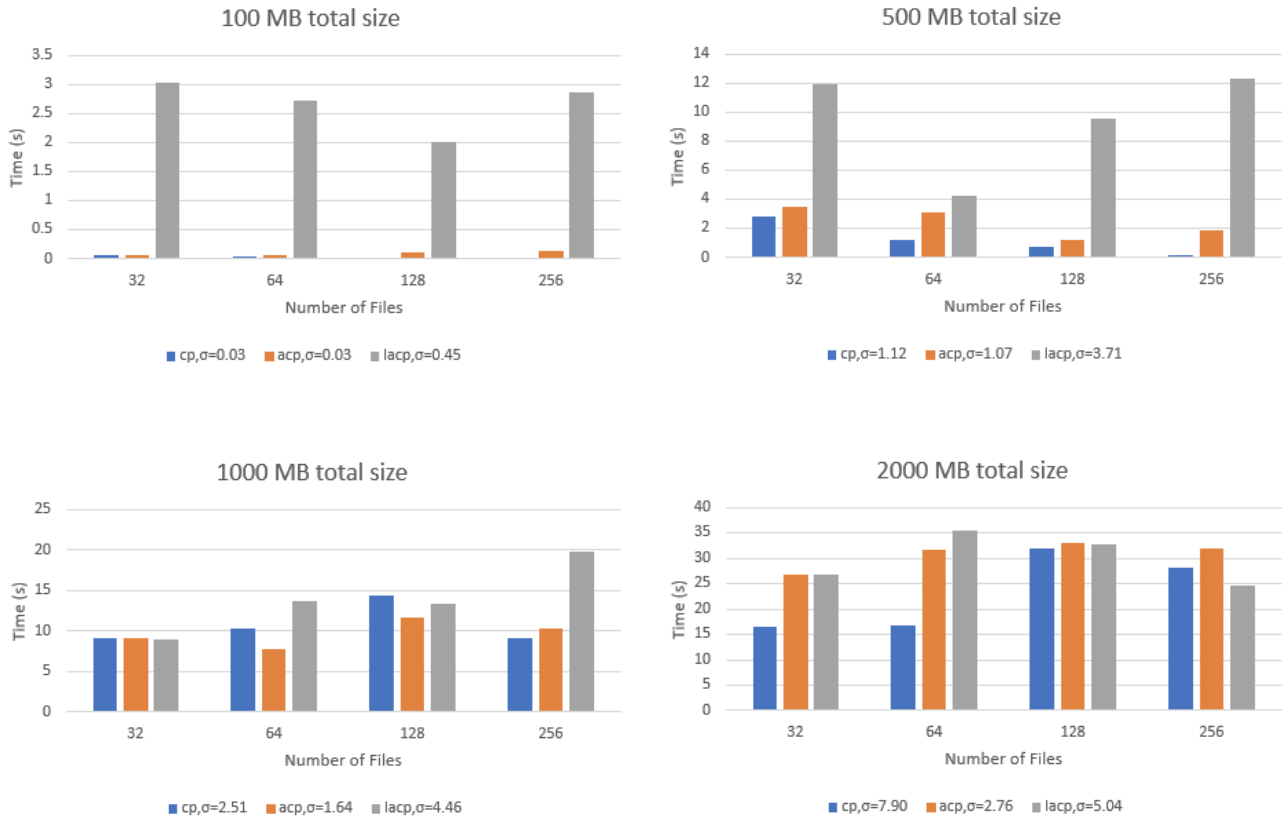


`acp` now works quite well, on par with or even faster than `cp` for file sizes up to 2GB. This is because `acp` performs blocking I/O using threads, and the readahead operates similarly to an asynchronous read request into the cache. `lacp` performs better than before, but is still much slower than `cp` and `acp` at small file sizes. At larger file sizes, it begins to outperform `cp`, and it even becomes the fastest program at 2GB total file size and 256 files. Though the `O_DIRECT` flag should cause requests to `lacp`'s file descriptors to bypass the cache, there are probably residual effects on caching on the device-level as a result of the readahead.

I briefly considered checking for loaded pages, but as a result of the experimentation structure (clearing system caches before every run), this would not have yielded any benefit, especially to the Linux aio library which bypasses system caches.

5.4 `fallocate()`

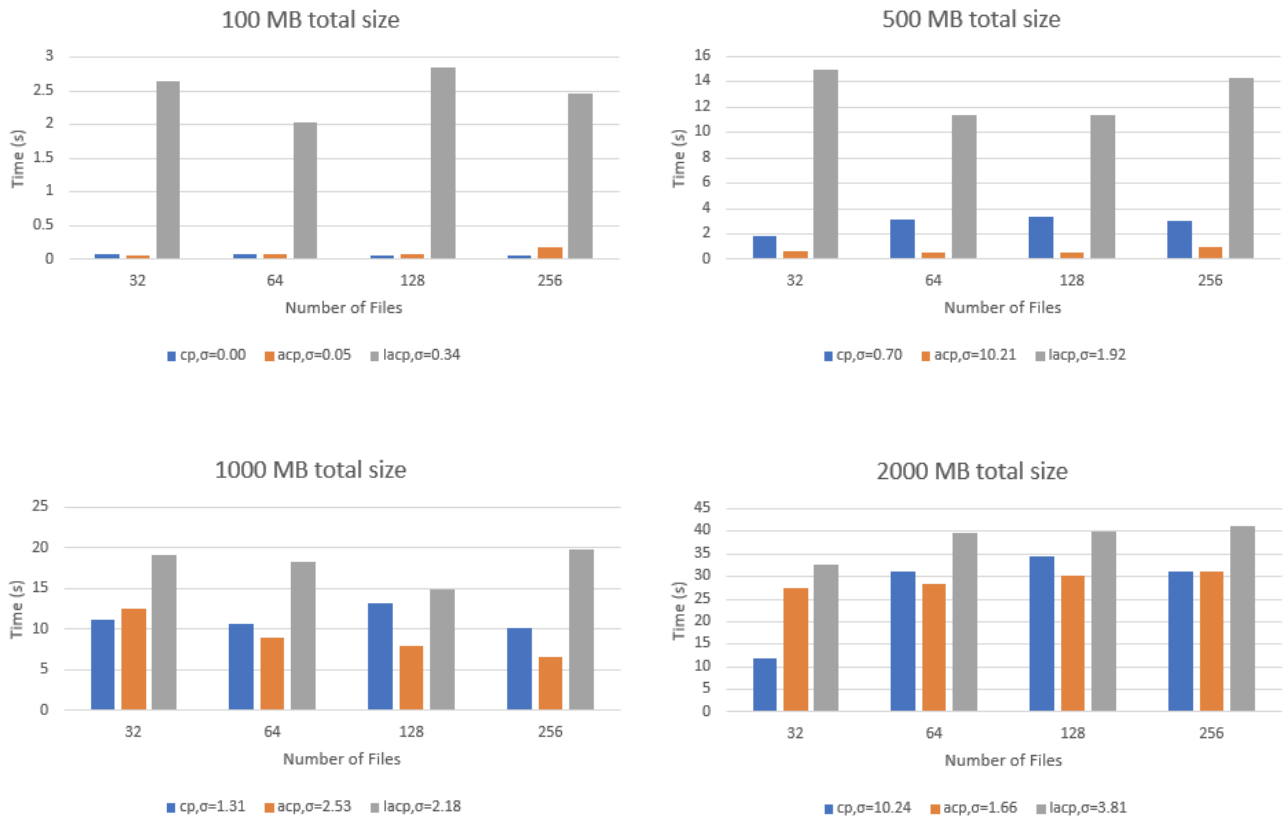
`fallocate()` is a Linux system call that can be used to manipulate a file's space on the disk. The advantage is that a certain file can immediately be grown to the full final size instead of lengthening in chunks as the program writes into it. This optimization should have more of an effect on larger files, as smaller files usually won't have the overhead of multiple expansions.



Interestingly enough, calling `fallocate()` seems to make `acp` and `lacp` slower compared to `cp` in some situations, usually at small file size. This could indicate that the extra system call is unneeded because the write requests will perform the allocation of the entire file size when submitted. For larger files, though, `fallocate()` may save some time as multiple size-extending calls to the disk are needed. The general trend of asynchronous writes becoming more efficient at larger file sizes and file numbers holds.

5.5 Priority

I ran this set of tests as root, and programmatically set the nice value to -15 to see if higher priority made a difference in the performance.



Again we see a trend of `acp` being comparable to or faster than `cp`, and `lacp` lagging behind until large file sizes. For I/O-bound programs like this copy utility, priority - which translates into more opportunities to be scheduled on the cores - should not have a large impact on performance. Indeed, if all the program is doing is spinning in a loop waiting to reap completion events, a higher priority could just mean more wasted cycles. A call to `nanosleep()` in the monitoring loop could help remedy this problem.

6 Conclusion

Accurate I/O benchmarking is hard for many reasons, including these: it's nearly impossible to be the process exclusively accessing the disk or memory during any run, caching occurs at several levels, not just in user- or kernel-space, overhead dominates small file copies, memory limits and swapping can dominate large file copies, and disk cache and scheduling behavior is pretty opaque to the utility, parallel code is hard to develop and debug, and errors during asynchronous calls are difficult to track down. Despite these drawbacks, almost all the data collected in my experiments indicate that both POSIX and Linux asynchronous I/O libraries scale better at large file sizes or large numbers of files. This makes intuitive sense: for small files, execution time is dominated by the overhead of setting up and using asynchronous I/O requests. For large files, the overhead of using these aio libraries is dwarfed by the benefit from doing parallel I/O. Hypothetically, `cp` copying two giant files would have half the performance as a Linux aio

utility copying the same two giant files (assuming enough cores, a smart disk scheduler, disk bandwidth not being the bottleneck, etc.). For large numbers of files, the bottleneck for `cp` is still how quickly it can open, write, and close files one at a time, whereas the versions using aio can issue many hundreds of requests at a time and let the kernel and disk schedulers try to optimize accesses.

As always, the answer boils down to "it depends". I could write a version of `cp` using a heuristic to determine whether it would be better to use the low-overhead single-threaded version or a parallel, asynchronous version, but the performance improvement would be negligible for my typical uses. Unless the workload is heavy on copying hundreds of massive files daily (which happens to happen at my job), optimizing a single-machine `cp` won't be a huge priority. For truly large file copying workloads, parallel filesystems like Lustre provide much more performance for much less headache on the programmer side. It can be fun to see how much one can actually manually improve common file operations, though.