CS 380L Advanced Operating Systems

Final Project : Implementing cp -r with Linux AIO

Due 2^{nd} Dec, 2014

Madhura Parikh
mparikh@cs.utexas.edu

Sreedevi Surendran
sreedevi@cs.utexas.edu

Abstract

In this report we describe our experience of reimplementing the GNU coreutils `cp -r` functionality via the asynchronous I/O (AIO) interface offered by Linux. Currently, the synchronous I/O model on linux means that a thread which issues an I/O request is blocked, until the I/O operation completes. In contrast, the AIO interface will allow the requesting thread to proceed without blocking and submit other requests while the original request is being processed. There are two interfaces to linux AIO - posix AIO and kernel AIO. We implement `cp -r` via both these interfaces and present our findings.

1 Problem statement and motivation

The project reimplements the GNU coreutils `cp -r` available on standard Linux systems using AIO. It takes as its input two parameters, the directory which is to be recursively copied and the new location where it is to be copied.

The driving hypothesis behind this project is that asynchronous I/O can help implement a more optimized version of `cp -r`, since it need not block on any individual read or write but can quickly traverse and build up the directory structure, even as the I/O requests are being submitted and reaped.

2 Overview of Linux AIO

The traditional I/O model on linux is synchronous. This means that a thread that makes an I/O request is blocked until that request is completed. In contrast, the asynchronous I/O model will allow a thread or process to make I/O requests without having to block for these to complete. The thread will be notified later when the requests are completed, and can then retrieve the corresponding results. The Linux AIO has been has been available for linux kernel versions >= 2.4 and was standardized since the linux kernel version 2.6. Unlike the synchronous model, in AIO a thread can overlap processing with I/O requests, resulting in more efficient CPU utilization.

In addition to the kernel AIO we described above, POSIX has an API for AIO as well. However this API is implemented at the user level rather than natively in the kernel. POSIX AIO uses multiple threads, to put up appearances of AIO, but it uses the synchronous blocking I/O in the underlying implementation.

Kernel AIO requires the `libaio` library whereas POSIX AIO requires the `librt` library. There are currently several restrictions on the kernel AIO - it has limited support on many filesystems and works well only for raw devices as compared to buffered files. It requires the files to be opened with the `O_DIRECT` flag, otherwise the invocation may fail or silently fall back on the synchronous I/O implementation. This also means that the reads and writes will not go through the kernel buffer cache. By contrast, the POSIX AIO is more portable and will work with buffered files and most filesystems.

Both the APIs introduce a control block structure - `aiocb` in case of POSIX and `iocb` in case of the kernel - that allows storing the context of the current request so that it can be later identified when it is completed. In addition they have a very simple API that allows making these requests.

2.1 Kernel AIO API

The kernel AIO API has an `io_context_t` datatype. Requests are submitted via an `iocb` structure to an `io_context_t` object and later on completion can be retrieved via that object. The `io_context_t` object is internally a pointer to the queue of completed requests. The length of this queue can be modified by writing to the `/proc/sys/fs/aio-max-nr`. The `io_context_t` object is created via a call to `io_setup(...)`

and destroyed via a call to `io_destroy`. Once the `io_context_t` object has been setup, requests (i.e `iocb` objects) can be submitted via a call to `io_submit(...)` Completed requests can be obtained via a call to `io_getevents(...)` Finally there is an `io_cancel(...)` method to cancel an already submitted I/O request. The entire API consists of these 5 functions. More details may be found by looking up the corresponding Linux man pages.

2.2 POSIX AIO API

The POSIX API also has a very small number of functions. All requests are made via an `aiocb` structure, as mentioned earlier. The `aio_read(...)` function allows submitting an asynchronous read request and similarly the `aio_write(...)` function allows submitting an asynchronous write request. There is also an `lio_listio(...)` function that allows submitting batches of requests. The `aio_error(...)` checks if a particular request is completed and such completed requests can then be retrieved via a call to `aio_return(...)`. There is an `aio_suspend(...)` function that will block the caller until the particular request is completed. However the API also supports notification of request completions via signals or via a callback mechanism. Again complete details may be found via the Linux man pages.

3 Implementation

We decided to implement the `cp -r` functionality via both the POSIX API as well as the kernel API.

3.1 Hardware and software requirements

All our implementation and evaluation was done on a 64-bit x-86 architecture machine that runs the Ubuntu 12.04 LTS flavor of the linux OS with linux kernel version 3.5. It has 4GB RAM (500GB rotating hard disk) and 2.5 GHz Intel core i5 processor (2 cores, 3MB cache). To disable frequency scaling we installed the `cpufreq` app available in the Ubuntu software center. With this app, we ran the CPU in the `performance` mode, thus ensuring that it ran at a constant frequency of $2.5GHz$. For our software requirements, we used the libaio library(version 0.3.109-2) for the kernel AIO and the librt library (version 2.4) for the POSIX API. Both these libraries are available via the Ubuntu package manager. We use some C++11 features like `std::atomic` in our code, which is compiled via the `g++` compiler (version 4.8.1) with the `std=c++11` flag. The code is run on the `ext4` filesystem.

3.2 Common implementation details

Before we discuss the specifics of implementations with regard to each API, we would like to discuss some common decisions. We use `fallocate(...)` to preallocate blocks on the disk for the new regular files to be written. While this doesn't directly help our application it will make later reads faster because the kernel can now allocate them contiguously. Also we use the `fadvise(...)` with the `FADV_WILLNEED` flag for each regular file to be copied so that the kernel can be informed about read-aheads. We currently use a buffer size equal to the system page size. This allows us to keep a larger number of requests inflight - which is often crucial for extracting efficiency out of AIO operations. Apart from that we use standard functions like `stat(...)` and macros like `S_ISREG(...)`, etc while copying the files. For traversing the directory tree recursively, we use the `nftw(...)` function. For the kernel-based implementation, we create directories upfront, whereas we lazily create directories for the POSIX-based implementation, since this was more optimal.

3.3 cp -r with Posix AIO

In this implementation we use the POSIX API which we described earlier. We use the callback mechanism to collect requests that are completed. We define two handlers the `read_handler` and the `write_handler`. The `read_handler` is called whenever a read from the source file completes. We created a context object that keeps additional information such as the source and destination file descriptors. Callbacks are registered via the `aio_sigevent` structure. We pass our context object to this structure - which will be provided to the callback - so that we can keep track of both the destination file, and the source file. Information such

as the file offset is already maintained by the `aiocb` structure. We use the `SIGEV_THREAD` option in the `aio_sigevent.sigev_notify` field so that the callback function is called in a new thread.

When the `read_handler` is invoked it uses the information from the `aiocb` struct in the completed request and our context object to check error conditions and setup a corresponding asynchronous write request, that copies the read data to the appropriate destination file. Whenever a write request is completed, the `write_handler` is called. This is much simpler than the `read_handler`, it simply checks for error conditions and returns. The callback threads communicate with the main thread via a semaphore (`sem_t`), so that the main thread can wait until all the asynchronous requests have completed.

Complete implementation details may be found in our source code.

3.4 cp -r with Kernel AI

For the kernel based implementation, we setup two different threads, the `read_queue` and the `write_queue`. They serve conceptually similar roles as the handlers described above. However rather than being invoked via a callback, they will reap the requests that are ready by calling `io_getevents(...)`. We setup the `io_getevents(...)` so that it returns whenever a single completion is available, though in practice it can be made to block until multiple completions are available. Also as described above, we pass the file descriptor of the destination file in our custom defined context object. Both the threads will keep reaping the completed requests. The `read_queue` will setup a write request for the destination file corresponding to the completed read request and submit it. The `write_queue` will simply reap completed write requests and check them for error conditions.

Since the kernel based AIO only works with the `O_DIRECT` flag, we took care to ensure that our buffer and offset were aligned with the block size of the `ext4` filesystem. We also tried to run the implementation with the `O_NONBLOCK` flag instead of the `O_DIRECT` flag - this is currently not supported via AIO and will fall back to synchronous I/O. However we in fact found it to be faster than the kernel based AIO and even GNU `cp -r`, so we report the results for completeness.

4 Evaluation

We performed our evaluation for different file sizes and for different depths of the directory structure, since this was a re-implementation of `cp -r`. We used the hardware and software setup, already described earlier and report our results below. For convenient testing, we wrote a small python script `workload.py` that takes as parameters the number of files, the file size (common for all files), the directory depth, and then sets up a binary tree like directory tree that has node directories upto the given depth and each of these node directories at different levels in the tree has the given number of files of the given size. The files were generated by reading random bits from `/dev/urandom`. With this script we could quickly setup workloads of different file sizes and with different recursive nesting of directories.

4.1 Sanity tests

To ensure that our code was indeed correctly copying the files even at several levels of nesting in the directories, we ran a recursive diff (using the `diff` utility available on linux) between the source directory and the destination directory (created by our program). This was a huge help to us and helped discover several synchronization bugs that appeared only for large files. While the new files had the same size as the original files, the data was jumbled up. After much debugging, we were able to verify our program easily and quickly via recursive diff.

4.2 Performance tests

For testing the performance of our implementations, we setup different directory trees via the python script we described earlier. First we varied the size of the files in these trees. We measured performance for files of size 4KB, 8KB, 16KB, 32KB and 64KB. Here 4KB is the size of the buffer in both our implementations and also the systems page size. Secondly we tested for these file sizes across different depths in the directory tree, while keeping the actual amount of data being copied constant. Thus if our directory tree had 2 levels,

it would have 8 files of a particular size at each level, whereas if had 8 levels. it would have 2 files of a particular size at each level. We tested for depths 2, 4 and 8. With this setup, we measured the completion time of four programs : our kernel based implementation with the `O_DIRECT` flag(`k-D`), our kernel based implementation with the `O_NONBLOCK` flag(`k-N`), our POSIX based implementation(`posix`) and finally the GNU `cp -r` implementation (`cp`). We measured each value over three runs, and we present a mean of those runs below in the table 1. Before every run, we cleared the kernel page cache, to ensure cold start for each program. The plots provide a more visual perspective of the table.

File size(KB)	depth= 2				depth = 4				depth = 8			
	cp	k-D	k-N	posix	cp	k-D	k-N	posix	cp	k-D	k-N	posix
4	0.194	0.127	0.181	0.171	0.253	0.315	0.266	0.265	1.14	2.88	0.798	0.878
8	0.259	0.267	0.233	0.190	0.514	0.599	0.417	0.366	1.33	3.12	1.20	1.35
16	0.406	0.362	0.332	0.205	0.441	0.687	0.502	0.329	1.78	4.47	1.31	1.14
32	0.364	0.527	0.345	0.258	0.465	0.889	0.511	0.332	3.61	7.74	3.65	2.81
64	0.377	0.750	0.4	0.267	0.632	1.45	0.677	0.440	3.72	10.3	3.59	3.39

Table 1: Completion time in seconds for different file sizes and different nesting levels.

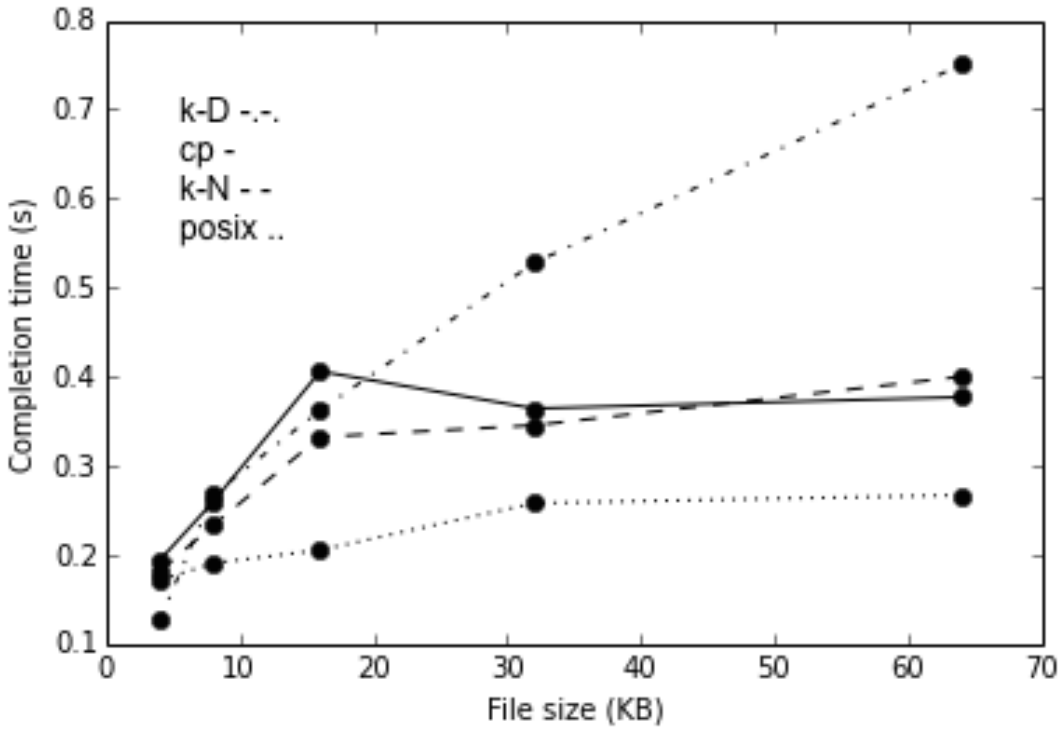

Figure 1: Completion time when the directory tree to be copied has depth= 2

4.3 Discussion

Clearly our POSIX implementation has the best performance of all the tested programs. It shows nearly $8 - 23\%$ improvement over `cp` with the improvements at the larger end of the range being much more frequent. The kernel implementation with the `O_DIRECT` flag is the slowest of all the implementations. As we mentioned earlier, since this flag supports reading only from raw devices and bypasses the kernel cache, there is a significant drop in the performance.

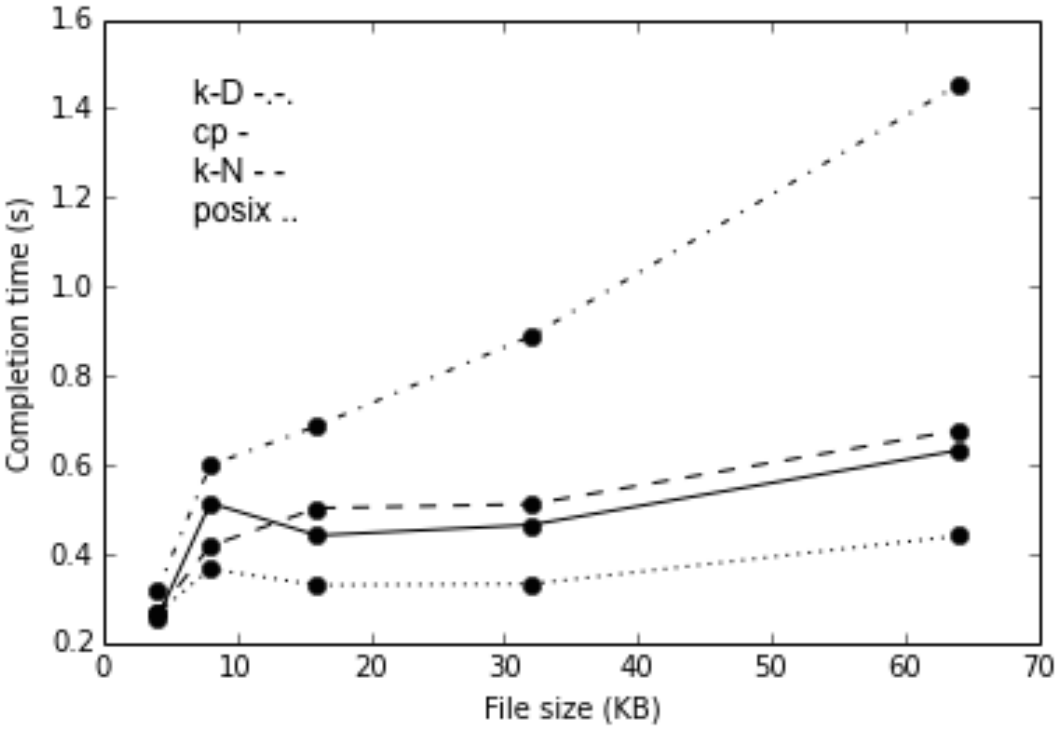

Figure 2: Completion time when the directory tree to be copied has depth= 4

Interestingly for depth = 2, and files of size 4KB, the k-D implementation is in fact faster than cp. Since in these experiments we initially cleared the cache, and the system page size is 4K, the kernel doesn't have much opportunity for read-aheads per file in this scenario. Thus k-D is faster, however for larger files, read-aheads per file will help cp that uses the page cache. Another reason for the higher speed may be that the buffer size in this scenario equals the size of the file to be copied.

The kernel implementation using the `O_NONBLOCK` flag is also mostly faster than cp. While this actually should degenerate to synchronous blocking I/O on the `io_submit` call, the slight performance improvement may be be cause in this case the read and write workloads are managed by separate threads.

All the programs become slower as the data to be copied increases. The programs also become slower, as the directory tree to be copied has more levels. This is likely because it will involve more look-ups. The k-D implementation becomes especially slow in this case. One reason for this may be that [1] the kernel AIO is only partially supported on the `ext4` filesystem. It will also block if the metadata for the required files is not in memory. This may become more apparent when there is more metadata as in case of deeper directory trees.

5 Future work

Our project revealed several interesting directions to explore in the future:

- Asynchronous I/O is supported on Linux via `epoll` and `O_NONBLOCK` flag. In the future we would like to implement `cp -r` with that and test it against our current implementations.
- We used profiling via `strace -c` and the `perf` tool to understand some of the bottlenecks in our implementation. In the future we would like to perform more diagnostics to uncover additional optimizations we could apply.

[1] https://code.google.com/p/kernel/wiki/AIOUserGuide

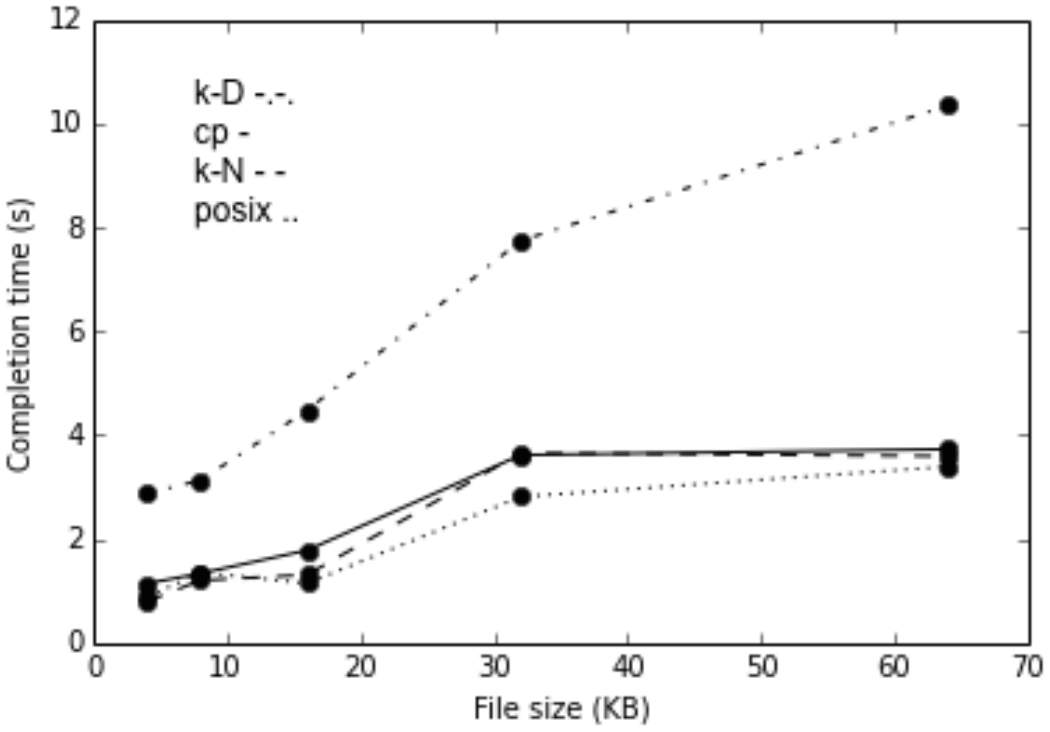

Figure 3: Completion time when the directory tree to be copied has depth= 8

6 Conclusion

In this project we implemented `cp -r` via the native linux kernel API and via the POSIX API. From our experiments we conclude that the kernel-based AIO is currently not mature enough to be successfully adapted for recursive copy and other buffered filesystem tasks. However it does offer several exciting possibilities if properly supported by the filesystem. To this end we illustrate the POSIX-based AIO which has a superior performance as compared to GNU cp. Clearly asynchronous non-blocking I/O can be promising for several filesystem operations - since it allows the CPU to be utilized while overlapping the I/O requests - it is especially useful for applications like databases that compute checksums or perform other cpu-bound operations while also maintaining a high-rate of I/O requests. We hope that kernel asynchronous I/O is better supported natively in the Linux kernel in the future.