# CS380L: Advanced Operating Systems
# Final Project Report

Anubhav Goel (ag82989)
Aneesh Shetty (aks4724)

7 December 2022

## Introduction

`cp` is the standard bash command used for copying files and directories in Linux. The recursive flag `cp -r` is added if the copying task involves copying an entire directory structure consisting of multiple files and folders. However a simple `strace` of `cp -r` reveals several interesting points about its implementation.

1. `cp -r` first runs `fstat` on the source directory, to obtain the structure of the directory to be copied.
2. `cp -r` sequentially calls `open` on each file in the source directory, creates a new corresponding file in the destination directory using `open`, followed by sequential calls of `read` and `write`.
3. `cp -r` uses the idea of synchronous IO, which essentially means that it makes blocking read and write calls sequentially. Every subsequent call will not be executed till the previous call has successfully returned. This imposes a high overhead in terms of time.
4. `cp -r` calls `close` on the source and the destination file to conclude the copying operation.

Note the call to `fadvise` with the flag `POSIX_FAVD_SEQUENTIAL`. This system call tells the kernel that the source file will be read sequentially in the future so that the kernel can do `readahead` and perform cache optimizations to favor sequential reads.

Additionally, we note that the buffer size of reads and writes is 128 KB, which might lead to optimization for calls where the file sizes are 128 KB.

```
newfstatat(AT_FDCWD, "source/file.txt", {st_mode=S_IFREG|0664,
    ↪  st_size=1002, ...}, AT_SYMLINK_NOFOLLOW) = 0
openat(AT_FDCWD, "source/file.txt", O_RDONLY|O_NOFOLLOW) = 3
newfstatat(3, "", {st_mode=S_IFREG|0664, st_size=1002, ...},
    ↪ AT_EMPTY_PATH) = 0
openat(AT_FDCWD, "dest/file.txt", O_WRONLY|O_CREAT|O_EXCL,
    ↪ 0664) = 4
newfstatat(4, "", {st_mode=S_IFREG|0664, st_size=0, ...},
    ↪ AT_EMPTY_PATH) = 0
fadvise64(3, 0, 0, POSIX_FADV_SEQUENTIAL) = 0
read(3, "#!␣/usr/bin/bash\n\n#␣sleep␣15\n#␣t"..., 131072) =
    ↪ 1002
write(4, "#!␣/usr/bin/bash\n\n#␣sleep␣15\n#␣t"..., 1002) =
    ↪ 1002
read(3, "", 131072) = 0
close(4) = 0
close(3) = 0
```

We write our own implementation to copy a directory recursively using the idea of asynchronous IO. Asynchronous IO allows IO operations to be processed through non-blocking calls, that is, other calls and processes are allowed without requiring a previous call to have returned successfully.
Linux has an existing asynchronous IO interface called `aio` but it suffers from various limitations including that it supports asynchronous IO only for `O_DIRECT` (or un-buffered) accesses. For this reason, we look at `io_uring`, a new interface for asynchronous IO operations which was introduced in Linux 5.1.

## Using `io_uring`

The primary goal of `io_uring` is efficiency. To avoid copies of data in memory, `io_uring` uses shared memory structures between the application and the kernel. The memory structures are implemented as a pair of Single Produce Single Consumer (SPSC) ring buffers, namely **submission queue** and **completion queue**.

The application adds a read/write request to the tail of the submission queue and updates it. The kernel reads requests from the head of the submission queue and updates it. The requests are processes by the kernel and the completion events are added to the tail of the completion queue, following which the kernel updates the tail of the queue. Subsequently, the application reads the completion event from the head of the completion queue and updates it. The above set of data structures and operations form the basis for copying files using `io_uring`.

# Hypothesis

We write our own implementation for copying directory structures in Linux `mycp` which utilizes `io_uring` to perform asynchronous reads and writes. We expect the performance of `mycp` to be superior in comparison to the existing implementation of `cp -r` in terms of the execution time for the above-mentioned reasons. We plan to test our implementation over various file sizes and number of files in the source directory to validate our hypothesis.

# Design and Implementation

## Data Structures

We create the following data structures for our implementation:

1. We create an `Error` class and `Result` class which we use for error handling, we took inspiration for this class and the use of its variants from standard Rust implementations.

2. We create a `free_list` class to allocate blocks for holding data associated with read/write requests. The size of the blocks allocated by the `free_list` is a configurable parameter in our code.

3. We create a `wrapped_fd` class which is essentially a wrapper over file descriptors to share it across workers. We create member functions to check whether the file associated with a given `fd` is open or closed (`is_fd_open()` and `is_fd_closed()`), as well as `notify_fd_closed()`) and `ensure_fd_open()`.

4. We create an `io_queue` class which holds all functions for scheduling and running copy jobs and workers. The main functions in this class are `add_one_worker()` which creates a `worker` corresponding to a given data block and adds it to a queue of pending workers and `run_uring()` which submits copying to request to the `io_uring` submission queue.

5. We create a `worker` class which holds the data associated with one block. Each `worker` corresponds to one read/write submission in the `io_uring` which is done using the member function `submit_read_write_jobs()`.

## Implementation

A flowchart demonstrating the key steps in our implementation has been added below.

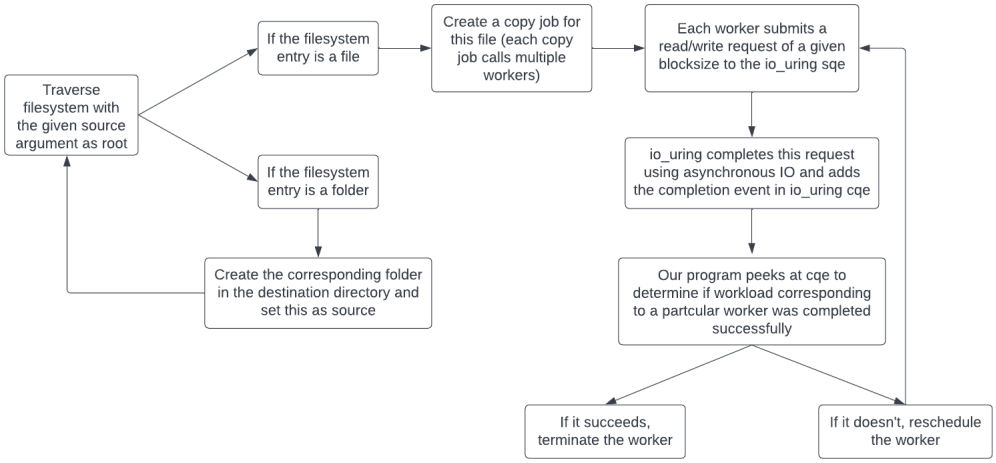


Figure 1: Implementation flowchart for `mycp`

The implementation can be broken down into the following steps:

1. We traverse the source directory using

   `filesystem::recursive_directory_iterator` and check each directory entry

   (a) if the directory entry is a folder, we call `mkdir` at the corresponding location in the destination directory

   (b) if the directory entry is a regular file, we create add workers which copy the file from the source to destination

   (c) if the directory entry is not one of the above, we do not handle this file type for the scope of this project

   We traverse the source directory using Depth-First-Search.

2. For every file to be copied, we call `fallocate` using the file descriptor for the file created at the destination with the size of the source file.

3. We maintain a `free_list` which we use for obtaining a block which forms a single submission unit to the submission queue of `io_uring`.

4. For every block that we wish to copy, we create a `worker` corresponding to it and this is represented by `add_one_worker()` call in our code. We add this worker to the `workers_pending_start` queue which is protected by a lock.

5. We create a separate thread in our main program which constantly looks at the queue of `workers` which are pending and sends read and write requests corresponding to the data that each `worker` is associated with. This thread is created in `start()` function of our `io_queue` class. Subsequently, `run_uring()` function submits requests to the `io_uring` corresponding to each `worker`.

6. The `run_uring()` function picks the `worker` from the front of the queue and pops the queue. This instance is represented by the `need_to_add` variable.

7. For this instance, the member function `submit_read_write_jobs()` is called. The function calls `io_uring_prep_read()` and `io_uring_prep_write()` which are a part of `liburing.h` library. An important implementation detail at this point is that we use the `IOSQE_IO_LINK` flag to ensure that **the write operation is aborted automatically if the read operation fails**.

8. The `run_uring()` function peeks at the completion queue using `io_uring_peek_cqe()` and interprets the result of the completion event using `cqe->user_data` and `cqe->res`. Depending on the result, the `worker` has executed successfully and the file is closed or there has been an error and the `worker` is rescheduled.

9. A `join()` function finally cleans up the threads when all the `workers` have finished running.

We make use of `atomic` keyword to declare sensitive variables and protect `worker` queues using locks.

# Evaluation

## System Environment

We tested our implementation on our local systems which have the following environments:

### Anubhav's PC

- Output of `lscpu`:

```
Architecture: x86_64
  CPU op-mode(s): 32-bit, 64-bit
  Address sizes: 39 bits physical, 48 bits virtual
  Byte Order: Little Endian
CPU(s): 4
  On-line CPU(s) list: 0-3
Vendor ID: GenuineIntel
  Model name: Intel(R) Core(TM) i5-7300HQ CPU @ 2.50GHz
    CPU family: 6
    Model: 158
    Thread(s) per core: 1
    Core(s) per socket: 4
    CPU max MHz: 3500.0000
    CPU min MHz: 800.0000
Caches (sum of all):
  L1d: 128 KiB (4 instances)
  L1i: 128 KiB (4 instances)
  L2: 1 MiB (4 instances)
  L3: 6 MiB (1 instance)
```

- Output of `grep MemTotal /proc/meminfo`:

```
MemTotal: 16268136 kB
```

- Output of `cat /proc/version`:

```
Linux version 5.15.0-56-generic (buildd@lcy02-amd64-004) (
    ↪ gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0, GNU ld (GNU
    ↪  Binutils for Ubuntu) 2.38) #62-Ubuntu SMP Tue Nov 22
    ↪ 19:54:14 UTC 2022
```

- Output of `cat /sys/block/sdb/queue/rotational`

```
1
```

**Aneesh's PC**

- Output of `lscpu`:

```
Architecture: x86_64
  CPU op-mode(s): 32-bit, 64-bit
  Address sizes: 39 bits physical, 48 bits virtual
  Byte Order: Little Endian
CPU(s): 4
  On-line CPU(s) list: 0-3
Vendor ID: GenuineIntel
  Model name: Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz
    CPU family: 6
    Model: 142
    Thread(s) per core: 2
    Core(s) per socket: 2
    CPU max MHz: 3100.0000
    CPU min MHz: 400.0000
Caches (sum of all):
  L1d: 64 KiB (2 instances)
  L1i: 64 KiB (2 instances)
  L2: 512 KiB (2 instances)
  L3: 3 MiB (1 instance)
```

- Output of `grep MemTotal /proc/meminfo`:

```
MemTotal: 8003252 kB
```

- Output of `cat /proc/version`:

```
Linux version 5.15.0-56-generic (buildd@lcy02-amd64-004) (
    ↪ gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0, GNU ld (GNU
    ↪  Binutils for Ubuntu) 2.38) #62-Ubuntu SMP Tue Nov 22
    ↪ 19:54:14 UTC 2022
```

- Output of `cat /sys/block/sdb/queue/rotational`

```
1
```

## Workloads and Experimental Results

In all results, a configuration of 128MB block size has been used in each block in the ring. The ring size is 8 blocks (1 GB), and the maximum number of file descriptors that can be open at any time is 900. Therefore, whenever the number of files to be copied is significantly higher than this number, the performance of `mycp` is negatively affected.

## Time of Copy

We start by measuring the execution time of our implementation `mycp` and compare it against baselines `cp -r` as well as `rsync`. We measure the execution time of the methods against the number of files in the source directory for different file sizes. We plot these values and the plots for 6 different file sizes, 128 KB, 400 KB, 1 MB, 4 MB, 16 MB, and 128 MB have been presented below. We use `/usr/bin/time` to measure execution time and we use the following command to clear caches to ensure a higher level of uniformity between the experiments and simulate similar initial conditions for all the copy methods `sudo bash -c "echo 3 > /proc/sys/vm/drop_caches"`. Furthermore, we verify the correctness of all our copy operations using `diff -r`.

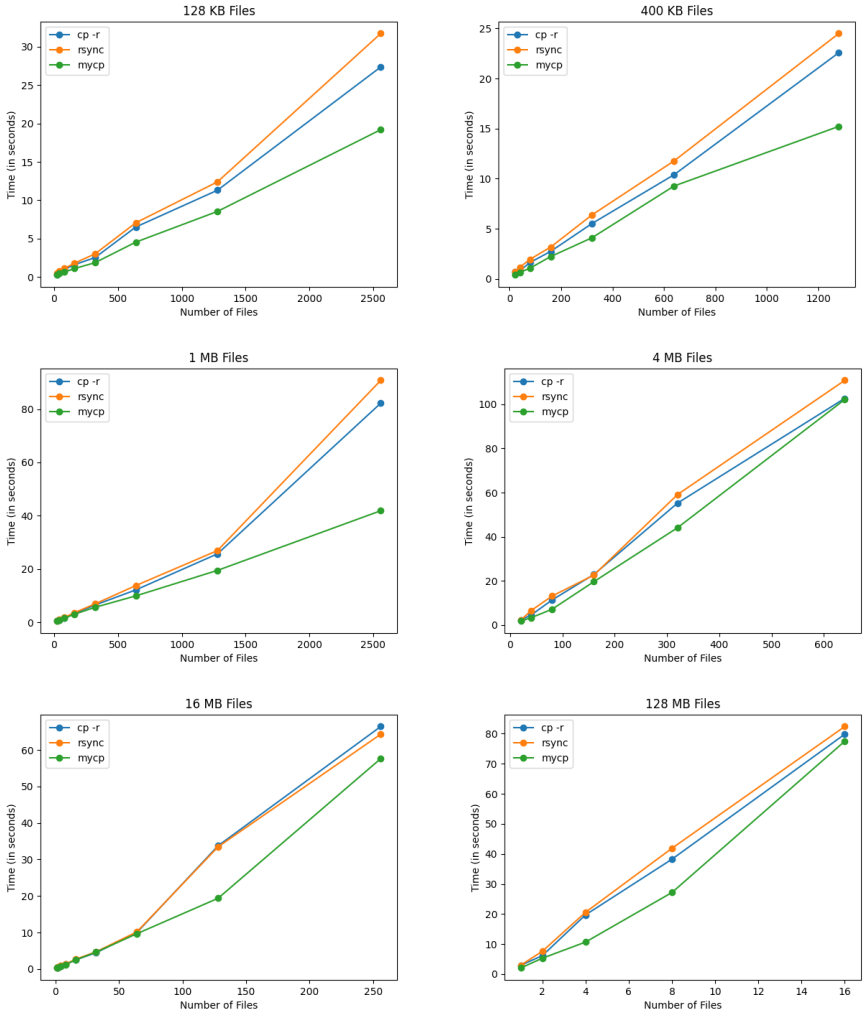


Figure 2: Copying times vs Number of files for different file sizes

As can be seen from the above plots, our implementation always does as well as `cp -r` and `rsync`, if not significantly better in a majority of the experiments. We also looked at performance for a small number of large files and observed that our implementation continued to outperform `cp -r`. A table containing our observations for execution times for large files is presented below.

| Workload | `cp -r` | `rsync` | `mycp` |
|---|---|---|---|
| Single file of 4 GB | 19.96 s | 27.99 s | 19.33 s |
| Single file of 8 GB | 25.68 s | 34.64 s | 19.56 s |
| Single file of 16 GB | 19.04 s | 26.62 s | 18.78 s |
| 2 files of 4 GB | 63.16 s | 72.74 s | 55.00 s |

Table 1: Execution times for large files

We consider cases when the total size of the workload to be copied was the same but the workload was split into varying number of files. We expected this to be an interesting experiment since increasing number of files increases the number of `open`, `close`, `read` and `write` system calls in `cp -r`. This does not happen in `mycp` since `io_uring` essentially allows us to batch schedule a number of system calls in the submission queue, thus reducing the number of switches to kernel mode.

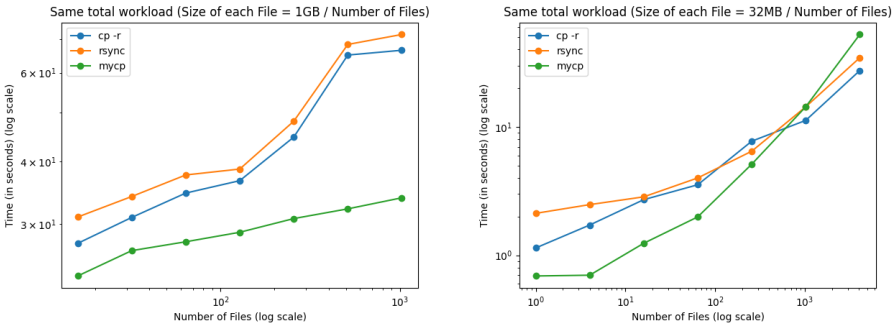


Figure 3: Copy times for same total workload

Figure 3 (left) shows the running time for a 1 GB workload, divided into a number of files. As expected, with the increase in the number of files, the overhead of system calls causes `cp -r` and `rsync` to slow down in comparison to `mycp`. Notice that in this experiment, we don't go much beyond the file descriptor cap of 900.

We expect that increasing the number of files beyond this would cause contention between the `wrapped_fd` structures to open their corresponding files, which would cause `mycp` to slow down. Indeed this happens in the total workload of 32 MB with varying the number of files it is split into, as shown in Figure 3 (right).

So far, the experiments have been focusing on increasing the breadth of the source directory (a high number of files in the source directory). We conduct some experiments with regard to increasing depth in the source directory as well. We write a bash script to make a source directory with a given depth and place a single file inside the final folder. For example, a source directory of depth 4 and placing a file of size 128 MB inside the final directory using the

above script gives the following output on using `tree`.

```
dir1/
 dir2
     dir3
         dir4
             file_128MiB

3 directories, 1 file
```

However, in this case, we observe that the results show high variation. Our implementation continues to consistently perform better than our baselines but no clear dependence on depth can be observed. We also expect this since the source directory walk happens in memory and does not involve IO. We have presented the results here for completeness.

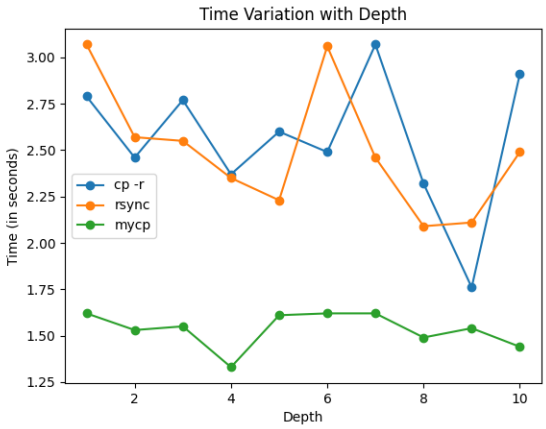


Figure 4: Copy times variation with changing depth in source directory structure

## Using SSD

We also experimented with running our code on SSD by moving our codebase to `tmp` folder. Since SSD supports faster random reads and writes, we expect our implementation to have a significant advantage. The plots from our experiments have been shown below.

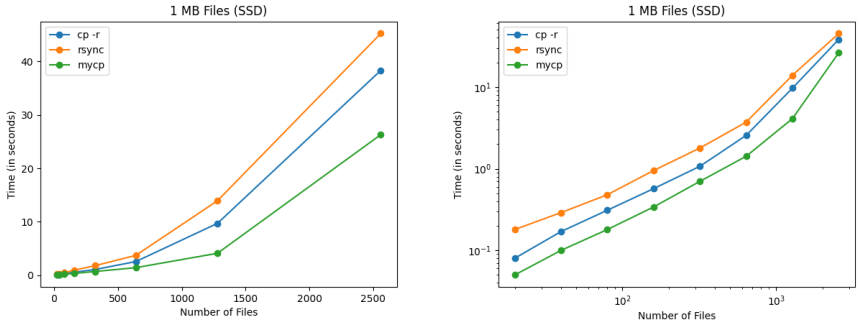


Figure 5: Experiments run on SSD, regular scale (left) and log scale (right)

## Memory Usage

One drawback which we noticed with our implementation compared to `cp -r` and `rsync` is the high memory usage by our program. We measure the maximum resident set size and observe that it is significantly high over all runs compared to `cp -r` and `rsync`. As an example, we tabulate the values for one run which involved copying 10 32 MB files.

| **Maximum RSS for `cp -r` (in KB)** | **Maximum RSS for `rsync` (in KB)** | **Maximum RSS for `mycp` (in KB)** |
|---|---|---|
| 2742 | 4836 | 161035 |

Table 2: Maximum Resident Set Sizes for different copy methods

## Additional Observations

During our experiments, we observed high variability in execution times of our baselines `cp -r` and `rsync` compared to our implementation `mycp`. We believe this is due to the low-level optimization that is present in the implementations for these calls, however, we did not have the opportunity to investigate this further under a formal hypothesis. However, we present our observation using a simple measure of the standard deviation of the execution times of the three methods over 10 runs of the following experiment: copying 50 4 MB files.

| **Copy Method** | **Execution Time Standard Deviation** |
|---|---|
| `cp -r` | 0.67 |
| `rsync` | 0.59 |
| `mycp` | 0.08 |

Table 3: Standard deviation for execution times of different copy methods

The standard deviations for `cp -r` and `rsync` are $\sim$ 25% of their means whereas for our implementation, it is $<$ 10% of the mean. We also plot the execution times to demonstrate the high variability of `cp -r` and `rsync` compared to `mycp`.

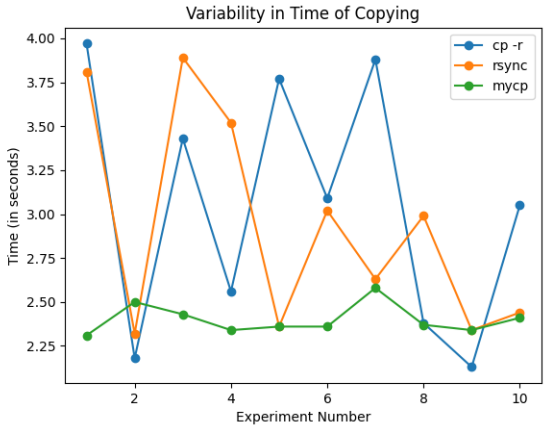


Figure 6: Execution times of different copy methods over 10 runs

### Conclusion

We observe that our implementation `mycp` outperforms `cp -r` and `rsync` across a variety of workloads. We use this evidence to successfully validate our initial hypothesis that the use of `io_uring` for asynchronous IO can make read-/write and subsequently copy operations faster. Additionally, our implementation takes advantage of the faster random access provided by SSD and leverages it for even higher performance.
Additionally, we observe that our implementation has significantly less variability in execution time compared to `cp -r` and `rsync`. This makes our implementation suitable for applications that require high reliability in terms of the time of copying over speed.

## Improvements and Future Work

- Our current implementation does not handle symlinks within the source directory structure. Given more time, we would have liked to handle this case as well.
- Currently we make a simple `fallocate` call to allocate space for the entire file size at the destination. We can improve this using different flags associated with `fallocate`.

# References


- https://www.cs.utexas.edu/~rossbach/cs380l/lab/project.html
- https://kernel.dk/io_uring.pdf
- https://developer.ibm.com/articles/l-async/
- https://man7.org/linux/man-pages/man2/fallocate.2.html
- https://man7.org/linux/man-pages/man2/readahead.2.html
- https://linux.die.net/man/2/fadvise
- https://wheybags.com/blog/wcp.html
- https://unixism.net/loti/
- https://github.com/axboe/fio
- https://stackoverflow.com/questions/61525015/how-to-build-liburing
- https://www.tecmint.com/clear-ram-memory-cache-buffer-and-swap-space-on-linux/
- https://stackoverflow.com/questions/2336242/recursive-mkdir-system-call-on-unix


# Appendix: Terminal Screenshots

As proof of our results, we have attached selected terminal screenshots. Although most of our results have been averaged over 5 iterations, we have attached screenshots containing a single iteration for readability.

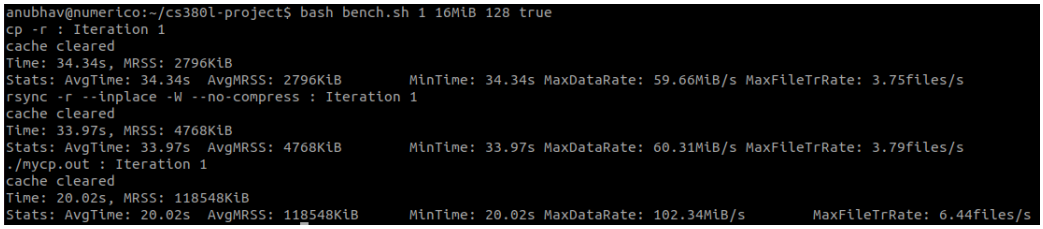


Figure 7: Terminal screenshot showing the significant advantage of our implementation compared to `cp -r` and `rsync`. The high maximum RSS for `mycp` is also evident in this screenshot.

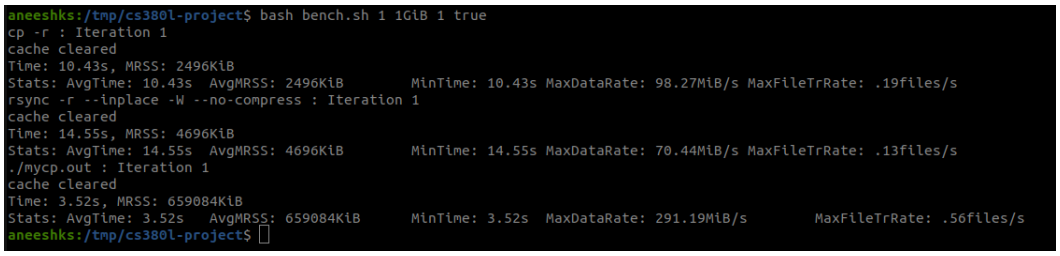


Figure 8: Terminal screenshot showing significant time advantage in copying a single large file (Single file of 1 GB) (SSD)

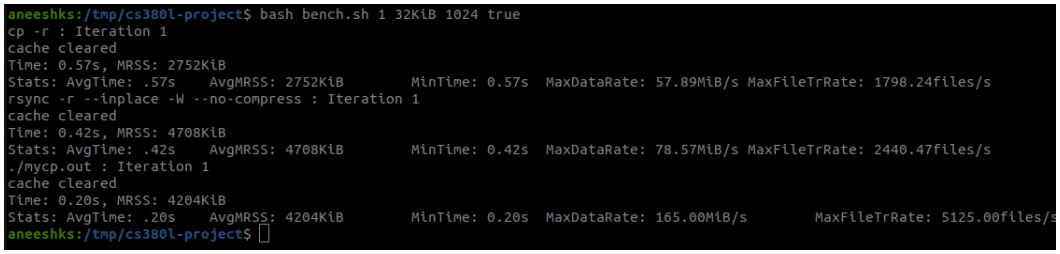


Figure 9: Terminal screenshot showing significant time advantage in copying a large number of small files (1024 files of 32 KB) (SSD)

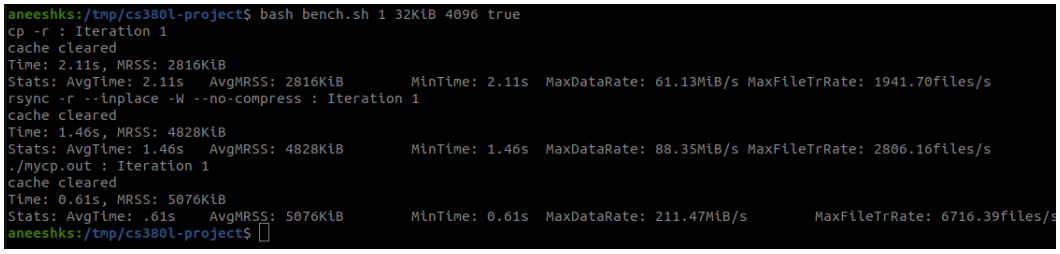


Figure 10: Terminal screenshot showing significant time advantage in copying a large number of small files (4096 files of 32 KB) (SSD)

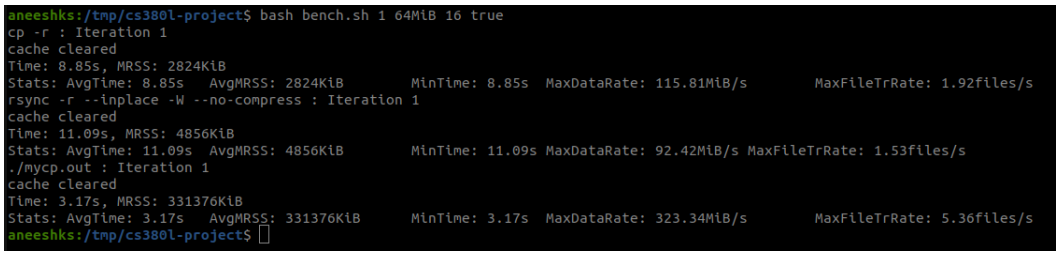


Figure 11: Terminal screenshot showing significant time advantage in copying a moderate number of medium-sized files (16 files of 64 MB) (SSD)

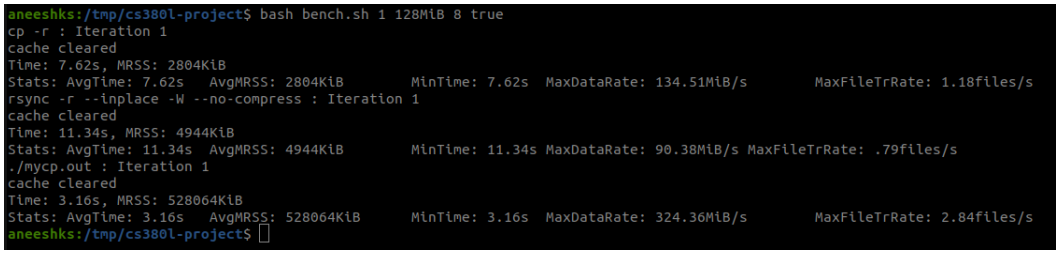


Figure 12: Terminal screenshot showing significant time advantage in copying a moderate number of medium-sized files (8 files of 128 MB) (SSD)