

# CS-GY-6233 Final Project Report

Name	Net ID
Chirag Mahajan	cm6591

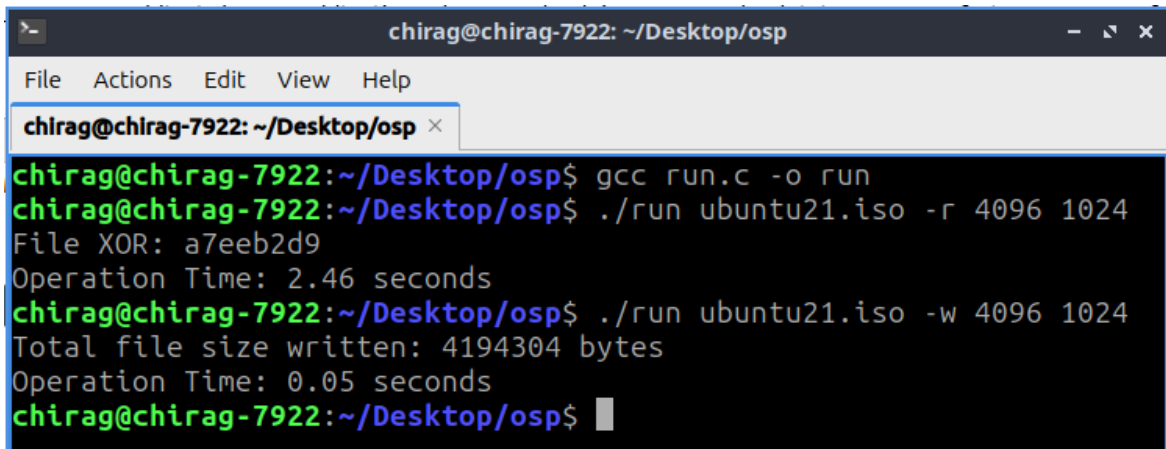
*Instructions to Run the code are included in the README.txt file attached with the report.*

## 1. Basics

Our program utilizes system calls such as open, read, write, and close to interact with files. To verify the accuracy of the read operation, our program calculates and returns the XOR value of the read file. The read() system call is employed to read the file according to user input provided via the command line. Similarly, the write() system call is used to write to the file based on user input, specifically the block size and block count. The file size written will be `block_size*block_count`. While reading the file, make sure the file size is at least `block_size*block_count` otherwise it will give segmentation fault (error).

Usage: `./run <filename> [-r|-w] <block_size> <block_count>`

Output:



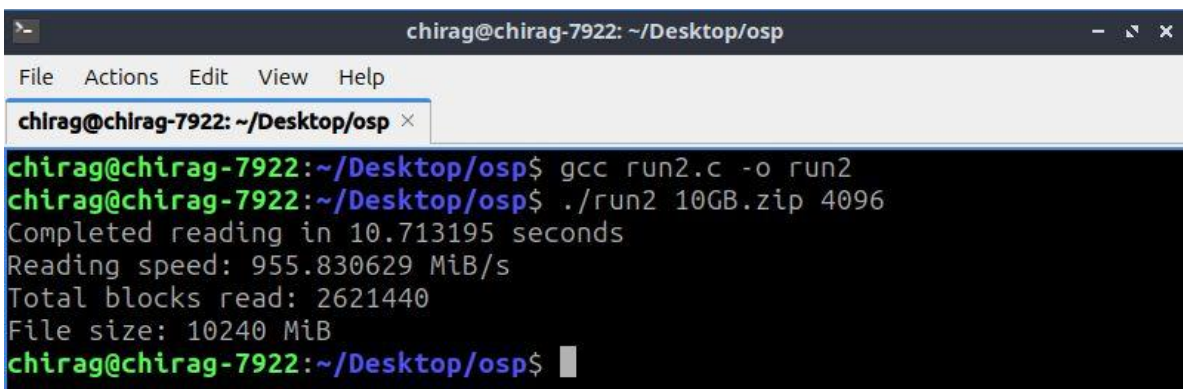
```
chirag@chirag-7922: ~/Desktop/osp
File Actions Edit View Help
chirag@chirag-7922: ~/Desktop/osp x
chirag@chirag-7922:~/Desktop/osp$ gcc run.c -o run
chirag@chirag-7922:~/Desktop/osp$ ./run ubuntu21.iso -r 4096 1024
File XOR: a7eeb2d9
Operation Time: 2.46 seconds
chirag@chirag-7922:~/Desktop/osp$ ./run ubuntu21.iso -w 4096 1024
Total file size written: 4194304 bytes
Operation Time: 0.05 seconds
chirag@chirag-7922:~/Desktop/osp$
```

## 2. Measurement

To ensure a file size that can be processed in a reasonable amount of time, a minimum time limit of 5 seconds is set. If the current file is too small and can be read in less than 5 seconds, we double the file size and re-execute the read subroutine. This process is repeated until the read subroutine takes at least 5 seconds to complete. On our specific machine, the file size that met this criteria was determined to be 10240 MiB or 10737418240 bytes.

Usage: ./run2 <filename> <block\_size>

Output:



```
chirag@chirag-7922: ~/Desktop/osp
File Actions Edit View Help
chirag@chirag-7922: ~/Desktop/osp x
chirag@chirag-7922:~/Desktop/osp$ gcc run2.c -o run2
chirag@chirag-7922:~/Desktop/osp$ ./run2 10GB.zip 4096
Completed reading in 10.713195 seconds
Reading speed: 955.830629 MiB/s
Total blocks read: 2621440
File size: 10240 MiB
chirag@chirag-7922:~/Desktop/osp$
```

**Extra Credit:** dd is a command-line utility for Unix and Unix-like operating systems whose primary purpose is to convert and copy files. We implemented a script to compare performance of our program and dd.

Working of script:

Our Bash script is designed to compare the performance of a custom file copy program (`./runtest`) with the standard `dd` command in Linux. It does this by creating test files of various sizes, copying them using both methods, and measuring the time taken and speed of each operation.

dd Command: `dd if=<input_file> of=<output_file> bs=<block_size> count=<block_count>`

Usage of script: `./ddtest.sh`

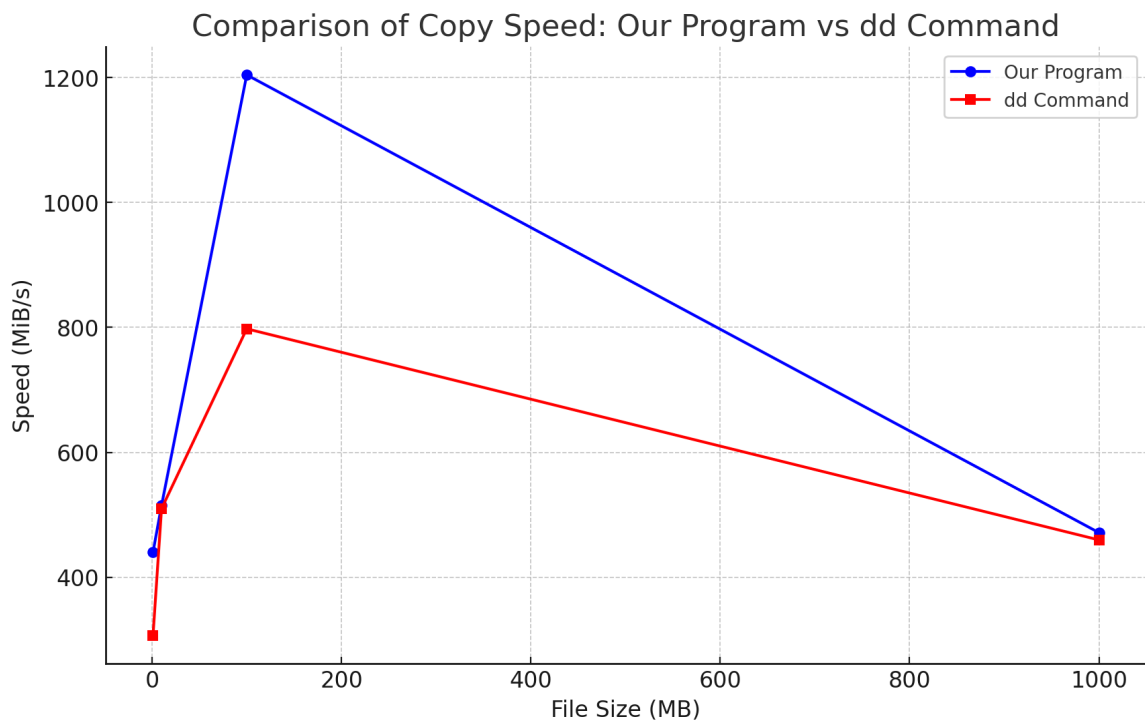
Output:

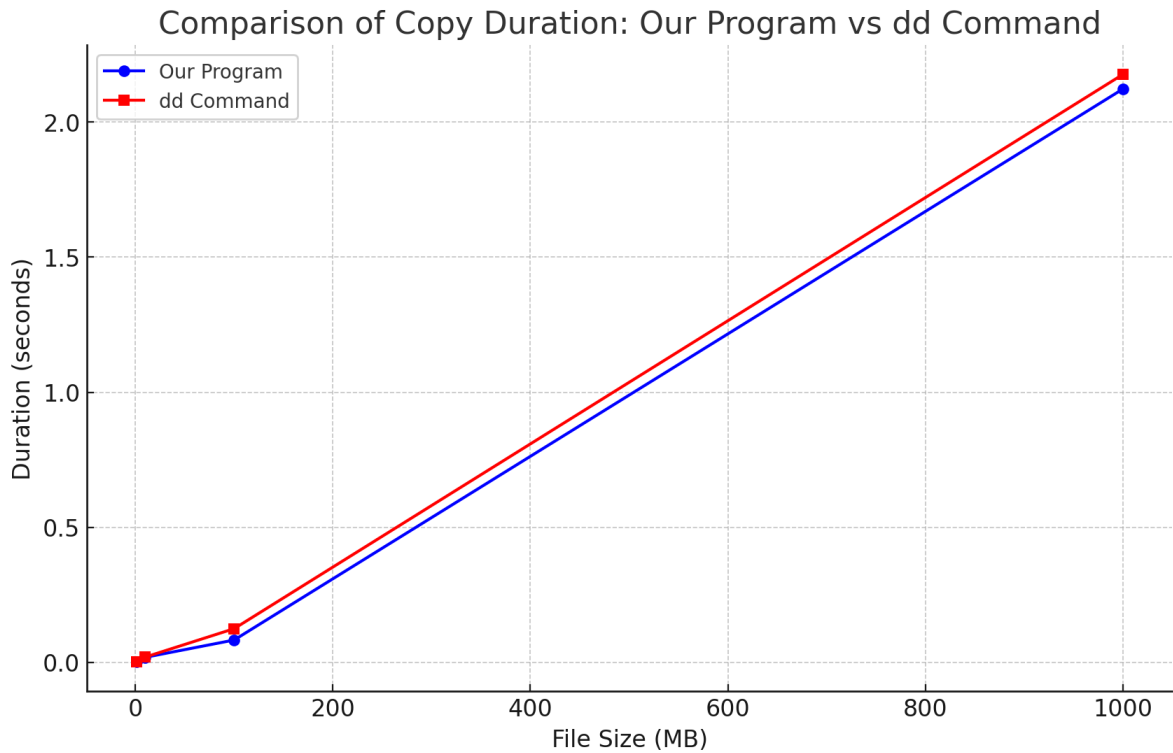
```
chirag@chirag-7922: ~/Desktop/osp ×
chirag@chirag-7922:~/Desktop/osp$ ./ddtest.sh
Testing with file size: 1MB
Running your program for copying fileA_1MB to fileB_fileA_1MB...
Copy operation time: 0.00 seconds
Duration: 0.002 seconds, Speed: 439.932 MiB/s
Running dd for copying fileA_1MB to fileB_dd_fileA_1MB...
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.00164052 s, 639 MB/s
Duration: 0.003 seconds, Speed: 306.328 MiB/s
Testing with file size: 10MB
Running your program for copying fileA_10MB to fileB_fileA_10MB...
Copy operation time: 0.02 seconds
Duration: 0.019 seconds, Speed: 514.674 MiB/s
Running dd for copying fileA_10MB to fileB_dd_fileA_10MB...
10485760 bytes (10 MB, 10 MiB) copied, 0.0179169 s, 585 MB/s
Duration: 0.020 seconds, Speed: 509.595 MiB/s
Testing with file size: 100MB
Running your program for copying fileA_100MB to fileB_fileA_100MB...
Copy operation time: 0.08 seconds
Duration: 0.083 seconds, Speed: 1204.547 MiB/s
Running dd for copying fileA_100MB to fileB_dd_fileA_100MB...
104857600 bytes (105 MB, 100 MiB) copied, 0.122009 s, 859 MB/s
Duration: 0.125 seconds, Speed: 797.815 MiB/s
Testing with file size: 1000MB
Running your program for copying fileA_1000MB to fileB_fileA_1000MB...
Copy operation time: 2.12 seconds
Duration: 2.122 seconds, Speed: 471.330 MiB/s
Running dd for copying fileA_1000MB to fileB_dd_fileA_1000MB...
1048576000 bytes (1.0 GB, 1000 MiB) copied, 2.16163 s, 485 MB/s
Duration: 2.176 seconds, Speed: 459.650 MiB/s
chirag@chirag-7922:~/Desktop/osp$ █
```

Observations:

File Size (MB)	dd Duration (seconds)	dd Performance (MiB/s)	Our Program Duration (seconds)	Our Program Performance (MiB/s)
1	0.003	306.328	0.002	439.932
10	0.020	509.595	0.019	514.674
100	0.125	797.815	0.083	1204.547
1000	2.176	459.650	2.122	471.330

We plotted the graph of File Copying Speed and Time Duration comparison of our program and dd. Here are the results:





## Comparison:

(Keeping block size: 4096)

**Speed Variations:** For smaller file sizes (1MB and 10MB), our program demonstrates higher speeds compared to the `dd` command. However, at 100MB, our program's speed peaks dramatically, suggesting an optimization for handling mid-sized files. At 1000MB, the speeds of both methods converge more closely.

**Duration Trends:** In terms of duration, both methods show an expected increase in time with larger file sizes. Our program generally completes the operations quicker than the `dd` command for smaller files (1MB and 10MB), but the difference becomes less pronounced as file size increases. At 1000MB, the durations are nearly identical.

**Optimal File Size Handling:** The results suggest that your program is particularly efficient with mid-sized files (around 100MB), where it greatly outperforms `dd` in speed. For very large files (1000MB), both methods tend to level out in performance.

In conclusion, our program outperforms `dd` in terms of speed and time duration across various file sizes.

## Extra Credit:

This code is designed to benchmark the performance of the previous program used in dd performance comparison. It is integrated with the Google Benchmark library, which is used for benchmarking code snippets. The code performs file copy operations on test files of various sizes and measures their performance in terms of speed and efficiency.

Steps:

1. Compile the C++ benchmark code (bench\_test.cpp). Make sure to link against Google Benchmark and pthread.

Command:

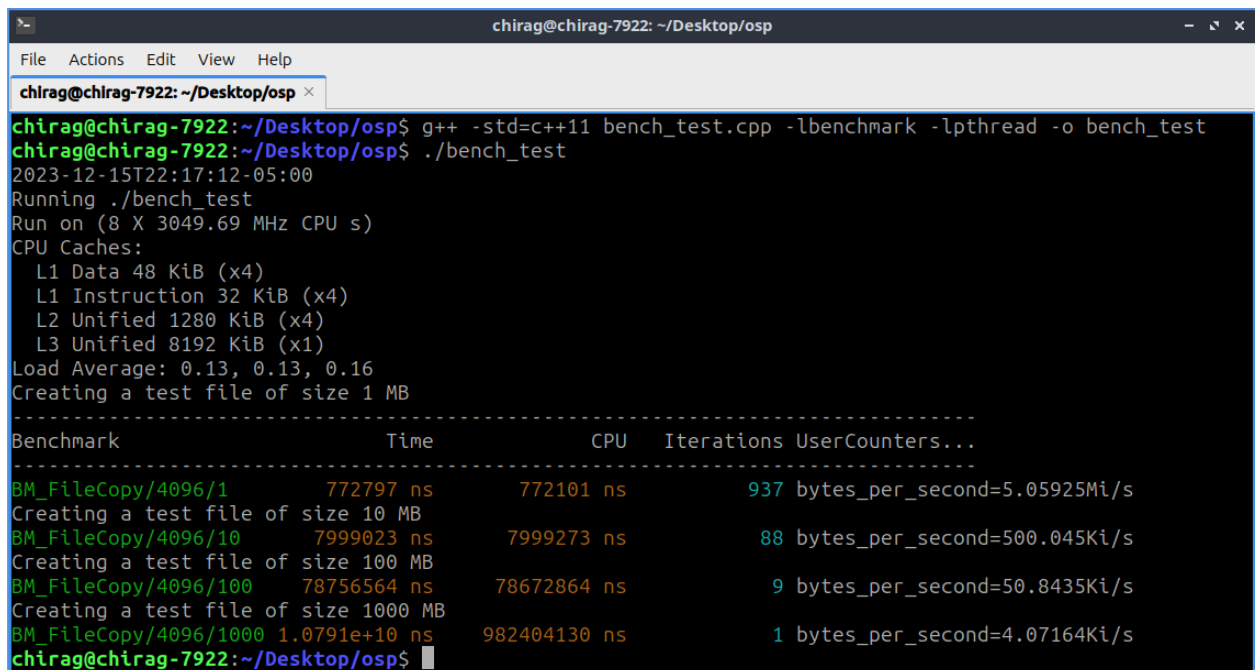
```
g++ -std=c++11 bench_test.cpp -lbenchmark -lpthread -o bench_test
```

2. Run the benchmark:

Command:

```
./bench_test
```

This setup will perform benchmarks on our file copying program with block size of 4096 and file sizes of 1MB, 10MB, 100MB and 1000MB. The Google Benchmark framework will handle the timing and iteration of the tests.



```
chirag@chirag-7922: ~/Desktop/osp
File Actions Edit View Help
chirag@chirag-7922: ~/Desktop/osp x
chirag@chirag-7922:~/Desktop/osp$ g++ -std=c++11 bench_test.cpp -lbenchmark -lpthread -o bench_test
chirag@chirag-7922:~/Desktop/osp$ ./bench_test
2023-12-15T22:17:12-05:00
Running ./bench_test
Run on (8 X 3049.69 MHz CPU s)
CPU Caches:
  L1 Data 48 KiB (x4)
  L1 Instruction 32 KiB (x4)
  L2 Unified 1280 KiB (x4)
  L3 Unified 8192 KiB (x1)
Load Average: 0.13, 0.13, 0.16
Creating a test file of size 1 MB
-----
Benchmark                                Time          CPU    Iterations  UserCounters...
-----
BM_FileCopy/4096/1                      772797 ns      772101 ns        937 bytes_per_second=5.05925Mi/s
Creating a test file of size 10 MB
BM_FileCopy/4096/10                     7999023 ns     7999273 ns         88 bytes_per_second=500.045Ki/s
Creating a test file of size 100 MB
BM_FileCopy/4096/100                   78756564 ns    78672864 ns          9 bytes_per_second=50.8435Ki/s
Creating a test file of size 1000 MB
BM_FileCopy/4096/1000 1.0791e+10 ns    982404130 ns          1 bytes_per_second=4.07164Ki/s
chirag@chirag-7922:~/Desktop/osp$
```

File Size (MB)	Time (ns)	CPU Time (ns)	Iterations	Speed (MiB/s)
1	772,797	772,101	937	5.05925
10	7,999,023	7,999,273	88	0.500045
100	78,756,564	78,672,864	9	0.0508435
1000	10,791,000,000	982,404,130	1	0.00407164

### Interpretation:

1. **Performance Decrease with File Size:** There is a noticeable decrease in performance (speed) as the file size increases. The speed drops significantly from 5.05925 MiB/s for a 1 MB file to just 0.00407164 MiB/s for a 1000 MB file.
2. **Time and CPU Time:** Both the time taken and CPU time increase as the file size increases. This is expected since larger files require more data to be read and written, consuming more time and CPU resources.
3. **Speed Inconsistency:** The drastic drop in speed (bytes per second) suggests that the file copying operation becomes significantly less efficient as the size of the file increases. This might be due to various factors such as increased disk I/O, cache misses, or other system resource limitations for larger files.

### 3. Raw Performance

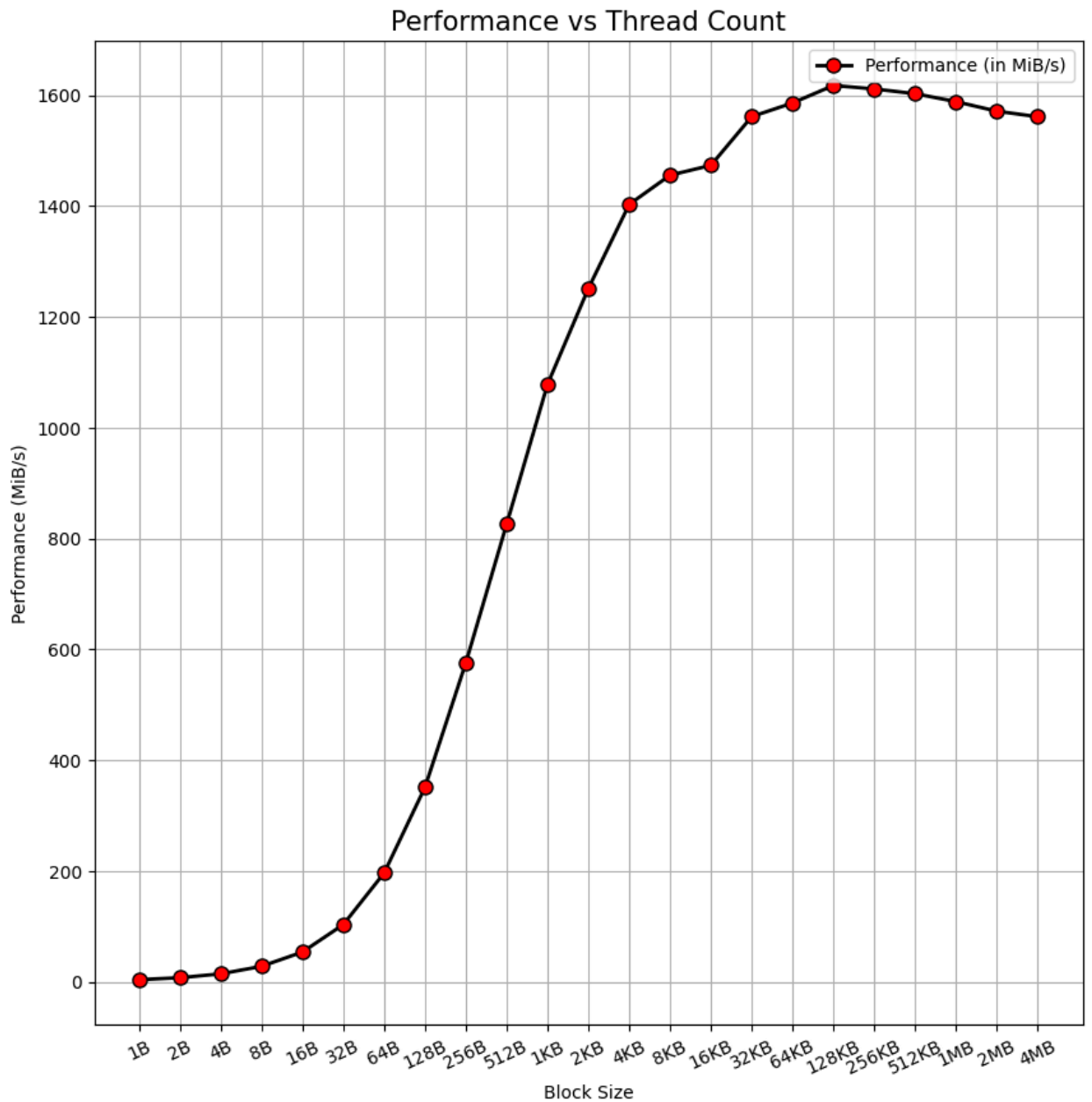
With the file of reasonable size, we ran our read subroutine on block sizes varying from 1B to 4MB.

Block Size (B)	Performance (MiB/s)
1B	3.57
2B	7.184
4B	14.12
8B	27.79
16B	53.53
32B	103.08
64B	196.26
128B	350.9
256B	575.58
512B	825.97
1KB	1078.18
2KB	1251.19
4KB	1403
8KB	1455.62
16KB	1473.5
32KB	1561.39
64KB	1586.06
128KB	1617.65
256KB	1611.33
512KB	1602.83
1MB	1588.47
2MB	1571.22
4MB	1561.23

Usage: ./run3 <filename> <block\_size>



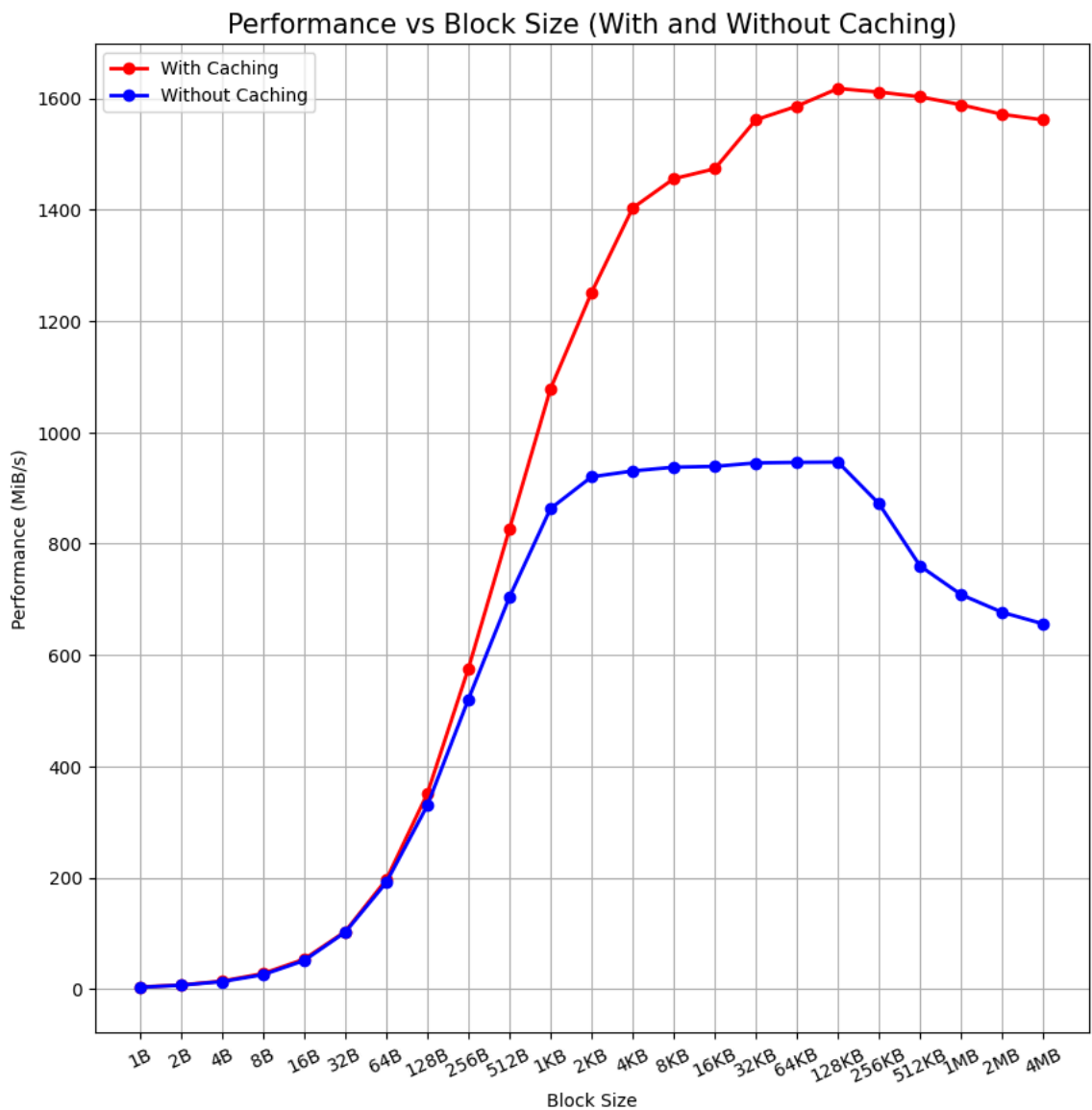
Plotting the above values, we obtain the following graph:



It has been observed that as the block size increases, the reading speed also increases up to a certain point. Beyond this point, the speed plateaus, as it reaches the maximum achievable speed for reading a file using the read system call.

## 4. Caching

We noticed that executing the code consecutively multiple times resulted in significantly faster read speeds. This is due to the system's ability to cache data that has already been read, allowing it to retrieve the same data from RAM when possible. This method is more efficient and optimal than accessing the same file from physical memory, as it is less resource-intensive.



While measuring uncached runtimes using `run3.c`, we cleared the disk using the suggested command: `sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"`

Block Size (B)	Performance with caching (MiB/s)	Performance without caching (MiB/s)
1B	3.57	3.1
2B	7.184	6.87
4B	14.12	13.36
8B	27.79	25.68
16B	53.53	51.11
32B	103.08	101.92
64B	196.26	191.28
128B	350.9	329.58
256B	575.58	520.29
512B	825.97	703.88
1KB	1078.18	863.02
2KB	1251.19	920.12
4KB	1403	930.7
8KB	1455.62	937.43
16KB	1473.5	938.89
32KB	1561.39	945.1
64KB	1586.06	946.1
128KB	1617.65	946.61
256KB	1611.33	871.68
512KB	1602.83	759.9
1MB	1588.47	708.44
2MB	1571.22	676.71
4MB	1561.23	655.96

Usage: ./cachetest.sh <filename>

## Extra Credit:

There are 3 options to clear the cache without interrupting any processes or services:

1. echo 1 : to clear page cache only
2. echo 2 : to clear dentries and inodes
3. echo 3 : to clear page cache, dentries and inodes

An **Inode** is a data structure which provides a representation of a file.

**Dentries** is a data structure which represents a directory or a folder. So, dentries can be used to store cache, if they exist then store or check from memory.

The **Page cache** is anything the OS could store in the memory or hold it. Record of pages which are read from secondary storage.

While evaluating the raw performance of our non-cached reads, it was crucial to clear the cache entirely to obtain accurate results. That's why we opted for mode 3 of cache clearing, which ensures that all cached data is removed and allows us to measure the true performance of our system.

## 5. System Calls

Keeping block size as 1 Byte, we tried various system calls to see their performance.

We experimented with the following system calls:

read

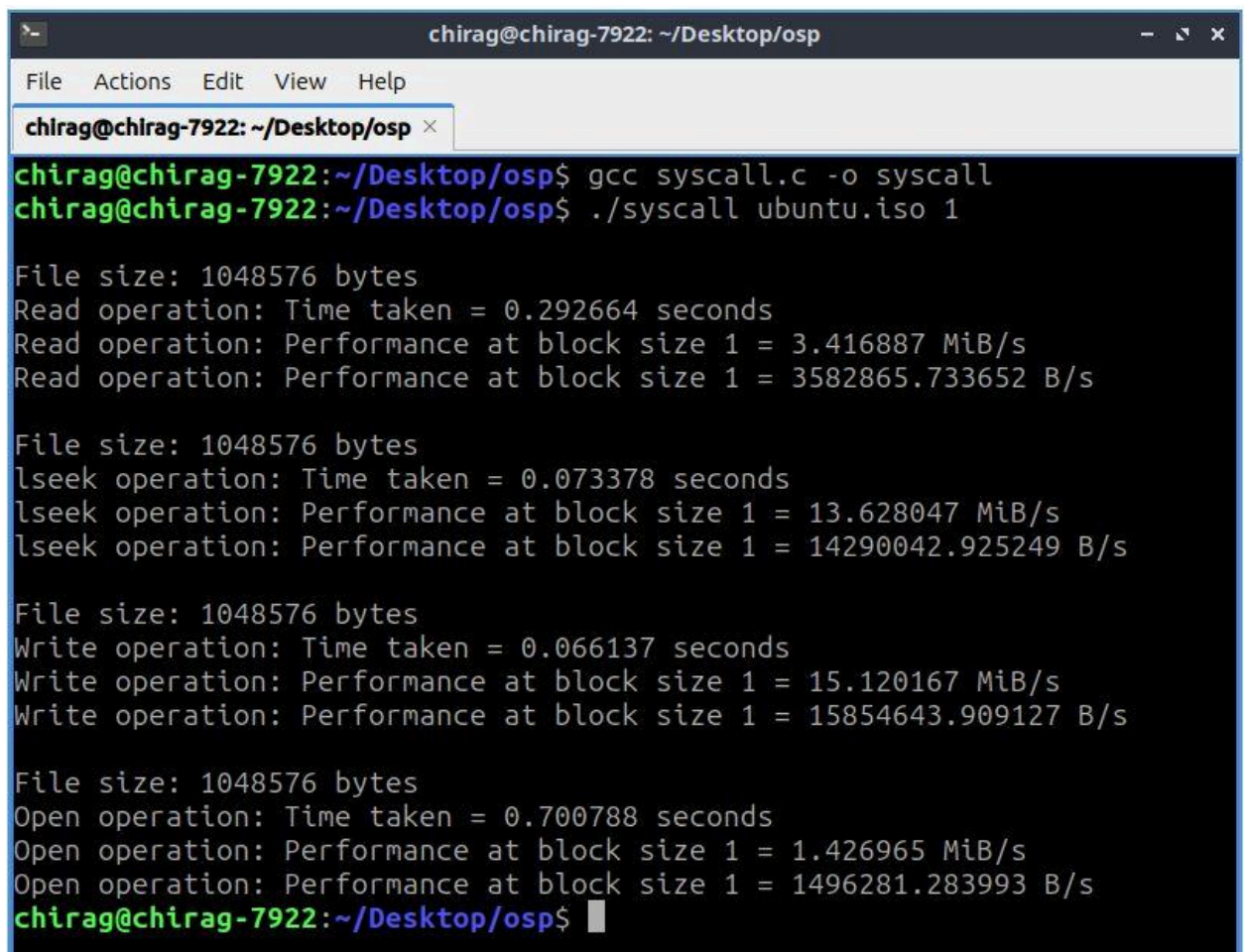
write

open

lseek

Usage: ./syscall <filename> <block\_size>

Output:

A terminal window titled 'chirag@chirag-7922: ~/Desktop/osp' with a menu bar (File, Actions, Edit, View, Help) and a tab labeled 'chirag@chirag-7922: ~/Desktop/osp'. The terminal shows the execution of a program 'syscall' which measures the performance of various system calls on a file named 'ubuntu.iso' (1048576 bytes). The program is run with the command './syscall ubuntu.iso 1'. The output shows three sets of results: Read operation, lseek operation, and Write operation. Each set includes the time taken and performance in MiB/s and B/s at a block size of 1.

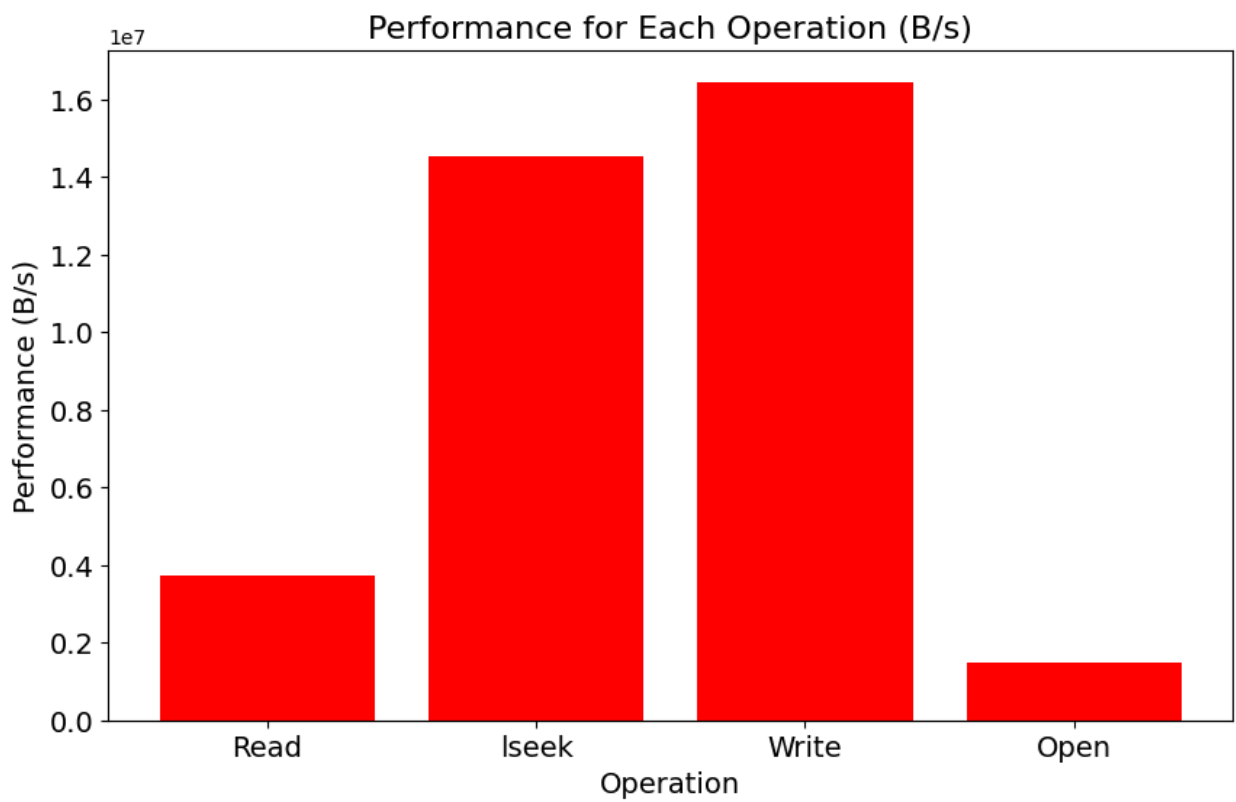
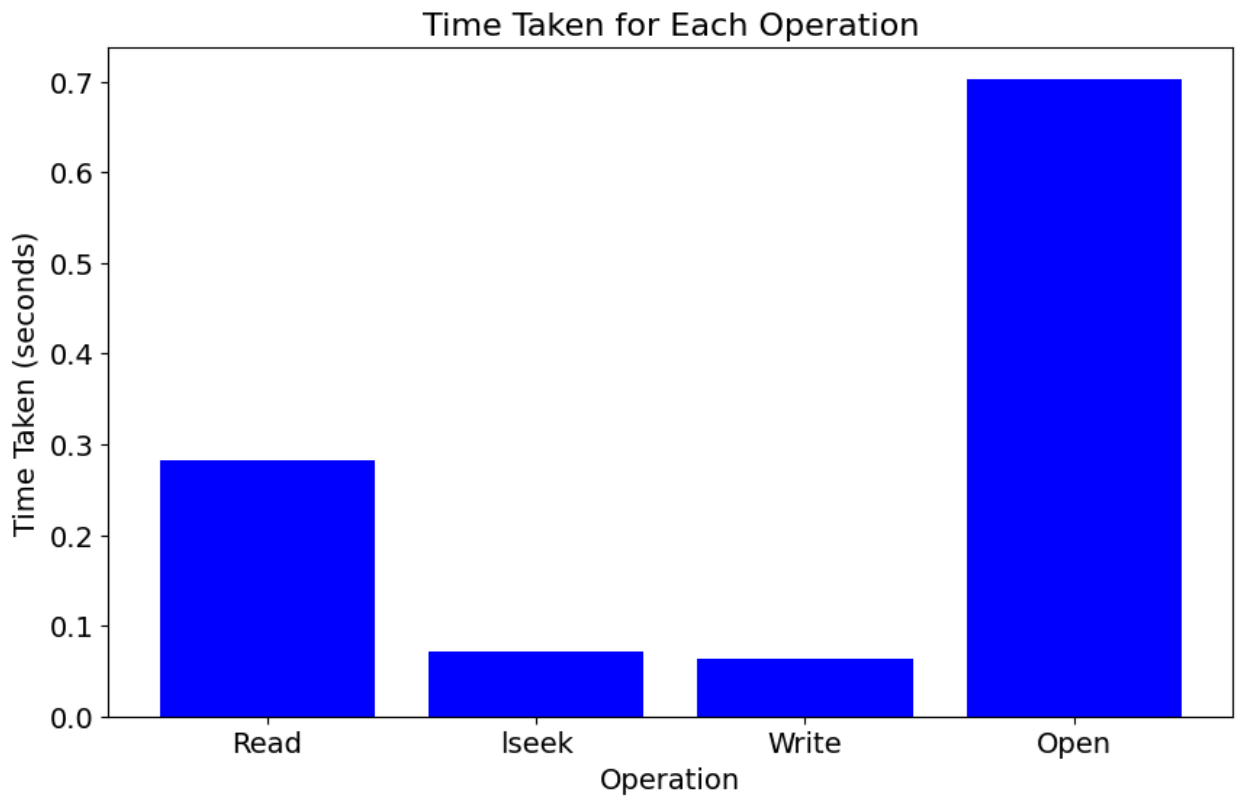
```
chirag@chirag-7922:~/Desktop/osp$ gcc syscall.c -o syscall
chirag@chirag-7922:~/Desktop/osp$ ./syscall ubuntu.iso 1

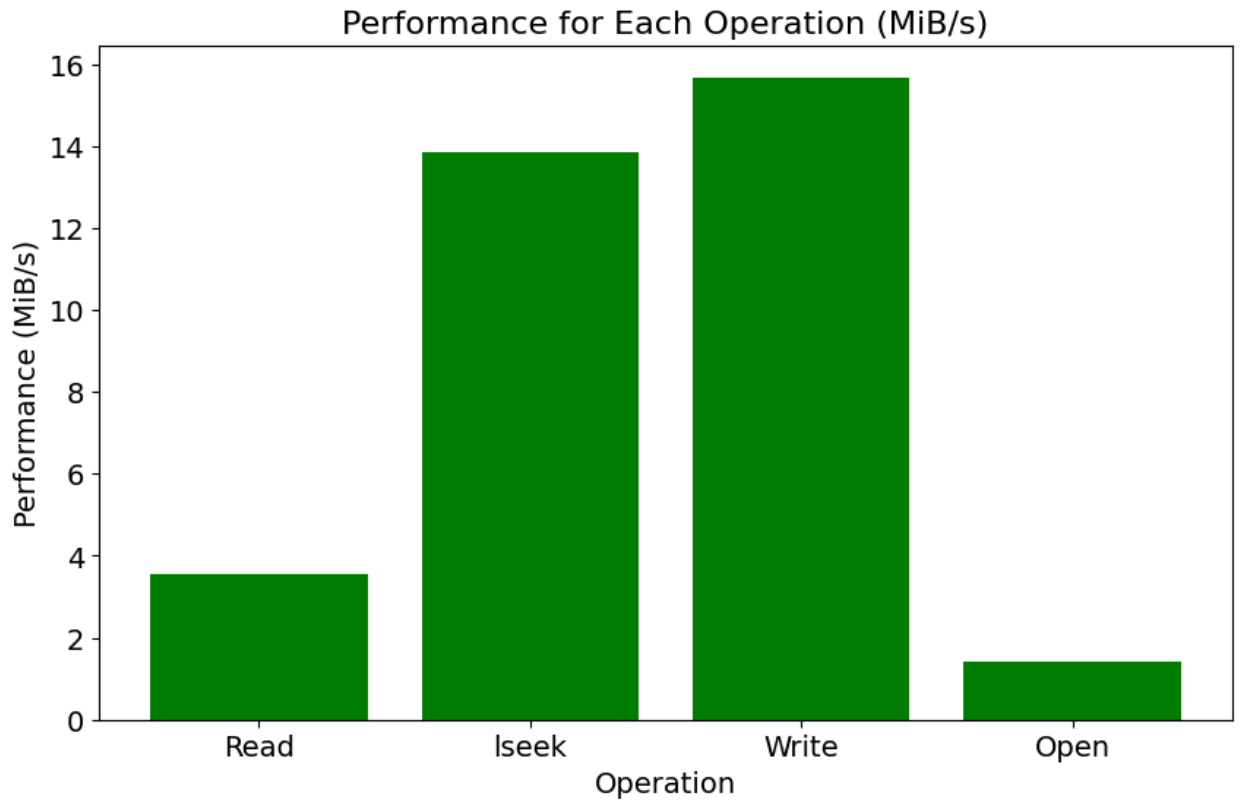
File size: 1048576 bytes
Read operation: Time taken = 0.292664 seconds
Read operation: Performance at block size 1 = 3.416887 MiB/s
Read operation: Performance at block size 1 = 3582865.733652 B/s

File size: 1048576 bytes
lseek operation: Time taken = 0.073378 seconds
lseek operation: Performance at block size 1 = 13.628047 MiB/s
lseek operation: Performance at block size 1 = 14290042.925249 B/s

File size: 1048576 bytes
Write operation: Time taken = 0.066137 seconds
Write operation: Performance at block size 1 = 15.120167 MiB/s
Write operation: Performance at block size 1 = 15854643.909127 B/s

File size: 1048576 bytes
Open operation: Time taken = 0.700788 seconds
Open operation: Performance at block size 1 = 1.426965 MiB/s
Open operation: Performance at block size 1 = 1496281.283993 B/s
chirag@chirag-7922:~/Desktop/osp$
```





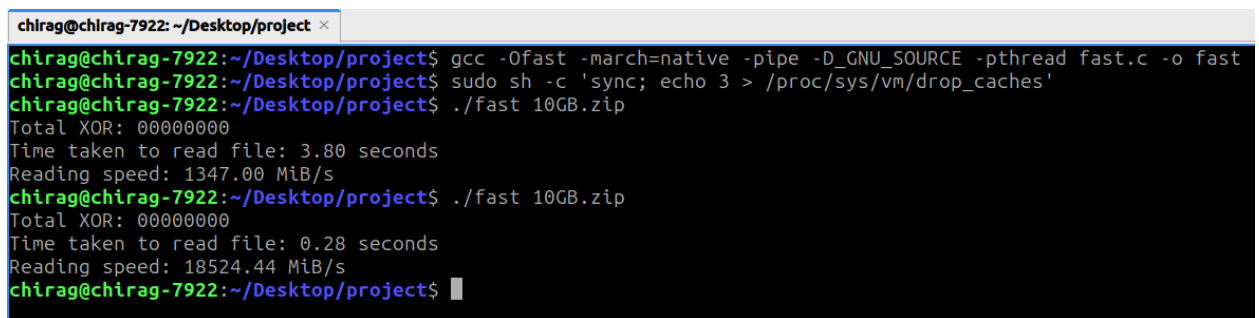
We conducted a comparison of the system calls `lseek()`, `read()`, `open()`, and `write()` to measure the overhead associated with these operations. Our observations revealed that system calls are significantly more expensive than normal function calls. This is due to the overhead incurred when a system call is made, such as the context switching between user space and kernel, and the transfer of metadata across the user-space boundary.

## 6. Raw Performance

To determine the ideal block size, we explore values around the peak performance obtained in Part 3 and 4. We found 128KB to be the ideal block size.

So, for our system we use `block_size` as 131072 bytes or 128KB in the code. Need to change that according to the system used.

Output:

A terminal window titled 'chirag@chirag-7922: ~/Desktop/project' showing the compilation and execution of a program. The user runs 'gcc -Ofast -march=native -pipe -D\_GNU\_SOURCE -pthread fast.c -o fast', then 'sudo sh -c 'sync; echo 3 > /proc/sys/vm/drop\_caches'', and finally './fast 10GB.zip' twice. The first run shows a reading speed of 1347.00 MiB/s, and the second run shows a reading speed of 18524.44 MiB/s.

```
chirag@chirag-7922:~/Desktop/project$ gcc -Ofast -march=native -pipe -D_GNU_SOURCE -pthread fast.c -o fast
chirag@chirag-7922:~/Desktop/project$ sudo sh -c 'sync; echo 3 > /proc/sys/vm/drop_caches'
chirag@chirag-7922:~/Desktop/project$ ./fast 10GB.zip
Total XOR: 00000000
Time taken to read file: 3.80 seconds
Reading speed: 1347.00 MiB/s
chirag@chirag-7922:~/Desktop/project$ ./fast 10GB.zip
Total XOR: 00000000
Time taken to read file: 0.28 seconds
Reading speed: 18524.44 MiB/s
chirag@chirag-7922:~/Desktop/project$
```

We can see without caching the Reading Speed is 1347.00 MiB/s while with caching it is 18524.44 MiB/s. So we definitely have optimized the program as much as we could.

For compiling we used :

```
gcc -Ofast -march=native -pipe -D_GNU_SOURCE -pthread fast.c -o fast
```

-Ofast: This flag enables all the optimizations from -O3 and additionally enables -ffast-math, which relaxes some IEEE or ISO rules/specifications for mathematical calculations.

-march=native: This flag tells the compiler to generate code optimized for the specific architecture of the machine on which you are compiling. It enables all instruction subsets supported by the local machine.

-pipe: This flag uses pipes rather than temporary files for communication between the various stages of compilation. This can speed up the compilation process by reducing disk I/O, but it might increase memory usage.



-D\_GNU\_SOURCE: This is a preprocessor directive that defines the \_GNU\_SOURCE macro. It enables GNU extensions to the standard libraries, allowing the code to use additional features and functionalities provided by the GNU system, beyond what's available in the standard libraries.

-pthread: This flag tells the compiler to link in the POSIX threads library (libpthread). It sets flags for both the preprocessor and linker. This is necessary for compiling code that uses the POSIX threads (pthreads) library.

Usage: ./fast <filename>

We are now trying multiple threads. To facilitate multiple threads reading the same file, we initially split the total number of blocks by the number of threads. Each thread is then assigned to read its specific, non-intersecting blocks. After each thread has read its assigned block and computed the xor value for that block, the results from all threads are combined (xor'ed) to produce the final output. Each thread is given its block\_offset and the number of blocks it needs to read, and it also maintains a record of the number of bytes it has read, which aids in calculating speed.

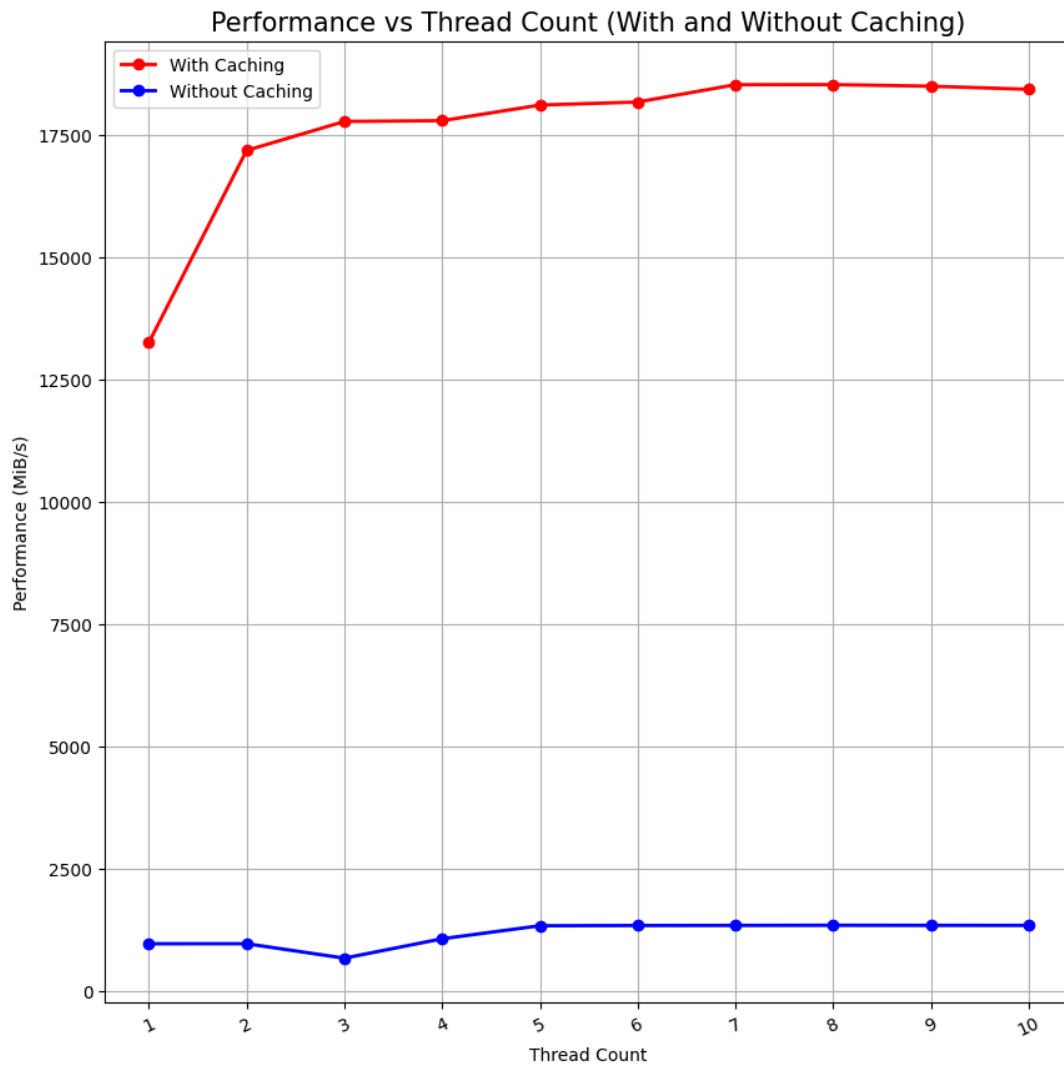
In our program, the default usage is ./fast <filename> where it will use 8 threads and 131072 block size which we determined to be the most optimal for our system. But, we can also mention the threads and block\_size if we want custom input.

Usage: ./fast <filename> <thread\_count> <block\_size>

Using thread\_count from 1 to 10, we ran our code for performance check on different thread\_count.

Thread Count	Performance with caching (MiB/s)	Performance without caching (MiB/s)
1	13252.85	969.41
2	17181.09	969.67
3	17773.43	674.15
4	17789.19	1070.65
5	18108.42	1336.96
6	18170.55	1341.68
7	18525.03	1343.59
8	18526.6	1346.06
9	18493.57	1344.4
10	18430.81	1344.04

Output:



We observe that as the number of threads increases, so does the performance. Given that we're operating on a 4 cores, 8-threads system, it's unsurprising that the performance dips when we use 9 or 10 threads. To maximize file reading speed, we employ a block size of 128KB or 131072 bytes and utilize 8 threads for execution.

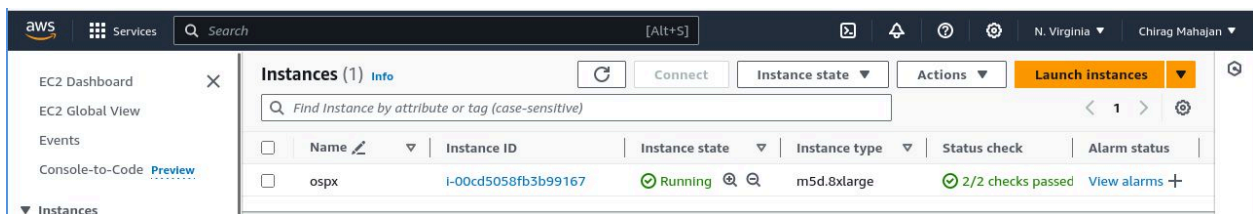
For performance without caching, we can see a plateau with not a steep increase in performance.

## Extra Credit:

We created an AWS EC2 instance 'm5d.8xlarge' (since we are only allowed this one which is of maximum capacity). We then connected to the instance using SSH.

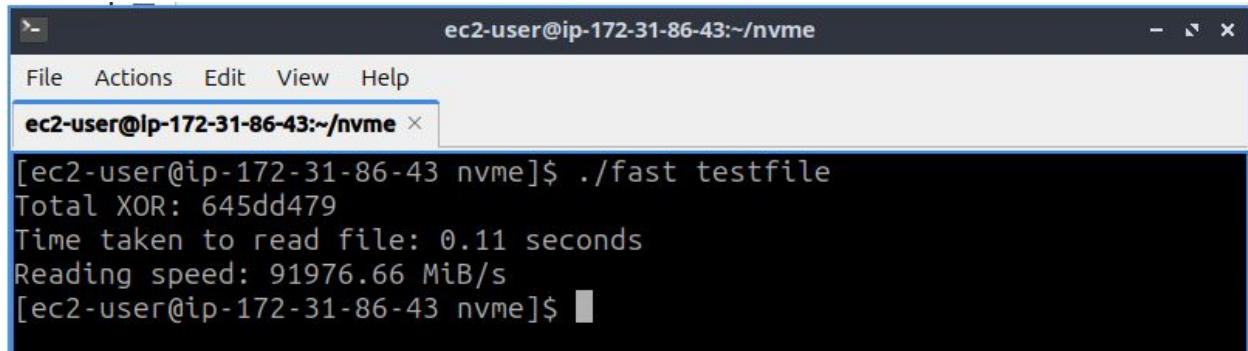
Steps:

1. Launch an EC2 instance.
2. Connect to the instance.
3. Prepare the NVMe drive and mount it to a directory.
4. Install necessary software like C compiler.
5. Transfer code to the instance.
6. Compile and run the program.



Output:

```
ec2-user@ip-172-31-86-43:~/nvme
File Actions Edit View Help
ec2-user@ip-172-31-86-43:~/nvme x
[ec2-user@ip-172-31-86-43 ~]$ cd nvme
[ec2-user@ip-172-31-86-43 nvme]$ gcc -Ofast -pthread -pipe -march=native -D_GNU_SOURCE fast.c -o fast
[ec2-user@ip-172-31-86-43 nvme]$ sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"
[ec2-user@ip-172-31-86-43 nvme]$ ./fast testfile
Total XOR: 645dd479
Time taken to read file: 8.18 seconds
Reading speed: 1251.14 MiB/s
[ec2-user@ip-172-31-86-43 nvme]$
```

A screenshot of a terminal window titled "ec2-user@ip-172-31-86-43:~/nvme". The terminal shows the command `./fast testfile` being executed. The output is as follows:  
[ec2-user@ip-172-31-86-43 nvme]\$ ./fast testfile  
Total XOR: 645dd479  
Time taken to read file: 0.11 seconds  
Reading speed: 91976.66 MiB/s  
[ec2-user@ip-172-31-86-43 nvme]\$

In our program for this instance, the default usage is `./fast <filename>` where it will use 64 threads and 1048576 block size which we determined to be the most optimal for the EC2 instance.

Performance without caching: 91976.66 MiB/s

Performance with caching: 1251.14 MiB/s

The significant difference in performance between the EC2 instance (md5.8xlarge) and our local system when running the `fast.c` program can be attributed to several factors:

**Instance Type and Resources:** The m5d.8xlarge instance on EC2 has far superior hardware resources compared to our personal system. These instances are designed for high performance and can handle I/O operations much faster.

**Caching Impact:** The presence of caching can greatly enhance read speeds, as seen in both environments. Caching allows data to be served from faster storage like RAM instead of re-reading from the disk. The larger and faster cache in the EC2 instance contributes to its higher performance with caching.

**Disk Type and Configuration:** EC2 instances generally use SSDs or even faster NVMe drives, which have significantly higher read/write speeds compared to standard HDDs that are used in our personal system. Additionally, the configuration and setup of the storage system on EC2 instances are optimized for high performance.

**Network and Infrastructure:** In an EC2 environment, the underlying infrastructure including network and storage architecture is optimized for high performance and low latency, which can further contribute to faster I/O operations.

**Operating System and Driver Efficiency:** Differences in operating systems, file systems, and driver efficiencies between our local system and the EC2 instance can also lead to variations in performance.

Overall, the performance difference highlights the impact of hardware capabilities, caching mechanisms, and system optimizations in high-demand environments like EC2 compared to typical personal computing systems.