CS-GY 6233 Final Project Report

Students	Name	NetId
1.	Harsh Sanjay Apte	ha2179@nyu.edu
2.	Savani Manoj Gokhale	sg6428@nyu.edu

Instructions to Run the code-

Executing *./build* command will compile every file required for Step 1, 2 and 6 in the root repo. Each compiled can be executed by following commands -

1 : *chmod +x build.sh*

2 : ./build.sh

Step 1: ./run “<filename>” [-r|-w] <block_size> <block_count>

Step 2 : ./run2 “<filename>” <block_size>

Step 6 : ./fast “<filename>”

IMP - File to be read must be kept in the same folder where executables are kept

Step 1: Basic Read Write -

Functionality to read file as per given by user from command line was implemented by using *read()* system call. Similarly, *write()* system call used to write the file as input given by the user i.e. block size and block count.

Execution → *./run.out <filename> -r/-w <block_size> <block_count>*

Step 2 : Measurement -

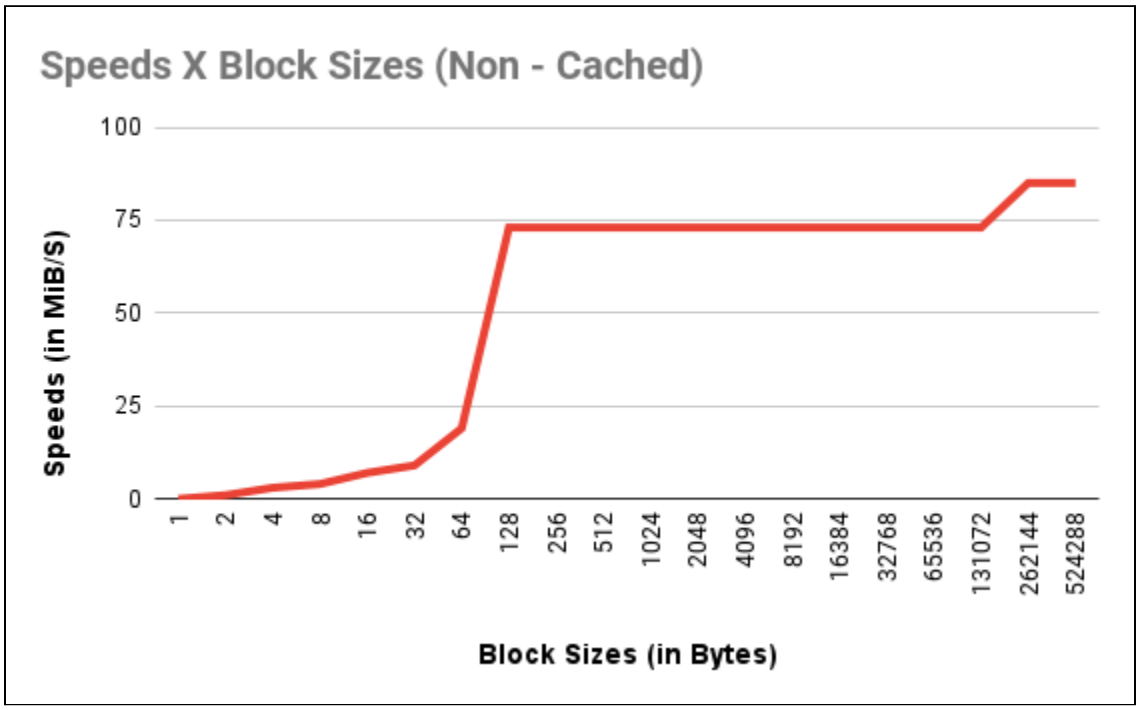
A functionality to find out an optimal block count which can be read in reasonable time i.e. between 5-15 seconds was implemented. We picked up a block size starting from 1 and kept it increasing till the maximum limit stated - 128 MiB, for each block size, we found an optimal block count. The logic used was passing the block size and block count combination to Step 1's code, doubling the block count for every iteration until the time threshold of 5 seconds is breached. Returned block count and file size which can be read in reasonable time.

Execution → *./run2.out <filename> <block_size>*

Step 3 - Raw Performance -

Measured raw performance by passing different block size - block count combinations. Observed that speed increases as we keep increasing the block size till a certain limit, after that it plateaus out as that's the maximum speed which can be obtained for reading a file by read system call.

Performance Graph for Step 3 -

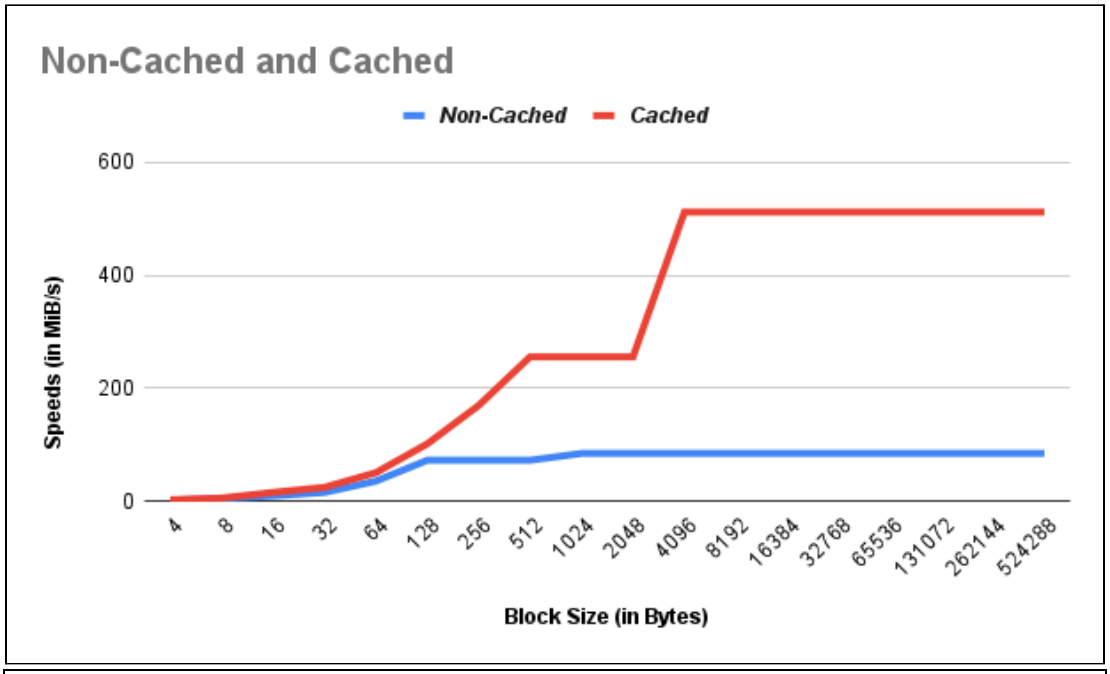


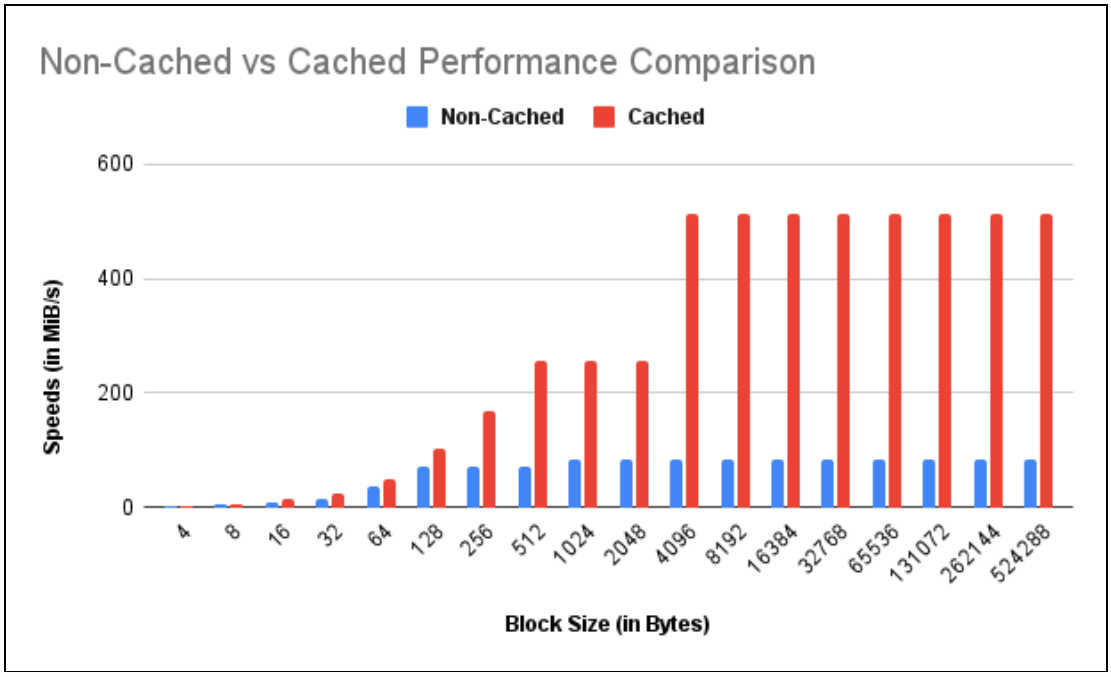
As seen in the graph diagram performance was flattened out at around speed of **85 MiB/s** at the block size of **524288 bytes / 0.5 MBits**

Step 4 - Caching -

It was observed that read speeds were much faster if Step 1's code was executed back to back for multiple times. That's because system caches already read data and fetches the same data from RAM when possible which is less expensive and optimal than going into physical memory and accessing the same file.

Performance Comparison Graph for Step 4 -





As we can observe for larger block sizes, cached performance is much faster than non-cached one. While clearing the system caches *sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"* was used.

Why 3 in the cache clearing command?

Answer - As we're experimenting on linux systems, while reading about it we got to know that each linux system/distribution has 3 modes to clear caches. We specify those modes 1,2 or 3 by passing it as a parameter from CLI/shell. 1 clears only page caches from memory, 2 clears dentries and inodes, whereas 3 clears all of it i.e. page cache, dentries and inodes.

Page Cache - Record of pages which are read from secondary storage
Dentries - Data Structure in OS which represents folders/directories
Inodes - Metadata files of OS

As we're evaluating our fresh reading performance for every non-cached read, it was important to clear everything from cache to note raw performance, that's why we used mode 3 of cache clearing.

Step 5 - System Call Performance -

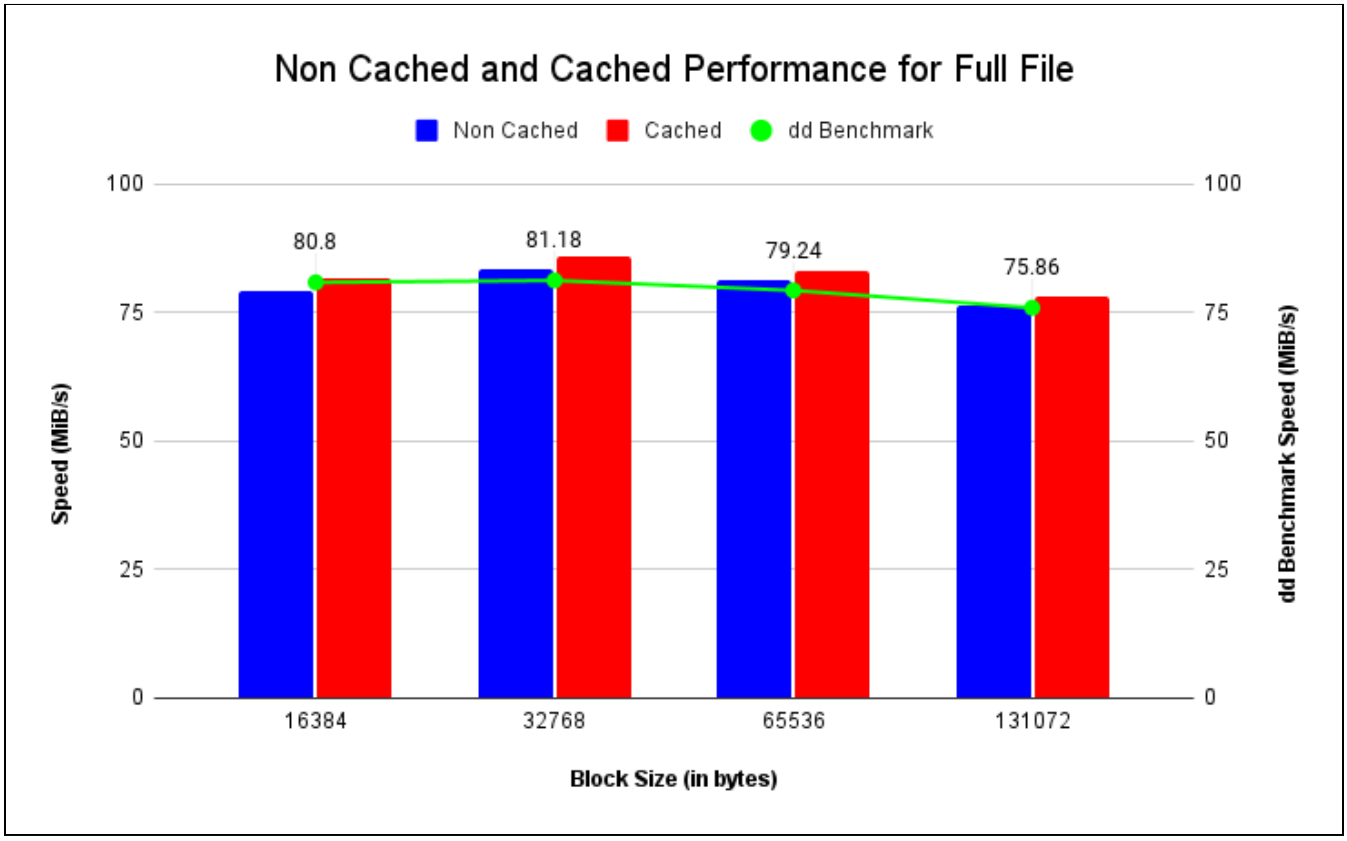
We compared lseek() and read() system calls to quantify how expensive system calls are. Performance observation were as follows -

	Block Size	Bytes/Sec	MiB/s
lseek()	1	5,208,266	4.96
read()	1	838,860	0.80

Step 6 - Final Raw Performance and XOR Check -

Optimal block size was found from previous steps' data which can be used to read large files in optimal size. When compared, it is observed that our testing functionality is beating linux's default "dd" performance.

Performance observation Graph -



As it can be seen in the diagram, ***32768 bytes*** is the optimal block count found for our system to read large files in optimal time. Also, the code performance has achieved faster read rates than *dd.*