NYU Tandon School of Engineering
Computer Science and Engineering
CS-GY 6233, Fall 2021

Final Project Report (due December 16, 11:55 p.m)

Contributors

This homework is done and being submitted collectively by:

[1] Manish Singh Saini (myself): "mss9238"

[2] Geethanjali Devendiran (hw partner): "gd2148"

How to run the code?

First things first, how to run the code:

Steps:

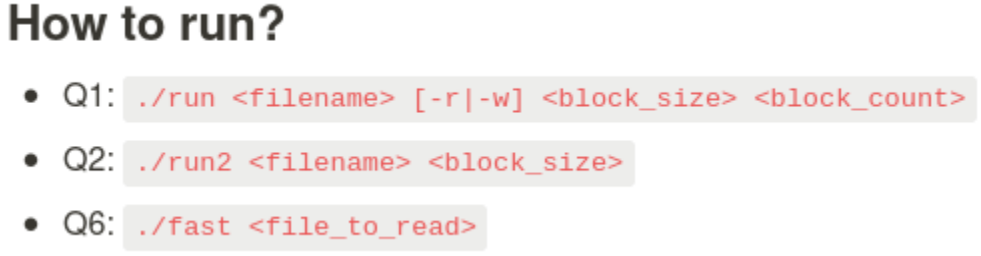
[1] Extract the tar gz of the code in any directory. You should see a bunch of code files in it.

[2] Run the below command on the terminal (make sure you are in the same directory as the code files):

bash build.sh

The above command will compile all that's needed to run the code.

[3] Run the code as per the instructions published in the project description on Notion website by the prof.. Attaching them below for your reference:



Notes:

If at any point you are wondering about the code structure and want to take a closer look at something, below is the description of the overall code structure:

- *run.c:* contains the main function required to execute **Part 1**
- *run2.c:* contains the main function required to execute **Part 2**
- *fast.c:* contains the main function required to execute **Part 6**

- *analysis.c:* contains the main function required to do analysis/measurements for Part 3,4 and 5. Ideally, you would not interact with this since we will already publish the results as graphs later in this report, but if you are curious, the command would be: *./analysis <filename> [2 | 3 | 4]* , where 2 -> Part 2, 3 -> Part 3, 4 -> Part 4. If you run it, expect to wait anywhere b/w a few minutes to an hour to look at the complete results. For more information, please check the code.

- *utils.c:* contains various utility/core functions (like read, write, readFast, lseekFile etc); this is sort of the core code driving functionality in literally all other files: run.c, run2.c, fast.c and analysis.c.

Breakdown

Part 1: Basics

We wrote the code for this as per the instructions in the project description. You can run it as per the instructions.

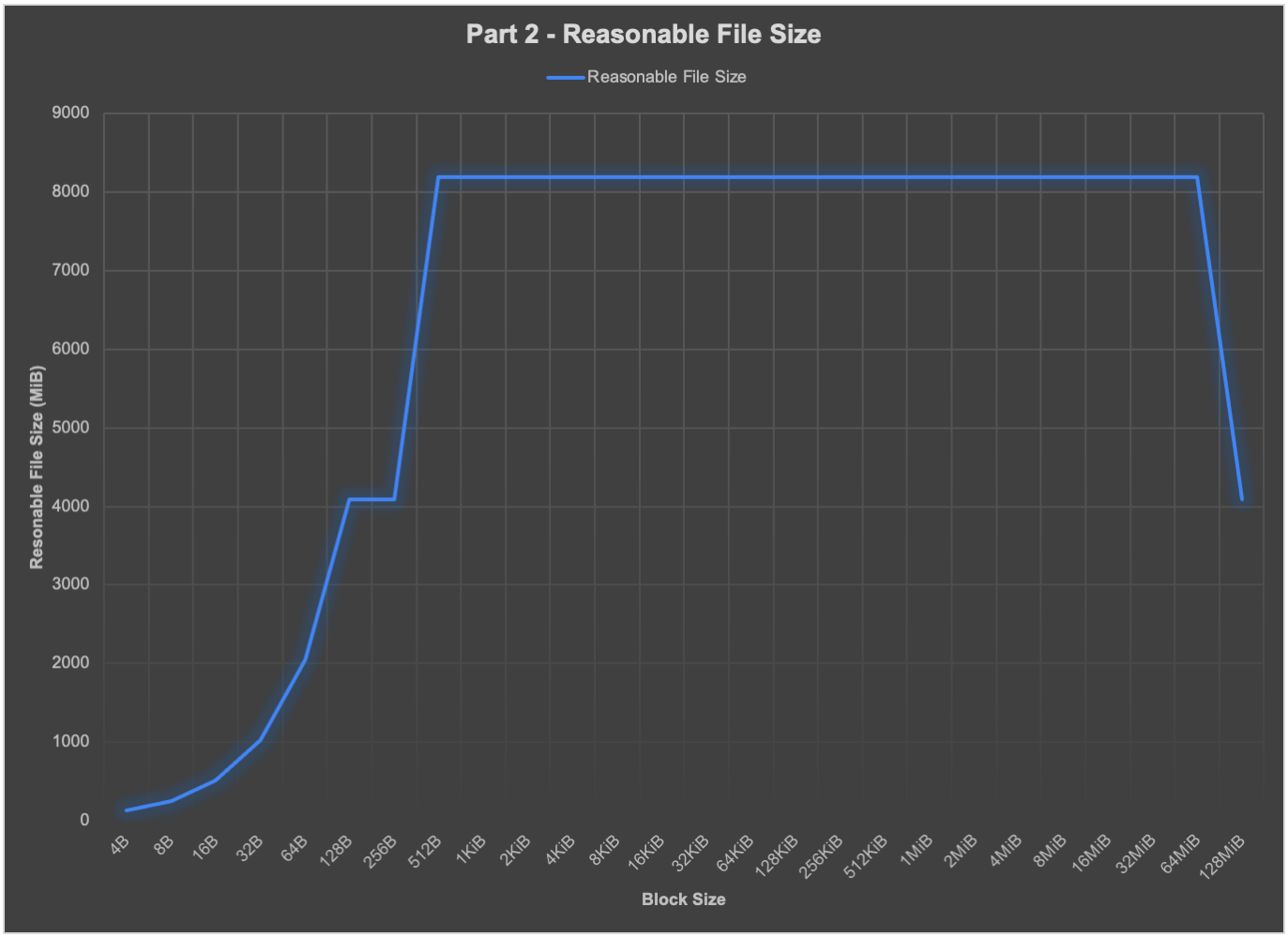
Part 2: Measurements

For this part, we have chosen **8 seconds** as the maximum reasonable time => anything that takes **more than 8 seconds** is NOT reasonable for our experiments.

Having said that, we ran experiments on **block sizes** ranging from **2^2 - 2^27 bytes** i.e **4 bytes - 128 MiB** (each power of 2 being a separate data point on X-axis).

For each block size, we started our analysis from **block count = 2^0,** and going up to as far as **block count = 2^31** to find the reasonable read time (<=8 seconds) for it (although we never really hit that upper bound for any of the block sizes in reasonable time). To make sure we have correct readings (and which are not skewed by outliers), we run **5 iterations each** of reading the file**,** for a given block size and a given block count, to compute the **average read time** and then determine whether it's reasonable enough or not.

Below is our graph that shows **block size on X-axis** and corresponding **"reasonable" file size** (i.e read in <= 8 seconds) **on Y-axis**:



As expected, reasonable file size first increases as we increase the block size till a certain point; and then it plateaus.

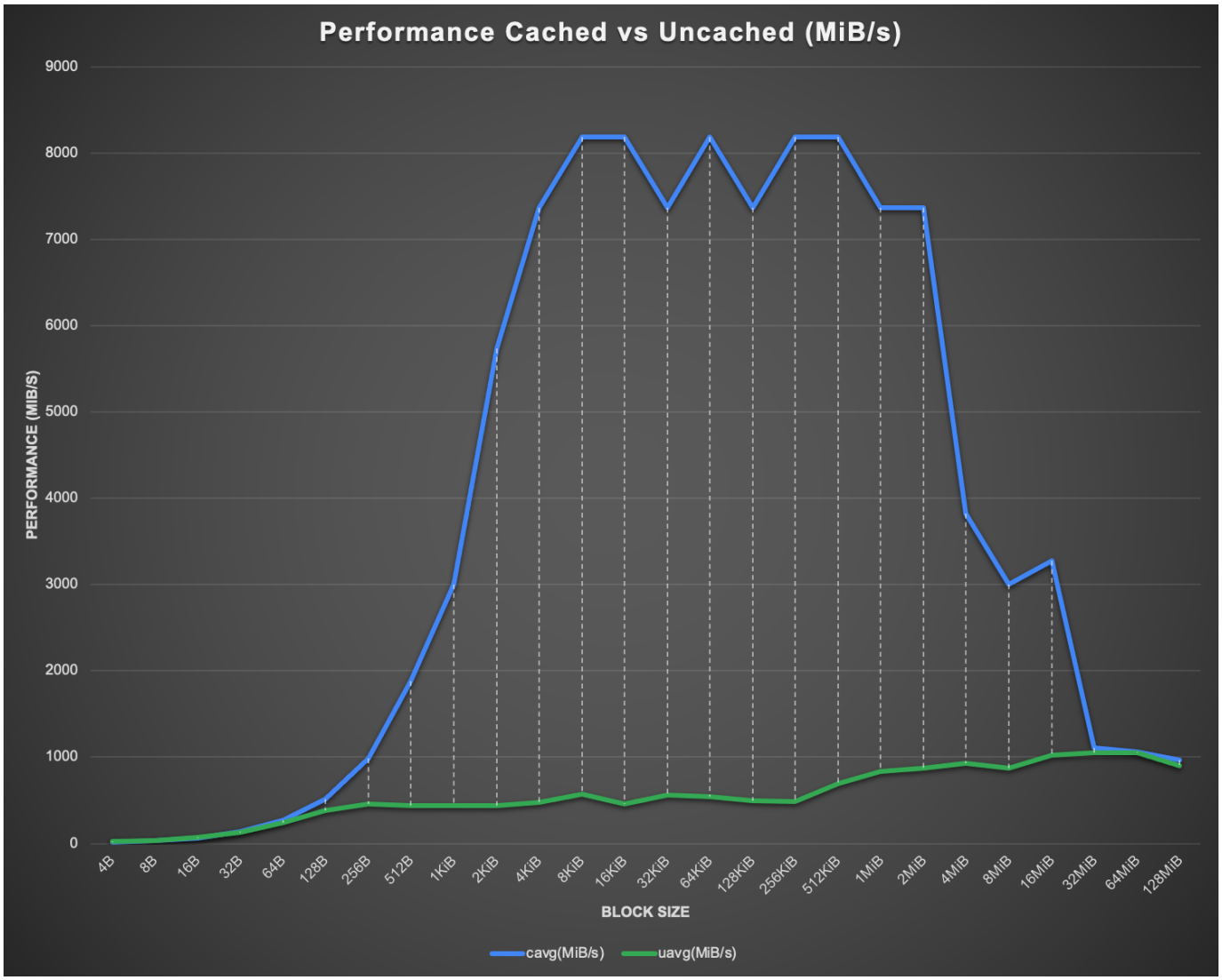
Part 3 & Part 4: Raw Performance & Caching

Since Part 4 subsumes Part 3 (as also mentioned by the prof. on Slack), we will NOT talk about Part 3 separately, rather discuss Part 4.

Just like for Part 2, for Part 4 also, we ran experiments on **block sizes** ranging from **2^2 - 2^27 bytes** i.e **4 bytes - 128 MiB** (each power of 2 being a separate data point on X-axis).

For each **block size**, we first find out the **reasonable file size (or block count)** by running Part 2 implicitly. Once we have a reasonable block count for the given block size, we read the file ((block count * block size) bytes) multiple times, in both cached and non-cached mode, to take multiple measurements to compute **average read performance** and standard error.
Note that we drop the cache after each read (in a loop) while taking un-cached read measurements to get "truly" uncached values.

Below is our graph that shows **block size on X-axis** and **read performance (MiB/s)** (while reading the reasonable file size for the given block size) **on Y-axis**:



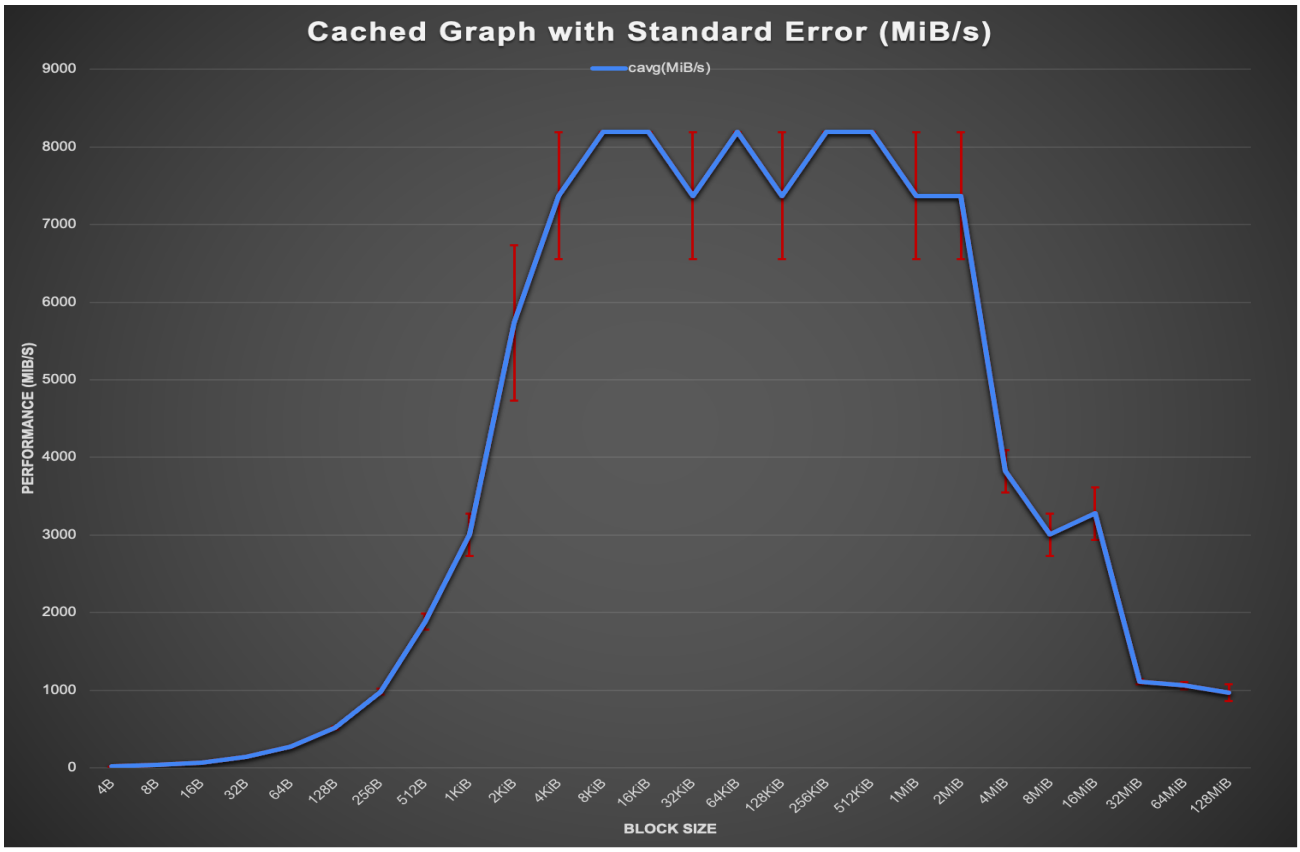
Note that "**cavg**" in the label stands for **"cached average"** and **"uavg"** stands for **"UNcached average"**.

As expected, cached and uncached performances are pretty close when the block size is small (most likely because the corresponding read size is too small for observing any caching benefits), but diverges widely as we start reading with larger block sizes. After a certain point, cached and uncached performances start converging back, since the read/block sizes get too big to be benefitted from the limited disk cache. At far end (>= 32 MiB block size), cached and uncached performances are pracatically the same.

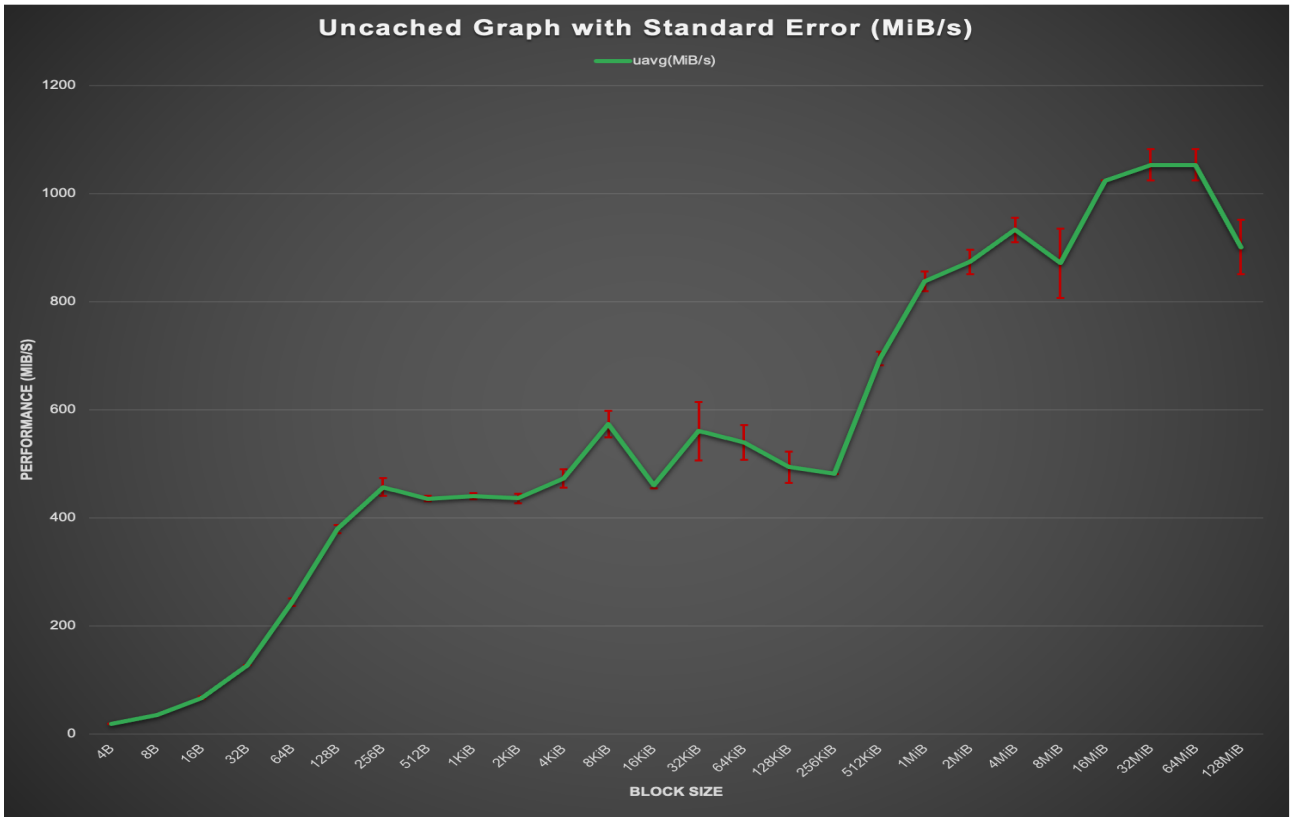
Notes:

Below are our graphs, one each for cached and uncached performances, also showing the **standard error** we observed in our experiments for each data point.

Cached Performance:



UNcached Performance:



Extra credit Question: Why "3" in below command to clear the disk caches on linux? Read up on it and explain.

```
sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"
```

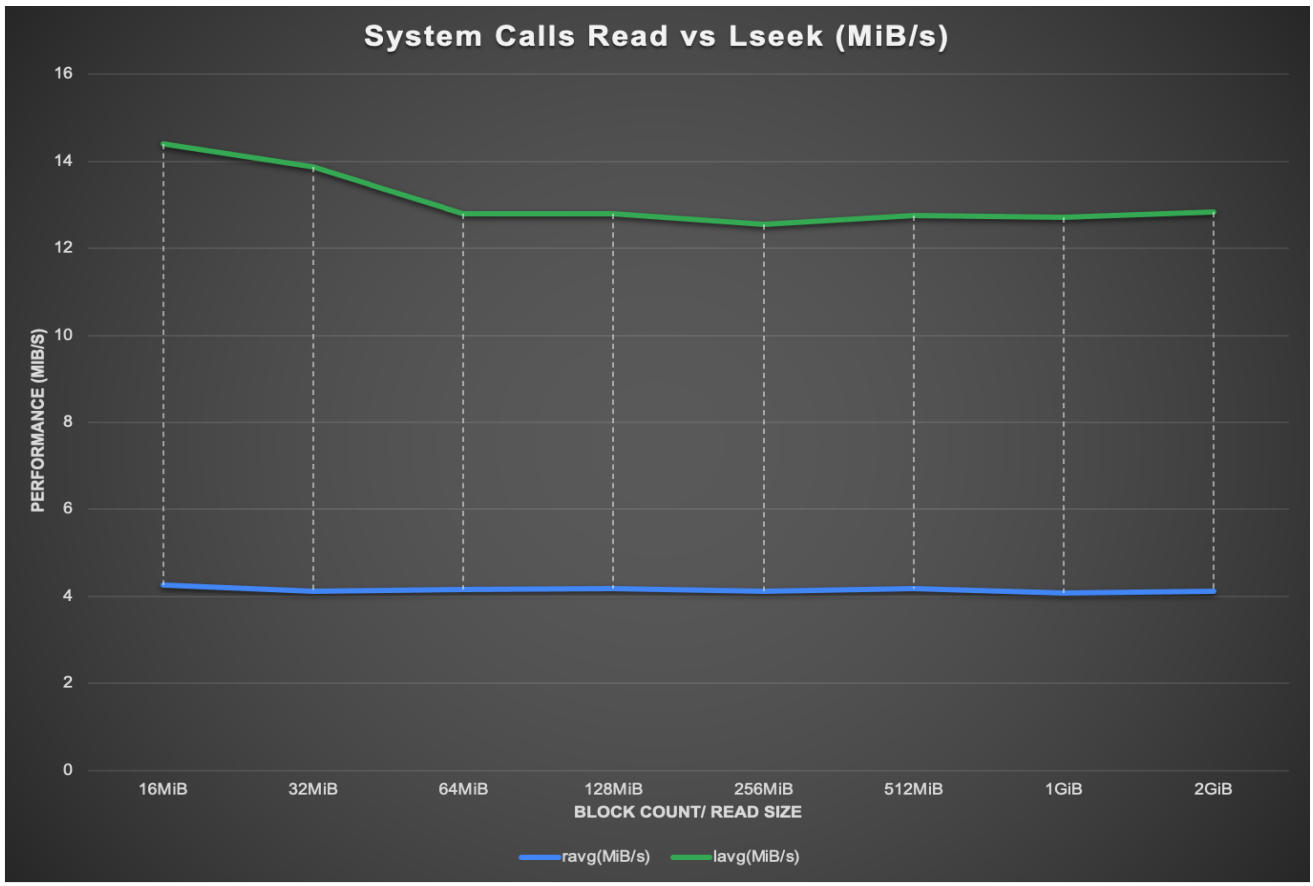
Answer:

There are three types of caches that we can flush in a linux system: page cache, dentries and inodes. Using "1" in the above command would just flush "page cache". Using "2" would only flush "dentries and inodes". To test our "uncached performance" properly, we want to obtain a "truly uncached" environment (without restarting the OS) and therefore we want to drop all three caches: page cache, dentries and inodes. The "3" in the above command does exactly that. That's why "3".

Part 5: System Calls

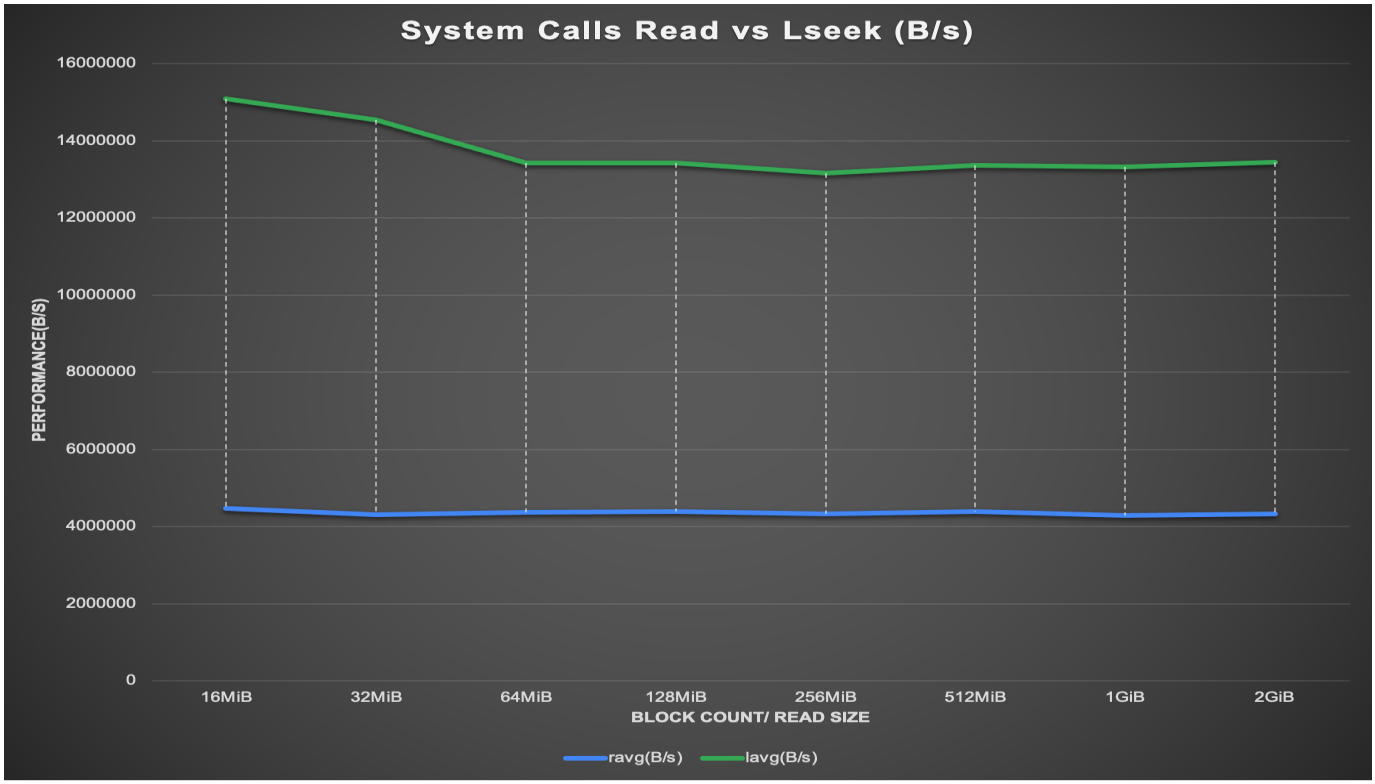
For this part, we fixed the **block size = 1 byte** and ran experiments on **block count** ranging from **2^0 - 2^31** => **read sizes** ranging from **1 B - 128 MiB.**

As expected, reads were extremely fast (~0 seconds) for initial few block counts/read sizes. Therefore, we have excluded these data points from our plots and show observations for block counts/read sizes ranging from **16 MiB - 2 GiB.**
Below is our graph showing **block count/read size** on **X-axis** and **performance (MiB/s)** on **Y-axis**:



Note that, we have also plotted a performance measure for another system call called "lseek" that does even less real work than "read", as also supported by our observations above. Clearly, we are able to do significantly more (by a factor) "lseek" calls than "read" calls per second. Note that **"ravg"** stands for **"read average"** and **"lavg"** stands for **"lseek average".**

Below is the same graph as above, just the Y-axis readings are in **B/s,** instead of MiB/s:

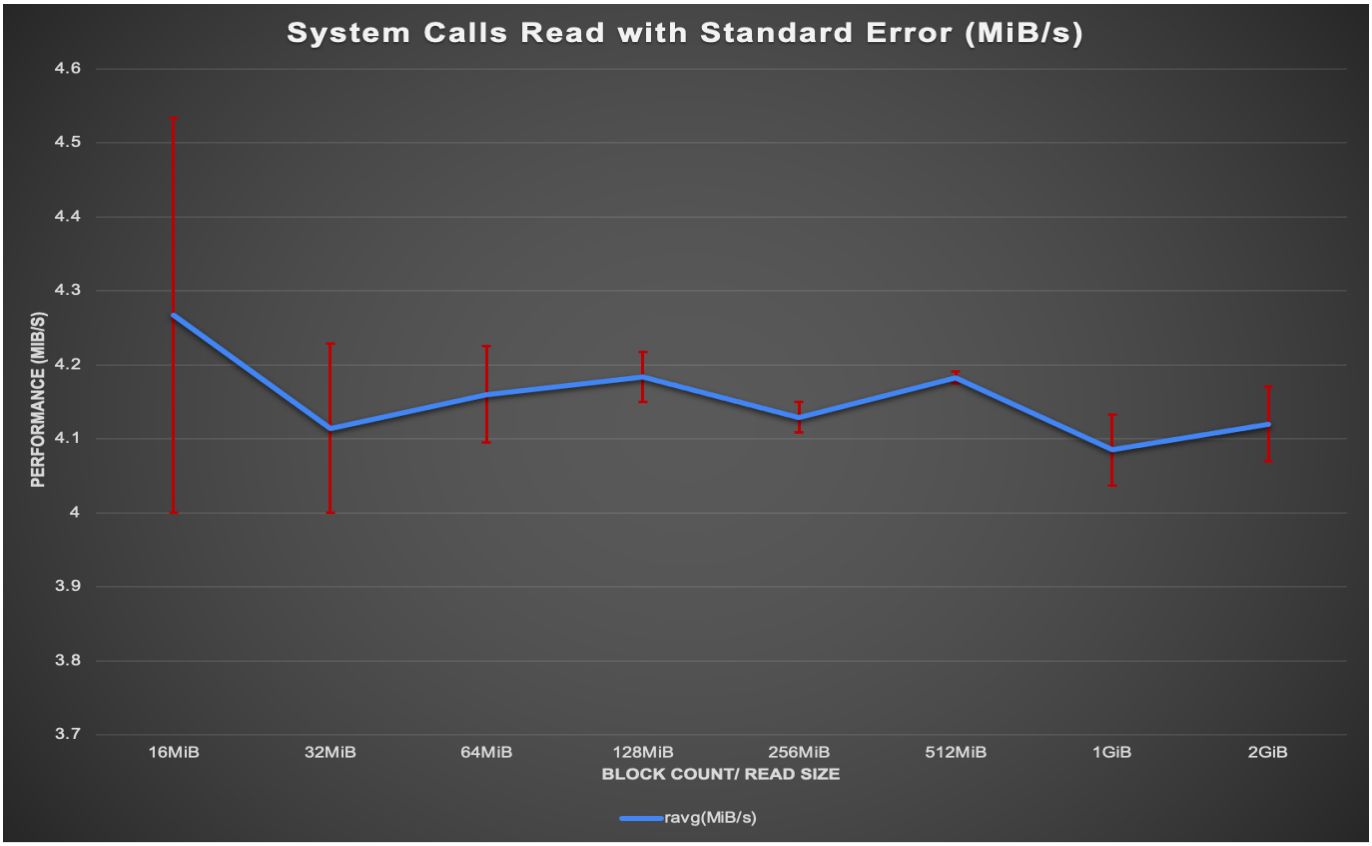


The numbers on **Y-axis** above tells us exactly how many calls we can do per second for various block count/read size/lseek count on X-axis.

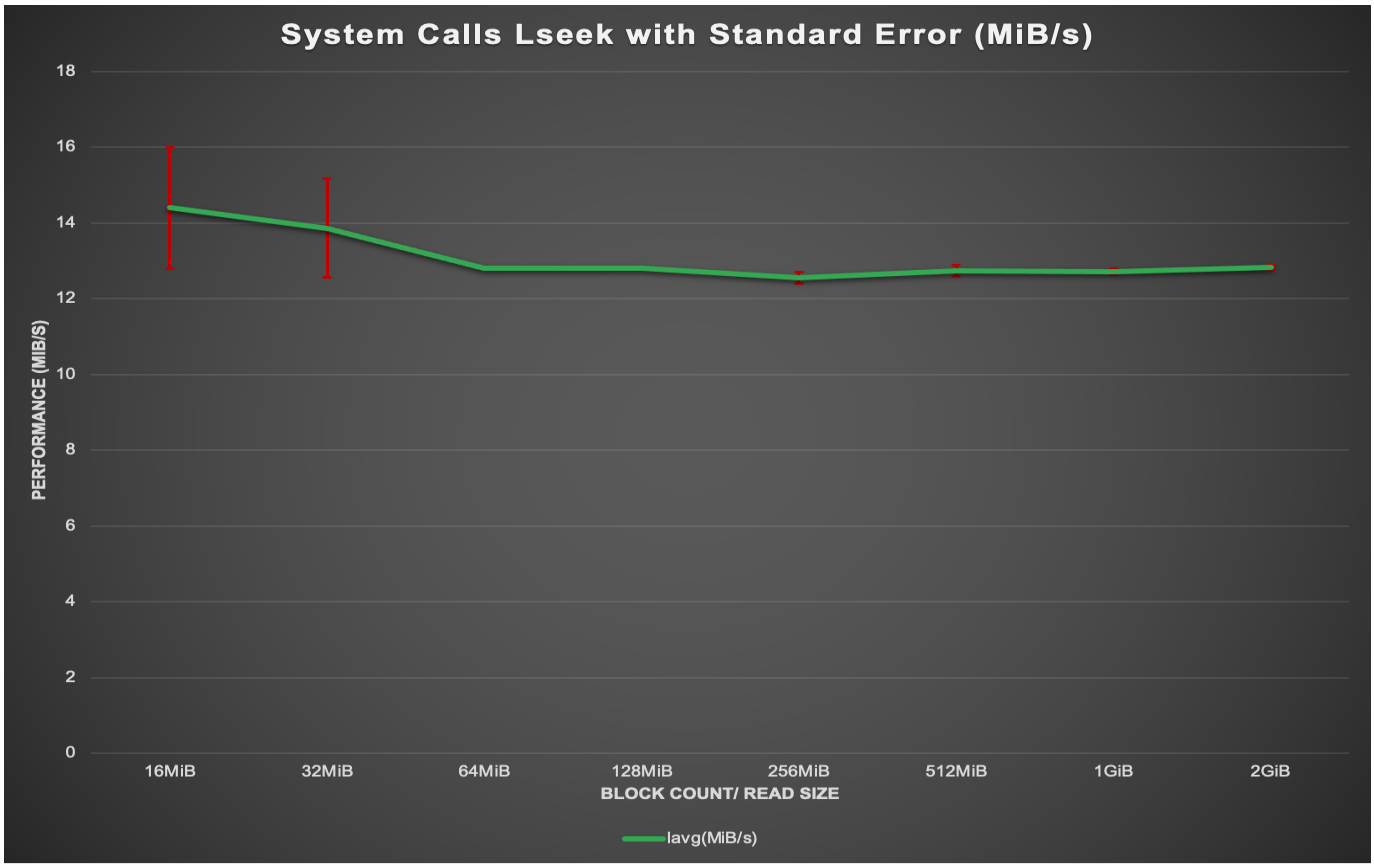
Notes:

Below are our graphs, one each for “read” and “lseek” performances, also showing the **standard error** we observed in our experiments for each data point.

Read Performance:



Lseek Performance:

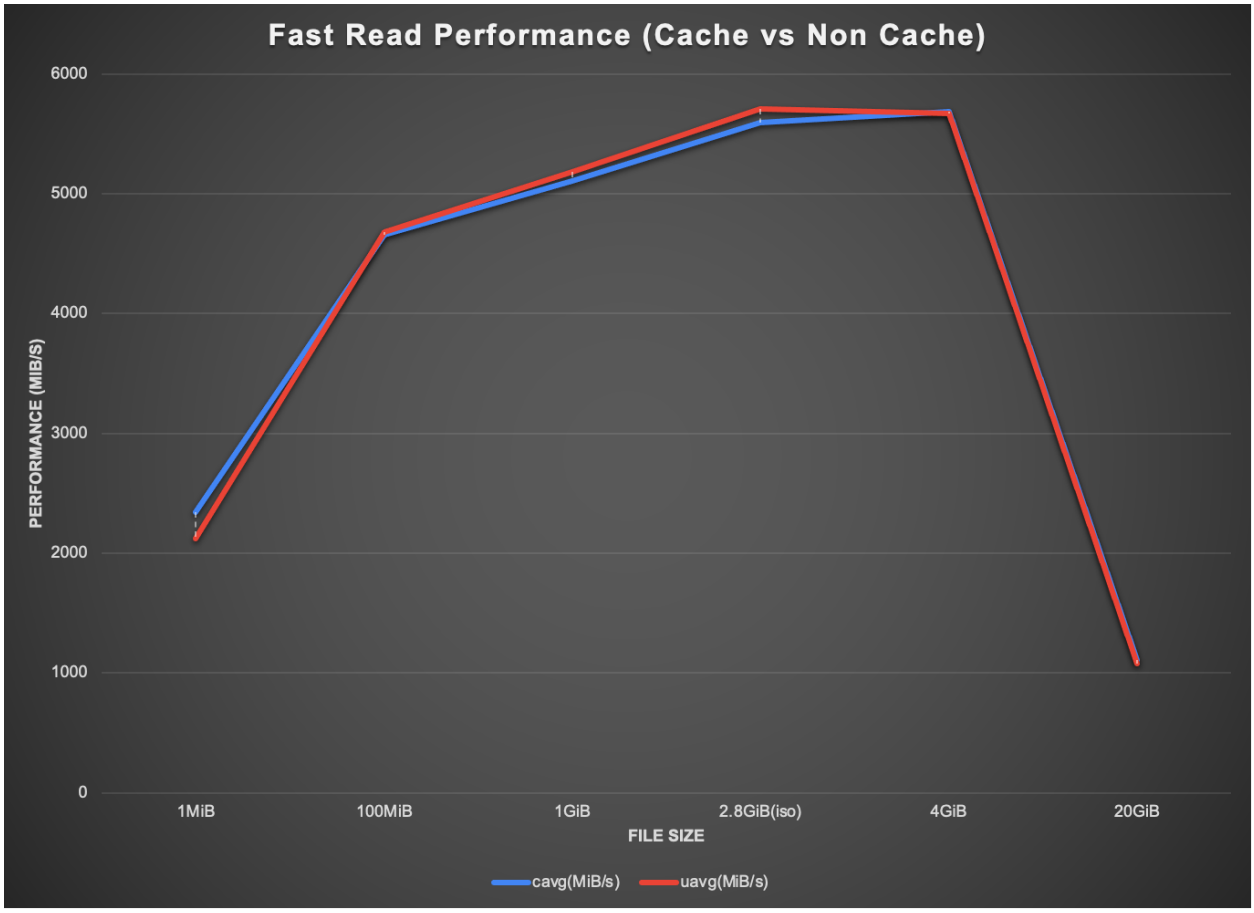


Part 6: Raw Performance

For this part, we ran our experiments on multiple **file sizes**, ranging from **1 MiB to 20 GiB.**

To optimize read performance, we ran multiple experiments using combinations of various **block sizes** and **thread count** (we are doing multithreading), leading us to pick a **block size of 4 MiB** and **8 threads** (for most cases) for getting the best possible read performance on our system.

Below is our graph (under the above chosen parameters) showing **file size** on **X-axis** and **read performance (MiB/s)** on Y-axis, for **both cached and UNcached reads**:



Note that "**cavg**" in the label stands for **"cached average"** and **"uavg"** stands for **"UNcached average"**.

Appendix/References

We are also attaching a spreadsheet (along with the rest of the submission content) where we have all our readings and plots for the several experiments we conducted while working on the project. It goes without saying that it is more like a meta-work (or scratch pad/notes if you would say) and might be very confusing to anyone looking at it for the first time. But we just wanted to attach it in case you are wondering where all our finished readings and plots (in the rest of the report above) come from.