Report

!!!!!!
The whole final project (code in C) is run on an environment of a Ubuntu container on Docker with a physical memory of 16GB and a CPU of 10-core M1 Pro.

!!!!!!
When running a Ubuntu container in Docker, we should make the container run in privileged mode by adding the `--privileged` parameter, which will give the container higher privileges, including permission to modify the filesystem.

1. Basic `run.c` Program

Commands to execute the basic part are as follows.

```
gcc run.c -o run
./run <filename> [-r|-w] <block_size> <block_count>
```

This program is a tool for testing file system performance, especially in terms of reading and writing large amounts of data. It can be used to measure the time taken for operations and data throughput.

Here's a brief explanation of the code in `run.c`:

- `xorbuf` function: Calculate the XOR checksum of the data that has been read.
- `writeToFile` function: Write a specified number of data blocks to a given file descriptor. Each block is of the given size and is filled with a random character.
- `readFromFile` function: Read specified data blocks with the given size from a specified file descriptor and computes the XOR checksum of the read data.
- `main` function: The main entry point of the program. It handles command-line arguments including `block_size` and `block_count`, opens a file, performs read or write operations, and calculates and displays performance data such as total size, elapsed time, and throughput.

Special scenarios in `run.c` include:

- If an insufficient or incorrect number of command-line arguments is provided, the program will display usage information and exit.
- The program will print an error message and exit in case of errors in memory allocation, file opening, or read/write operations.
- In read mode, when the file size is not a multiple of the size of `unsigned int` (4 bytes), a warning message is printed and the file is padded with zeros.
- In read mode, if the file size is smaller than the requested read size, the program will print an error message and exit.
- In read mode, if the given parameter of block count is 0, then read the whole given file.

2. Measurement (Find Reasonable File Size)

Commands to execute the measurement part are as follows.

```
gcc run2.c -o run2
./run2 <filename> <block_size>
```

In `run2.c`, we reuse the `readFromFile` function from `run.c`. This program accepts `filename` and `block_size` as arguments to measure the time taken for the read operations. With a fixed block size, it starts by reading a single data

block, then doubles the number of blocks based on the time taken, aiming to find a reasonable file size to keep the duration between 5 and 15 seconds to find a reasonable time for our following experiments.

The file we use for this `run2.c` measurement part is `test2.txt`, a text file of 8 GiB size generated by `run.c`. We also use `4096` as our block size to conduct this part to find a reasonable time between 5 and 15 seconds.

In the end, the returned block count is 1048576 and the returned total size is 4096 MiB (4 GiB), which means that the program takes 9.526729 seconds to read 4 GiB of data (429.948202 MiB/s), within our "reasonable" time. Part of the output is as shown below。

```
Block count: 1048576
Total size: 4096.000000 MiB
Elapsed time: 9.526729 seconds
Performance: 429.948202 MiB/s
```

It should be noted that the results of each run may vary due to caching and so on, and different environments lead to different outputs for this program.

Extra credit - dd

We also use the `dd` program in Linux to compare our performance. Assume that we already have a `test2.txt` file of 8 GiB size and the machine is already in a sudo mode or run by a root user. The command to execute the `dd` program and its output are as follows.

```
sudo dd if=test2.txt of=/dev/null bs=4096 count=1048576
```

```
1048576+0 records in
1048576+0 records out
4294967296 bytes (4.3 GB, 4.0 GiB) copied, 3.4376 s, 1.2 GB/s
```

When reading 4 GiB data, compared to our former experiment, in the case of caching, we can tell that the `dd` program is substantially optimized and faster than our implementation in `run2.c`.

It should be noted that there is a variation each time the programs are run possibly due to caching, hardware and system configuration, and other various factors.

Extra credit - Google Benchmark

We also utilize Google Benchmark in `run2_benchmark.cc` to test our `readFromFile` function. Assume that we already have a `test2.txt` file of 8 GiB size and the Google Benchmark has already been installed globally. We use three test cases with a block size of 4096 bytes and block counts of 524288, 1048576, and 2097152, separately. The command to execute the C++ program and its output are as follows.

```
g++ run2_benchmark.cc -o run2_benchmark -lbenchmark -pthread
./run2_benchmark
```

```
2023-11-28T00:42:16+00:00
Running ./run2_benchmark
Run on (10 X 48 MHz CPU s)
Load Average: 9.40, 10.18, 10.17
-------------------------------------------------------------------------------
Benchmark                             Time             CPU   Iterations UserCounters...
-------------------------------------------------------------------------------
BM_readFromFile/4096/524288  7793899879 ns   2575565225 ns            1 Performance (MiB/s)=262.771
BM_readFromFile/4096/1048576 1.1961e+10 ns   5047278005 ns            1 Performance (MiB/s)=342.445
BM_readFromFile/4096/2097152 2.9254e+10 ns   1.0730e+10 ns            1 Performance (MiB/s)=279.975
```

From the benchmark test output, it's evident that we have successfully run the benchmark test and included a custom performance metric "Performance (MiB/s)". This metric shows the performance of file reading in megabytes per second (MiB/s). The results indicate how performance varies with different block counts.

3. Raw Performance (Find Optimal Block Size)

Commands to execute this raw performance part are as follows.

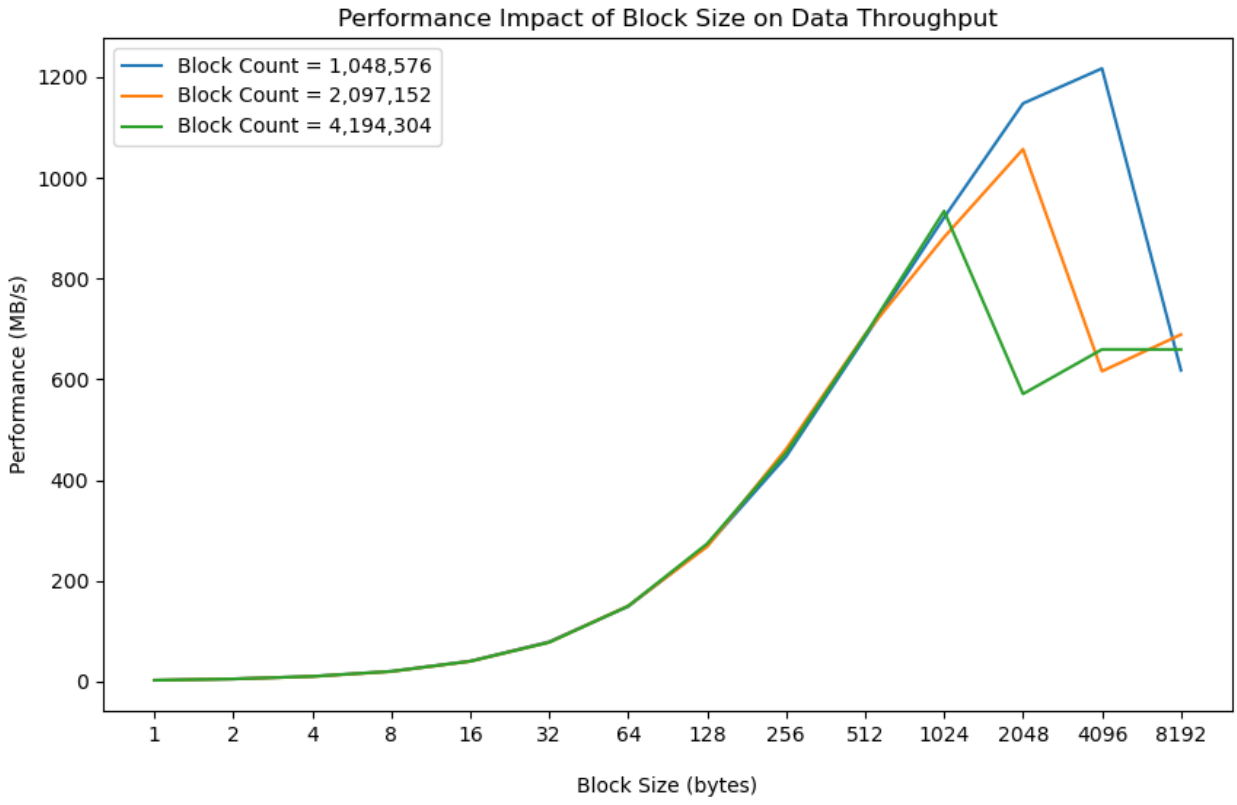
```
chmod +x ./run3.sh
./run3.sh <filename>
```

We write a shell script `run3.sh` based on the `run.c`. This script is used to measure the raw performance of no-cache read operations at different block sizes when reading a file of the same size in order to find an optimal block size.

Here is how the script works in context:

- The script compiles the C program `run.c` into an executable run and runs cache-clearing commands before the loop.
- It then enters a loop where it starts with a block size of 1024 bytes (1 KiB) and doubles it in each iteration until it reaches or exceeds 8388608 bytes (8 MiB).
- In each iteration of the loop, the script executes the run program for reading the whole given file and clears the caches after reading, ensuring that the next read operation starts with an empty cache.

Assume that we already have a file `test3.txt` of 4 GiB size created by `run.c` and the machine is already in a sudo mode or run by a root user. We conduct this experiment with the 4 GiB `test3.txt`. With the results that the script outputs, we make a line graph for the read operations that shows its performance as we change the block size when reading the same file, as shown below.



From the graph, we can observe that the performance in terms of data throughput generally increases with the block size. Starting at a lower performance around 200 MiB/s for a 1 KiB block size, there's a notable upward trend as the block size increases. There are a few places where the performance plateaus or slightly dips, but the overall trend is an increase in throughput with larger block sizes. The data also shows that the reading performance reaches a peak at 330.043662 MiB/s with a block size of 2097152 bytes (2 MiB).

This pattern typically occurs because larger block sizes can reduce the overhead of system calls and context switches, and they can also take advantage of the underlying storage media's higher sequential read/write speeds. However, this

benefit usually has a threshold, after which increasing the block size further does not result in significant performance gains, as the system's other limitations (like bus speed, CPU processing for I/O operations, etc.) become the bottleneck.

It should be noted that there is a variation each time the programs are run possibly due to caching, hardware and system configuration, and other various factors. However, the reading performance trend remains generally unchanged with block size.

4. Caching

Commands to execute this caching part are as follows.

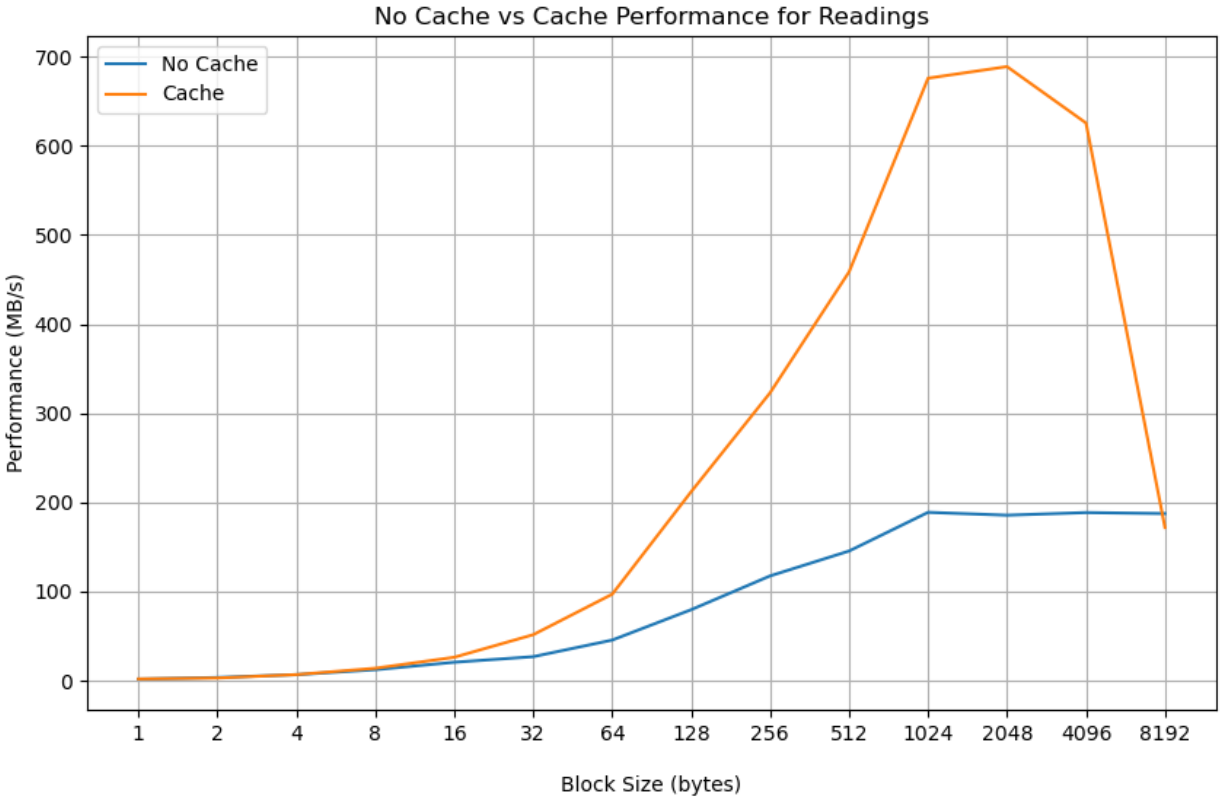
```
chmod +x ./run4.sh
./run4.sh <filename>
```

We write a shell script `run4.sh` based on the `run.c`. This script aims to measure the performance difference between reading a file when its contents are not cached versus when they are potentially cached.

Here is how the script works in context:

- The script compiles the C program `run.c` into an executable run and runs cache-clearing commands before the loop.
- It then enters a loop where it starts with a block size of 1024 bytes (1 KiB) and doubles it in each iteration until it reaches or exceeds 8388608 bytes (8 MiB).
- In each iteration of the loop, the script executes the run program for reading the whole given file twice (first for no-cache reading and second for cache reading) and clears the caches after reading, ensuring that the next read operation starts with an empty cache.

Assume that we already have a file `test4.txt` of 4 GiB size created by `run.c` and the machine is already in a sudo mode or run by a root user. In our experiment, we use this 4 GiB `test4.txt`. With the results that the script outputs, we make a line graph for the read operations that shows the effect of caching, as shown below.



The graph compares the performance of reading a 4 GiB file with and without using the cache, plotted against various

block sizes.

The performance without cache (blue line) starts much lower but increases with the block size. As the block size grows, the number of system calls decreases, and the efficiency of disk reads improves, leading to better throughput. This is because larger reads can better utilize the disk's sequential read speeds and reduce overhead.

The performance when using the cache (orange Line) remains relatively high and consistent across different block sizes, with minor fluctuations. This consistency is indicative of the benefits of caching, where the operating system stores recently accessed data in memory, which is much faster than re-reading from disk. The cached data can be accessed quickly, leading to higher throughput.

Comparing the two line in the graph, we can also interpret the followings:

- Across all block sizes, using the cache significantly outperforms not using the cache. This is expected since memory access speeds are orders of magnitude faster than disk access speeds.
- The performance benefit of caching is more pronounced at smaller block sizes, where the overhead of system calls and disk seek times would otherwise be a bigger proportion of the total operation time.
- As the block size increases, the performance difference narrows somewhat but still remains substantial, indicating that even for larger I/O operations, having data in the cache provides a significant speed advantage.

It should be noted that there is a variation each time the programs are run possibly due to hardware and system configuration, and other various factors. However, the overall trend of "No Cache" line and "Cache" line remains generally unchanged with block size.

Extra credit - Why "3"

In Linux systems, the `/proc/sys/vm/drop_caches` file is used to tell the kernel to drop clean caches, dentries (directory entries), and inodes from memory, causing that memory to become free. The pagecache is used by the system to reduce the number of disk reads by caching file data. Dentries and inodes cache hold metadata about the filesystem.

Writing different values to this file has different effects: 1 for clearing the pagecache, 2 for clearing dentries and inodes, and 3 for clearing the pagecache, dentries, and inodes. The command `sudo sh -c "/usr/bin/echo 3 > /proc/sys/vm/drop_caches"` clears all the above, which is why "3" is used. This is the most comprehensive cache-dropping operation, ensuring that all cached disk data that can be freed is freed.

It's important to note that this does not clear dirty objects (objects with content that needs to be written to disk). To ensure all cached objects are dropped, any dirty objects must be cleaned (written to disk) first, which can be done with the `sync` command. So in a full cache-clearing operation, we might run sync first to flush data to disk and then drop caches with the `echo 3` command.

5. System Calls

Commands to execute this system calls part are as follows.

```
chmod +x ./run5.sh
./run5.sh <filename>
```

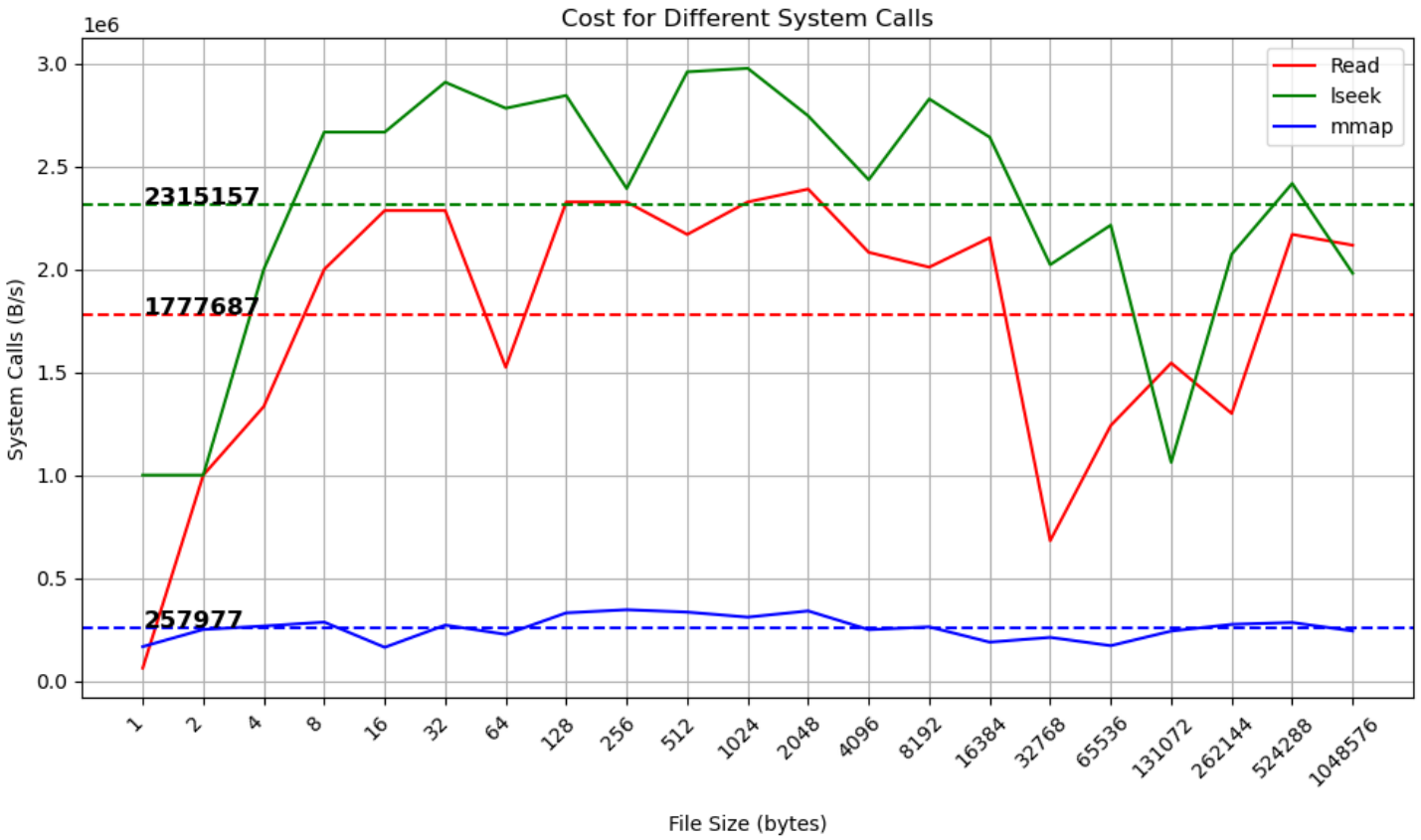
We write several C source files, `run5_read.c`, `run5_lread.c`, and `run5_mmap.c`, representing applying different system calls to process files, `read`, `lseek`, and `mmap`, separately. In addition, we also write a shell script `run5.sh` based on these C files to conduct this system calls part of the experiment. These multiple files aim to measure the performance in B/s to quantify how many different system calls are executed per second.

Here is how the script works in context:

The script compiles three different C programs into executables. Each of these programs is expected to perform file operations using different system calls— `read`, `lseek`, and `mmap`, respectively. Then cache-clearing commands are run. After compilation and cache-clearing, the script runs a loop where it executes each of the three programs with a doubling block count starting from 1024 up to 8388608 and a fixed block size of 1 byte on a given file (file size is equal

to block count). In addition, cache-clearing commands are also run after each file operation program is executed.

Assume that we already have a file `test5.txt` of 4 GiB size created by `run.c` and the machine is already in a sudo mode or run by a root user. In our experiment, we use this 4 GiB `test5.txt`. With the results that the script outputs, we make a line graph for the performance in B/s of the three system calls, as shown below.



The graph indicates that `read` has the highest overhead for very small file sizes but quickly becomes more efficient as the file size increases. `lseek` performs a high number of system calls per second, which makes sense as it doesn't transfer data. `mmap` shows the lowest number of system calls per second, which is expected since memory mapping is a one-time operation, after which the file can be accessed as if it were in memory.

The dashed lines represent the average number of system calls per second for each operation, with `mmap` averaging around 235611, `lseek` around 2361041, and `read` around 1605376. So, on average, calling `mmap` has a higher cost than the other two system calls.

6. Raw Performance (Optimize Program to Run as Fast as It Could)

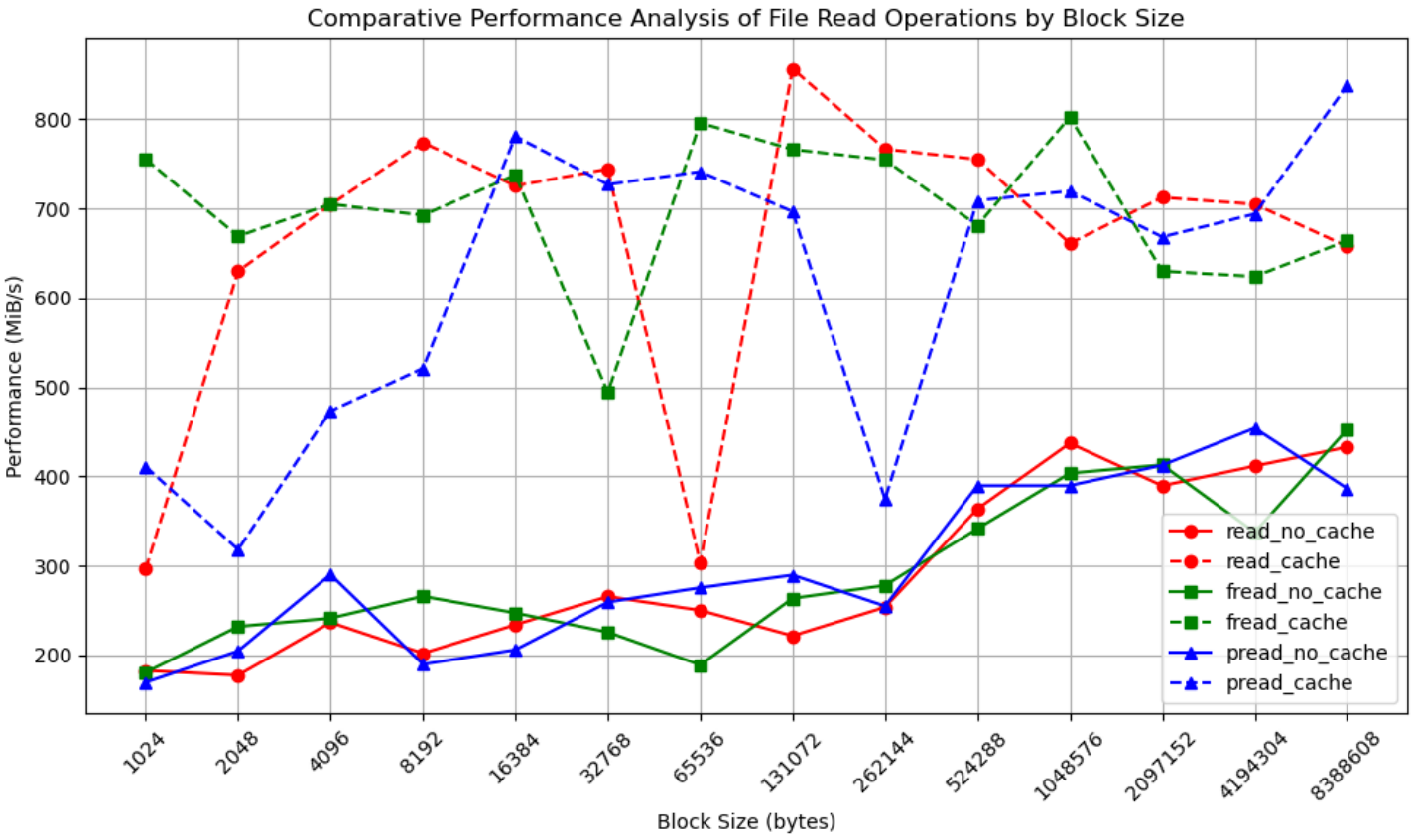
Upon reference from the Internet, we discover that there are many functions in C we can use to implement a disk read operation.

We analyze how each of them retrieves data from a disk. When accessing data from a hard disk using techniques like `read()`, `fread()`, and `pread()`, users need to specify the amount of data to read. These functions involve the operating system's kernel in the data transfer process. The kernel reads the data from the disk into a kernel buffer (also known as kernel space) and then copies this data to the user space buffer provided by the application. This approach introduces an additional data copying step between the kernel and user space, which can result in performance overhead due to the context switching and data movement involved in these operations. However, `mmap()` is another technique that allows a process to map a file into memory. This mapping between the file and the process's memory space allows the process to access that portion of the file directly in memory as if it were contiguous memory. Instead of reading from the kernel buffer pool, `mmap()` enables a process to read the file directly through memory, which could lead to performance improvements due to reduced system calls and back-and-forth data copying. However, `mmap()` is

just establishing a memory-to-disk mapping, instead of actual reading work. When we try to fetch the data at the pointer (eg. compute XOR), the operating system will encounter a page fault, then the data will be fetched from the disk, which is when the actual read happens.

With the preceding discussions, we plan to conduct experiments on these four functions with the given file `ubuntu-21.04-desktop-amd64.iso` (about 2.63 GiB) to find the fastest read method. It should be noted that there is a variation each time the programs are run possibly due to multiple various factors. However, the performance trends we obtained remain generally unchanged.

First, we use `read()`, `fread()`, and one thread `pread()` to read the given file and compare their no-cache and cache performance by block size. With the output of our scripts, we make a line graph, as shown below.



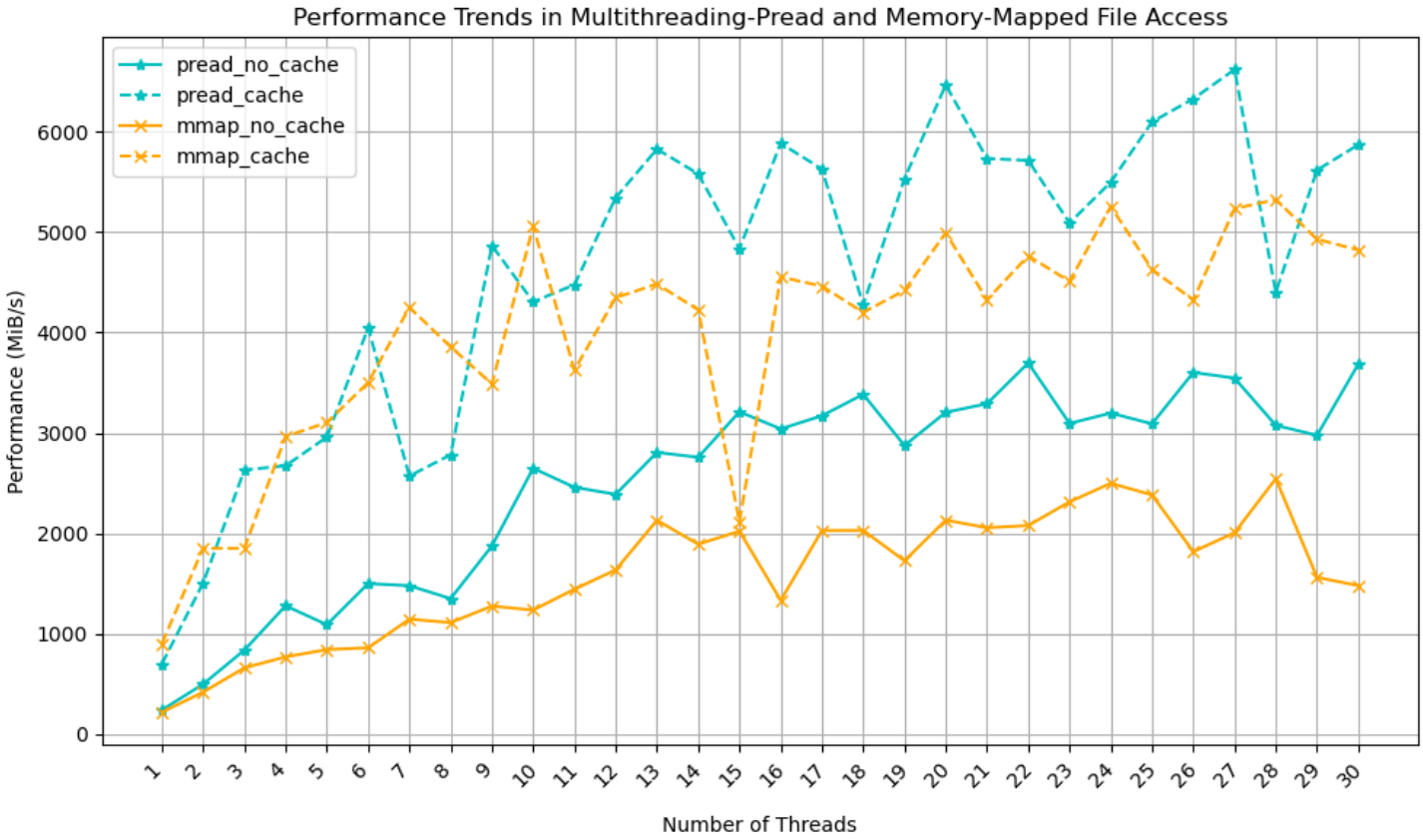
The performance of each operation type varies significantly with the block size. This suggests that the choice of block size can have a major impact on file read performance. Generally, operations with caching outperform those without. This is expected because reading from the cache is faster than reading from the disk.

For no-cache read operations, the performance of `read()`, `fread()`, and one thread `pread()` overall trends upward with block size, basically stabilizing and fluctuating slightly at 2097152 bytes (2 MiB), which also doesn't slow down the average performance for the three functions in cache read operations.

For cache read operations, the performance of `read()`, `fread()`, and one thread `pread()` generally shows higher performance than their non-cached counterparts. However, there are cases of isolated extreme swings in the trend of cache read operations including all three functions. Even with multiple times of running tests, this phenomenon keeps occurring in different block sizes. We interpret this as a cache miss. When a cached read function encounters a cache miss, it means that the requested data is not available in the cache and must be fetched from the main memory. In this situation, the efficiency of the cached read function is significantly lower compared to a cache hit because accessing the main memory takes more time and is slower than accessing the cache. Nevertheless, if the experiment is run a large number of times, the average performance of `read()`, `fread()`, and one thread `pread()` should be close to a stable speed.

It should be noted that there is a variation each time the programs are run possibly due to caching, hardware and system configuration, and other various factors. However, the overall trend of the performance lines in the above graph remains generally unchanged with the block size.

Second, since multithreading has the potential to increase efficiency, we consider two other situations: `pread()` with multiple threads, and `mmap()` with multithreaded XOR computation. We implement two types of read operations with a fixed block size of 2097152 bytes (2 MiB) and compare their no-cache and cache performance by the number of threads. With the output of our scripts, we make a line graph, as shown below.



For `pread()` with multiple threads, whether cached or not, there is a general upward trend as the number of threads increases, basically stabilizing and fluctuating slightly at 20 threads, which suggests that performance improves with more threads up to a point. However, the performance appears to be volatile and doesn't increase linearly. This might indicate that there are other limiting factors or overheads associated with creating or managing a higher number of threads.

For `mmap()` with multithreaded XOR computation, whether cached or not, there is a similar trend but generally at a lower performance level than `pread()` with multiple threads. This could imply that memory-mapped file access has a lower throughput in this particular scenario, or it might have a different scaling behavior with the number of threads. In addition, it should be noted that there is a drastic performance degradation for cache read operation when the number of threads is 15, which may be caused by page fault and cache miss.

Both of the two read operations likely represent scenarios where caching mechanisms are employed, which can significantly impact performance. The cache appears to have a beneficial effect, particularly for `pread()` with multiple threads, which shows the highest performance metrics on the graph.

It should be noted that there is a variation each time the programs are run possibly due to caching, hardware and system configuration, and other various factors. However, the overall trend of the performance lines in the above graph remains generally unchanged with the number of threads.

Finally, according to the above experimental analysis, we choose the `pread()` function and optimize it with 20 threads and a block size of 2 MiB as our final method to implement our `fast.c`. We run this program with `ubuntu-21.04-desktop-amd64.iso` (about 2.63 GiB).

Commands to execute the fast program are as follows.

```
gcc fast.c -o fast
./fast <file_to_read>
```

We need to clear the cache for the first run to get the result for the uncached result. For the cached results, just run normally a second time. The results of our `fast.c` are as follows.

fast.c	XOR	Elapsed Time (s)	Performance (MiB/s)
No-Cache	a7eeb2d9	1.663190	1616.266454
Cache	a7eeb2d9	0.387231	6942.001552

Extra credit - AWS Cloud

First, we write `fast_aws.c` by changing the number of threads in the former `fast.c` from 20 to 192, to achieve higher performance when running on the "m5d.metal" EC2 instance with 96 vCPU. Besides, we also write a script `fast_aws.sh` to run the executable program three times to observe performance: the first time is for reading without cache, and the second and the third time is for reading with cache.

Second, we prepare for the test on AWS Cloud:

- Apply the vCPU limit quota increase through Amazon Web Services (increasing from 32 to 100 for me)
- Launch an "m5d.metal" (96 vCPU) EC2 instance on AWS Cloud
- Connect the VPS through SSH in our own computer's terminal
- Initialize the NVMe drive which is attached to the VPS
- Uploading test files to the NVMe disk on the VPS through SFTP, including a C source file `fast_aws.c`, a corresponding script `fast_aws.sh`, and the `ubuntu-21.04-desktop-amd64.iso` for reading.
- Use `sudo yum install gcc` to install dependencies

Third, with all preparation done, we type commands to execute the script. The commands to execute and its output are as follows.

```
chmod +x ./fast_aws.sh
./fast_aws.sh <file_to_read>
```

```
[ec2-user@ip-172-31-70-190 nvme]$ ./fast_aws.sh
Pread file 'ubuntu-21.04-desktop-amd64.iso' with a block size of 2097152 bytes and a thread number of 192
XOR: a7eeb2d9
File size: 2688.158203 MiB
Elapsed time: 0.888528 seconds
Performance: 3025.406293 MB/s
Pread file 'ubuntu-21.04-desktop-amd64.iso' with a block size of 2097152 bytes and a thread number of 192
XOR: a7eeb2d9
File size: 2688.158203 MiB
Elapsed time: 0.159901 seconds
Performance: 16811.390818 MB/s
Pread file 'ubuntu-21.04-desktop-amd64.iso' with a block size of 2097152 bytes and a thread number of 192
XOR: a7eeb2d9
File size: 2688.158203 MiB
Elapsed time: 0.094391 seconds
Performance: 28478.967308 MB/s
```

From the result, we can tell that on the Amazon "m5d.metal" EC2 instance with high-end hardware (96 vCPU, 384 GiB Memory), using the NVMe drive, the read performance is much higher than the `fast.c` that is run on our own device, whose comparison is more obvious in the table below.

	Elapsed Time (s)	Performance (MiB/s)
No-Cache Read (fast.c)	1.663190	1616.266454
Cache Read (fast.c)	0.387231	6942.001552
No-Cache Read (fast_aws.c)	0.888528	3025.406293
Cache Read (fast_aws.c)	0.094391	28478.967308