CS-GY 6233 Final Project

Daniel Kerrigan (djk525), Racquel Fygenson (rlf9859)

December 16, 2021

Part 1: Reading & writing to the disk

Our code is available in the following GitHub repository. To compile our code, you can execute the script `./build`. Reading and writing files can then be done with the following command:

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

This will read or write a file with a single thread. When reading, the program prints the XOR value of the 4-byte integers in the file.

Part 2: Finding a reasonable block count

We wrote a benchmark program to find a reasonable block count when given a block size. A reasonable block count for a given block size is one that makes our `./run` program from Part 1 take more than 5 seconds to read. Starting with a block count of 1, the benchmark program measure how long it takes the `./run` program from Part 1 to read that many blocks of the given block size. The program then loops and doubles the block count until the execution time of `./run` is greater than 5 seconds. Finally, we output the block count that results in a read time of more than 5 seconds, along with the corresponding file size and read time.

For the timings, we measure how long it takes to `fork()` and `exec()` the `./run` program.

To find the reasonable block count for a given block size, you can run the following command:

```
./run2 <filename> <block_size>
```

Part 3 & 4: Performance with & without caching

For Parts 3 and 4, we measured how fast our program reads data with various block sizes. All reads were using a single thread. We ran all experiments on a computer with the latest version of Ubuntu (release 20.04), a ∼10 year old Samsung 830 series SSD, 16 GB of RAM, and an Intel i7-2600 CPU.

For Figure 1, we ran our program 10 times for each block size, clearing our cache before each run. We then calculated the mean, median, minimum, and maximum read speeds over the 10 runs for each block size. All values are in MiB/second. In Figure 1, there is one bar for each block size. The length of the bar encodes the mean read speed at that block size. There is a black circle for each block size, which shows the median read speed. Each bar also has a vertical black line over it. The bottom of this line shows the minimum read speed of the 10 runs and the top of the line shows the maximum read speed. We see that performance starts to plateau with block sizes of 512 B.

To generate the data for Figure 1, the following command can be used:

```
sudo ./benchmark --perf_no_cache <filename>
```

For Figure 2, we repeated the same process, except we did not clear our cache before each run. For each block size, we did one extra run at the beginning to make sure that the file's data is cached, and then did 10 runs without clearing the cache. With caching, we see that performance plateaus with block sizes of 64 KiB.

To generate the data for Figure 2, the following command can be used:

```
./benchmark --perf_cache <filename>
```

Figure 3 is a grouped bar chart that shows the data from both Figure 1 and Figure 2 side-by-side. There is a pair

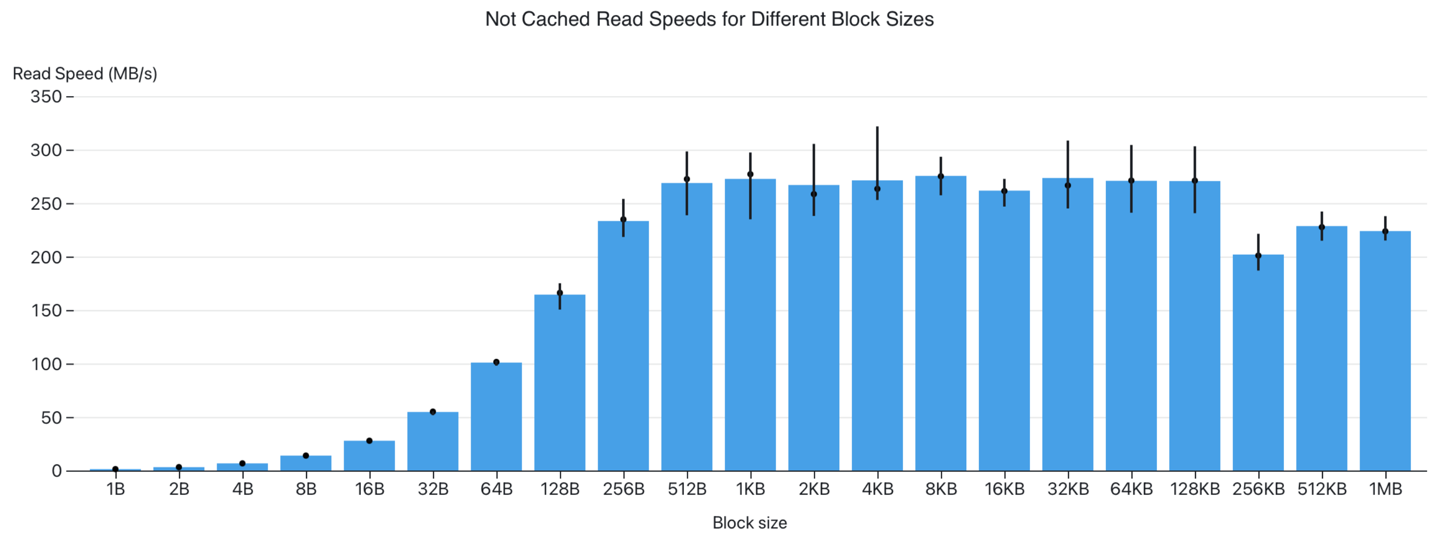

Figure 1: Single-threaded reads with cold cache. Bar length encodes mean read speed of 10 runs. The position of the black circles encode median read speed. The black vertical lines on top of the bars show the minimum and maximum read speeds.

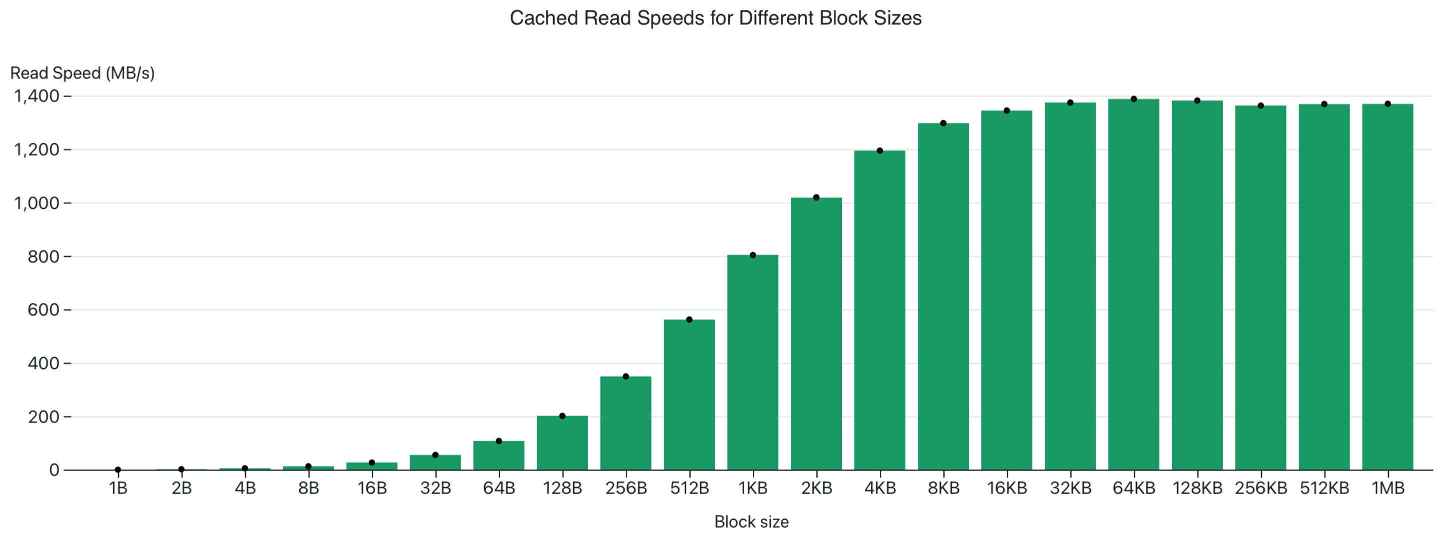

Figure 2: Single-threaded reads with warm cache. Bar length encodes mean read speed of 10 runs. The position of the black circles encode median read speed. Read speeds were consistent, so the black lines showing the minimum and maximum read speeds are not visible.

of bars for each block size. In each pair, the blue bar shows the not cached read speed and the green bar shows the cached read speed. We can see that read speeds are similar up until block size 128 B, after which the cached reads become significantly faster.

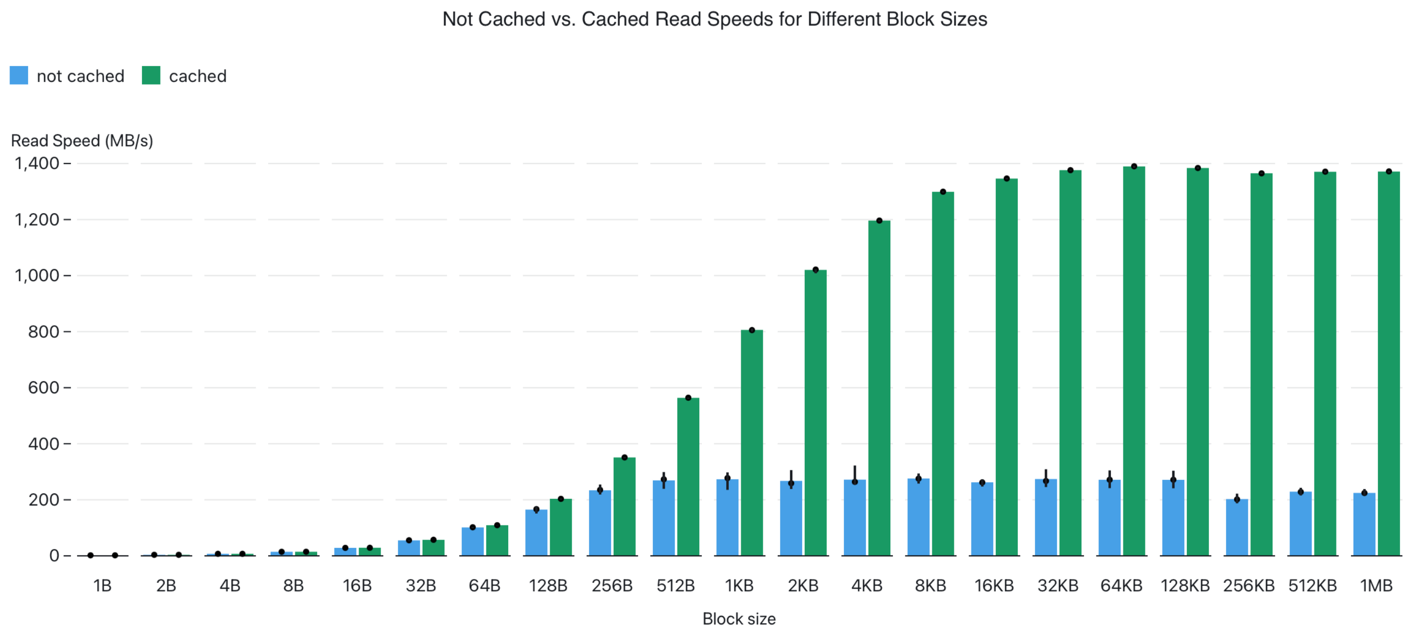

Figure 3: Comparing cold vs. warm cache for single-threaded reads.

Part 5: Performance compared to system calls

In Part 5, we measured the reading speed of our `./run` program for a block size of 1 byte. We measured the speed in bytes per second, which is essentially the number of times we called the `read` system call per second. We repeated this 10 times and recorded the mean, median, maximum, and minimum calls per second.

We compared the number of `read` calls per second to four other system calls: `fstat`, `lseek`, `getuid`, and `getpid`. We measured how much time it took to call each system call 25 million times. We made sure to choose a large enough number of calls so that the measuring time for a given system call would be longer than 5 seconds. We repeated this 10 times for each system call and recorded the mean, median, maximum, and minimum calls per second.

Results from this experiment can be seen in Figure 4, where the mean is encoded by the height of each bar, the median is represented by the black dot, and the minimum and maximum speeds are shown by the bottom and top of the thin black lines at the top of each bar.

We highlight the `read` system call in the graph because the way that we are measuring the `read` system call through executing the `./run` program is different than how the other system calls were measured. The additional time to `fork()` and `exec()` the `./run` program and to calculate the XORs should be negligible compared to the time spent doing the `read` system calls, but it may have some slight influence on the results.

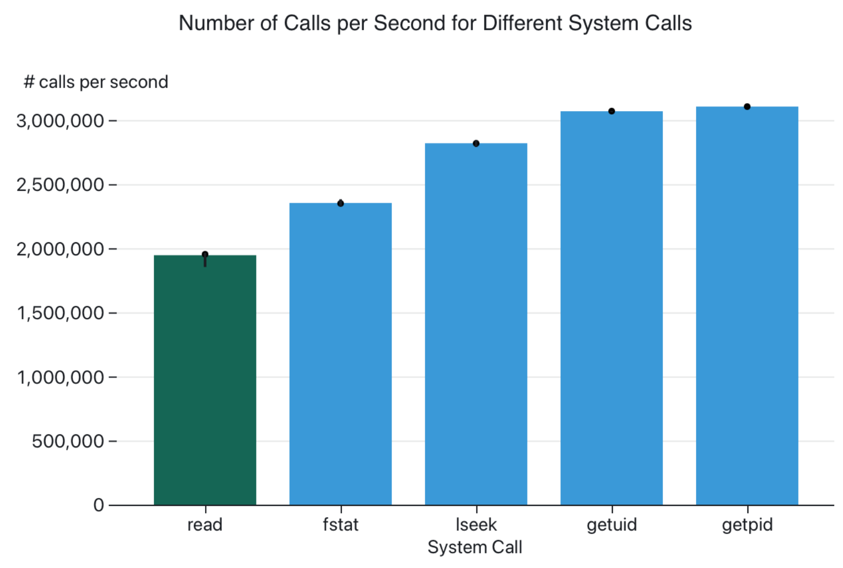

Figure 4: Single-byte reads from our ./run program compared to system calls.

To generate the data for Figure 4, the following command can be run:

```
./benchmark --sys_calls <filename>
```

Part 6: Optimizing our program

For part 6, we looked at altering two variables to optimize our `./run` program's speed: block size and number of threads. To do this, we implemented multi-threaded reading. We had to make some adjustments to how we determined the reasonable block count when multiple threads were used for cached reads. During initial experiments of cached reads with multiple threads, we found that the reasonable file size for certain block sizes and number of threads was determined to be 16 GiB. The computer that we ran our experiments on has 16 GB of RAM. For the combinations of block sizes and number of threads that were reading 16 GiB, we noticed that the read speeds were drastically slower, indicating that the data was not able to all fit in the cache. To account for this, we made two changes to how we determined the reasonable block count when measuring multi-threaded cached reads. First, we used a lower time threshold of 2.5 seconds instead of 5 seconds. Second, we restricted our function to only double block count while the total file size ($blockSize \times blockCount$) is $\leq$ 12 GiB. This caps the maximum block count that can be returned for a given block size. This ensures that we don't use a file size that is too large to entirely cache in 16GB of RAM, which would decrease our cached performance.

Finally, we attempted to optimize the performance of our `./run` program by varying block size and number of threads. We ran several experiments both without caching and with caching, and reported our findings in Figure 5 and Figure 6. Each graph is a grouped bar chart that has one group for each block size. Within each group, there is one bar for each number of threads. To find the height of each bar reported in Figure 5 and Figure 6, we averaged 5 runs completed with a particular block size and number of threads. We also report the median of these 5 runs via the small black dot, and the minimum and maximum of these runs using the black line as an error bar at the top of each bar.

Without caching and at smaller block sizes, we see that more threads generally increases read speeds up until 8 or 16 threads. We see performance for all threads plateau at block sizes of 256 B. At this point and beyond, there is generally not a clear trend in which number of threads performs the best and using multiple threads does not

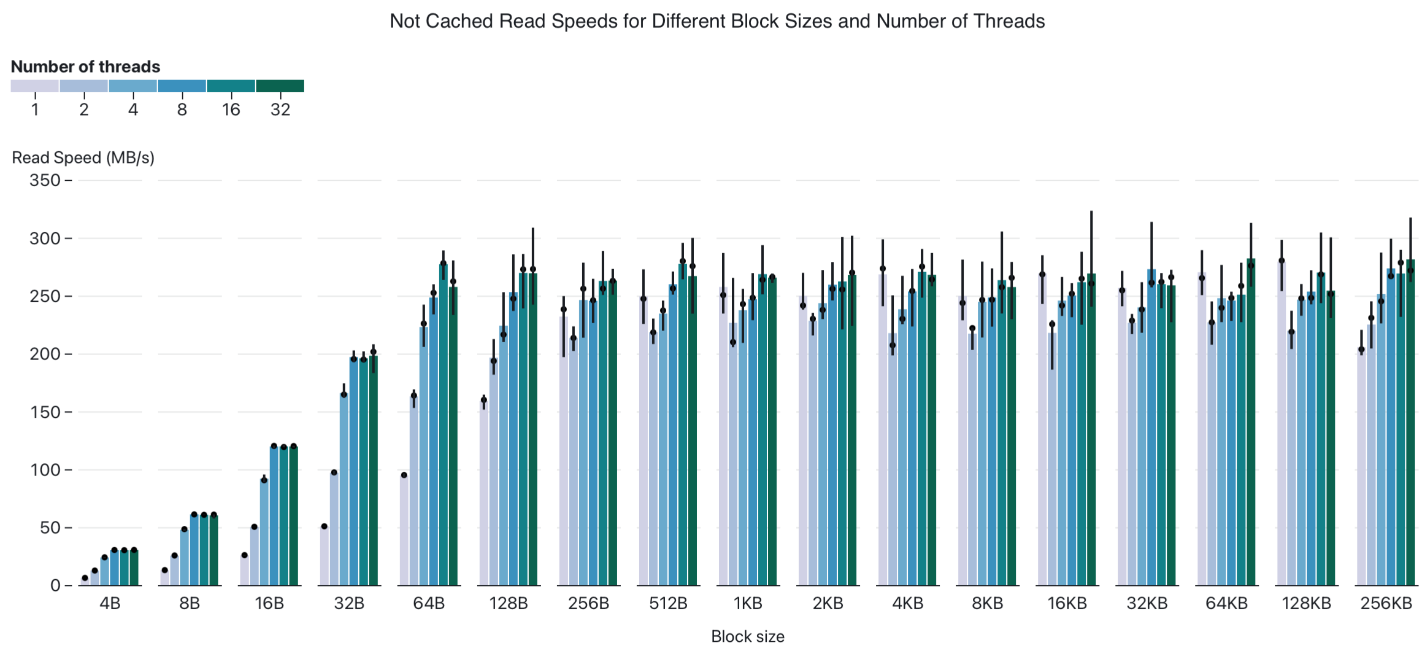

Figure 5: Multi-threaded reads with cold cache.

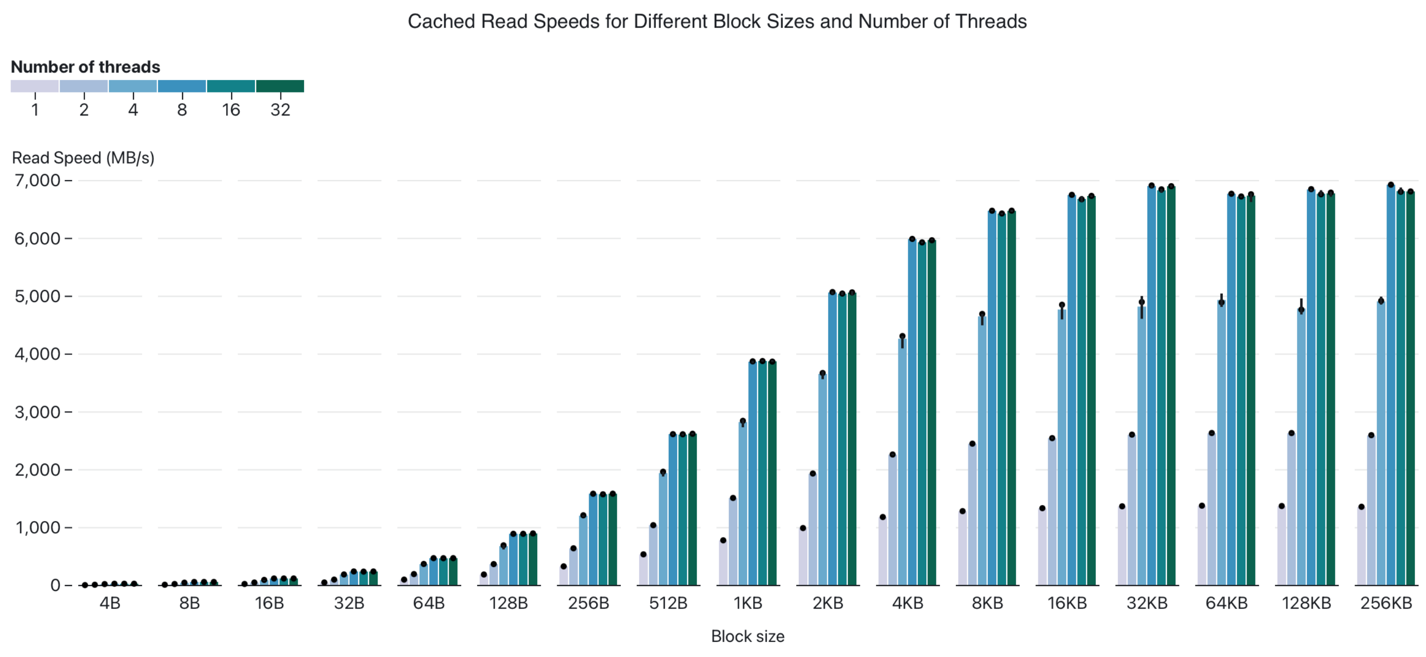

Figure 6: Multi-threaded reads with warm cache.

always outperfom a single thread. With caching, the trend is clearer. For each block size, we see performance increase with more threads, up until 8 threads. There is no additional gain in speed with 16 or 32 threads compared to 8. With caching, performance plateaus at a block size of 32 KiB. To cover the cases of users running our `./run` program both with and without caching, we decided to finalize our `./fast` program to call `./run` with a block size of 32 KiB and 8 threads. This allows us to achieve speeds at the performance plateau of both cached and not cached files. We provide the read speeds of 5 runs with 32 KiB block size and 8 threads with both cold and warm caches in Table 1.

	Not cached	Cached
mean (MiB/s)	273.364	6,908.04
median (MiB/s)	261.641	6,916.377
min (MiB/s)	257.611	6,876.113
max (MiB/s)	314.215	6,921.959

Table 1: Read speeds for block size of 32 KiB and 8 threads.

To generate the data for Figure 5, the following command can be used:

```
sudo ./benchmark --perf_no_cache_threads <filename>
```

To generate the data for Figure 6, the following command can be used:

```
./benchmark --perf_cache_threads <filename>
```

To run the fast read version of our `./run` program, the following command can be used:

```
./fast <file_to_read>
```