OS Final Project Report

Abirbhav Dutta: ad5548
Yashika Dhawan: yd2281

Note: Throughout the report when we say ‘with cache’, we mean the cached scenario, and when we say without cache, we mean the non-cached scenario.

Part 1 : Basics

Program written: *run.c*.
Execution command: *./run <filename> [-r|-w] <block_size> <block_count>*
Output: *xor of all 4-byte integers for the block count and size specified*

Part 2 : Measurement

Program written: *run2.c*
Execution command: *./run2 <filename> <block_size>*
Output: *BlockCount*

Implementation details:

We ran the run2.c program for different block sizes starting with a small block size (1 Byte) and kept multiplying by 2. The run2.c program calls the code in part1 internally. On finding a ‘reasonable’ time i.e. between 3 to 15 seconds, we returned the block count.

We also ran this program in a loop and kept doubling the block size to find the ideal block count for each block size. Result is given below.

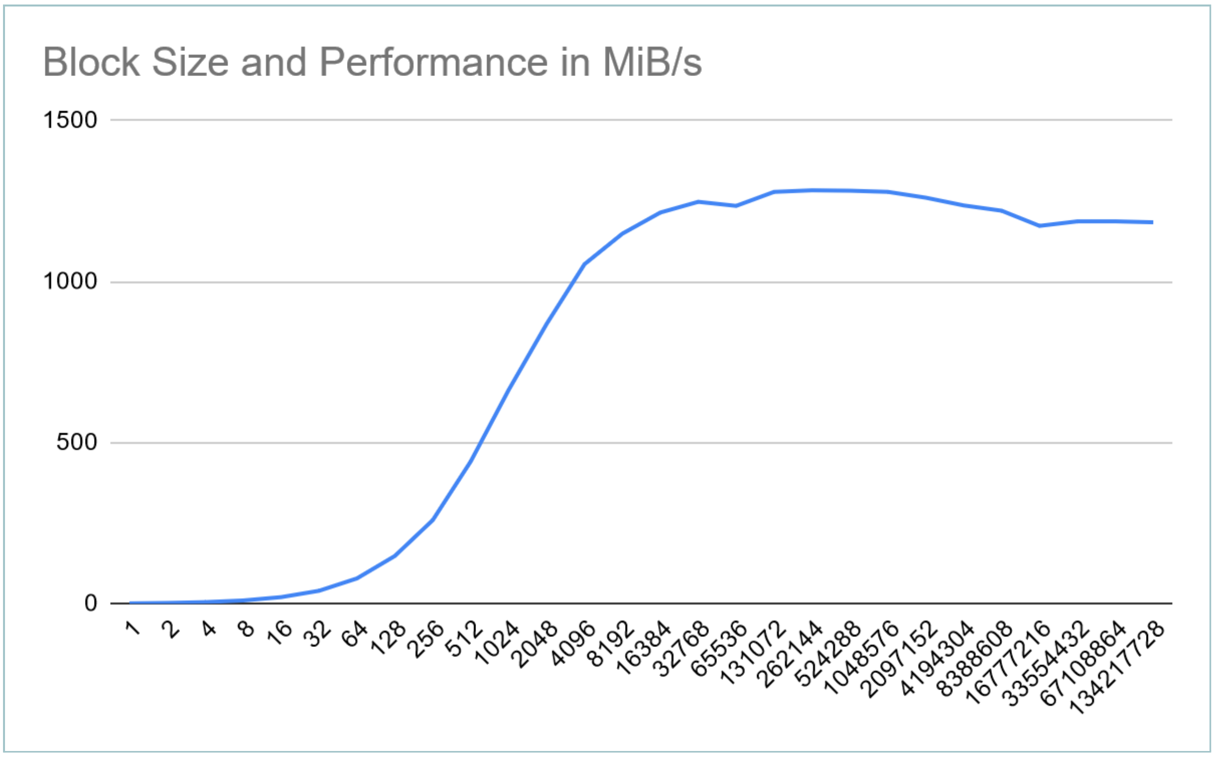
Block Size	Block Count	Time in seconds
1	4194304	3.086139
2	4194304	3.005538
4	4194304	3.063287
8	4194304	3.083581
16	4194304	3.083881
32	4194304	3.162794
64	4194304	3.244777
128	4194304	3.444072

256	4194304	3.936941
512	4194304	4.632057
1024	2097152	3.084435
2048	2097152	4.714712
4096	1048576	3.884537
8192	524288	3.563714
16384	262144	3.371497
32768	131072	3.281545
65536	65536	3.314986
131072	32768	3.202419
262144	16384	3.19015
524288	8192	3.193201
1048576	4096	3.202269
2097152	2048	3.248159
4194304	1024	3.311235
8388608	512	3.355782
16777216	256	3.490498
33554432	128	3.449231
67108864	64	3.449196
134217728	32	3.457746

Part 3 : Raw Performance

Note: For parts 3 to 5 we wrote the program *complete.c*.
Execution command: *./complete <filename>*

The graph below plots performance in MiB/s v/s Block Size. We can observe from the graph that the performance starts plateauing at around a block size of 8192, and the maximum performance is achieved for a block size of 262144 which is 256 KiB.



The same data in the form of a table:

Performance Table :

Block Size	Performance in MiB/s
1	1.296118
2	2.661753
4	5.223148
8	10.377544
16	20.75307
32	40.47055
64	78.896031
128	148.661225
256	260.100403
512	442.136149
1024	663.97905
2048	868.76993
4096	1054.43721
8192	1149.362604

16384	1214.890712
32768	1248.192421
65536	1235.60088
131072	1279.033273
262144	1283.952284
524288	1282.725269
1048576	1279.093044
2097152	1261.022008
4194304	1237.000596
8388608	1220.579755
16777216	1173.471526
33554432	1187.511185
67108864	1187.523005
134217728	1184.586712

Part 4 : Caching

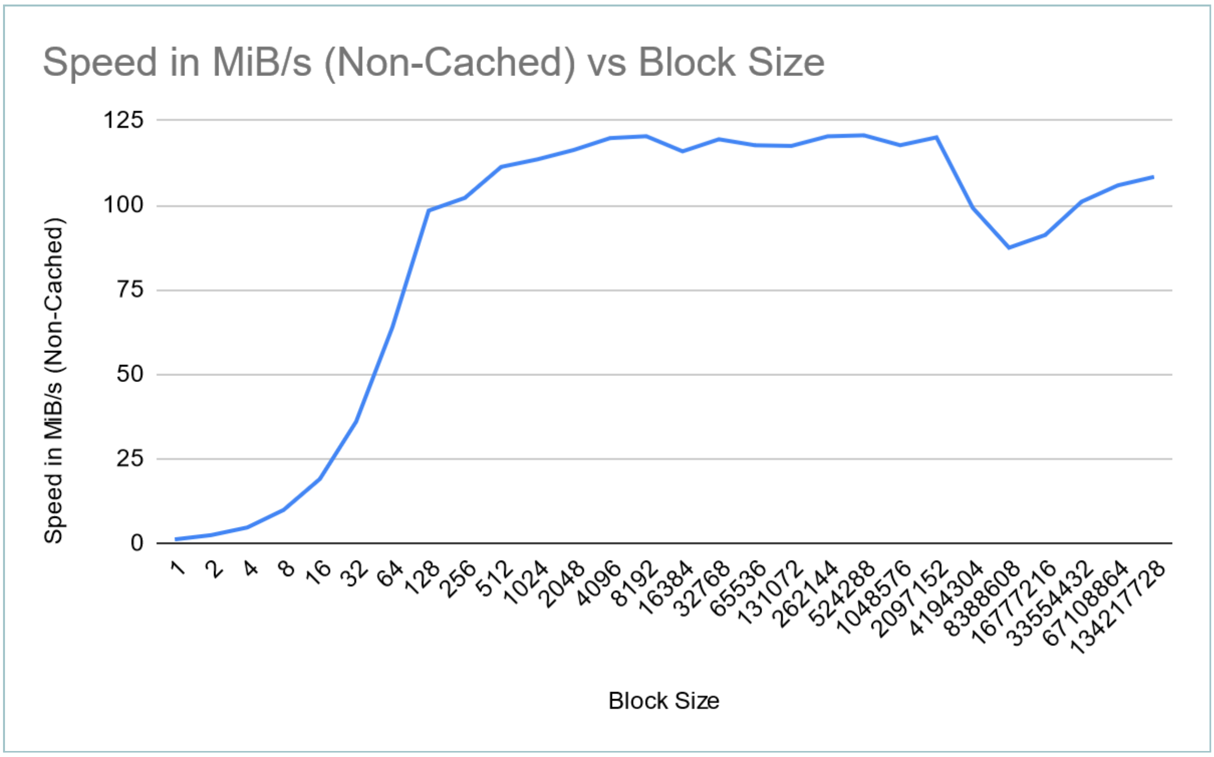
Note: For parts 3 to 5 we wrote the program *complete.c*.
Execution command: *./complete <filename>*

Implementation details:

We ran the same tests above for each block size after clearing the cache.
To clear the cache, we used the command:

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

The result we got is shown below. The graph plots performance in MiB/s v/s Block Size:
Each test is run 5 times.

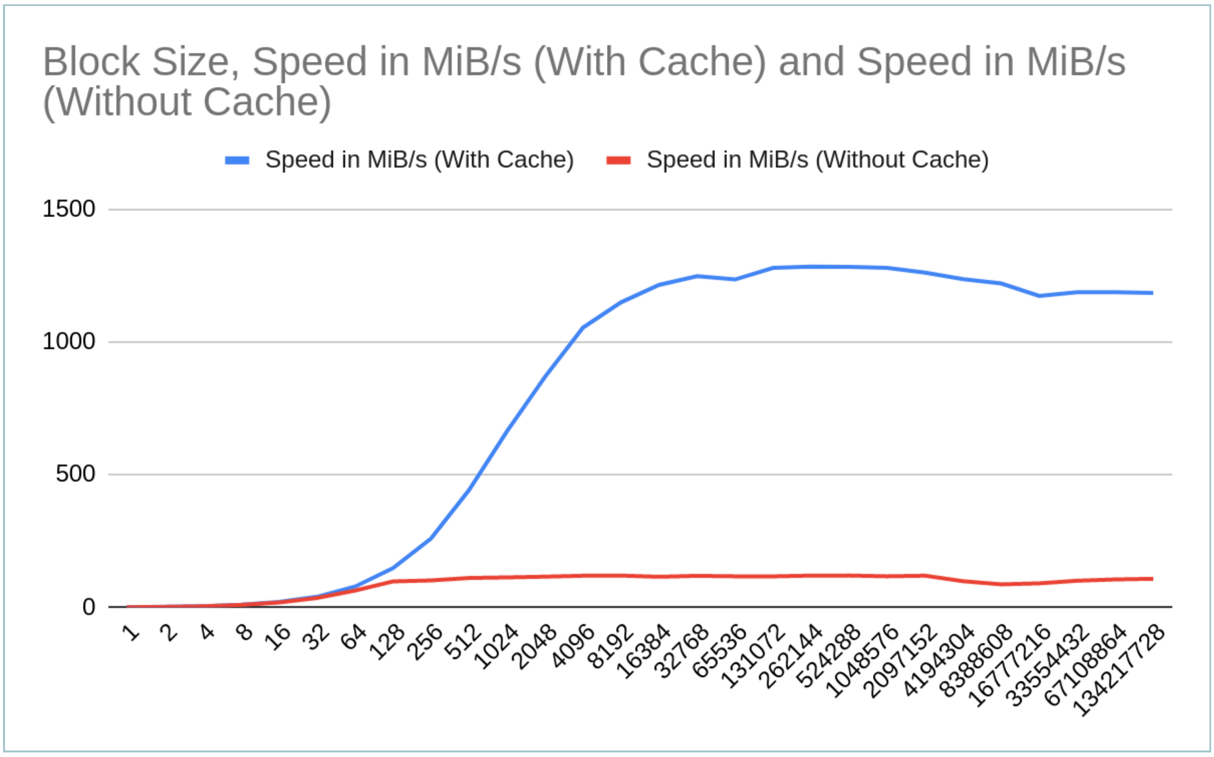


The same data in the form of a table:

Block Size	Speed in MiB/s (Non-Cached)
1	1.248193
2	2.504868
4	4.771619
8	9.95553
16	19.108872
32	36.058154
64	63.960244
128	98.434695
256	102.202593
512	111.358206
1024	113.583745
2048	116.353843
4096	119.854252
8192	120.407951
16384	115.946804
32768	119.509101

65536	117.734969
131072	117.544451
262144	120.379179
524288	120.677978
1048576	117.754905
2097152	120.075822
4194304	99.23488
8388608	87.460922
16777216	91.233011
33554432	101.063456
67108864	105.893167
134217728	108.400913

What would be more interesting is if we plot both the cached and non-cached performances in the same graph. The following graph contains just that. This will help in comparing the performance between cached and non-cached reads. We can clearly see that cached reads are much faster.

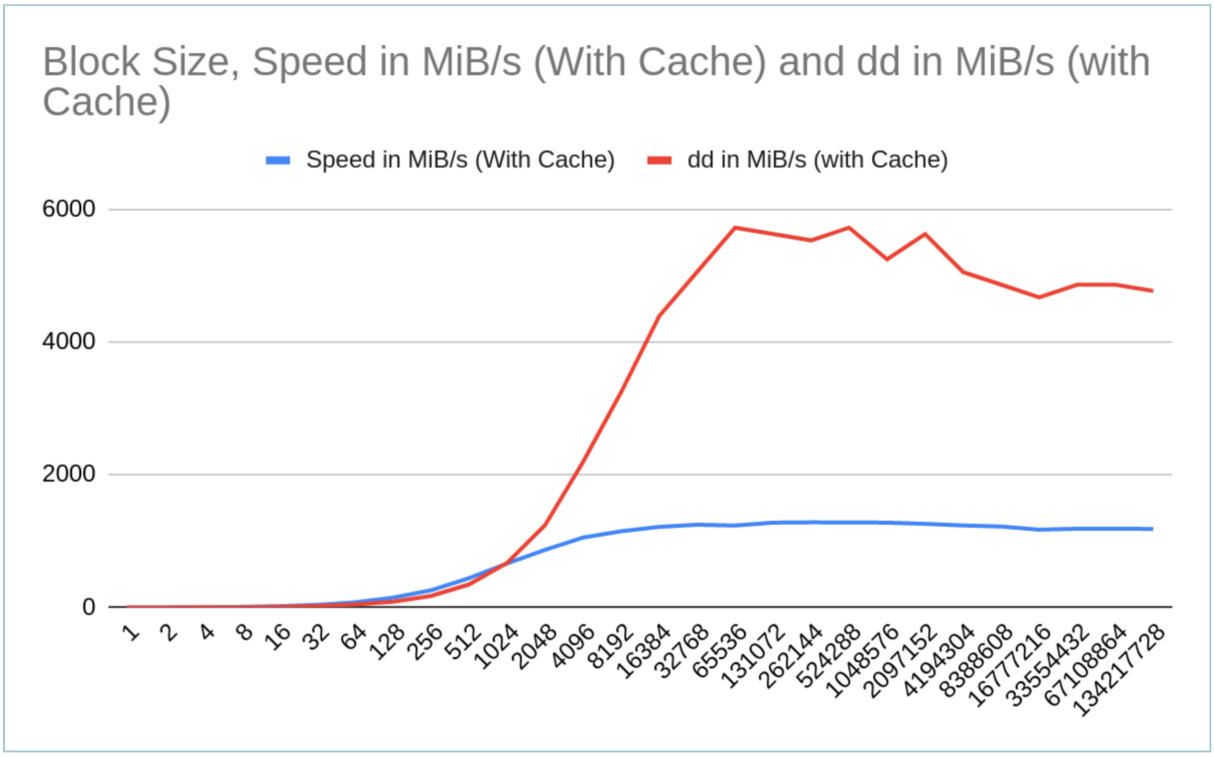


Extra Credit: Comparing with dd

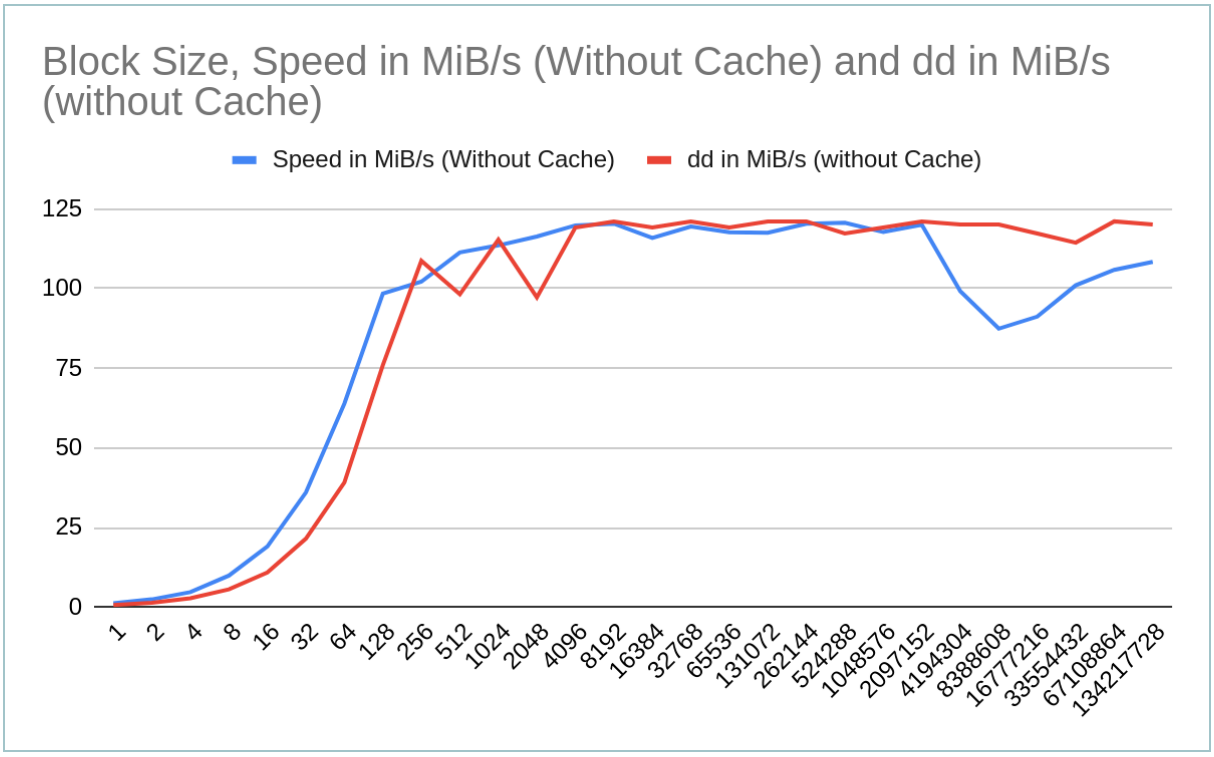
Implementation details: The code to run this experiment is also present in complete.c.

We ran the dd command for the same block sizes and block counts that we used in Part 3 and 4, and measured both the cached and non-cached performance. Comparison with our program is shown below. (NOTE that this is the comparison with our run.c file where dd outperforms run.c. However, as mentioned in Part 6, **our fast.c program outperforms dd**)

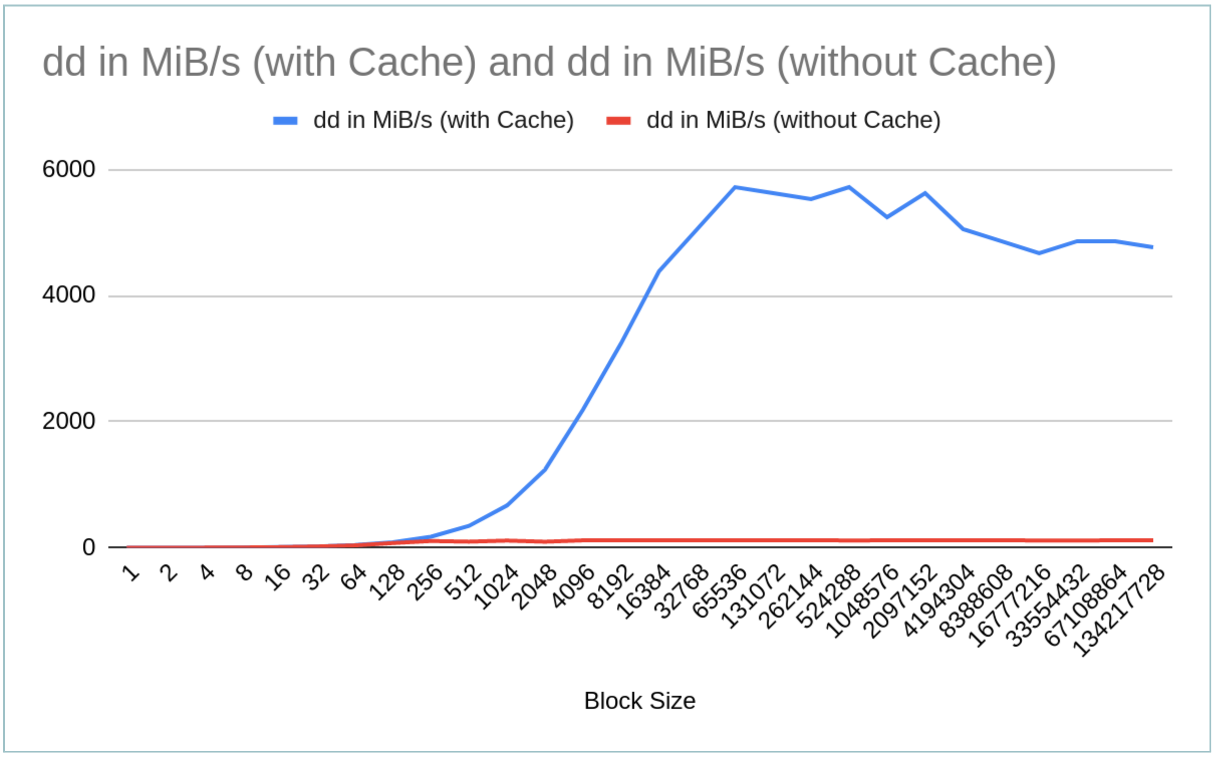
Below graph shows the comparison between the speed (performance) of our code with cache v/s speed of dd with cache.



Below graph shows the comparison between speed of our code without cache vs speed of dd without cache. (ie the non-cached performance).



Below graph shows the comparison of speed of dd with cache and without cache.



The same data in the form of a table:

Block Size	Run.c speed in MiB/s (With Cache)	Run.c speed in MiB/s (Without Cache)	dd in MiB/s (with Cache)	dd in MiB/s (without Cache)
1	1.296118	1.248193	1.236074038	1.190369211
2	2.661753	2.504868	2.538444631	2.388827485
4	5.223148	4.771619	4.981180446	4.550568978
8	10.377544	9.95553	9.896793897	9.494330117
16	20.75307	19.108872	19.79166328	18.2236344
32	40.47055	36.058154	38.5957113	34.38772396
64	78.896031	63.960244	75.24109347	60.99722174
128	148.661225	98.434695	141.7743451	93.87460932
256	260.100403	102.202593	248.0509917	97.46795568
512	442.136149	111.358206	421.6537498	106.1994257
1024	663.97905	113.583745	633.2195565	108.3218644
2048	868.76993	116.353843	828.5232942	110.9636349
4096	1054.43721	119.854252	1005.589352	114.3018839
8192	1149.362604	120.407951	1096.117232	114.8299323
16384	1214.890712	115.946804	1158.609685	110.5754524
32768	1248.192421	119.509101	1190.368659	113.9727224
65536	1235.60088	117.734969	1178.360434	112.2807788
131072	1279.033273	117.544451	1219.780778	112.0990868
262144	1283.952284	120.379179	1224.47191	114.8024932
524288	1282.725269	120.677978	1223.301738	115.08745
1048576	1279.093044	117.754905	1219.83778	112.2997913
2097152	1261.022008	120.075822	1202.603902	114.5131895
4194304	1237.000596	99.23488	1179.695306	94.63772495
8388608	1220.579755	87.460922	1164.035177	83.40920733
16777216	1173.471526	91.233011	1119.109284	87.00655053
33554432	1187.511185	101.063456	1132.498542	96.38159034
67108864	1187.523005	105.893167	1132.509814	100.9875601
134217728	1184.586712	108.400913	1129.709548	103.3791323

Extra Credit: Why ‘3’? In the command to clear cache?

Writing to *drop_cache* cleans cache without killing any application/service. Command echo is doing the job of writing to file.
echo 1 clears pagecache only. Pagecache is cached files; recently accessed files are stored here.
echo 2 clears dentries and inodes only. Dentries and inode cache are directory and file attributes.
echo 3 clears pagecache, inodes and dentries, i.e. all three.

Part 5 : System Calls

Note: For parts 3 to 5 we wrote the program *complete.c*.
Execution command: *./complete <filename>*

We got the following results for read() by changing the block size to 1 Byte:

Performance in MiB/s with Block Size of 1 byte : 1.296118 MiB/s
Performance in B/s with Block Size of 1 byte : 1359078.279162 B/s

We further tried with three system calls: lseek, getpid and stat. Each was run in a loop for a time greater than 5 seconds. The table below shows how read compares to lseek , getpid and stat.

System Call	Speed (Sys Calls/sec)
read	1359078.28
lseek	1870927.94
getpid	1976489.55
stat	971242.54

Part 6 : Raw Performance

Program written: *fast.c*.
Execution command: *./fast <filename>*
Output: xor of all 4-byte integers in the file.

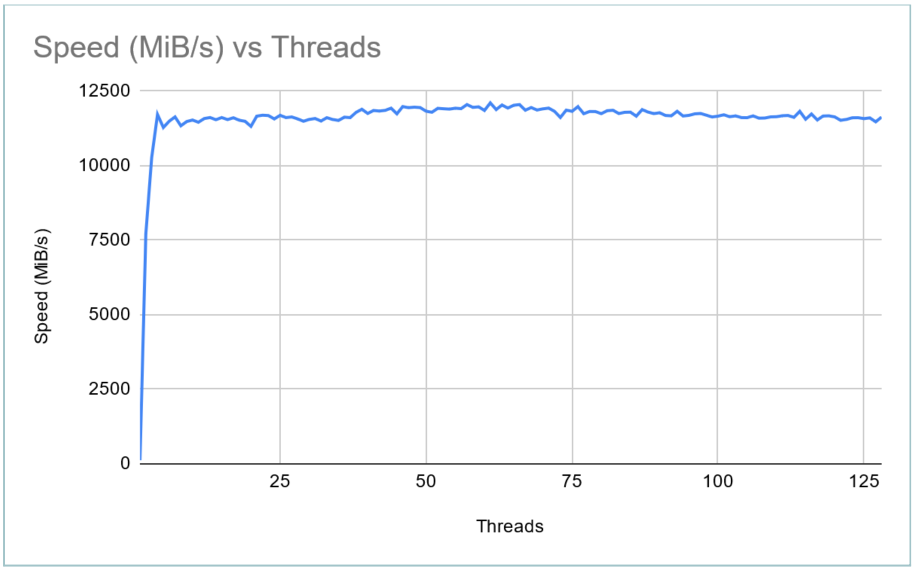
We wrote two more programs here: *fasttest.c* and *fast2.c*. Details of the programs are mentioned below and the way to run is included in README.

NOTE: Our program outperforms dd. On average, the speed for dd was 8.66 *GiB/s*, while the speed for our fast program is *11.18 GiB/s*.
If you don't add block size (bs) to the dd command, the speed for dd is on average about *120.16 MB/s*, which is even less.

Fast.c	dd	dd (without bs)
11.18 GiB/s	8.66 GiB/s	126 MB/s

Details about how we implemented fast.c:

- We implemented multithreading, which significantly improved the performance.
- The block size that we chose was the block size for which we were getting the maximum value in Part 3, which turned out to be **256 KiB**.
- To select the ideal number of threads, we ran our program multiple times for different thread counts. The code for this is present in *fasttest.c*. You can find the graph below:



If we zoom in on the earlier threads, we get the following graph:

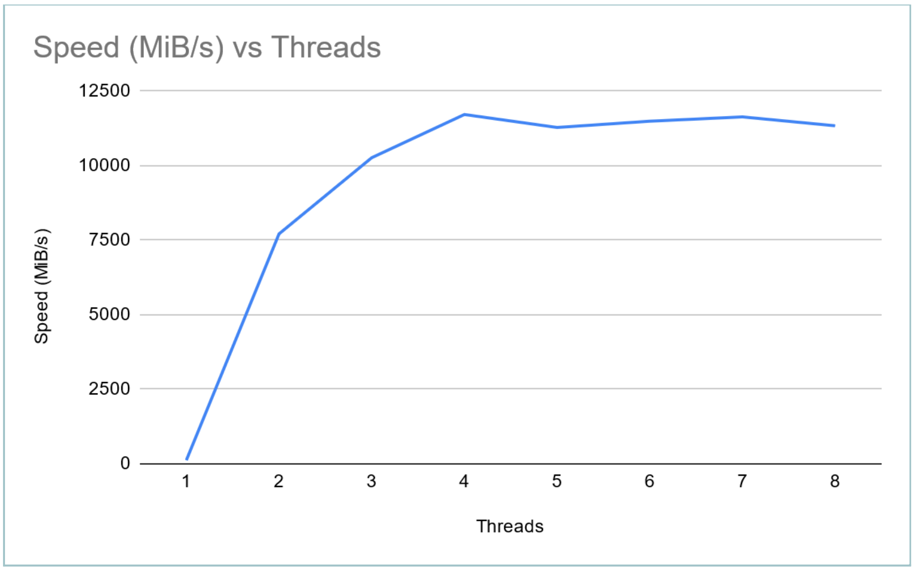


Table (Data for this graph): See Appendix (at the end)

The system we tested on had 4 logical processors, and we can clearly see from the graph that performance increases sharply till 4 threads. After that it mostly remains constant for larger threads. If we look at the data closely, we can see that performance keeps increasing slightly till about ~64 *threads* and then keeps decreasing slightly, but this change is a negligible amount, so not much can be concluded.

Cached v/s Non-Cached Performance (Fast.c)

To test our program's cached and non-cached performance, we fixed the threads to 64, and on an average got the following result:

Cached (Fast.c)	Non-cached (Fast.c)
11455.35 MiB/s	91.51 MiB/s

The fast.c program significantly outperforms run.c in the cached case. (Our performance for a block size of 256 KiB for run.c was around 1284 MiB/s). However, the non-cached result of fast.c is lesser than the result we got in Part 3 for our run.c program, where the speed was 120.379179 *MiB/s* for the block size of 256 KiB. To make sure we got the right result, we ran both run.c and fast.c in a **different system** after clearing the cache. For the block size of 256 KiB, we got the following result:

Non-Cached (Fast.c)	Non-Cached (Run.c)
2206.66 MiB/s	383.19 MiB/s

This result shows that our fast.c outperforms our run.c program in non-cached scenarios as well. Note that this result is contradictory to the performance in our own system.

Below is our thought process for choosing the number of threads in our final fast.c program:

Based on our observations we believe that taking a number of threads that is slightly more than the number of logical processors in a system seems a good and safe choice. We are doing the same in our file: *fast2.c* where we are taking the number of threads to be equal to 1.5 times the number of logical processors. However, there is no truly portable code to get the logical processors of a system. The solution we used: *int num_logical = sysconf(_SC_NPROCESSORS_CONF*); might not work on every system. Therefore we didn't do this in our fast.c program. (This code is present in fast2.c that can be checked).

The number of threads that we chose finally was **128**. The m5d.metal instance has 96 logical processors, and therefore we chose our number of threads to be greater than that number, assuming that the system in which our code will be tested will be of a similar or lesser specification to m5d.metal.

Appendix

Speed (MiB/s) v/s Threads

Threads	Speed (MiB/s)
1	120.209326
2	7712.755072
3	10267.98198
4	11717.65752
5	11282.45554
6	11492.7167
7	11637.24336
8	11339.41729
9	11481.52836

10	11532.30768
11	11457.88937
12	11581.30129
13	11616.28564
14	11543.05162
15	11615.17273
16	11547.72143
17	11608.29655
18	11525.54102
19	11490.01124
20	11319.13127
21	11663.25754
22	11693.18277
23	11681.96401
24	11568.58657
25	11691.00033
26	11614.01218
27	11634.67354
28	11571.26978
29	11490.65529
30	11554.31288
31	11583.39536
32	11498.11971
33	11611.27323
34	11550.53697
35	11517.7821
36	11628.28992
37	11613.07912
38	11788.31003
39	11894.13773
40	11753.10424
41	11852.33942
42	11832.10276
43	11857.41252
44	11929.66463

45	11740.42535
46	11984.01066
47	11946.62796
48	11965.23933
49	11948.69163
50	11828.30444
51	11792.81039
52	11925.27365
53	11911.31469
54	11898.93274
55	11928.86947
56	11914.80137
57	12050.89797
58	11958.05669
59	11975.36803
60	11853.69764
61	12108.48898
62	11883.25661
63	12036.862
64	11933.52833
65	12026.85996
66	12049.17227
67	11861.96581
68	11951.13603
69	11868.24634
70	11907.67914
71	11932.35381
72	11822.94626
73	11618.14097
74	11865.57349
75	11823.09504
76	11981.87112
77	11744.48549
78	11818.8317
79	11816.96127

80	11746.12502
81	11843.5248
82	11861.17965
83	11750.6422
84	11790.78788
85	11796.26505
86	11661.37571
87	11883.30671
88	11799.79582
89	11748.23016
90	11777.25252
91	11684.55477
92	11674.08984
93	11825.79839
94	11667.50594
95	11688.05533
96	11738.11526
97	11752.47945
98	11694.51689
99	11633.83319
100	11663.60743
101	11707.85048
102	11643.25205
103	11672.2166
104	11614.11985
105	11611.21345
106	11671.70911
107	11594.28243
108	11599.24438
109	11636.5948
110	11643.04766
111	11676.2176
112	11684.5911
113	11621.19458
114	11823.51658

115	11559.03921
116	11734.07173
117	11534.19528
118	11667.80779
119	11673.4855
120	11635.17781
121	11522.12534
122	11553.47226
123	11608.46387
124	11610.71125
125	11578.25672
126	11604.29418
127	11471.87898
128	11630.40102