Canvas Group Name: Student 1: Kevin Le, [kevle@ucsc.edu](mailto:kevle@ucsc.edu), #1463503 Student 2: Matthew Wark, [mtwark@ucsc.edu](mailto:mtwark@ucsc.edu)

**Notes to our work:** [**https://docs.google.com/document/d/1UjQTg2suEkgFWIL1h3qXAjVp1pO0B2Hdu6NU\_OBb2Q4/edit?usp=sharing**](https://docs.google.com/document/d/1UjQTg2suEkgFWIL1h3qXAjVp1pO0B2Hdu6NU_OBb2Q4/edit?usp=sharing)

CMPE110 HA1: Performance Detective

Due Date: Saturday 04/21/18

Clone the video transcoder x264 via git (http://git.videolan.org/git/x264.git).

Use ./configure and make to compile two different binaries of the same program. Compile one binary using the -O0 and another binary using the -O3 optimization level. Read the .configure script to determine how to pass additional cflags parameters to make such as defining the optimization level. You will also need to add two more flags when executing .configure to compile successfully: "--enable-pic --enable-shared"

If you have any issues in doing this, checkout "man git", "man make", "man gcc"

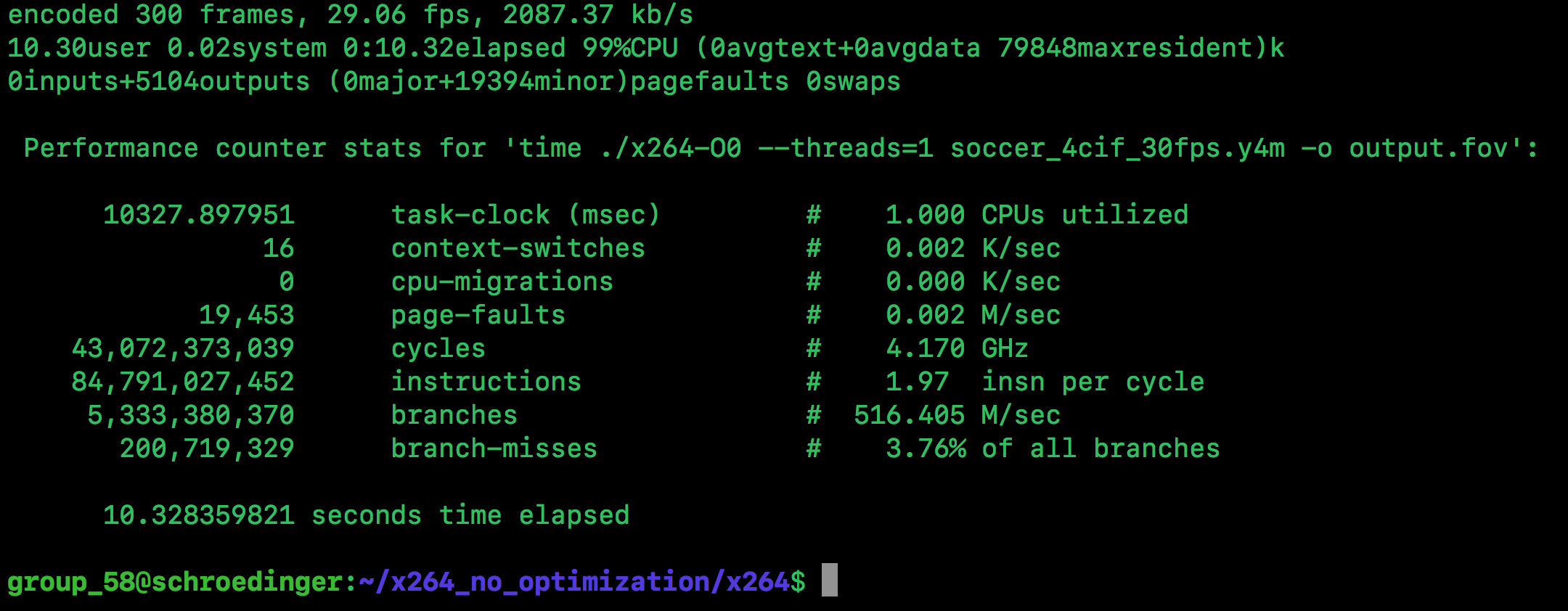
Download the "soccer\_4cif\_30fps.y4m" video file from <https://media.xiph.org/video/derf/y4m/>

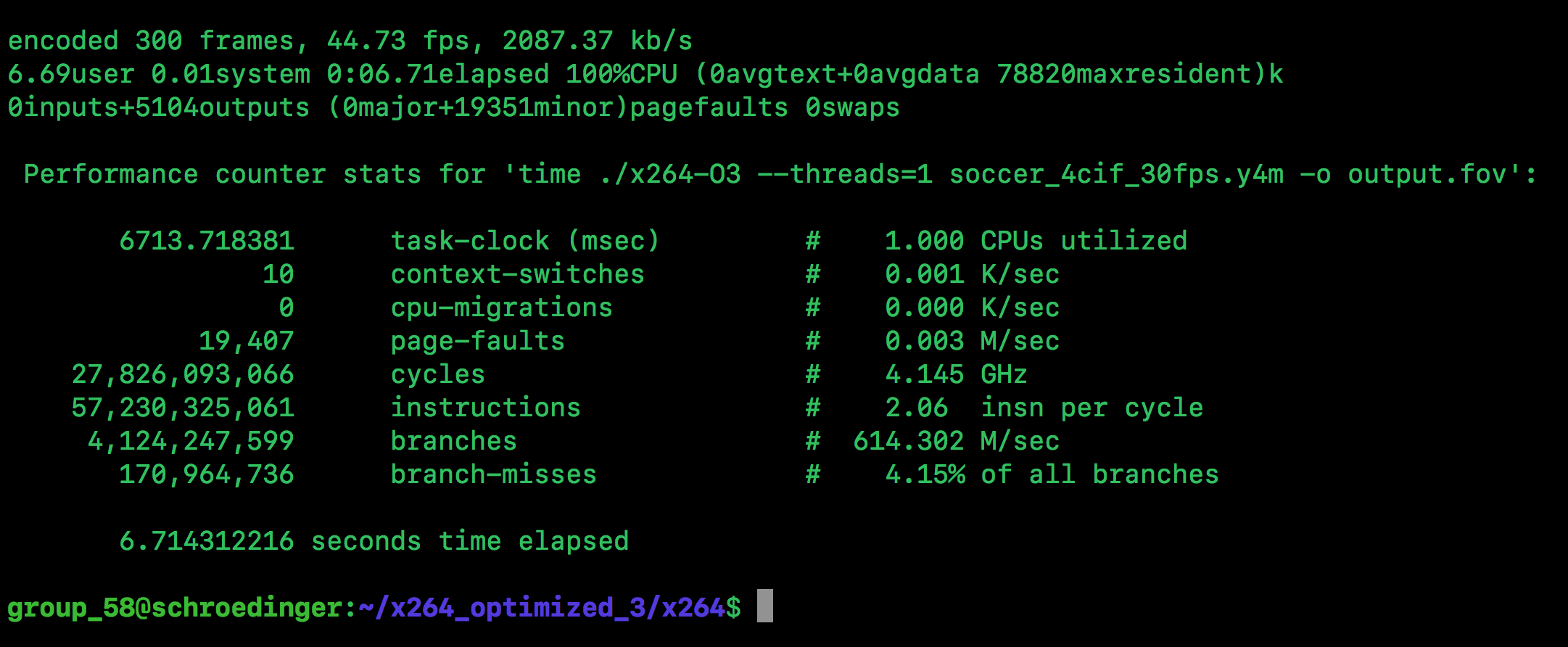
To run the experiments, login to “schroedinger.soe.ucsc.edu”. Use your canvas group name as user (e.g. group\_0 in case you are member of group 0. Use “aJe73Lb4oQ” as the password which you should change after you have logged in. As multiple students will be sharing the same machine, run your tests multiple times until you get reproducible results. If you need to copy a locally downloaded file to the server, use scp.

# 1) Execution Time:

Run the two x264 binaries in single threaded mode (--threads=1) to transcode the "soccer\_4cif\_30fps.y4m" video file.

Use time to measure the execution time of both binaries. What is the speedup you achieve by compiling with the -O3 optimization level?

Time O0: 10.328359821 seconds

Time O3:

6.714312216 seconds

Speedup: Time O1/Time O3 = 10.3283/6.7143 = 1.5382x speedup

(3 Points)

# 2) Parallel Speedup:

Run the two x264 binaries with 4 threads. What speedup do you achieve over single threaded? What is the parallelizable fraction of the x264 application according to Amdahl’s law?

* <https://www.youtube.com/watch?v=BxH93LTSOFo>
* <https://www.youtube.com/watch?v=KAfiOjGUl6o>

Speedup(1->4) O0:

1 Thread: 10.3283 seconds

4 Threads: 3.7031 seconds

Speedup of 2.789x

Speedup(1->4) O3:

1 Thread: 6.7143 seconds

4 Threads: 4.0004 seconds

Speedup of 1.6784x

Parallelizable Fraction O0: 7156/8361 = 0.85528

Parallelizable Fraction O3: 2.7136/5.0352 = 0.53893

(4 Points)

# 3. Perf Record

Use perf record to determine where the application spends most of its time. Determine the 4 function names that consume the most clock cycles for the gcc optimization levels 0. How much of the total execution time is spent in these functions? By how much was the compiler able to improve the performance (speedup) of the 4 functions? Fill in the table below:

* Google how to use perf wiki
* <https://www.youtube.com/watch?v=M6ldFtwWup0>
* Perf die.net website
* Perf record + perf report command for this

1 Thread:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Function | % O0 - proportion of running time for each function? | % O3 | Exec time O0 | Exec time O3 | Speedup |
| x264\_8\_me\_search\_ref | 8.12% | 6.98% | 0.83866281746  seconds | 0.4686589926768 seconds | 1.78949477246x speedup for O3 |
| get\_ref\_avx2 | 7.40% | 5.87% | 0.76429862675  seconds | 0.3941301270792 seconds | 1.939203766060784x speedup for O3 |
| refine\_subpel | 6.26% | 7.10% | 0.64655532479  seconds | 0.476716167336  seconds | 1.35626892707x speedup for O3 |
| x264\_8\_pixel\_satd\_8x8\_internal\_avx2 | 4.41% | 5.89% | 0.4554806681  seconds | 0.3954729895224 seconds | 1.151736478008446x speedup for O3 |

(4 Points)

# 4. Perf Stat

Use perf stat to determine the instruction count and average CPI for the two compiled binaries. Compute the clock cycle time or your system.

1 Thread:

O0: 43,072,373,039 cycles

O3: 27,826,093,066 cycles

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Optimization | Exec time | IC | CPI | Cycle Time |
| O0 | 10.328359821 seconds | 84,791,027,452 | 0.507982676155011 | 2.39792 x 10^-10 seconds |
| O3 | 6.714312216 seconds | 57,230,325,061 | 0.486212388909919 | 2.41297 x 10^-10 seconds |

Exec time = IC \* CPI \* Cycle time

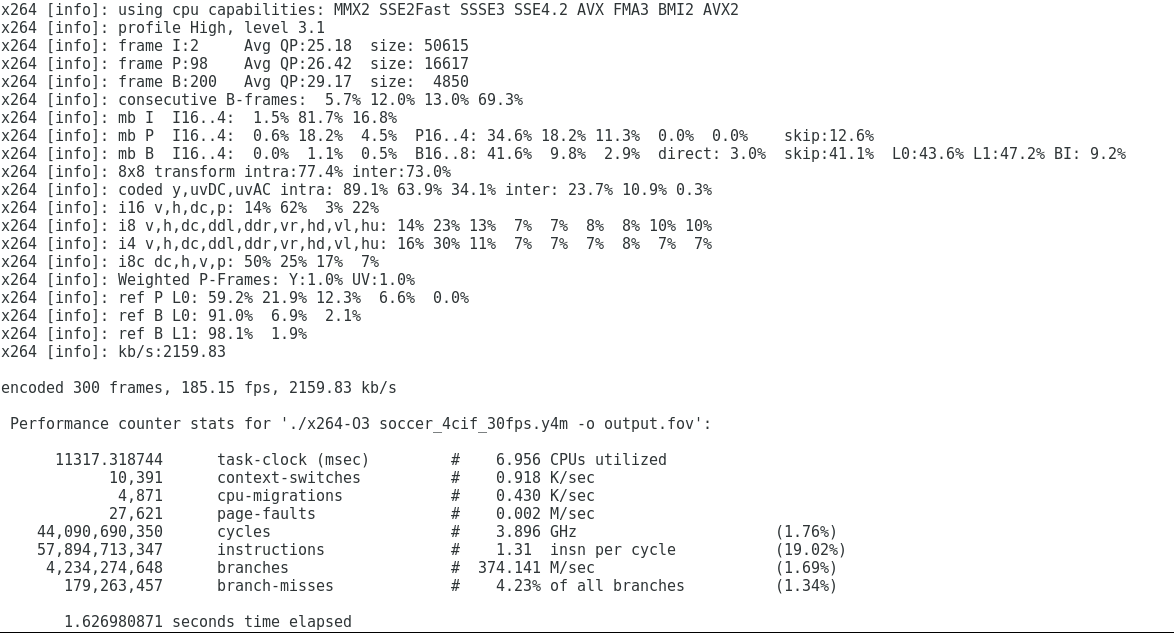
So Cycle time = Exec time / IC \* CPI

(2 Points)

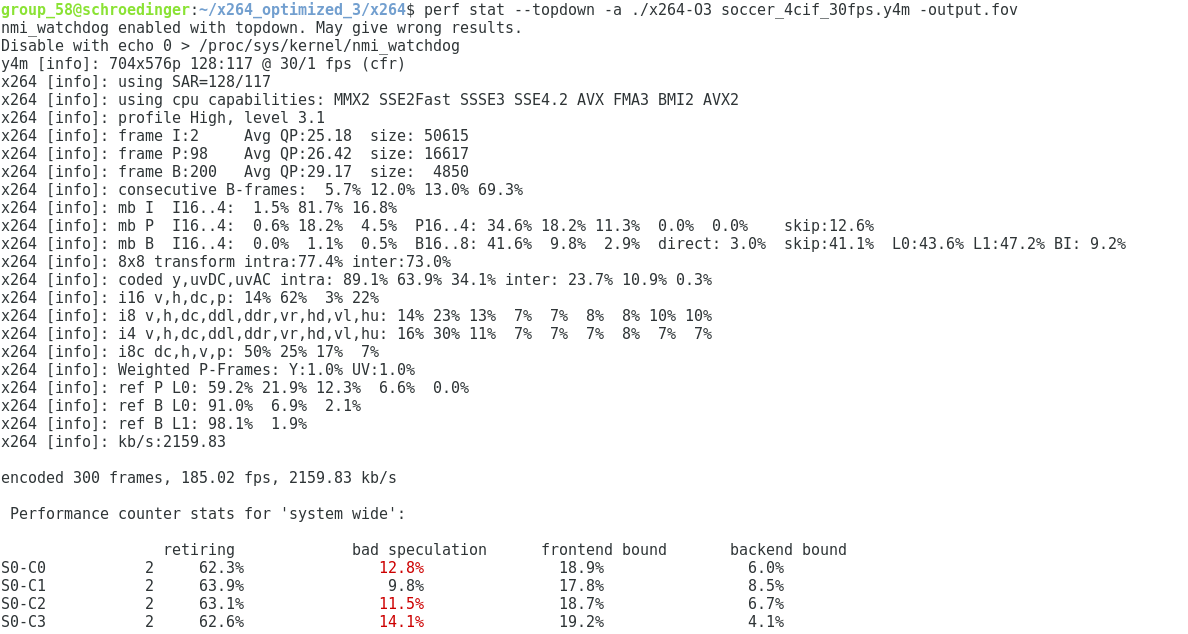
# 5. Perf Topdown

Use perf stat's top down methodology to determine the fraction of cycles spent on frontend, backend, retired and bad speculation [1]. Compute the theoretical optimal average CPI that a code can run on the system. Interpret your result (what does the processor need to be able to do to achieve this CPI?)

Running perf stat with threads not specified produces the following results:



Running perf top down, also with the threads not specified, produces the following results:



In order to achieve a theoretical optimal average CPI, we want to have 100% retiring. Additionally, we want to completely reduce bad speculation, front end and back end bounds to 0%. This would mean having no, or very close to 0 fetch latency, so we would have to eliminate all of our cache misses. We would also have to have no core, or memory bounds.

To calculate our theoretical optimal average CPI, we created a proportion from our “perf stat” results. When we don’t specify threads in perf, we achieve an average rating of 63.075%. We also observed that our program was running at 1.31 instructions per cycle.

Equation below:

63.075% 100%

------------ = -----------

1.31 IPC x IPC

=> 63.075x = 131

=> x = 131

----------

63.075

=> x = 2.079365079 IPC

=> x = theoretical optimal average CPI = .4809160306 CPI

[1] Ahmad Yasin: A Top-Down Method for Performance Analysis and Counters Architecture <https://sites.google.com/site/analysismethods/yasin-pubs/TopDown-Yasin-ISPASS14.pdf>