CSE 520 Computer Architecture II – Spring 2020

Programming Assignment 1 (100 Points)

Generic performance evaluation of a multicore system

In this assignment, you will be asked how the performance could scale up with number of parallel threads used in a multi-core system. With the given 3 benchmark programs (in https://www.dropbox.com/sh/je437v3853d5ux8/AAAz0CRb2I7PpbCeheGa-dmja?dl=0), please collect the following data using **perf** (see command usage: perf) command.

- Execution time
- Instruction Per Cycle (IPC)
- L1D cache load count, miss count, and load miss rate
- L2 cache demand data read count, miss count, and miss rate (L2 cache demand data read is a perf raw event defined in Chapter 19 Performance-monitoring Events, Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3B, https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html)
- Last-level cache (LLC) load count, miss count, and miss rate

The measurement data must be collected when the benchmark programs are run with multiple threads and with/without using hardware hyperthreading. That is, two sets of data will be collected from each benchmark program using taskset (see command usage: taskset):

1. Without using hardware hyperthreading and n threads which are affiliated with n physical cores, where n=1, 2, 4, 8.
2. With hardware hyperthreading and n threads which are affiliated with n/2 physical cores, where n=2, 4, 8, 16.

To automate the measurement, you should develop a shell script for each benchmark that executes perf command iteratively to collect various measures. Note that perf uses a limited number of PCM (Performance Counter Monitor) registers to count hardware events. To avoid any inaccuracy caused by a sampling method, you should run the benchmark multiple times and, during each run, collect only 2 to 3 measures. Between consecutive runs, you should have a simple program to remove any benchmark-related data from L1, L2, and L3 caches. This can be done by creating a vector of the size of L3 cache, filling in the vector with random data, and finding the maximum element of the vector. Note that the collected data may have some variations. In the assignment, you should compute the average of 3 runs for each measure.

An analysis report is required in the assignment. In the report, you will provide the table of above collected data, and plot 4 bar graphs with x-axis in the number of threads, for each benchmark, . With y-axis, multiple bars represent a group of measures in each bar graph for:

1. Execution time speedup normalized to 1 thread with/without hardware hyperthreading.
2. L1D cache load miss rate and MPKI (misses per 1000-instructions) with/without hardware hyperthreading.
3. L2 cache demand data read miss rate and MPKI with/without hardware hyperthreading.
4. LLC load miss rate and MPKI with/without hardware hyperthreading.

In additional to data tables and bar graphs, your comments and observations from the measured data should be reported. You may discuss why a benchmark program cannot scale up linearly and give supporting data to your statements. You can use the CPU spec such as how many physical cores and logical cores your CPU has, or other metrics you observed like various level cache hit rates if you can draw the correlation.

Note: All the data for submission must be collected from the machines available in the BYENG 217 lab, although personal machines can be used to test the run script.

Due Date

This assignment will be due at 11:59pm on Feb. 14

What to turn in for grading.

1. Create a working directory for each benchmark program to include your bash script file, the program to clear cache content, and a readme text file. We will follow the instruction in your readme file to compile your program and re-run your measurement. Also, please comment your script properly.
2. Compress the directories and your report (in pdf) into a zip archive file named **cse520-lastname-firstinitial_assgn01.zip.**
3. Submit the zip archive to Canvas by the due date and time.
4. There will be 20 points penalty per day if the submission is late. Note that submissions are time stamped by Canvas. If you have multiple submissions, only the newest one will be graded. If needed, you can send an email to the instructor and TA to drop a submission.
5. The assignment must be done individually. No collaboration is allowed, except the open discussion in the forum on Canvas. The instructor reserves the right to ask any student to explain the work and adjust the grade accordingly.
6. ASU Academic Integrity Policy (http://provost.asu.edu/academicintegrity), and FSE Honor Code (http://engineering.asu.edu/integrity) are strictly enforced and followed.

Appendix

Benchmarks to be used:

You will be provided with 3 benchmarks with their corresponding data input for this assignment. They are fluidanimate ,bodytrack and h264dec. All benchmarks allow you to specify how many threads they will used in the benchmark-dependant argument. Here are the command examples of running these benchmarks with which argument is for thread setting being pointed out.

Fluidanimate (from PARSEC https://parsec.cs.princeton.edu/index.htm):

Usage:fluidanimate <number of thread> 20 in_500K.fluid out.fluid

Example:

```
1 thread: ./fluidanimate 1 20 in_500K.fluid out.fluid

4 thread: ./fluidanimate 4 20 in_500K.fluid out.fluid
```

Bodytrack (from PARSEC https://parsec.cs.princeton.edu/index.htm):

Usage: bodytrack sequenceB_261 4 20 4000 5 2 <number of thread> 4

Example:

```
1 thread: ./bodytrack sequenceB_261 4 20 4000 5 2 1 4

4 thread: ./bodytrack sequenceB_261 4 20 4000 5 2 4 4
```

H264dec (from Starbench https://www.aes.tu-berlin.de/menue/forschung/projekte/abgeschlossene_projekte/starbench_parallel_benchmark_suite/):

Usage: h264dec -i ./park_joy_2160p.h264 -t <number of thread>

Example:

```
1 thread: ./h264dec -i ./park_joy_2160p.h264 -t 1

4 thread: ./h264dec -i ./park_joy_2160p.h264 -t 4
```

You can also refer to the run.sh script in each benchmark folder for how to run them with different threads in a more programmable way using shell script, together with perf for collecting various data.

Command usage: taskset

In this assignment, you will need to use taskset to forcefully assign the given benchmark running on the specific CPU core. Here is the common usage of taskset

```
taskset -c <logical core list> <target command and its argument>
```

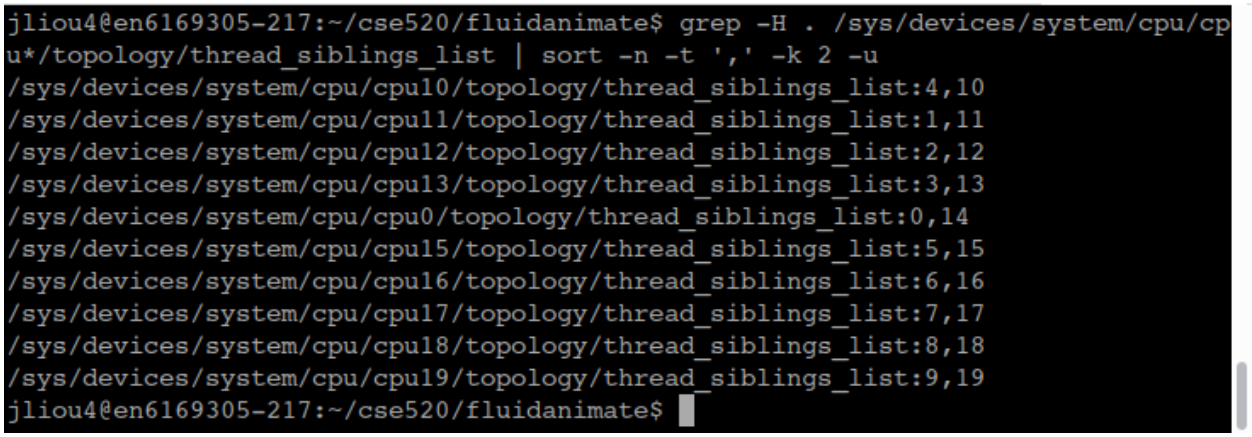
For example, the following command is to run fluidanimate with 4 threads **only** on core 1 and 3.

```
taskset -c 1,3 ./fluidanimate 4 20 in_500K.fluid out.fluid
```

Note that the cores specified here are logical cores. Modern CPUs all have hyperthreading feature, meaning each physical CPU core will have 2 logical cores. So, the more important thing is to determine which logical cores are sharing the same physical core. Here is a command showing such mapping (Just copying the entire command and run it on the system).

```
grep -H . /sys/devices/system/cpu/cpu*/topology/thread_siblings_list | sort -n
-t ',' -k 2 -u
```

The result will look like this



This shows the paired logical cores, such as logical core 4 and 10, who share the same physical core. This is to say, to run 2-thread fluidanimate on 1 physical core with hyperthreading, the following command as an example will be used.

```
taskset -c 4,10 ./fluidanimate 2 10 in_500K.fluid out.fluid
```

whereas the following command is to run the same program on 2 physical cores without hyperthreading.

```
taskset -c 4,1 ./fluidanimate 2 10 in_500K.fluid out.fluid
```

Command usage: perf

Modern CPUs usually contain limited PCM (Performance Counter Monitor) for software developers to better understand the performance impact to the CPU microarchitecture caused by developed program so that they can further optimize their program against this specific CPU.

On Linux, perf is the most common tool to retrieve PCM's reading in high level language. Here is a simple tutorial about perf command where you will need for the assignment:

1. List available perf everts
   `perf list`
   you will see the output similar to the image below.

```
List of pre-defined events (to be used in -e):

  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  bus-cycles                                         [Hardware event]
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  ref-cycles                                         [Hardware event]
```

2. Profile your program with perf

Let's say we want to profile our test program with cache-misses, cache-references, and cpu-cycles events. The perf command will be

```
perf stat -e cache-misses,cache-references,cpu-cycles /path/to/your/program
```

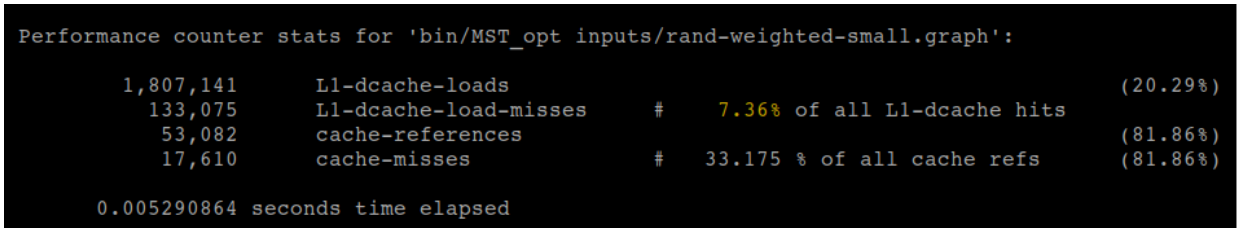
```
Performance counter stats for './test':

           2,217      cache-misses              #    5.484 % of all cache refs
          40,430      cache-references
     135,358,826      cpu-cycles

     0.035086799 seconds time elapsed
```

More examples can be found in the `run.sh` script of each benchmark

You must be aware that the number of PCM on a CPU is usually small, meaning that the number of an event can be profiled at once is limited. Some events even need more than 1 PCM to calculate. When required number of PCM is larger than how many CPU has, perf will use multiplex mode, which uses PCM in a time-sharing manner. When this occurs, you will see a percentage number behind a reported event like the image below.



In such a mode, the perf might not be able to report accurate number. To prevent this from happening, you might want to profile target program with only a few events, says 2, but in multiple times with different events combination.