Report Lab 1

Silvio Baratto, Tobias Ganzmann

13 ottobre 2022

Indice

1 PERF (I)

1.1 Definition profiler

A profiler in software engineering, is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization, and more specifically, performance enginee-ring. Profilers, which are also programs themselves, analyze target programs by collecting information on their execution. Based on their data granularity, on how profilers collect information, they are classified into event based or statistical profilers.

1.2 Code explanatin for benchmarking

The code given to do some performance analysis using perf as a profiler consist in a simple function that initialize a matrix of dimension $2000x2000$ and fill all the matrix with a value equal to 1. The program repeat this operation 100 times.

1.3 Performance analysis tool (perf)

We are interested in see how the program behaves with different optimizers and code optimizations using perf to check the performance results. To do that with this instructions we want to look at:

   a. CPU clock cycles at user level

   b. Machine code instructions at user level

   c. First-level data cache load hitst at user level

   d. First-level data cache load misses at user level

   e. First-level data cache store hits at user level

   f. First-level data cache store misses at user level

   g. Last-level cache load hits at user level

   h. Last-level cache load misses at user level

   i. Last-level cache store hits at user level

   j. Last-level cache store misses at user level

The way how we obtain the following benchmark results is from this terminal command:

```
$ sudo perf stat -e cycles:u -e instructions:u \
                 -e L1-dcache-loads:u -e L1-dcache-load-misses:u \
                 -e L1-dcache-stores:u -e L1-dcache-store-misses:u \
                 -e LLC-loads:u -e LLC-load-misses:u \
                 -e LLC-stores:u -e LLC-store-misses:u ./task1
```

From the first run of the code we obtain the following results:

```
CPU = 5.893150 ms

 Performance counter stats for './task1':

    2,743,735,005        cycles:u                     (32.74%)
    3,199,246,694        instructions:u               (44.26%)
      402,366,995        L1-dcache-loads              (55.79%)
      803,099,963        L1-dcache-load-misses        (67.31%)
      799,527,269        L1-dcache-stores             (67.46%)
    <not supported>      L1-dcache-store-misses
        1,628,527        LLC-loads                    (67.46%)
            1,094        LLC-load-misses              (44.21%)
        2,479,174        LLC-stores                   (21.69%)
              258        LLC-store-misses             (21.69%)


      0.590644181 seconds time elapsed

      0.590584000 seconds user
      0.000000000 seconds sys
```

Changing the code in such way that array is now traversed by rows instead of by columns, hence taking advantage of both the row major order used in C and the cache hierarchy we obtain the following results:

```
CPU = 1.423880 ms

 Performance counter stats for './task1':

      648,114,804        cycles:u                     (32.97%)
    1,624,440,721        instructions:u               (44.14%)
      396,820,828        L1-dcache-loads              (55.31%)
       12,204,130        L1-dcache-load-misses        (66.48%)
      383,946,246        L1-dcache-stores             (66.48%)
    <not supported>      L1-dcache-store-misses
            1,680        LLC-loads                    (66.48%)
               40        LLC-load-misses              (44.69%)
        6,988,534        LLC-stores                   (22.35%)
               67        LLC-store-misses             (22.35%)


      0.143955657 seconds time elapsed

      0.135983000 seconds user
      0.007999000 seconds sys
```

From the graph below we can see how much changing to the row major perform better on cycles, instructions and cache

1.4 Conclusion

2 PERF (II)

3   GPROF

4 OpenMP (Compilation and basic directives)

5 Optional assignment