DSSC - Profiling

Presta Alberto

December 2018

Introduction

The aim of this exercise is to learn how to profile a code, using specific tools from *gprof*, *perf* and *Valgrind*. What is profiling? It's a way to evaluate our code, in order to find out hotspots and bottlenecks: Profiling is the first step for optimization (obviously after writing a working code).

The Code

First of all we have to choose an "interesting" code to profile. We wrote a code which calculates the transpose of a matrix, filled with random values. My code receives in input from the user the dimension of the matrix, then it fills (through a function *fill*) the matrix with random numbers and finally we calculate the transposed matrix. We repeat this procedure for 1000 times, always considering different matrices filled with random numbers[1]. For all the report we will deal with matrices of syze 100×50. Below there are the declarations of the functions that we used in the code.

```
void fill(double*p, int row,int col);
double* transpose (double* matrix,int row,int col);
void print( double* matrix, int row, int col);
```

Also we have calculated the transpose in a naive way; without any kind of optimization technique like blocking or loop unrolling. Here we are the function *transpose*:

```

double*  transpose(double* matrix, int n, int m){
    double* b = new double[n*m]{};
     int cont=0;
     for(int j{0}; j<n;j++) {
         for(int i{0};i<m;i++) {
            b[i*n+j]=matrix[cont];
             cont+=1;
                }
             }
     return b;
    }
```

[1] we used the function *rand()*.

Profiling with GProf

The first profiling tool which we want to use is called Gprof. In order to use it, we have to compile the code with the flag *-pg*: here we have the results.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 65.15      0.13     0.13     2000    65.15    65.15  print(double*, int, int)
 15.04      0.16     0.03     1000    30.07    30.07  transpose(double*, int, int)
 10.02      0.18     0.02     1000    20.05    20.05  fill(double*, int, int)
 10.02      0.20     0.02                             main
  0.00      0.20     0.00        1     0.00     0.00  _GLOBAL__sub_I_main
  0.00      0.20     0.00        1     0.00     0.00  __static_initialization_and_destruction_0(int, int)
```

Where the columns stand for:

1. *Each % time*: the percentage of the total running time of program used by this function.
2. *Cumulative seconds*: a running sum of the number of seconds accounted for by this function and those listed above it.
3. *self seconds*: the number of seconds accounted for by this function alone. This is the major sort for this listing.
4. *calls* : the number of times this function was invoked, if this function is profiled, else blank.
5. *self/ms call*: the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.
6. *name*: This is the name of the function.

We notice that gprof is a *sample based profiling*, where every 0.01 seconds the state of the code is controlled. We observe that much that the 65% of the time is spent inside the function *print*: this is not so unexpected because for each matrix we print both the matrix and its transposed (which means that we have 2000 prints). We have to say that Gprof is no longer used so much (indeed it is consedered a dinosaur) for several reasons (lacks real multithread support, lacks real line-by-line capability,does not profile shared libraries,need recompilation).

profiling with Valgrind

In this section we will focus on another important tool to profile a code: *Valgrind*. In particular Valgrind has got two important devices:

1. Memcheck: It is a *memory checking control* which, during the execution of the code, takes over many problems concerning memory management (for instance incorrect allocation or memory leaks).
2. Cachegrind: it is a *cache profiler*. It simulates data movements throughout different levels of caches and it counts (we can say it forecasts) how many *cache misses* happen at each level of memory. With cachegrind we can also create a call-tree.

Memcheck

After having compiled our code, we run the Memcheck tool with this command:

```
valgrind --tool=memcheck ./trasposta
```

And we have the following result:

```
HEAP SUMMARY:
==13461==     in use at exit: 80,002,588 bytes in 2,001 blocks
==13461==   total heap usage: 2,001 allocs, 0 frees, 80,002,588
    bytes allocated
==13230== LEAK SUMMARY:
==13230==    definitely lost: 79,520,000 bytes in 1,988 blocks
==13230==    indirectly lost: 0 bytes in 0 blocks
==13230==      possibly lost: 480,000 bytes in 12 blocks
==13230==    still reachable: 2,588 bytes in 1 blocks
==13230==         suppressed: 0 bytes in 0 blocks
==13230== Reachable blocks (those to which a pointer was found) are
    not shown.
==13230== To see them, rerun with: --leak-check=full --show-
    reachable=yes
==13230==
==13230== For counts of detected and suppressed errors, rerun with:
    -v
==13230== ERROR SUMMARY: 20 errors from 9 contexts (suppressed: 6
    from 6)
Profiling timer expired
```

As we can see, our code has some problems with memory leak: indeed we ha a huge number of *definitely lost bytes* and we didn't do anything to free memory in the code.

Cachegrind

We used this tools to create a call-tree which describe our code and every call of this one.[2]. The arguments used in the compilation ensure that both branch and cache prediction are performed fully[3]. We compile in this way:

```
valgrind --tool=callgrind --dump-instr=yes --cache-sim=yes --branch
    -sim=yes --collect-jumps=yes ./trasposta
```

we have the following call-tree:

[2]Moreover we can control (using cachegrind) the percentage of time spent in each function

[3]In this case I also decided to decrease the number of prints from 2000 to 200, in order to give much more relevance to the other functions

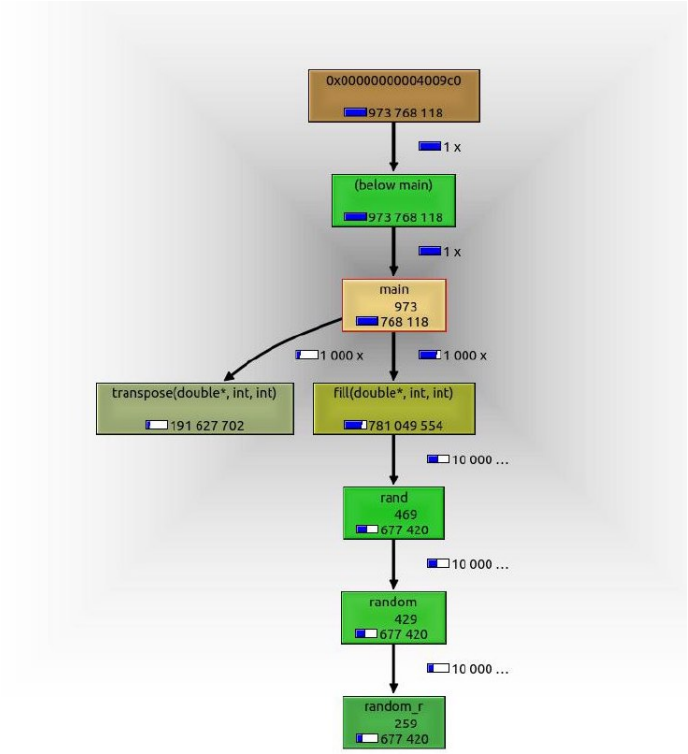

Figure 1: Call-tree generated by Cachegrind

Profiling with Perf

In the first section we have mentioned about some issues of Gprof: for these problems a lot of people use *Perf* to profile a code. Perf is the standard way to access the hardware performance counters, in both sampling and counting mode. First of all we use the command *perf -stat* in order to have a counting mode in analyzing the code; in particular we want to have a profile of core events (number of cycles, number of instructions,branch misses etc..). We give the following command:

```
perf stat -e cpu-cycles:u,instructions:u,cache-misses:u,branches:u,
    branch-misses:u  ./trasposta
```

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

        2170980424 cpu-cycles:u                #    0,000 GHz
        3610948205 instructions:u              #    1,66  insns per
    cycle
              4151 cache-misses:u
         663412449 branches:u
           1253021 branch-misses:u             #    0,19% of all
    branches

      11,354563146 seconds time elapsed
```

We notice that even if we have a huge number of branch-misses (1253021), they represent a small percentage of the total number of branches. Also the total number of cache-misses is not so large if we take into account that the code calculates the transposed of 1000 of different matrices of size 100×50.

the last thing we'll do is take a look to "perf in sampling mode", using the command *perf record*. The perf record command is used to sample the data of events to a file. The command operates in a per-thread mode and by default records the data for the cycles event. Below we have the first part of the output (where there are most frequent functions called in the code).

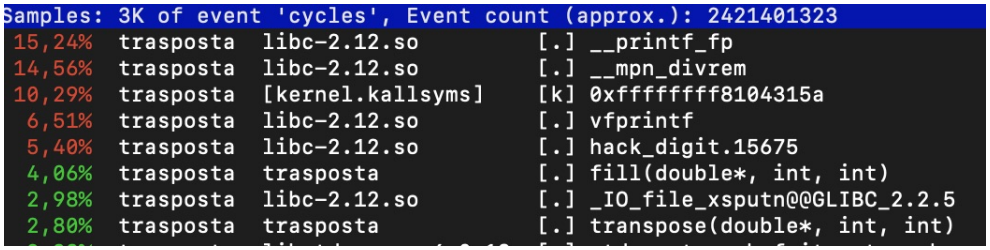

Figure 2: output of the command *perf record*