Exercise 2-Profiling

Michela Venturini

November 26, 2018

Abstract

In this exercise we profile a choosen code by using Fprof, Valgrind and Gperf.

1 The code

The code that i used is divided in two part: the first one performs four calls of functions that execute simple cycles, while the second one implements the Laplace algorithm to calculate the determinants of three matrices, by a recursive function.

2 Profiling

2.1 Gprof Gprof is a statistical performance analysis tool for Unix applications. To analize the program use the following steps:

1 compile the program with gcc by using -pg option to insert instrumentation code (for example a call to the monitor function `mcount` before each function call)

2 run the program that produced the file *gmon.out* where are saved sampling data

3 generate the output by the command `gprof program.x gmon.out > data.txt`

4 analize the output that consists in two parts: the flat profile and the call graph.

Flat profile The flat profile shown in figure 1 gives the total execution time spent in each function and its percentage of the total running time. Function call counts are also reported. Output is sorted by percentage, with hot spots at the top of the list.

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 52.99      0.73     0.73        1   731.24   731.24  func3
 31.21      1.16     0.43 18705900     0.00     0.00  togli_riga0_colonnai
 11.61      1.32     0.16        3    53.42   197.00  determinante
  3.63      1.37     0.05        1    50.08   781.32  func1
  0.73      1.38     0.01                             main
  0.00      1.38     0.00        1     0.00   731.24  func2
  0.00      1.38     0.00        1     0.00     0.00  func4
  0.00      1.38     0.00        1     0.00     0.00  prod_matr
```

Figure 1: Flat profile

Textual call graph The second part of the output is the textual call graph (see figure 2), which shows for each function: who called it (parent), who it called (child subroutines) and recursive calls. There is an external tool called gprof2dot capable of converting the call graph from gprof into graphical form, anyway we decided to show only the graphical output of Valgrind later.

```
granularity: each sample hit covers 2 byte(s) for 0.72% of 1.38 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.01    1.37                 main [1]
                0.05    0.73       1/1           func1 [2]
                0.16    0.43       3/3           determinante [5]
                0.00    0.00       1/1           func4 [7]
                0.00    0.00       1/1           prod_matr [8]
-----------------------------------------------
                0.05    0.73       1/1           main [1]
[2]     56.5    0.05    0.73       1         func1 [2]
                0.00    0.73       1/1           func2 [3]
-----------------------------------------------
                0.00    0.73       1/1           func1 [2]
[3]     52.9    0.00    0.73       1         func2 [3]
                0.73    0.00       1/1           func3 [4]
-----------------------------------------------
                0.73    0.00       1/1           func2 [3]
[4]     52.9    0.73    0.00       1         func3 [4]
-----------------------------------------------
                             18705900             determinante [5]
                0.16    0.43       3/3           main [1]
[5]     42.8    0.16    0.43       3+18705900 determinante [5]
                0.43    0.00 18705900/18705900     togli_riga0_colonnai [6]
                             18705900             determinante [5]
-----------------------------------------------
                0.43    0.00 18705900/18705900     determinante [5]
[6]     31.2    0.43    0.00 18705900          togli_riga0_colonnai [6]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[7]      0.0    0.00    0.00       1         func4 [7]
-----------------------------------------------
                0.00    0.00       1/1           main [1]
[8]      0.0    0.00    0.00       1         prod_matr [8]
-----------------------------------------------
```

Figure 2: Textual call graph

Annotated souce code By using the command `gprof -A sample.x gmon.out` we obtain the annotated souce code with indications on the most executed lines and the the percentage of code executed.

Obtained informations What we deduce is that the most expensive funtions are *func3*, *togi_riga0_colonnai* and *determinante*. Moreover *func3* is called only one time so we expect this funcion is the most expensive in terms of single execution, while *togli_riga0_colonnai* is called 18 milions of time so we expect this funcion not to be so expensive in terms of time. From the call graph we deduce that *func3* was not called in main but by *func2*, called by *func1* in the main; morever the function *determinante* is called recursively a huge amount of time and it is responable for the calls at the function *togli_riga0_colonnai*. The function *func4* is not actually mentioned in the profiling output due to its small impact on the execution - probably the sampling period used by Gprof is not sufficiently small.

Considerations on gprof At run-time, timing values reported by gprof are obtained by statistical sampling, done by probing the program counter at regular intervals using operating system interrupts. The resulting data is not exact, rather a statistical approximation and the amount of error is usually more than one sampling period (in our case 0.01 s). Moreover Gprof cannot measure time spent in kernel mode (syscalls, waiting for CPU or I/O waiting), and only user-space code is profiled.

2.2 Valgrind Valgrind is a programming tool for memory debugging, memory leak detection, and profiling. It is a virtual machine using just-in-time (JIT) compilation techniques and run on host and target (or simulated) CPUs of the same architecture. To profile the program we execute the following steps:

1 compilation with gcc by using -g option to include debugging information so that memcheck can report the exact line numbers

2 Run the program by using `valgrind --leak-check=yes ./sample`. The flag –leak-check=yes turns on the memcheck tool provided by Valgrind. The output is shown in figure 3. This tool analizes the behaivior of the Heap and the error

```
==4560==
==4560== HEAP SUMMARY:
==4560==     in use at exit: 0 bytes in 0 blocks
==4560==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==4560==
==4560== All heap blocks were freed -- no leaks are possible
==4560==
==4560== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 3: Output of Memcheck tool

detected. In this case no error happened.

3 run again the program by using `valgrind --tool=calgring ./sample`

4 with the obtained output run kcachegrind tool: `Kcachegrind calgrind.out...` At this point i can obtain the call graph shown in figure 4. This graphical representation of the functions' calls explains the structure of the execution. Also in this case func4 is not mentioned but it is possible to understand levels of recursion in function *determinante*.

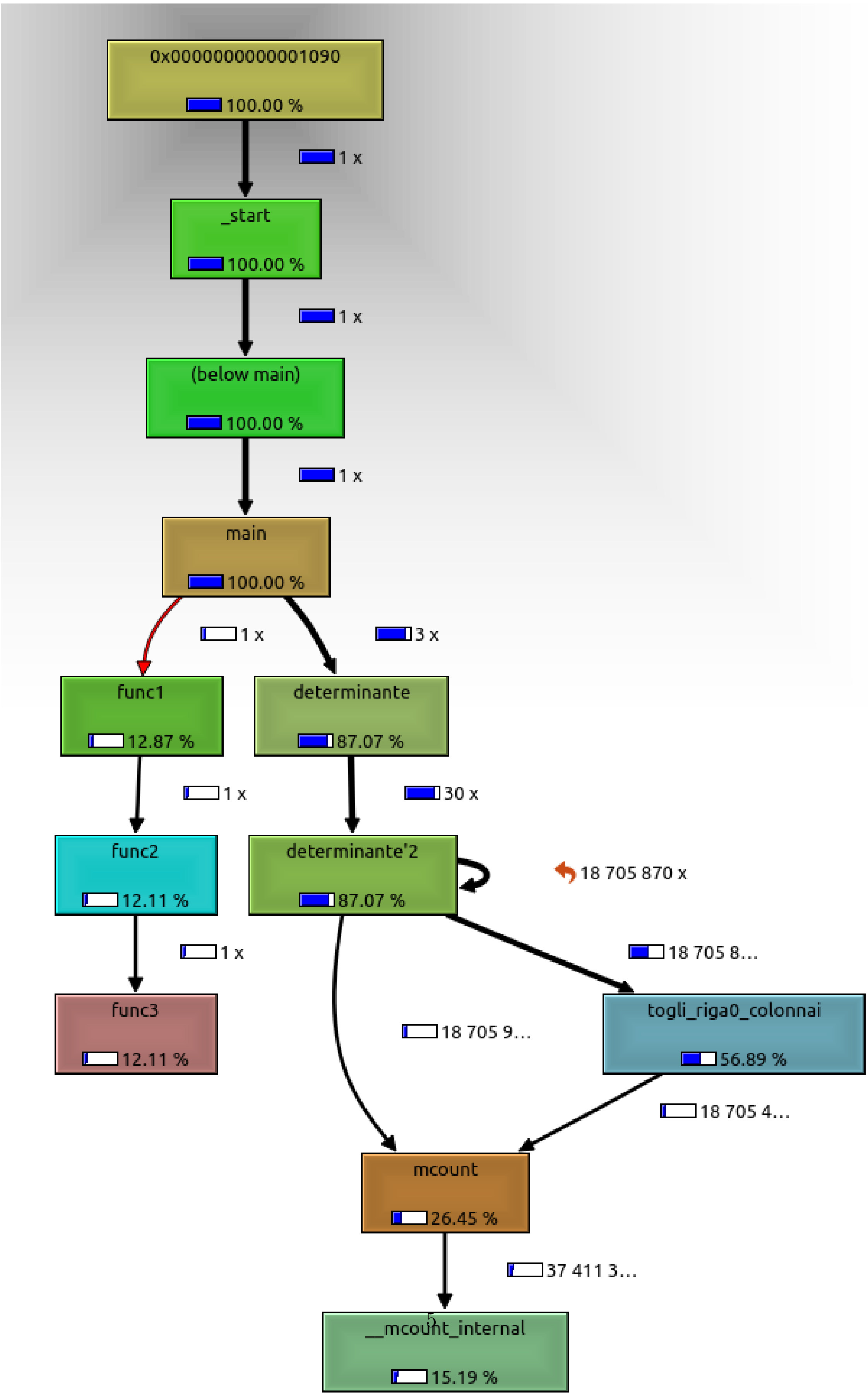


Figure 4: Callgraph by Kcachegrind

2.3 Perf Perf is a performance analyzing tool available on Linux and it is capable of statistical profiling of the entire system (both kernel and userland code). We use it to obtain deeper information about system and program behaiviour by using the commands:

1 `perf stat ./sample`
the output is shown in figure 5. From this first analisys we notice that the nuber of instructions per cycle is low. The hardware on which we run the code is a Intel core i5 5200U dual core with hyperthreading enabled so we expect a higher number of instructions per cycles. This fact could suggest us the program is memory bounded.

```
Performance counter stats for './sample1':

       877,519639      task-clock (msec)         #    0,998 CPUs utilized
               30      context-switches          #    0,034 K/sec
                0      cpu-migrations            #    0,000 K/sec
               87      page-faults               #    0,099 K/sec
    1.920.681.729      cycles                    #    2,189 GHz
    1.264.238.864      instructions              #    0,66  insn per cycle
      332.541.170      branches                  #  378,956 M/sec
          225.095      branch-misses             #    0,07% of all branches

      0,878872109 seconds time elapsed
```

Figure 5: First output of perf

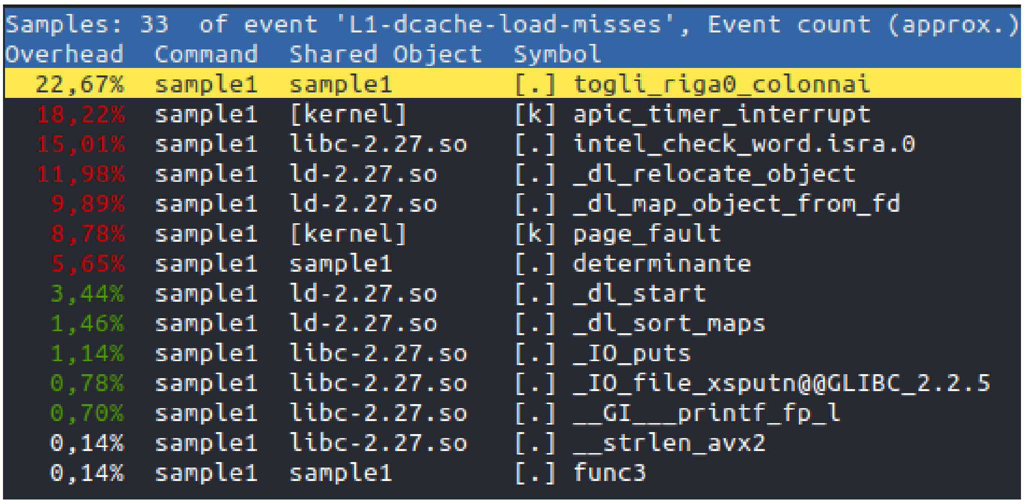
2 `sudo perf stat --repeat=5 -e cache-references:u,cache-misses:u ./sample`
the output is shown in figure 6. In this case we want to analize cache references compared with cache-misses and we obtaine a high percentage in average over five executions($\tilde{5}$6% of the total). This behaivior suggest me a bad usage of memory in the code.

```
Performance counter stats for './sample1':

           16.495      cache-references:u
            9.319      cache-misses:u            #   56,496 % of all cache refs

      0,861526376 seconds time elapsed
```

Figure 6: Output of perf stat

3 By using `sudo per record -e cache-references:u,cache-misses:u, L1-dcache-misses:u,L1-icache-misses:u ./sample` we are able to identify inside the code the routine that causes this behaiviour. The figure 7 shows that the

function that causes most of L1-dcache-misses is in particular `togli_riga0_colonnai` and this analysis allows us to improve the code in memory access pattern.



Samples: 33 of event 'L1-dcache-load-misses', Event count (approx.)

Overhead	Command	Shared Object	Symbol
22,67%	sample1	sample1	[.] togli_riga0_colonnai
18,22%	sample1	[kernel]	[k] apic_timer_interrupt
15,01%	sample1	libc-2.27.so	[.] intel_check_word.isra.0
11,98%	sample1	ld-2.27.so	[.] _dl_relocate_object
9,89%	sample1	ld-2.27.so	[.] _dl_map_object_from_fd
8,78%	sample1	[kernel]	[k] page_fault
5,65%	sample1	sample1	[.] determinante
3,44%	sample1	ld-2.27.so	[.] _dl_start
1,46%	sample1	ld-2.27.so	[.] _dl_sort_maps
1,14%	sample1	libc-2.27.so	[.] _IO_puts
0,78%	sample1	libc-2.27.so	[.] _IO_file_xsputn@@GLIBC_2.2.5
0,70%	sample1	libc-2.27.so	[.] __GI___printf_fp_l
0,14%	sample1	libc-2.27.so	[.] __strlen_avx2
0,14%	sample1	sample1	[.] func3

Figure 7: Output of perf report