Exercise 2 - Profiling

Alessia Paoletti
Matricola SM3500374

The code that I have choosen to profile calculates the inverse of a matrix. The matrix has 10 rows and 10 columns, with the diagonal elements equal to $row\_number * column\_number * 1000$ and all the other elements equal to 1. The code is very trivial: once the non-singularity of the matrix is proved, it calculates the determinant of the matrix. Then the main function calls the function cofactor which in turn calls the function transpose.

Profiling works by changing how every function in your program is compiled so that when it is called, it will stash away some information about where it was called from. From this, the profiler can figure out what function called it, and can count how many times it was called. This change is made by the compiler when the program is compiled with the `-pg` option which causes every function to call `mcount` as one of its first operations.
The `mcount` routine, included in the profiling library, is responsible for recording in an in-memory call graph table both its parent routine (the child) and its parent's parent. This is typically done by examining the stack frame to find both the address of the child, and the return address in the original parent. Since this is a very machine-dependent operation, `mcount` itself is typically a short assembly-language routine that extracts the required information, and then calls `__mcount_internal`. The `__mcount_internal` function is responsible for maintaining the in-memory call graph.

Once I have compiled the code with `-pg` option that generate extra code to write profile information suitable for the analysis program `gprof`, I am able to get the flat profile of the program, that shows the total amount of time the program spent executing each function. The flat profile is reported in Figure 1. As it shows the 99.74% of the total running time of the program is spent in the `determinant` function. This is due to the fact that primarily it has to prove the nonsingularity of the matrix, then it is used by the `cofactor` function and finally it is called by the `transpose` function. The `determinant` function is recursive, with the base case of 1 by 1 matrix. So we can affirm that the core of the program is the `determinant` function, hence to the fact is invoked 102 times and the number of seconds accounted for this function alone is 2.49 seconds. The cumulative seconds of the `main` function, that invoked all the other functions, is 2.50 seconds, and that prove that the time spent in the `cofactor` and `transpose` fucntions is almost inexistent.

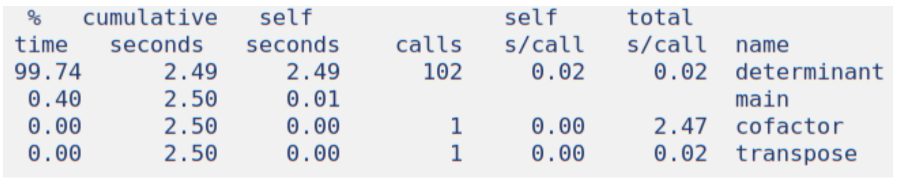

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 99.74      2.49     2.49      102     0.02     0.02  determinant
  0.40      2.50     0.01                             main
  0.00      2.50     0.00        1     0.00     2.47  cofactor
  0.00      2.50     0.00        1     0.00     0.02  transpose
```

Figure 1: Flat profile

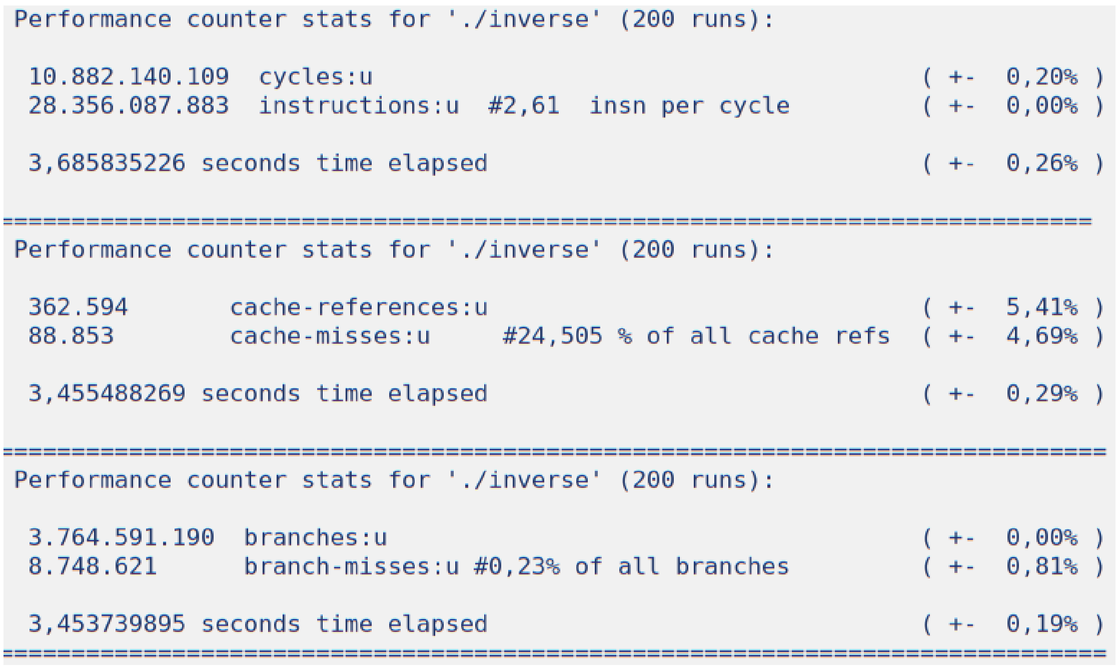
perf

I have profiled my program with `perf`, a performance analyzing tools that supports hardware performance counters, tracepoints, software performance pointers and dynamic probes. In order to collect information about the number of instruction, about the cache references and misses and about the branches and branc-misses I have runned the following commands:

```
perf stat --repeat=200 -e cycles:u,instructions:u, ./inverse
perf stat --repeat=200 -e cache-references:u,cache-misses:u, ./inverse
perf stat --repeat=200 -e branches:u,branch-misses:u, ./inverse
```

I have choosen to collect information about hardware events and in particular I have selected the cache information, in order to see how many cache misses we have, due to the fact that the time spent in access the memory and get the data affects a lot the total time of execution.

The results are reported below:



```
Performance counter stats for './inverse' (200 runs):

 10.882.140.109  cycles:u                                     ( +-  0,20% )
 28.356.087.883  instructions:u  #2,61  insn per cycle        ( +-  0,00% )

 3,685835226 seconds time elapsed                             ( +-  0,26% )

==========================================================================
Performance counter stats for './inverse' (200 runs):

 362.594        cache-references:u                            ( +-  5,41% )
 88.853         cache-misses:u     #24,505 % of all cache refs  ( +-  4,69% )

 3,455488269 seconds time elapsed                             ( +-  0,29% )

===========================================================================
Performance counter stats for './inverse' (200 runs):

 3.764.591.190  branches:u                                    ( +-  0,00% )
 8.748.621      branch-misses:u #0,23% of all branches        ( +-  0,81% )

 3,453739895 seconds time elapsed                             ( +-  0,19% )
===========================================================================
```

Figure 2: `perf` results

Call graph

The workflow of the program can be observerd by the compact call graph reported in Figure 3. As already mentioned, the `main` primarily calculate the determinant of the matrix, and, due to the fact that `determinant` is a recursive function, in the graph appears `determinant'2` that means its second level of recursion. Once proved the non singularity of the matrix, the `main` invoked the function to calculate the cofactor, that in turn, exploit the `determinant` function. Finally the `transpose` function is invoked and, of course, it uses the `determinant` once again.

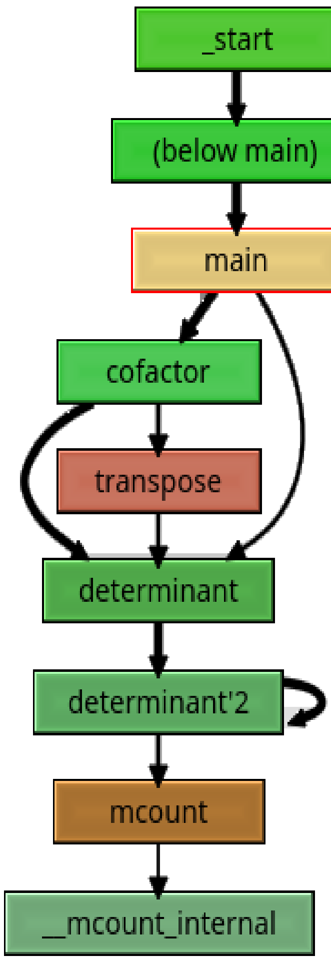
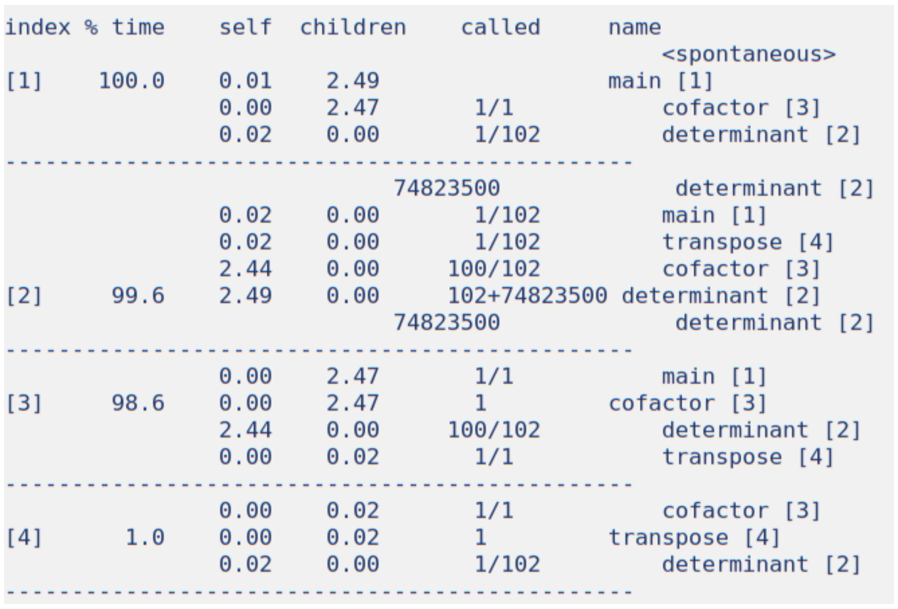


Figure 3: Compact call graph

We can also take a look a the call graph generated by `gprof` and reported in Figure 4. The same information are also reported in the full call graph created with Vallgrind and reported in Figure 5.



```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.01    2.49                 main [1]
                0.00    2.47       1/1           cofactor [3]
                0.02    0.00       1/102         determinant [2]
-----------------------------------------------
                             74823500             determinant [2]
                0.02    0.00       1/102         main [1]
                0.02    0.00       1/102         transpose [4]
                2.44    0.00     100/102         cofactor [3]
[2]     99.6    2.49    0.00     102+74823500 determinant [2]
                             74823500             determinant [2]
-----------------------------------------------
                0.00    2.47       1/1           main [1]
[3]     98.6    0.00    2.47       1         cofactor [3]
                2.44    0.00     100/102         determinant [2]
                0.00    0.02       1/1           transpose [4]
-----------------------------------------------
                0.00    0.02       1/1           cofactor [3]
[4]      1.0    0.00    0.02       1         transpose [4]
                0.02    0.00       1/102         determinant [2]
-----------------------------------------------
```

Figure 4: `gprof` call graph

The table describes the call tree of the program and it is sorted by the total amount of time spent in each function and its children. Here we can see the number of times the function was called. If the function called itself recursively (like `determinant`), the number only includes non-recursive calls, and is followed by a '+' and the number of recursive number.

So in this program the function `cofactor` is invoked only one time by the main and the function `transpose` in invoked only one time by the function `cofactor`. The calls of `determinat` are a little bit more complicated. It is invoked one time by the main in order to check the non-sngularity of the matrix, the it is invoked 100 time by the `cofactor` function and finally one time by the function `transpose`, for a total of 102 calls. Due to the recursive nature of the function, the function `determinant` calls the second level of recursion of itself 920 times, that in turn calls itself 74823500 times.

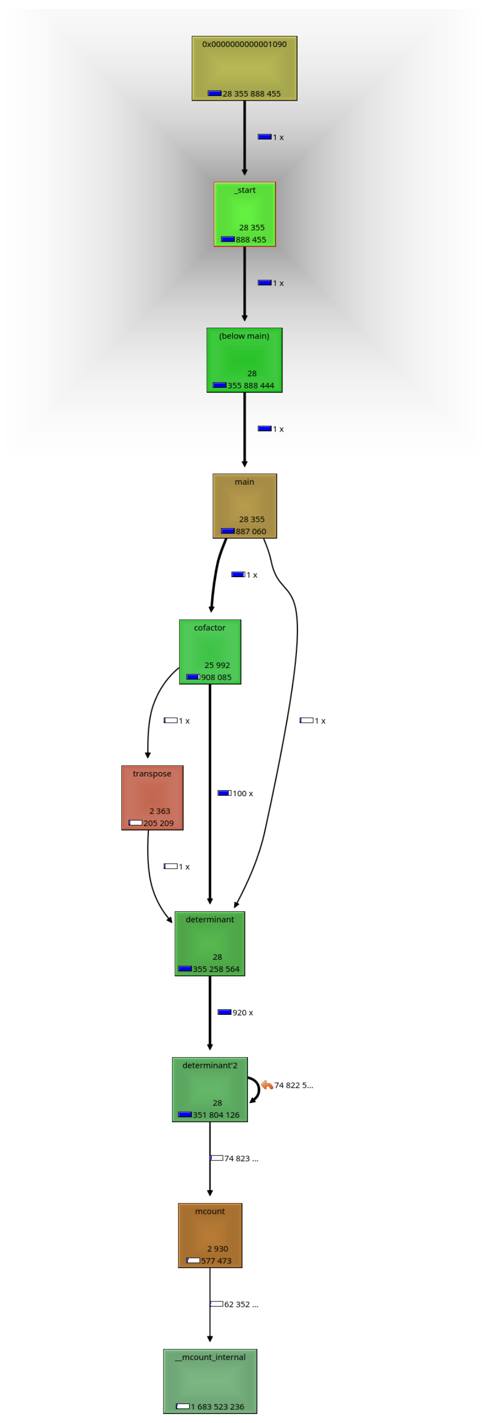


Figure 5: **gprof** call graph