Profiling - A2

Marco Franzon

December, 2018

Abstract

This analysis aim is to profile a code with gprof, perf and valgrind. Differences and similarities are discussed.

1 Introduction

The toy code to be profiled is profile me.c provided by Prof. Luca Tornatore.

```
#include <stdlib.h>
#include <stdio.h>

/* --=={ function prototypes declaration }==--
   ..................................... */

void loop       ( int );
void loop_caller( int );
unsigned long long int factorial  ( int );

/* -- END of funtion declarations  ========= -- */


/*       --=={ function definition }==--
   ..................................... */

unsigned long long int factorial (int n )
/*
  This function calculates the factorial of n
  how large can ben n to get a meaningful result ?
 */
{
  if ( n & 0xFFFFFFFE )
    {
      unsigned long long int f = factorial( n - 1 );
```

```
      return n * f;
    }
  return 1;
}



void loop(int n)
/*
  this function does not do much, actually
 */
{
  int volatile i;  // does not optimize out
  i = 0;
  while(i++ < n)
    ;

  unsigned long long int f;
  for ( i = 0; i < 100000; i++ )
    f = factorial( 20 );
  printf ( "factorial is %llu\n", f );
}



/*
   that is definitely  a phony function
 */
void loop_caller(int n) { loop(n); }

/* -- END of function definitions  ======== -- */



/*              --=={ M A I N }==--
   ..................................................... */

int main( void )
{
  loop_caller(10000000);
  return 0;

}
```

There are three main functions:

- `loop caller()` calls `loop();`

- `loop()` calls `factorial()` 100000 times;
- `factorial()`: calculates the factorial of 20;

The code is compiled in the following way:

```
gcc profile_me.c -o profile_me -fno-omit-frame-pointer -g -pg
```

2 Valgrind

As first profiler I used Valgrind, then callgrind tool to have a call three as an output and the tool cachegrind to check eventual memory leaks or cache thrashing. In Figure 1 you can see that the program spend almost all the time on `factorial()`.

3 Gprof

Thanks to `-pg` flag in compiling, is possible using Gprof. This one, in contrast with Valgrind, run only one time and scan the process giving back an output on the console and one call graph similar to the Valgrind's one. Result can be seen in Figure 2.

In the following piece of output of Gprof is shown the time usage by function in total and per call.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 75.00      0.03     0.03        1    30.00    35.00  loop
 12.50      0.04     0.01   100000     0.00     0.00  factorial
 12.50      0.04     0.01                             frame_dummy
  0.00      0.04     0.00        1     0.00    35.00  loop_caller
```

4 Perf

Perf is a tool provided by Linux in order to keep track of the event on the hardware. We launched the tool using the command:

```
perf stat -a -r 100 -e instructions -e branches -e branch-misses
 -e cache-references -e cache-misses -e L1-dcache-loads -e L1-dcache-stores
 -e L1-dcache-misses -e L1-icache-misses ./profile_me
```

where `-r` is used to run 100 iterations of the toy program in order to avoid fluctuations in the results. The events we were interested in are picked using the `-e` flag, in this case instructions, branch and cache L1D and L1I misses and stores. In Figure 3 is presented the output.

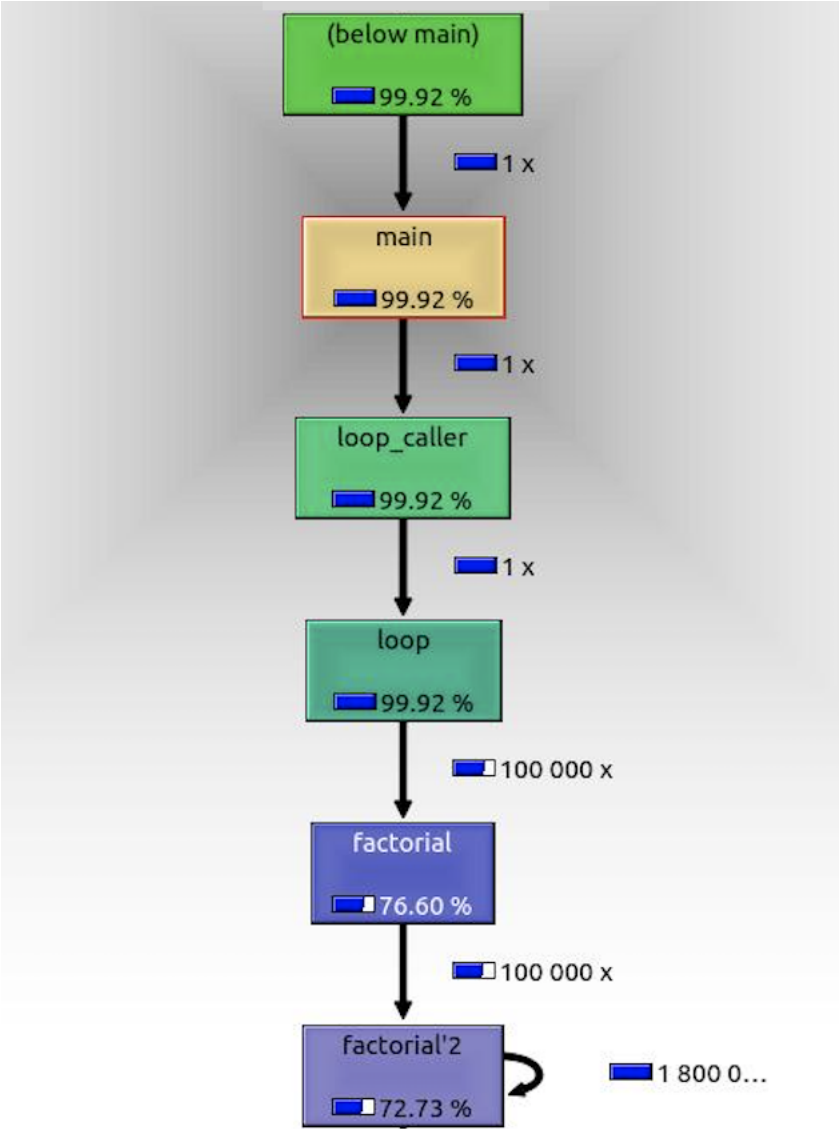

Figure 1: callgraph produced by callgrind, visualized with kcachegrind. From this graph you can see that, as one can expect, the majority of the time is spent in calling the `factorial()`

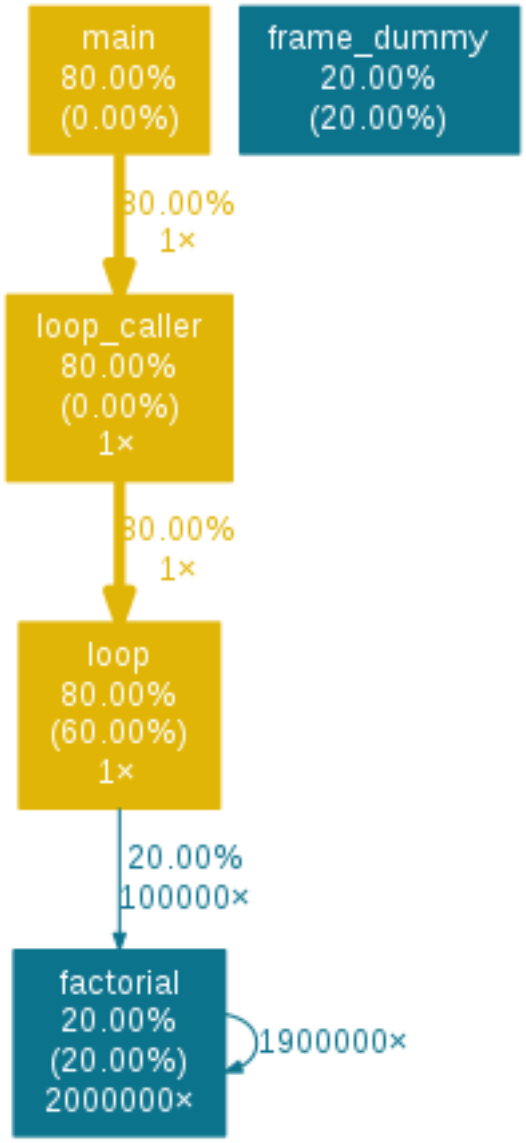

Figure 2: compiling with Gprof produce an output that can be used to create a png file containing the callgraph thanks to `gprof2dot` command.

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

     275.459.112      cpu-cycles:u                                                  (21,04%)
     217.034.799      instructions:u            #    0,79  insns per cycle         (21,85%)
              20      cache-misses:u                                                (19,73%)
      42.140.953      branches:u                                                    (20,93%)
         634.513      branch-misses:u           #    1,51% of all branches         (20,33%)

     0,146101830 seconds time elapsed
```

Figure 3: Output of the perf tool for cache chacking.