Systems Programming Laboratory, Spring 2022

Introduction to gprof

Abhijit Das
Arobinda Gupta

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

February 8, 2022

Profiling

- Debugging helps you remove implementation and logical bugs.
- You need a profiler to monitor the performance of your program.
- gprof is a profiler that helps you achieve that.
- gprof measures the relative performance of the functions in your program.
- The performance of a function in a program may be poor for two reasons.
  - Each invocation of the function takes too much time.
  - The function is called too many times.
- gprof helps you detect both.
  - The flat profile gives detailed data on the running times of functions.
  - The call graph generated by gprof tells which functions call which functions, and how many times.

How to run gprof

- First, compile your code with the –pg option.

  ```
  gcc -Wall -pg myprog.c
  ```

  This generates an executable file (it is `a.out` without the option –o).

- Then, you run the executable with the command-line parameters (if any).

  ```
  ./a.out
  ```

  This creates a profile-data file with the default name `gmon.out`.

- Finally, call gprof with the executable file name and the profile-data file. If the data file has the default name, you can omit it.

  ```
  gprof ./a.out gmon.out
  ```

- You get a long output showing the following:
  - The flat profile (timing profile).
  - The call graph.
  - A detailed instruction on how to interpret the above two tables.

Some options for calling gprof

- –b Compact output (without the interpretation instructions)
- –p Print only the flat profile
- –P Do not print the flat profile
- –pfname Print the flat profile of only the function fname
- –q Print only the call graph
- –Q Do not print the call graph
- –z Print the information of all functions (even if not called and/or taking zero time)
- –l Make line-by-line profiling (compile with –g and –pg). Works with old gcc versions. Use gcov instead.

Timing (or flat) profile

- A listing of the functions in your program with profiling information.
- The summary for each function shows the contributions of all invocations of the function.
  - **% time**: The percentage of time spent by the program while it was in that function (excluding the time spent in other function calls, if any, made from this function).
  - **Self time:** The time spent inside this function (excluding times spent in caller and called functions). The listing is sorted in the decreasing order of these times.
  - **Cumulative time:** The total self time spent by this function plus the self times of the functions appearing above this function in the table.
  - **Calls:** The number of times the function is called.
  - **Average self time per call:** This is the self time divided by the number of calls, in s (seconds), ms (milliseconds), us (microseconds), or ns (nanoseconds).
  - **Average total time per call:** Self time plus the time spent in other function calls made from this function, again in s, ms, us, or ns.

Limitations of gprof

- The estimates furnished by gprof are not fully accurate.
- gprof samples the execution of the program every 0.01 second (usually).
- Based on the samples, gprof makes a rough statistical analysis.
- You need to give gprof some time for gathering sufficiently many samples to make meaningful estimates. Your program should run for at least a few seconds.
- You cannot change the default sampling rate.
- The % estimates should add up to 100, but it is usually not the case. The sum may be less than or even larger than 100.
- Functions that are not called or that miss the samples are not listed (use the –z option to list all).
- Sometimes you will see functions (like **`frame_dummy`**) not in your program. These functions are called by the runtime system and should account for a very small percentage of the total time.
- gprof handles function-level profiling only. For line-by-line profiling, use gcov.

A sample output

```
$ gprof -b -p -z ./a.out
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 81.05      0.58     0.58 93324100     6.25     6.25  nextnum
 12.69      0.67     0.09 10000000     9.13    61.24  ishappy
  4.23      0.71     0.03                             main
  0.70      0.71     0.01                             frame_dummy
  0.00      0.71     0.00                             __do_global_dtors_aux
  0.00      0.71     0.00                             __gmon_start__
  0.00      0.71     0.00                             __libc_csu_fini
  0.00      0.71     0.00                             __libc_csu_init
  0.00      0.71     0.00                             _dl_relocate_static_pie
  0.00      0.71     0.00                             _fini
  0.00      0.71     0.00                             _init
  0.00      0.71     0.00                             _start
  0.00      0.71     0.00                             atexit
  0.00      0.71     0.00                             data_start
  0.00      0.71     0.00                             deregister_tm_clones
  0.00      0.71     0.00                             etext
  0.00      0.71     0.00                             register_tm_clones
$
```

Happy numbers

- Let n be a positive integer.
- Keep on replacing n by the sum of the squares of the (decimal) digits of n.
- If n eventually reduces to 1, then the initial n (and all the intermediate values of n generated in the process) is (are) happy.
- Otherwise, the sequence eventually becomes periodic and keeps on looping without ever reaching 1. These numbers are unhappy or sad.
- 2022 is unhappy: $2022 \to 2^2+0^2+2^2+2^2 = 12 \to 1^2+2^2 = 5 \to 25 \to 29 \to 85 \to 89 \to 145 \to 42 \to 20 \to 4 \to 16 \to 37 \to 58 \to 89$.
- 2026 is happy: $2026 \to 2^2+0^2+2^2+6^2 = 44 \to 4^2+4^2 = 32 \to 13 \to 10 \to 1$.
- Goal: To write an efficient function for checking whether a number is happy or not.
- We check its performance by calling it for all n in the range $[1, 100000]$.

The functions

ishappy(n) Returns 1 if n is happy, 0 if not.

nextnum(n) returns the sum of the squares of the digits of n.

init(n) A data structure is initialized to record that no number is generated in the sequence.

isvisited(A, n) Check whether n is already generated in the sequence.

markvisited(A, n) Mark in A that n is visited in the sequence.

main() This calls ishappy(n) for all n in the range $1 \leqslant n \leqslant 10^5$, and prints n if and only if the call returns 1.

Implementation of ishappy(n)

```
A = init(n);
markvisited(A,n);
while (1) {
   n = nextnum(n);
   if (!isvisited(A,n)) { markvisited(A,n); continue; }
   if (n == 1) return 1; else return 0;
}
```

The first attempt

- A is an array of size $n+1$ (or 200 if $n < 100$).
- init: Set all the cells $A[i] = 0$.
- isvisited(A,n): Just check whether $A[n] = 1$.
- markvisited(A,n): Set $A[n] = 1$.

Output of gprof

% time	cumulative seconds	self seconds	calls	self us/call	total us/call	name
99.15	9.32	9.32	100000	93.20	93.20	init
0.11	9.33	0.01	1246773	0.01	0.01	isvisited
0.00	9.33	0.00	1246773	0.00	0.00	markvisited
0.00	9.33	0.00	1246773	0.00	0.00	nextnum
0.00	9.33	0.00	100000	0.00	93.30	ishappy

The second attempt

- If $n \geqslant 100$, then nextnum$(n) < n$.
- If $n < 100$, then nextnum$(n) \leqslant 9^2 + 9^2 = 162$.
- For all 32-bit integers, nextnum$(n) \leqslant 3^2 + 9 \times 9^2 = 738$.
- For $n < 100$, we take A of size 200.
- For $n \geqslant 100$, we take A of size 1000.
- In ishappy(n), first replace n by nextnum(n) once, and then proceed as before.

Output of gprof

```
  %    cumulative   self              self     total
 time    seconds   seconds    calls  us/call  us/call  name
90.17       1.51      1.51  1000000     1.51     1.51  init
 4.84       1.59      0.08 12469340     0.01     0.01  nextnum
 3.63       1.65      0.06  1000000     0.06     1.69  ishappy
 1.82       1.68      0.03 11469340     0.00     0.00  markvisited
 0.61       1.69      0.01 11469340     0.00     0.00  isvisited
```

The third attempt

- We use a dictionary to store the numbers already generated in the sequence.
- A is now an array of size 1000. No need to initialize every cell of A.
- markvisited() appends to A each new number generated in the sequence. We also externally store how many integers are saved in A.
- A is not necessarily sorted. So isvisited() makes a linear search in A.
- For a sequence of a few tens of numbers, more sophisticated data structures may fail to produce better results.

Output of gprof

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 50.53      0.13     0.13 12469250    10.54    10.54  isvisited
 23.32      0.19     0.06 12469250     4.86     4.86  nextnum
 11.66      0.22     0.03  1000000    30.32   247.62  ishappy
  7.77      0.24     0.02  1000000    20.21    20.21  init
  3.89      0.25     0.01                             main
  1.94      0.26     0.01 12469250     0.41     0.41  markvisited
```

The fourth attempt

- Algorithmic improvement suggested by your Discrete-Maths professor.
- Every happy number reduces to 1.
- Every unhappy number ends up in the cycle containing 4.
- No need to maintain any data structure A to store the numbers generated in the sequence.
- ishappy() keeps on replacing n by nextnum(n) until n becomes 1 or 4.

Output of gprof

% time	cumulative seconds	self seconds	calls	self ns/call	total ns/call	name
82.27	0.54	0.54	93324100	5.82	5.82	nextnum
15.38	0.64	0.10	10000000	10.15	58.63	ishappy
1.54	0.65	0.01				main
0.77	0.66	0.01				frame_dummy

Call graphs

- The records for each function are delimited by a line consisting of dashes.
- The line starting with [index number] is the primary line for a function.
- Above the primary line appears a listing of all caller function. If there are no caller functions, a line containing `<spontaneous>` is printed.
- Below the primary line appears a listing of all called function.
- Each line gives information % time spent in that function, time spent inside that function, time spent inside the called functions, and call count(s).
- The primary line has a single call count if it is a non-recursive function. If it makes recursive calls to itself, two numbers appear as count1+count2, where count1 is the number of non-recursive calls, and count2 is the number of recursive calls.
- For a caller or called function, there are two call counts count1/count2 indicating that count1 calls in a total count2 calls are associated with the function in the primary line.

Understanding the call counts

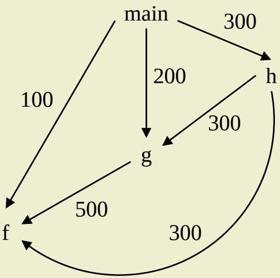
```
void f ()
{
}

void g() {
   f();
}

void h()
{
   f();
   g();
}

int main ()
{
   int i;

   for (i=0;i<100;++i) f();
   for (i=0;i<200;++i) g();
   for (i=0;i<300;++i) h();
}
```




```
index % time    self  children    called     name
                0.00    0.00    100/900         main [9]
                0.00    0.00    300/900         h [3]
                0.00    0.00    500/900         g [2]
[1]       0.0   0.00    0.00    900         f [1]
-----------------------------------------------
                0.00    0.00    200/500         main [9]
                0.00    0.00    300/500         h [3]
[2]       0.0   0.00    0.00    500         g [2]
                0.00    0.00    500/900         f [1]
-----------------------------------------------
                0.00    0.00    300/300         main [9]
[3]       0.0   0.00    0.00    300         h [3]
                0.00    0.00    300/900         f [1]
                0.00    0.00    300/500         g [2]
-----------------------------------------------
...
-----------------------------------------------
                                             <spontaneous>
[9]       0.0   0.00    0.00                main [9]
                0.00    0.00    300/300         h [3]
                0.00    0.00    200/500         g [2]
                0.00    0.00    100/900         f [1]
-----------------------------------------------
```

Call graph with timing: Happy numbers (third attempt)

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.01    0.25                 main [1]
                0.03    0.22 1000000/1000000     ishappy [2]
-----------------------------------------------
                0.03    0.22 1000000/1000000     main [1]
[2]     96.1    0.03    0.22 1000000         ishappy [2]
                0.13    0.00 12469250/12469250     isvisited [3]
                0.06    0.00 12469250/12469250     nextnum [4]
                0.02    0.00 1000000/1000000     init [5]
                0.01    0.00 12469250/12469250     markvisited [6]
-----------------------------------------------
                0.13    0.00 12469250/12469250     ishappy [2]
[3]     51.0    0.13    0.00 12469250          isvisited [3]
-----------------------------------------------
                0.06    0.00 12469250/12469250     ishappy [2]
[4]     23.5    0.06    0.00 12469250          nextnum [4]
-----------------------------------------------
                0.02    0.00 1000000/1000000     ishappy [2]
[5]      7.8    0.02    0.00 1000000         init [5]
-----------------------------------------------
                0.01    0.00 12469250/12469250     ishappy [2]
[6]      2.0    0.01    0.00 12469250          markvisited [6]
-----------------------------------------------
...
Index by function name

   [5] init                    [3] isvisited               [6] markvisited
   [2] ishappy                 [1] main                    [4] nextnum
```

Recursive call graph: Fibonacci numbers

- We use the following recursive implementation.

```
int Fib ( int n )
{
   if (n < 0) return -1;
   if (n == 0) return 0;
   if (n == 1) return 1;
   return Fib(n-1) + Fib(n-2);
}
```

- From main(), we call Fib(32).

Call graph by gprof

```
index % time    self  children    called     name
                              7049154             Fib [1]
                0.01    0.00       1/1            main [2]
[1]     100.0   0.01    0.00       1+7049154  Fib [1]
                              7049154             Fib [1]
-----------------------------------------------
                                                  <spontaneous>
[2]     100.0   0.00    0.01                  main [2]
                0.01    0.00       1/1            Fib [1]
-----------------------------------------------
```

Recursive call graph: Fibonacci numbers with memoization

- In main(), we initialize each element of an array $F[0 \ldots n]$ to -1.
- We pass F alongside n to Fib.
- In Fib(n, F), we first check if $F[n] \geqslant 0$. If so, we return this value.
- Otherwise, we set $F[n]$ (direct assignment for $n = 0, 1$, recursive calls for $n \geqslant 2$), and return $F[n]$.
- The main function calls Fib(32,F).

Call graph by gprof

```
index % time      self  children      called      name
                                      62              Fib [1]
                  0.00    0.00         1/1            main [7]
[1]        0.0    0.00    0.00         1+62       Fib [1]
                                      62              Fib [1]
-----------------------------------------------
                                                      <spontaneous>
[7]        0.0    0.00    0.00                    main [7]
                  0.00    0.00         1/1            Fib [1]
-----------------------------------------------
```