gprof: Performance Profiling Guide

1. Introduction and Basic Usage

What is gprof?

Purpose:[81] - gprof is a profiler that monitors performance of your program - Measures relative performance of functions - Helps detect performance bottlenecks

Why Profile?[81] A function's performance may be poor for two reasons: 1. Each invocation takes too much time 2. The function is called too many times

Three-Step Workflow

Step 1: Compile with -pg flag[81]

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

Creates executable with profiling instrumentation

Step 2: Run the program[81]

```
$ ./myprog
```

Executes normally and creates `gmon.out` profile data file

Step 3: Analyze with gprof[81]

```
$ gprof ./myprog
```

Displays flat profile and call graph

---

2. Understanding Profiling Output

Flat Profile (Timing Profile)

Purpose:[81] Lists functions with detailed profiling information on running times

Example Output:[81]

```
$ 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
```

Column Meanings:[81]

Column	Meaning	Example
% time	Percentage of time in this function (excluding called functions)	81.05%
Self seconds	Time spent inside this function only	0.58s
Cumulative seconds	Running total of self times	0.67s
Calls	Number of times function called	93324100
Self ns/call	Average self time per call in nanoseconds	6.25 ns
Total ns/call	Self time plus called functions per invocation	6.25 ns
Name	Function name	nextnum

Call Graph

Purpose:[81] Shows which functions call which functions and call counts

Example Output:[81]

```
index % time  self    children  called      name
                      <spontaneous>
[1]   100.0  0.01    0.25                  main [1]
                      1000000/1000000
                      ishappy [2]

                      1000000/1000000
[2]   96.1   0.03    0.22      1000000     ishappy [2]
                      12469250/12469250
                      isvisited [3]
                      12469250/12469250
                      nextnum [4]
```

Reading Call Graph:[81] - [index] is function index - Primary line shows total calls - Above: caller functions - Below: called functions - Format `count1/count2` shows count1 calls out of total count2

3. gprof Options and Commands

Compilation and Execution

Compile with profiling:[81]

```
$ gcc -Wall -pg myprog.c              # Creates a.out
$ gcc -Wall -pg -o myprog myprog.c    # Creates myprog
```

Run program with arguments:[81]

```
$ ./a.out arg1 arg2 arg3
```

Display Options

Basic analysis:[81]

```
$ gprof ./a.out                 # Full output (flat + call graph)
$ gprof ./a.out gmon.out        # Explicit profile file
```

Option: -b (Compact output)[81]

```
$ gprof -b ./a.out
```

Removes explanatory text, shows only data

Option: -p (Flat profile only)[81]

```
$ gprof -p ./a.out
```

Option: -q (Call graph only)[81]

```
$ gprof -q ./a.out
```

Option: -pfunctionname (Specific function)[81]

```
$ gprof -pnextnum ./a.out
```

Shows flat profile for `nextnum` function only

Option: -z (Include all functions)[81]

```
$ gprof -z ./a.out
```

Includes functions with zero time and system functions

Combined options:[81]

```
$ gprof -b -p -z ./a.out           # Compact flat profile with all functions
$ gprof -b -pfishappy -z ./a.out   # Compact profile of specific function
```

---

4. Case Studies

Case Study 1: Happy Numbers

Problem:[81] Check if numbers are happy (repeatedly sum squares of digits until reaching 1 or cycle)

Examples:[81] - 2026 is happy: 2026 → 44 → 32 → 13 → 10 → 1 - 2024 is unhappy: 2024 → 24 → 20 → 4 → 16 → 37 → 58 → 89 → 145 → 42 → 20 (cycle)

Four Optimization Attempts:

Attempt 1: Array initialization bottleneck[81]

```
$ gprof -b -p -z ./a.out
%        cumulative    self           self      total
time     seconds       seconds  calls  us/call   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
```

Problem: init() takes 99.15% initializing large array for each call

Attempt 2: Smaller arrays (math optimization)[81]

```
%        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
```

Improvement: 9.32s → 1.69s (5.5× faster) Problem: init() still 90.17%

Attempt 3: Dictionary approach[81]

```
%        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
```

Improvement: 1.69s → 0.26s (6.5× faster) Problem: isvisited() now bottleneck at 50.53%

Attempt 4: Algorithmic breakthrough[81]

```
%        cumulative    self           self      total
time     seconds       seconds  calls  ns/call   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
```

Key insight: Happy numbers → 1, unhappy numbers → cycle containing 4 No data structure needed, just check if reaches 1 or 4

Case Study 2: Recursive Fibonacci

Naive Recursion:[81]

```
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);
}
// Call: Fib(32)
```

Call graph output:[81]

```
index % time   self     children  called            name
                        7049154
[1]    100.0   0.01     0.00      1+7049154         Fib [1]
```

Calls: 1 + 7,049,154 recursive calls

With Memoization:[81]

```
int Fib(int n, int F[]) {
    if (F[n] >= 0) return F[n];
    if (n == 0) F[n] = 0;
    else if (n == 1) F[n] = 1;
    else F[n] = Fib(n-1, F) + Fib(n-2, F);
    return F[n];
}
// Call: Fib(32, F)
```

Call graph output:[81]

```
index % time   self     children  called        name
                        62
[1]    0.0     0.00     0.00      1+62          Fib [1]
```

Calls: 1 + 62 recursive calls Reduction: 7,049,154 → 62 calls (113,373× improvement!)

---

5. Limitations and Important Notes

Sampling-Based Approach[81]

How sampling works: - gprof samples execution every 0.01 seconds (default) - Based on samples, makes statistical analysis - Percentages are estimates, not exact

Accuracy Requirements:[81] - Program must run for at least a few seconds for meaningful results - Insufficient samples lead to inaccurate estimates - Sampling rate cannot be changed

Percentage Limitations:[81] - Percentages may not sum to exactly 100% - Sum may be less than or even larger than 100% - Normal limitation of sampling-based profiling

Functions in Output[81]

Functions not listed: - Functions not called during profiling - Missed all samples - Use `-z` option to include them

Unexpected system functions:[81] - Functions like `frame_dummy`, `__do_global_dtors_aux` - Called by runtime system - Usually account for small percentage

Call Count Notation:[81] - Regular: Single call count - Recursive: `count1+count2` format (count1=non-recursive, count2=recursive) - Caller/called lines: `count1/count2` format

Profiling Limitations[81]

Function-level only: - gprof handles function-level profiling - For line-by-line profiling, use `gcov`

No line-by-line in modern systems:[81] - Line-by-line profiling option `-l` works with old gcc - Recommended to use `gcov` for modern systems