Μεταγλωττιστές για Ενσωματωμένα Συστήματα

Χειμερινό Εξάμηνο 2023-24
«Debugging and Profiling»

Παναγιώτης Χατζηδούκας

Outline

- GDB
- Measuring execution time
- GNU gprof
- Perf
- Valgrind and Cachegrind
- PAPI

gdb (GNU Debugger)

- “GNU Debugger”
- A debugger for several languages, including C and C++
- It allows you to inspect what the program is doing at a certain point during execution.
- Errors like segmentation faults may be easier to find with the help of gdb.

gdb (GNU Debugger)

- Debuggers are programs which allow you to execute your program in a controlled manner, so you can look inside your program to find a bug.
- gdb is a reasonably sophisticated text based debugger. It can let you:
  - Start your program, specifying anything that might affect its behavior.
  - Make your program stop on specified conditions.
  - Examine what has happened, when your program has stopped.
  - Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.
- SYNOPSIS

```
gdb  [prog] [core|procID]
```

gdb

- GDB is invoked with the shell command gdb.
- Once started,it reads commands from the terminal until you tell it to exit with the GDB command quit.
  - The most usual way to start GDB is with one argument or two, specifying an executable program as the argument:

    ```
    $ gdb program
    ```

  - You can also start with both an executable program and a core file specified:

    ```
    $ gdb program core
    ```

  - You can, instead, specify a process ID as a second argument, if you want to debug a running process:

    ```
    $ gdb program 1234
    ```

    would attach GDB to process 1234

Compiling with the –g Option

- To use gdb best, compile your program with:

```
gcc –g –c my_math.c
gcc –g –c sample.c
gcc –o sample my_math.o sample.o
```

  or:

```
gcc –o sample -g my_math.c sample.c
```

- That is, you should make sure that –g option is used to generate the .o files.
  - This option tells the compiler to insert more information about data types, etc., so the debugger gets a better understanding of it.

Common Commands for gdb

- Here are some of the most frequently needed GDB commands:

b(reak) [file:]function	Set a breakpoint at function (in file).
r(un) [arglist]	Start program (with arglist, if specified).
bt or where	Backtrace: display the program stack; especially useful to find where your program crashed or dumped core.
print expr	Display the value of an expression.
c	Continue running your program (after stopping, e.g. at a breakpoint).
n(ext)	Execute next program line (after stopping); step over any function calls in the line.
s(tep)	Execute next program line (after stopping); step into any function calls in the line.
help [name]	Show information about GDB command name, or general information about using GDB.
q(uit)	Exit from GDB.
l(ist)	print the source code

Starting up gdb

- Just try “gdb” or “gdb prog1.x.” You’ll get a prompt that looks like this:
- (gdb)
- If you didn’t specify a program to debug, you’ll have to load it in now:
- (gdb) file prog1.x
- Here, prog1.x is the program you want to load, and “file” is the command to load it.

Getting help

- gdb has an interactive shell, much like the one you use as soon as you log into the linux grace machines. It can recall history with the arrow keys, auto-complete words (most of the time) with the TAB key, and has other nice features.

- If you're ever confused about a command or just want more information, use the "help" command, with or without an argument:
- (gdb) help [command]
- You should get a nice description and maybe some more useful hints

Running

- To run the program, just use:
- (gdb) run
- This runs the program.
  - If it has no serious problems (i.e. the normal program didn't get a segmentation fault, etc.), the program should run fine here too.
  - If the program had issues, then you (should) get some useful information like the line number where it crashed, and the parameters passed to the function that caused the error.

-
  ```
  Program received signal SIGSEGV, Segmentation fault.
  0x0000000000400524 in sum array region
  (arr=0x7fffc902a270, r1=2, c1=5, r2=4, c2=6) at sum-
  array-region2.c:12
  ```

Presence of bugs

- Okay, so you’ve run it successfully. But you don’t need gdb for that. What if the program isn’t working?
- Basic idea:
  - Chances are if this is the case, you don’t want to run the program without any stopping, breaking, etc. Otherwise, you’ll just rush past the error and never find the root of the issue. So, you’ll want to step through your code a bit at a time, until you arrive upon the error.
- This brings us to the next set of commands. . .

Setting breakpoints

- Breakpoints can be used to stop the program run in the middle, at a designated point. The simplest way is the command "break." This sets a breakpoint at a specified file-line pair:

```
(gdb) break file1.c:6
```

- This sets a breakpoint at line 6, of file1.c. Now, if the program ever reaches that location when running, the program will pause and prompt you for another command.

- You can set as many breakpoints as you want, and the program should stop execution if it reaches any of them.

Breakpoints

- You can also tell gdb to break at a particular function. Suppose you have a function my func:

```
int my func(int a, char *b);
```

- You can break anytime this function is called:

```
(gdb) break my func
```

continue and step

- Once you've set a breakpoint, you can try using the run command again. This time, it should stop where you tell it to (unless a fatal error occurs before reaching that point).
- You can proceed onto the next breakpoint by typing "continue" (Typing run again would restart the program from the beginning, which isn't very useful.)

```
(gdb) continue
```

- You can single-step (execute just the next line of code) by typing "step." This gives you really fine-grained control over how the program proceeds. You can do this a lot...

```
(gdb) step
```

Next command

- Similar to "step," the "next" command single-steps as well, except this one doesn't execute each line of a sub-routine, it just treats it as one instruction.
- (gdb) next

- Typing "step" or "next" a lot of times can be tedious. If you just press ENTER, gdb will repeat the same command you just gave it.
- You can do this a bunch of times

Querying other aspects of the program

- So far you've learned how to interrupt program flow at fixed, specified points, and how to continue stepping line-by-line. However, sooner or later you're going to want to see things like the values of variables, etc. This might be useful in debugging.
- The print command prints the value of the variable specified, and print/x prints the value in hexadecimal:

```
(gdb) print my var
(gdb) print/x my var
```

Setting watchpoints

- Whereas breakpoints interrupt the program at a particular line or function, watchpoints act on variables. They pause the program whenever a watched variable's value is modified. For example, the following watch command:

```
(gdb) watch my_var
```

- Now, whenever my_var's value is modified, the program will interrupt and print out the old and new values.

Other useful commands

- backtrace - produces a stack trace of the function calls that lead to a seg fault
- where - same as backtrace; you can think of this version as working even when you’re still in the middle of the program
- finish - runs until the current function is finished
- delete - deletes a specified breakpoint
- info breakpoints - shows information about all declared breakpoints

Remote debugging

- Remote debugging is the process of debugging a program running on a different system (called target) from a different system (called host).
  - a debugger running on host machine connects to a program which is running on the target via network.
  - The debugger in the host can then control the execution of the program on the remote system and retrieve information about its state.
- Remote debugging is often useful in case of embedded applications where the resources are limited.

GNU gdbserver

- We need the following two utilities to perform a remote debugging.
  - gdbserver – Run this on your target system
  - GDB – Execute this on your host system to connect to your target system

Target	Host
$ gdbserver localhost:2000 my_prg Process program created; pid = 2045 Listening on port 2000	$ gdb my_prg (gdb) (gdb) target remote 192.168.1.10:2000

How useful are debuggers?

- Debuggers can be great for seeing how small programs execute
- Great for certain types of problems
  - Identifying the line on which the program crashes
  - Seeing state of procedure stack at crash
- Less useful for non-crashing programs
- Disadvantages of debuggers
  - Not available on some systems
  - System-dependent user interface
  - Too many low-level details
    - Try debugging linked lists
  - Clicking over statements is tedious
  - Deal poorly with large amounts of data
  - Difficult to find intermittent bugs
- Debuggers are an important tool, but not the only, or even most important one for debugging programs

How useful are debuggers?

- Often more useful to dump information to screen
  - Easy to scan large amount of data
  - Only output relevant data
  - Debugging code can be left in program
    - Use #ifdef's and #defines to comment it out
    - Or just plain if statements, if you trust the compiler
    - Debugging sessions are transient

- Difficult bugs to find are ones where the program works for most of the input data, but some small part causes problems
- Bug may depend on combination of conditions
- Looking at a screen-full of data can allow you to reason backward

Backward Reasoning

- Debugging is the process of reasoning backwards from the undesired behavior (bad output, spectacular crash, running forever) to the cause of the behavior
- Debugging is backward reasoning
- Like solving murder mysteries
  - We have a dead body. How were they killed?
- Bugs show us a gap between how we think the program behaves and how it behaves
- Once we have identified the reason for the problem, fixing it is *usually* relatively easy
  - But sometimes bugs reveal fundamental flaws in our whole design

Some debugging tips

- Look for familiar patterns
  - int n;
    scanf(“%d”, n);

  - if (x = y)

  - double d = 3.14159265357;
    int i = 1;
    printf(“%d %f”, d, i);

  - You will make these and similar mistakes time and time again

Some debugging tips

- Examine the most recent changes
  - You should write your program incrementally
  - If a bug appears, it should be because of a recent change
  - Finding older bugs is very difficult
  - You should never write a big bunch of code and then try to debug it into workingness.
  - Need to test your program continuously as you develop it.
- Aside:
  - Extreme programming (XP) is a technique for small groups developing software
  - One aspect of XP is developing test cases in parallel with the code, and extending the code incrementally to pass more and new test cases
  - Continuous retesting as part of the process
    - Basically just (un)common sense

Some debugging tips

- Don’t make the same mistake twice
    - After you fix a bug, ask whether you made the same mistake elsewhere
    - Many bugs arise from a misunderstanding of how the program behaves
- Debug it now, not later
    - Tempting to press ahead if a bug seems unimportant
    - But old bugs are the hardest to find
    - May never find bug until after the software is delivered
- Mars Pathfinder
    - Spacecraft’s computers reset every day
    - Bug tracked down to something seen in testing
    - Engineers had been too busy working on something else
    - Fixing bugs over a radio link to Mars is more difficult
- Also very important for embedded software
    - Usually recalling large numbers of consumer devices because of software bugs is not practical

Some debugging tips

- Get a stack trace
    - Debuggers can be tedious to work with
    - But stack traces are the most useful debugging information when a program crashes spectacularly
- Read and think before typing
    - Debugging is fundamentally about understanding the difference between what you think you asked the computer to do and what you actually asked it to do
    - Get a cup of coffee
- Explain your code to someone else
    - Often end up explaining bug to yourself
    - Consider using cardboard cutout

Difficult Bugs

- Make the bug reproducible
  - Most difficult bugs to fix are those that are difficult to repeat
    - Big reason why concurrent programming is so difficult
  - Find input and parameter settings that cause the bug to appear every time
  - Will need to reproduce bug again and again
  - If bug cannot be made reproducible, that is also information
- Divide and conquer
  - Reduce the size of the input required to trigger bug
  - Make test case as small as possible
  - Use binary search
    - Throw away half the input
    - See which half of the input causes the bug
    - Repeat

Difficult Bugs

- Check real numbers
  - Round numbers in programs rarely appear randomly
  - E.g. program dropping characters apparently at random
    - Discovered one character every 1023 bytes
    - 1023 is suspiciously close to 1024, a very round number
    - Check the program for appearances of constant 1023 or 1024
    - Found 1024-byte buffer
    - The last character being overwritten by ‘\0’ character
  - This bug would be almost impossible to find with a debugger

  - *Always* be suspicious of round numbers

Difficult Bugs

- Display output
  - Write key information to the screen
  - Use %p for pointers
  - Watch out for pointer values that don't look like addresses
    - Small values
    - Odd numbers for non-char pointer
    - Addresses usually fall within a few ranges
  - Write out when program reaches key places
  - Use indentation to make output easier to read
  - Make sure all debugging information goes to stderr
    - stdout is buffered
  - Consider writing a log file
    - or just redirect stderr to a file
    - myprogram 2> logfile

Difficult Bugs

- Write self-checking code
  - If you believe something is true, test that it is true
  - Use C’s assert feature
    - #include <assert.h>
    - assert(p != NULL);
  - Consider writing your own functions to test more complicated conditions
  - Don’t throw away all debugging code after the bug is fixed
  - assert can be turned off without removing code
    - pass –DNDEBUG to compiler
    - but consider leaving assertions or other debugging code even in production versions

Difficult Bugs

- Draw a picture
  - Especially useful for debugging linked data structures
  - Can dump pointer values to the screen
  - Histograms can show statistical anomalies in data
    - Crude ones with ASCII art
- Use tools
  - Use grep, diff, etc to look through large volumes of debugging output.

Performance Analysis

When analyzing the performance of a program, there are some aspects that can be measured/considered:

- Execution time
- CPU utilization
- Memory usage
- Disk usage
- Bandwidth
- Power consumption

Execution Time

Even a simple aspect such as the execution time of a program can be difficult to measure or even define.

By execution time are we measuring:

- Wall clock time: total elapsed time.
- CPU time (or process time): the amount of time that the CPU spends executing that program.

What if we want to measure the execution of only one part of the program?

Measuring Execution Time

There are various approaches for measuring execution time in a program:

Bash `time`:

```
$time       ./myprogram
real        0m0.00s
user        0m0.00s
sys         0m0.00s
```

Measuring Execution Time

Programs to profile:

Row Major

```
// Memory allocation
unsigned int *p = new unsigned int[N*N];
// Initialization
for(int y = 0; y < N; y++) {
          for(int x = 0; x < N; x++) {
                    p[y*N + x] = x*y;
          }
}
// Initialization
unsigned int total = 0;
for(int y = 0; y < N; y++) {
          for(int x = 0; x < N; x++) {
                    total += p[y*N + x];
          }
}
cout << "Total is: " << total << endl;
delete[] p;
```

Column Major

```
// Memory allocation
unsigned int *p = new unsigned int[N*N];
// Initialization
for(int x = 0; x < N; x++) {
          for(int y = 0; y < N; y++) {
                    p[y*N + x] = x*y;
          }
}
// Initialization
unsigned int total = 0;
for(int x = 0; x < N; x++) {
          for(int y = 0; y < N; y++) {
                    total += p[y*N + x];
          }
}
cout << "Total is: " << total << endl;
delete[] p;
```

Measuring Execution Time

Row Major (N=1024)

```
$ g++ row-major.cpp -o row-major
$ time ./row-major
Total is: 3758368528

real            0m0.015s
user            0m0.008s
sys             0m0.004s
```

```
$ g++ row-major.cpp -o row-major -O3
$ time ./row-major
Total is: 3758368528

real            0m0.009s
user            0m0.003s
sys             0m0.004s
```

Column Major (N=1024)

```
$ g++ col-major.cpp -o col-major
$ time ./col-major
Total is: 3758368528

real            0m0.033s
user            0m0.026s
sys             0m0.004s
```

```
$ g++ col-major.cpp -o col-major -O3
$ time ./col-major
Total is: 3758368528

real            0m0.025s
user            0m0.019s
sys             0m0.004s
```

Measuring Execution Time

Row Major (N=8192)

```
$ g++ row-major.cpp -o row-major
$ time ./row-major
Total is: 16777216

real            0m0.601s
user            0m0.465s
sys             0m0.134s
```

```
$ g++ row-major.cpp -o row-major -O3
$ time ./row-major
Total is: 16777216

real            0m0.281s
user            0m0.139s
sys             0m0.140s
```

Column Major (N=8192)

```
$ g++ col-major.cpp -o col-major
$ time ./col-major
Total is: 16777216

real            0m7.579s
user            0m7.435s
sys             0m0.140s
```

```
$ g++ col-major.cpp -o col-major -O3
$ time ./col-major
Total is: 16777216

real            0m2.200s
user            0m2.051s
sys             0m0.147s
```

Measuring Execution Time

The wall clock or real time is the actual elapsed time of your program.

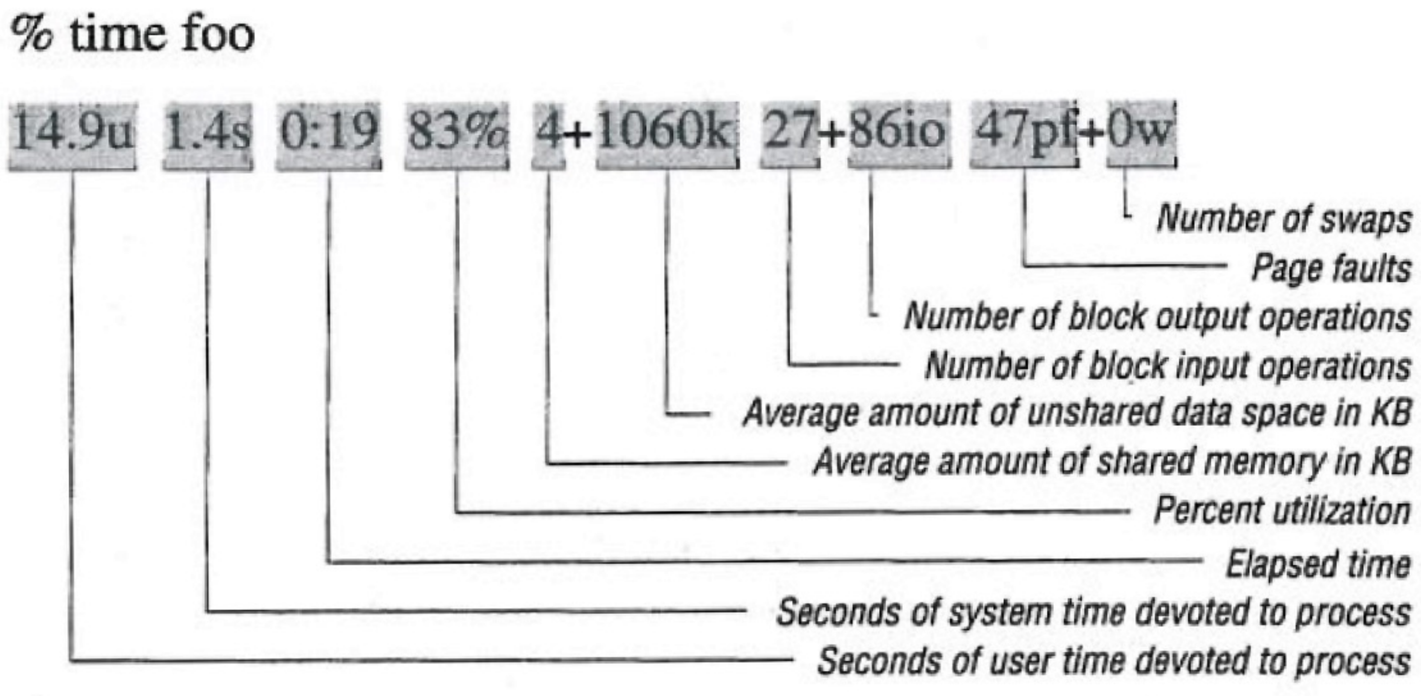
System time is the time spent doing system services.

User time is the time your program was actually running.

$$\text{Real Time} \approx \text{System Time} + \text{User Time}$$

Measuring Execution Time

The `time` utility on csh or tcsh gives somewhat more information:



```
% time foo
14.9u 1.4s 0:19 83% 4+1060k 27+86io 47pf+0w
```

Number of swaps
Page faults
Number of block output operations
Number of block input operations
Average amount of unshared data space in KB
Average amount of shared memory in KB
Percent utilization
Elapsed time
Seconds of system time devoted to process
Seconds of user time devoted to process

Measuring Execution Time

Row Major (N=8192)

```
$ g++ row-major.cpp -o row-major -O3
$ time ./row-major
Total is: 16777216
0.135u 0.140s 0:00.27 100.0%
   0+0k 0+0io 1pf+0w
```

Column Major (N=8192)

```
$ g++ col-major.cpp -o col-major -O3
$ time ./col-major
Total is: 16777216
2.05u 0.137s 0:02.18 100.0%
   0+0k 0+0io 1pf+0w
```

Measuring Time in Code

Execution time can be measured inside code, usually by recording the beginning and end of a code segment and taking the difference.

```
#include <ctime>

int main() {
        clock_t start = clock();
        ...
        clock_t end = clock();
        cout << (end - start) / (double)CLOCKS_PER_SEC << endl;
}
```

```
$ ./row-major
Total is: 16777216
0.26442
```

Measuring Time in Code

Could also use time_t and time. Limited to the number of elapsed seconds.

```
#include <ctime>

int main() {
        time_t start, end;
        time(&start);
        ...
        time(&end);
        cout << difftime(start, end) << endl;
}
```

```
$ ./row-major
Total is: 16777216
0
```

Measuring Time in Code

gettimeofday sets a timeval which contains the current time in tv_sec (seconds) and tv_usec (microseconds).

```
#include <sys/time.h>

int main() {
        timeval start, end;
        gettimeofday(&start);
        ...
        gettimeofday(&end);
        cout <<  (end.tv_sec  - start.tv_sec) +
                                (end.tv_usec - start.tv_usec)/1000000.0 << endl;
}
```

```
$ ./row-major
Total is: 16777216
0.262752
```

Profiling

While manually timing code execution is often useful, there are several profiling tools that can give us more information if we are looking to optimize our programs

Helpful for identifying where our program is spending most of its time to make sure we are actually optimizing the right part of the program.

Hardware Performance Counters

- For many years, hardware engineers have designed specialized registers to measure the performance of various aspects of a microprocessor.
- HW performance counters provide application developers with valuable information about code sections that can be improved.
- Hardware performance counters can provide insight into:
  - Whole program timing
  - Cache behaviors
  - Branch behaviors
  - Memory and resource contention and access patterns
  - Pipeline stalls
  - Floating point efficiency
  - Instructions per cycle
  - Subroutine resolution
  - Process or thread attribution

GNU gprof

- Profilers may show the time elapsed in each function and its descendants
  - number of calls, call-graph (some)
- gprof is an instrumenting profiler for every UNIX-like system

Using gprof

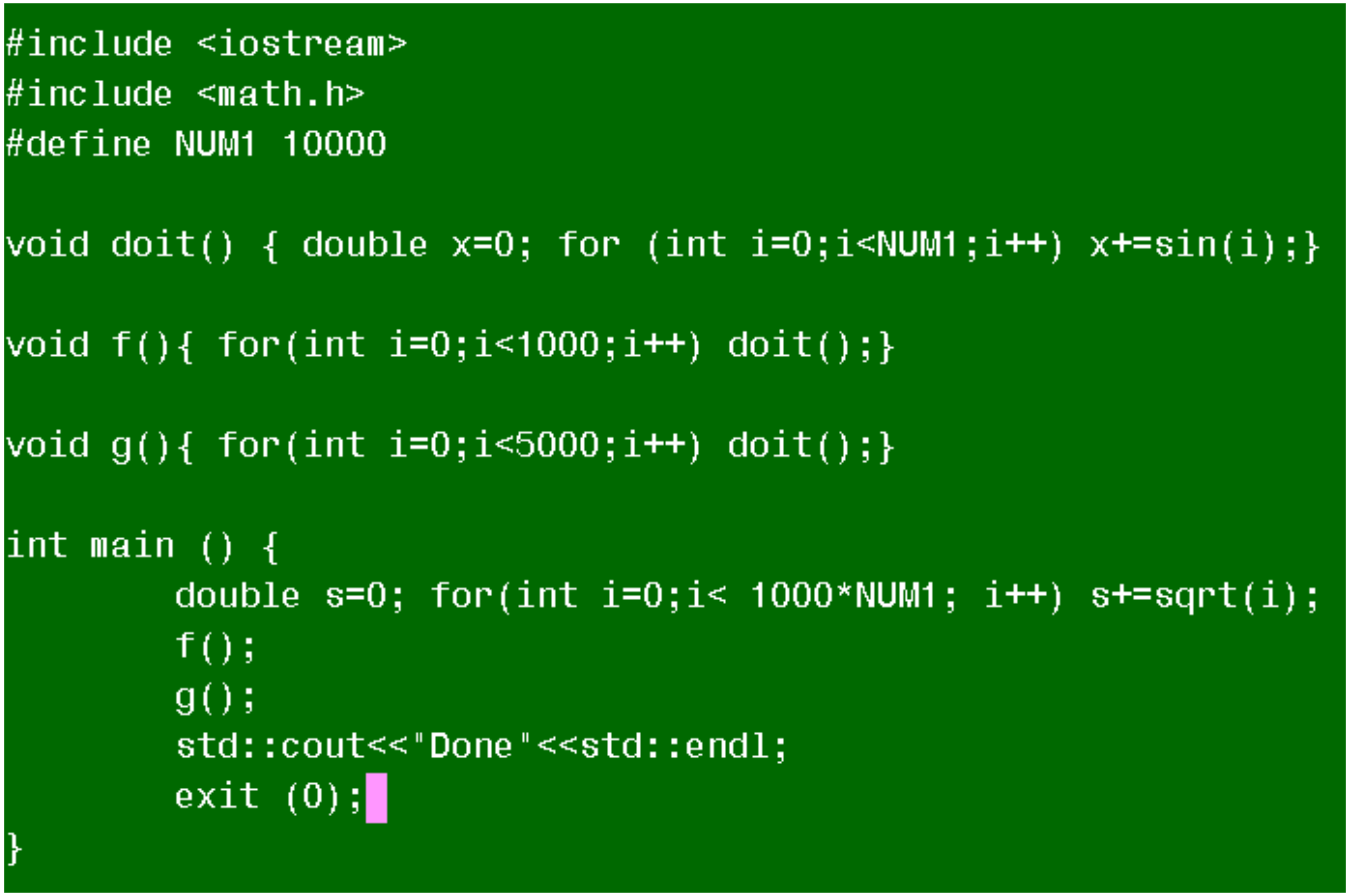
- Compile and link your program with profiling enabled

```
gcc -g -c myprog.c utils.c -pg
gcc -o myprog myprog.o utils.o -pg
```

- Execute your program to generate a profile data file
  - Program will run normally (but slower) and will write the profile data into a file called **gmon.out** just before exiting
  - Program should exit using exit() function

- Run gprof to analyze the profile data
  - `gprof a.out`

Example Program

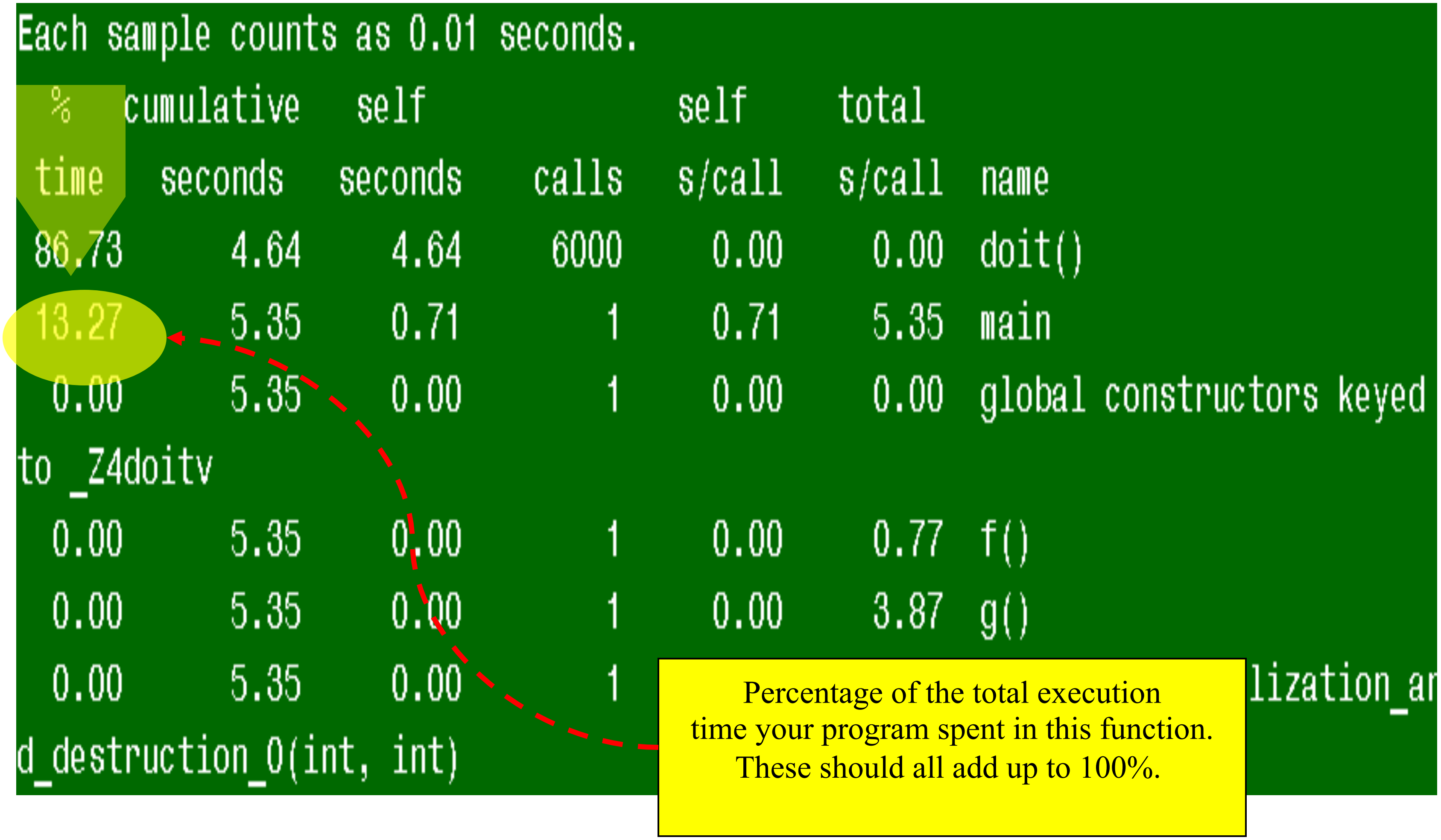


```
#include <iostream>
#include <math.h>
#define NUM1 10000

void doit() { double x=0; for (int i=0;i<NUM1;i++) x+=sin(i);}

void f(){ for(int i=0;i<1000;i++) doit();}

void g(){ for(int i=0;i<5000;i++) doit();}

int main () {
        double s=0; for(int i=0;i< 1000*NUM1; i++) s+=sqrt(i);
        f();
        g();
        std::cout<<"Done"<<std::endl;
        exit (0);
}
```

Understanding Flat Profile

- The *flat profile* shows the total amount of time your program spent executing each function.
- If a function was not compiled for profiling, and didn't run long enough to show up on the program counter histogram, it will be indistinguishable from a function that was never called

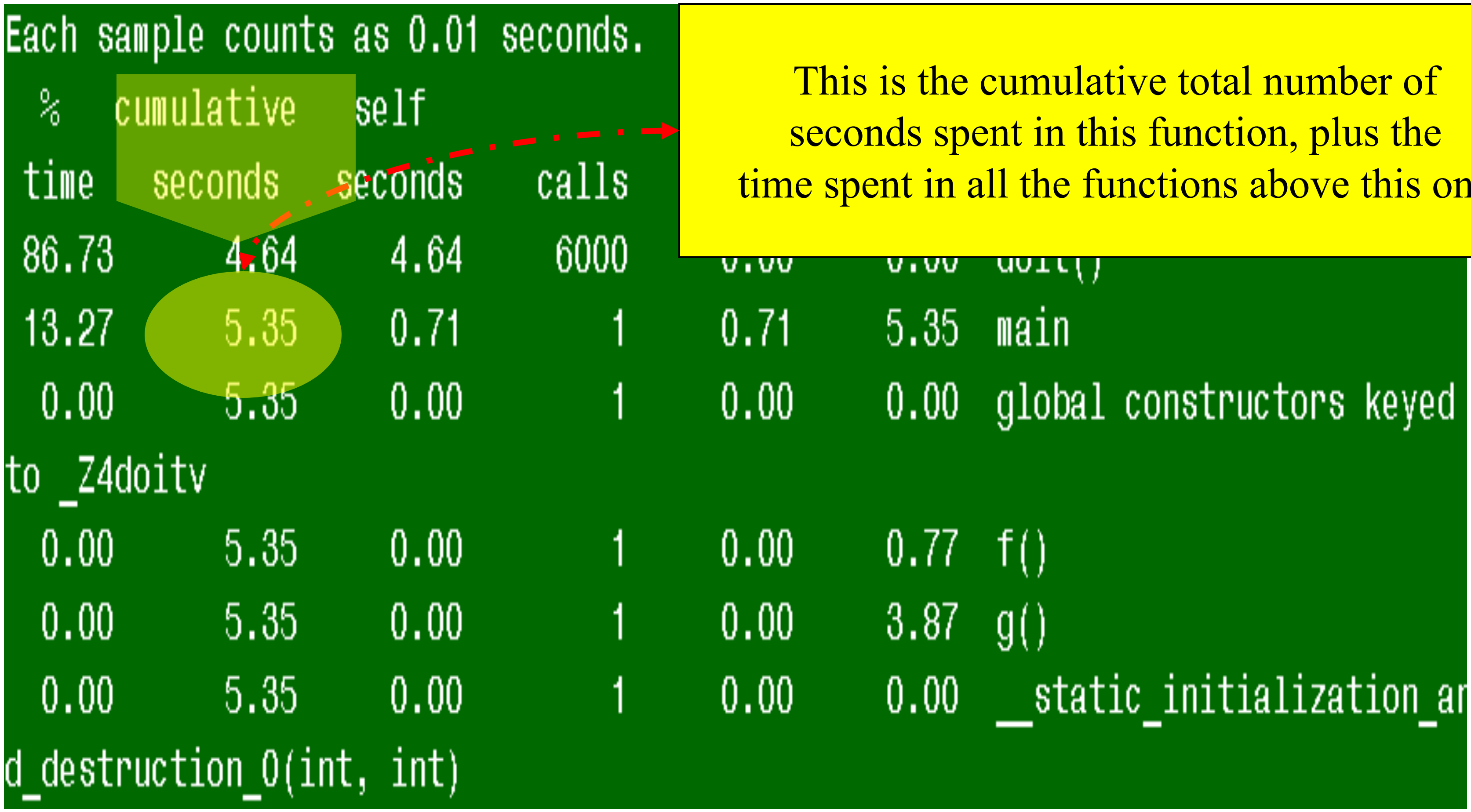
Flat profile : %time



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 86.73      4.64     4.64     6000     0.00     0.00  doit()
 13.27      5.35     0.71        1     0.71     5.35  main
  0.00      5.35     0.00        1     0.00     0.00  global constructors keyed
to _Z4doitv
  0.00      5.35     0.00        1     0.00     0.77  f()
  0.00      5.35     0.00        1     0.00     3.87  g()
  0.00      5.35     0.00        1                    lization_ar
d_destruction_0(int, int)
```

Percentage of the total execution time your program spent in this function. These should all add up to 100%.

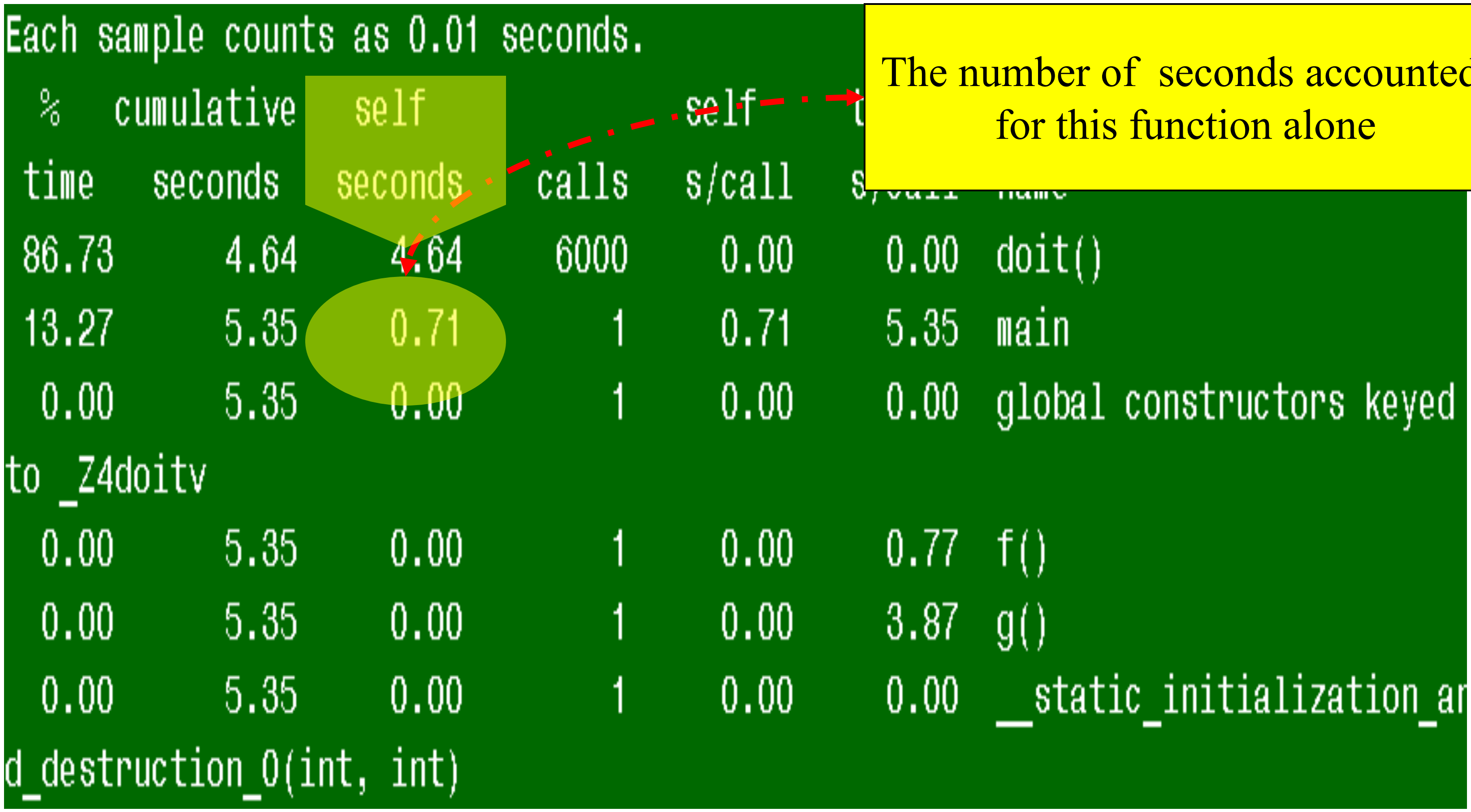
Flat profile: Cumulative seconds



```
Each sample counts as 0.01 seconds.
 %   cumulative   self
time   seconds   seconds    calls
86.73     4.64      4.64     6000     0.00     0.00  doit()
13.27     5.35      0.71        1     0.71     5.35  main
 0.00     5.35      0.00        1     0.00     0.00  global constructors keyed
to _Z4doitv
 0.00     5.35      0.00        1     0.00     0.77  f()
 0.00     5.35      0.00        1     0.00     3.87  g()
 0.00     5.35      0.00        1     0.00     0.00  __static_initialization_an
d_destruction_0(int, int)
```

This is the cumulative total number of seconds spent in this function, plus the time spent in all the functions above this one

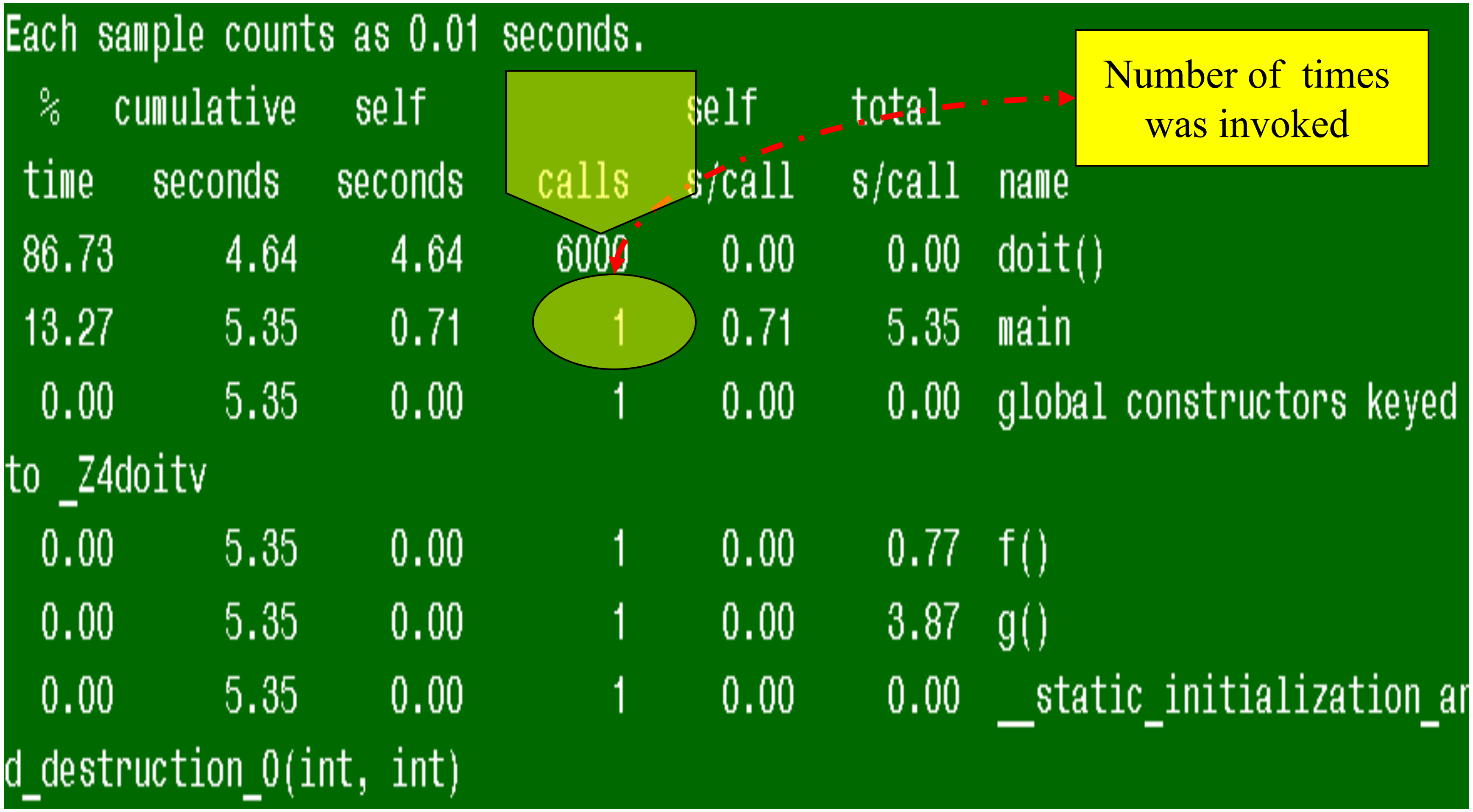
Flat profile: Self seconds



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     t
 time   seconds   seconds    calls  s/call   s/call  name
 86.73      4.64     4.64     6000    0.00     0.00  doit()
 13.27      5.35     0.71        1    0.71     5.35  main
  0.00      5.35     0.00        1    0.00     0.00  global constructors keyed
to _Z4doitv
  0.00      5.35     0.00        1    0.00     0.77  f()
  0.00      5.35     0.00        1    0.00     3.87  g()
  0.00      5.35     0.00        1    0.00     0.00  __static_initialization_ar
d_destruction_0(int, int)
```

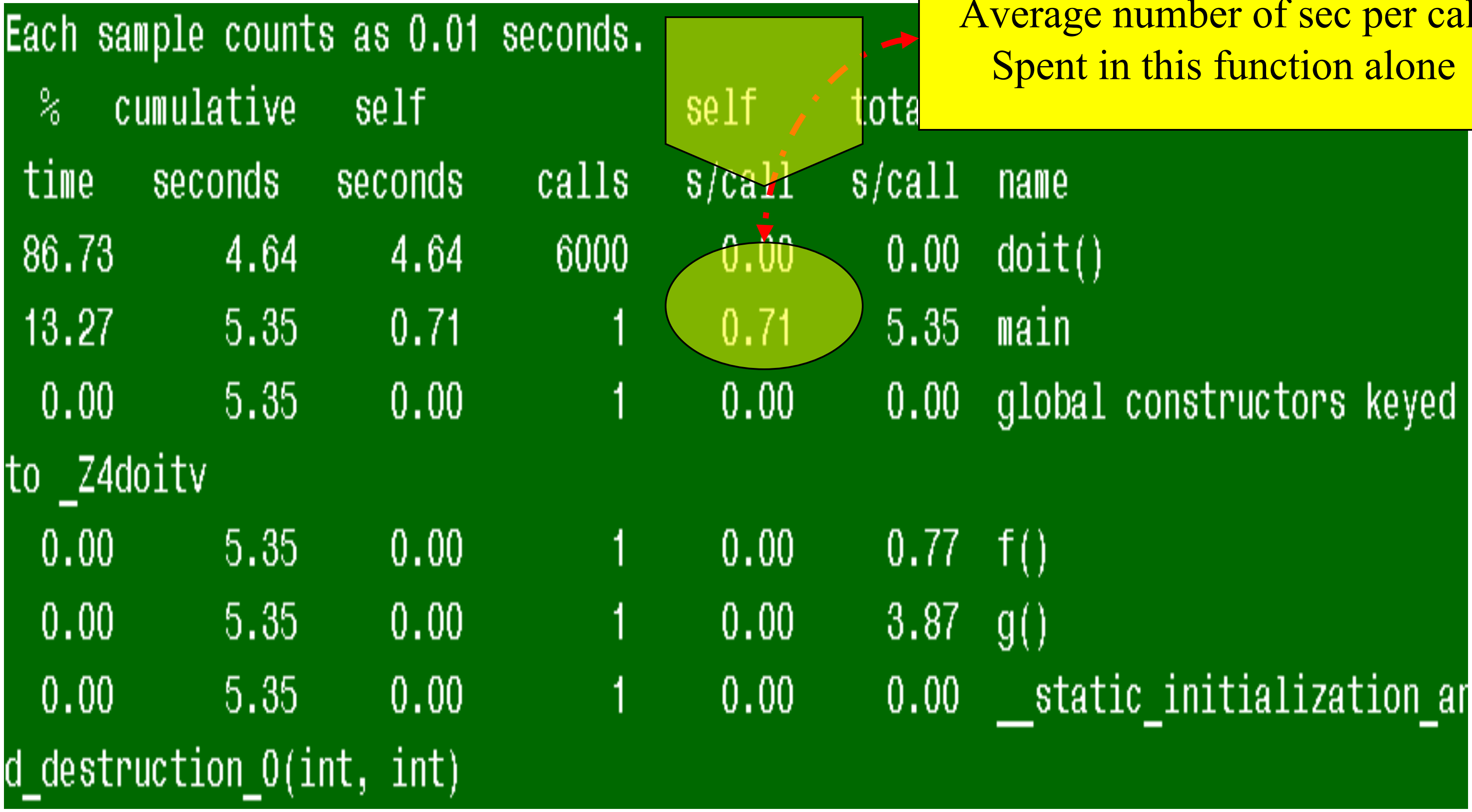
The number of seconds accounted for this function alone

Flat profile: Calls



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 86.73      4.64     4.64     6000    0.00     0.00  doit()
 13.27      5.35     0.71        1    0.71     5.35  main
  0.00      5.35     0.00        1    0.00     0.00  global constructors keyed
to _Z4doitv
  0.00      5.35     0.00        1    0.00     0.77  f()
  0.00      5.35     0.00        1    0.00     3.87  g()
  0.00      5.35     0.00        1    0.00     0.00  __static_initialization_ar
d_destruction_0(int, int)
```

Flat profile: Self seconds per call

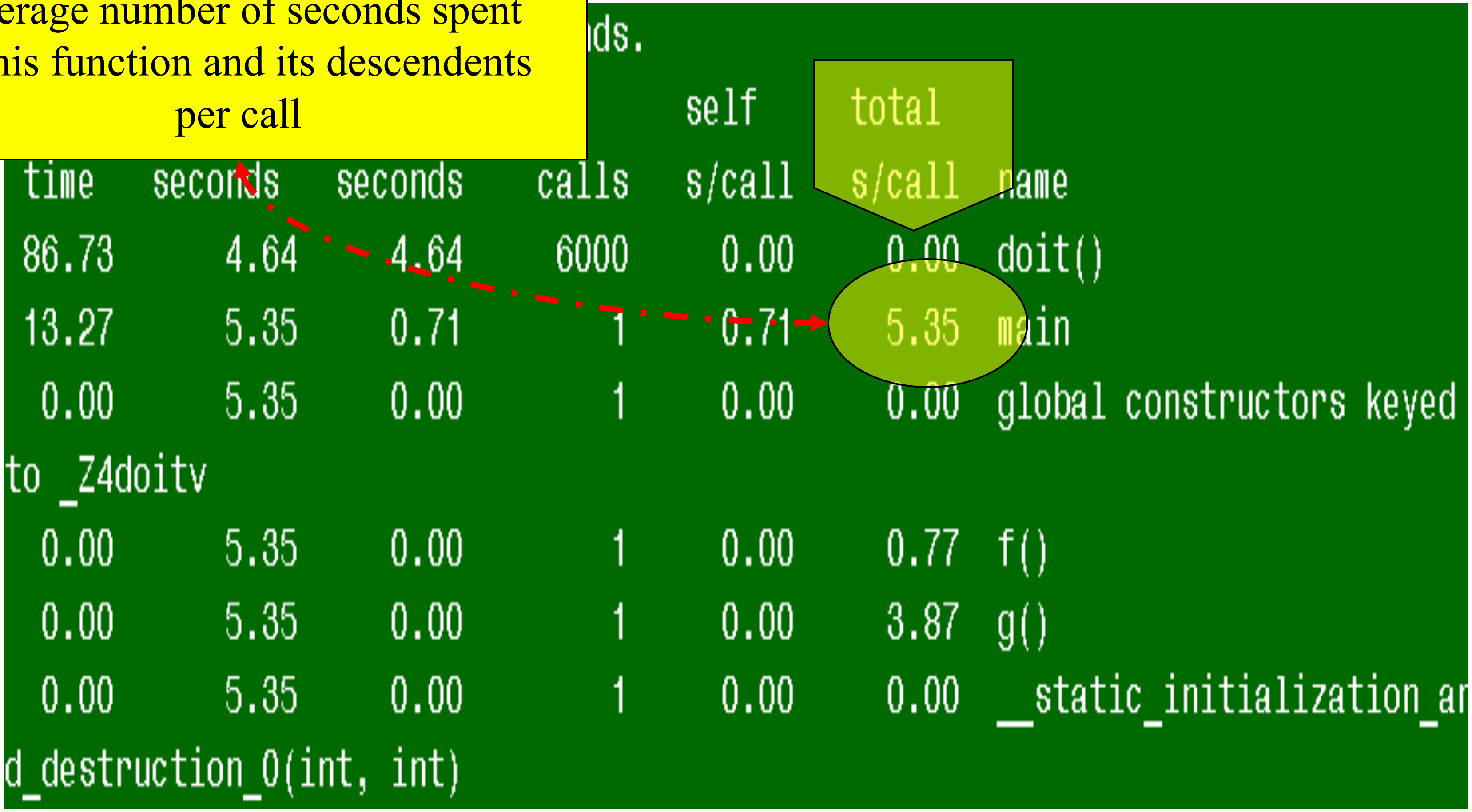


```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   s/call  name
 86.73      4.64     4.64     6000    0.00     0.00  doit()
 13.27      5.35     0.71        1    0.71     5.35  main
  0.00      5.35     0.00        1    0.00     0.00  global constructors keyed
to _Z4doitv
  0.00      5.35     0.00        1    0.00     0.77  f()
  0.00      5.35     0.00        1    0.00     3.87  g()
  0.00      5.35     0.00        1    0.00     0.00  __static_initialization_ar
d_destruction_0(int, int)
```

Average number of sec per call
Spent in this function alone

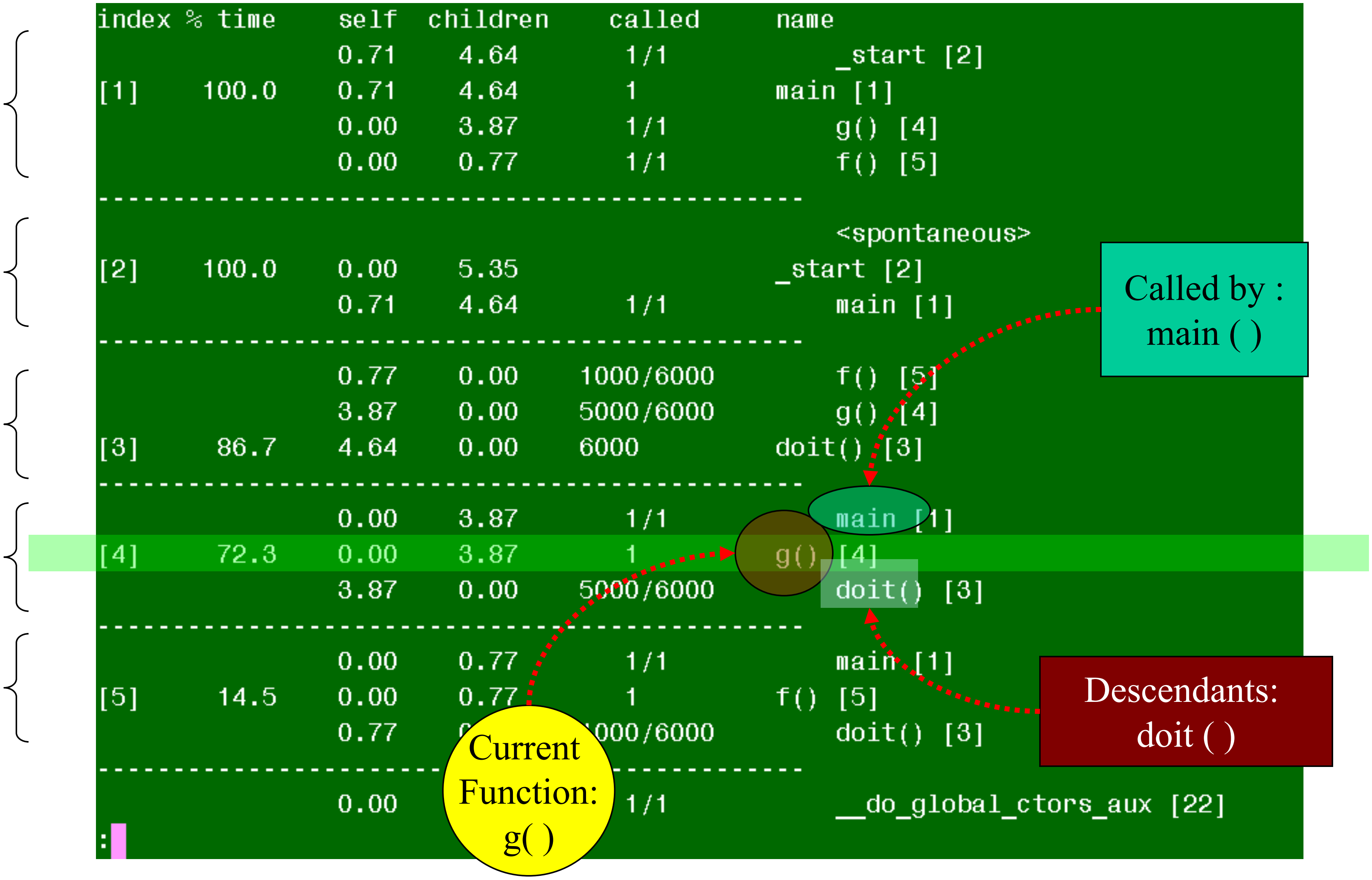
Flat profile: Total seconds per call

Average number of seconds spent in this function and its descendents per call



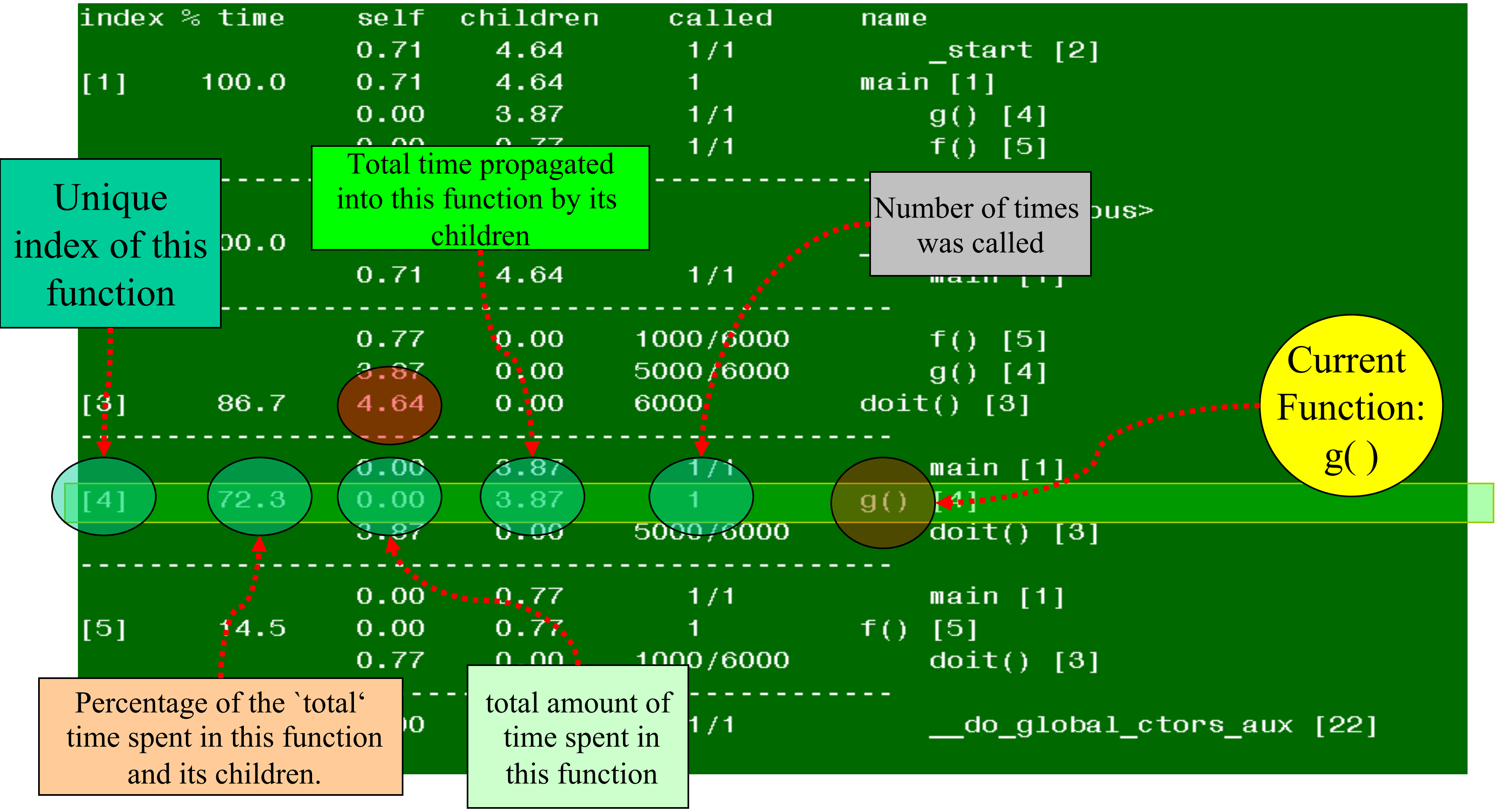
```
     ds.
                              self    total
time   seconds   seconds   calls  s/call  s/call  name
86.73     4.64      4.64    6000    0.00    0.00  doit()
13.27     5.35      0.71       1    0.71    5.35  main
 0.00     5.35      0.00       1    0.00    0.00  global constructors keyed
to _Z4doitv
 0.00     5.35      0.00       1    0.00    0.77  f()
 0.00     5.35      0.00       1    0.00    3.87  g()
 0.00     5.35      0.00       1    0.00    0.00  __static_initialization_an
d_destruction_0(int, int)
```

Call Graph: call tree of the program



```
index % time    self  children    called     name
                0.71    4.64       1/1           _start [2]
[1]    100.0    0.71    4.64       1         main [1]
                0.00    3.87       1/1           g() [4]
                0.00    0.77       1/1           f() [5]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00    5.35                 _start [2]
                0.71    4.64       1/1           main [1]
-----------------------------------------------
                0.77    0.00    1000/6000        f() [5]
                3.87    0.00    5000/6000        g() [4]
[3]     86.7    4.64    0.00    6000         doit() [3]
-----------------------------------------------
                0.00    3.87       1/1           main [1]
[4]     72.3    0.00    3.87       1         g() [4]
                3.87    0.00    5000/6000        doit() [3]
-----------------------------------------------
                0.00    0.77       1/1           main [1]
[5]     14.5    0.00    0.77       1         f() [5]
                0.77    0.00    1000/6000        doit() [3]
-----------------------------------------------
                0.00               1/1           __do_global_ctors_aux [22]
:
```

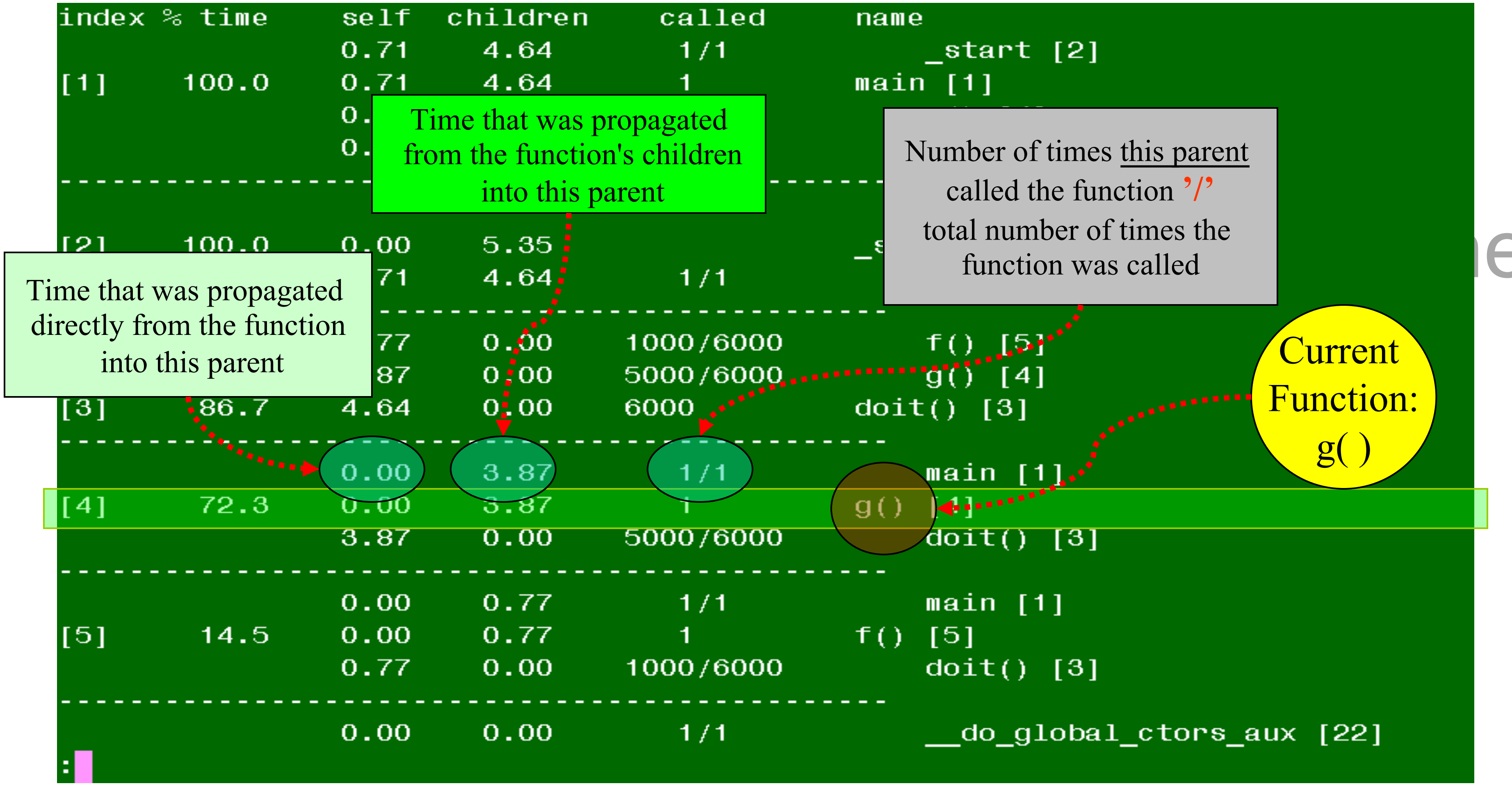
Call Graph: understanding each line



```
index % time    self  children    called     name
                0.71    4.64       1/1           _start [2]
[1]     100.0   0.71    4.64       1         main [1]
                0.00    3.87       1/1           g() [4]
                0.00    0.77       1/1           f() [5]
 ------                       -----------
                                                    ous>
         00.0
                0.71    4.64       1/1
 -----------------------------------------------------
                0.77    0.00    1000/6000        f() [5]
                3.87    0.00    5000/6000        g() [4]
[3]      86.7   4.64    0.00    6000         doit() [3]
 -----------------------------------------------------
                0.00    3.87       1/1           main [1]
[4]      72.3   0.00    3.87       1         g() [4]
                3.87    0.00    5000/6000        doit() [3]
 -----------------------------------------------------
                0.00    0.77       1/1           main [1]
[5]      14.5   0.00    0.77       1         f() [5]
                0.77    0.00    1000/6000        doit() [3]
                                 ------------
                  00               1/1           __do_global_ctors_aux [22]
```

Unique index of this function
Total time propagated into this function by its children
Number of times was called
Current Function: g( )
Percentage of the `total‘ time spent in this function and its children.
total amount of time spent in this function

Call Graph: parents numbers



```
index % time    self  children    called     name
                0.71    4.64       1/1           _start [2]
[1]    100.0    0.71    4.64       1         main [1]
                0.
                0.
-----------------                   -------
[2]    100.0    0.00    5.35                 _s
                 71     4.64       1/1       _
                 -------------------------------
                 77     0.00    1000/6000        f() [5]
                 87     0.00    5000/6000        g() [4]
[3]     86.7    4.64    0.00    6000         doit() [3]
-----------------------------------------------
                0.00    3.87       1/1           main [1]
[4]     72.3    0.00    3.87       1         g() [4]
                3.87    0.00    5000/6000        doit() [3]
-----------------------------------------------
                0.00    0.77       1/1           main [1]
[5]     14.5    0.00    0.77       1         f() [5]
                0.77    0.00    1000/6000        doit() [3]
-----------------------------------------------
                0.00    0.00       1/1           __do_global_ctors_aux [22]
:
```

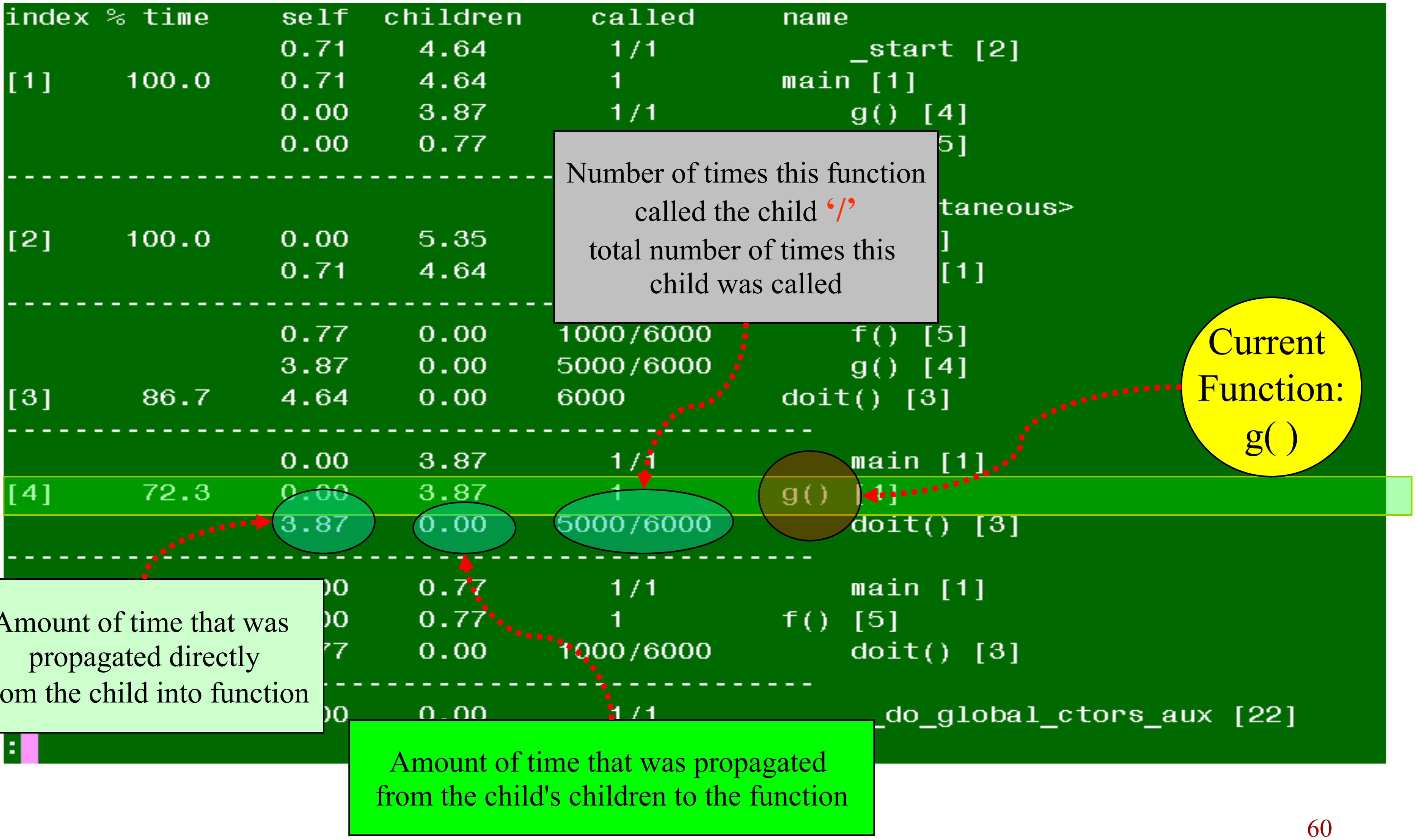
Time that was propagated from the function's children into this parent

Number of times this parent called the function '/' total number of times the function was called

Time that was propagated directly from the function into this parent

Current Function: g( )

Call Graph : “children” numbers



```
index % time    self  children     called       name
                0.71    4.64        1/1             _start [2]
[1]     100.0   0.71    4.64        1           main [1]
                0.00    3.87        1/1             g() [4]
                0.00    0.77                          5]
------------------------------
                                                    taneous>
[2]     100.0   0.00    5.35                        ]
                0.71    4.64                        [1]
------------------------------
                0.77    0.00     1000/6000          f() [5]
                3.87    0.00     5000/6000          g() [4]
[3]      86.7   4.64    0.00     6000           doit() [3]
--------------------------------------------------
                0.00    3.87        1/1             main [1]
[4]      72.3   0.00    3.87        1           g() [4]
                3.87    0.00     5000/6000          doit() [3]
--------------------------------------------------
                  00    0.77        1/1             main [1]
                  00    0.77        1           f() [5]
                  77    0.00     1000/6000          doit() [3]
--------------------------------------------------
                  00    0.00        1/1             __do_global_ctors_aux [22]
:
```

Number of times this function called the child ‘/’ total number of times this child was called
Current Function: g( )
Amount of time that was propagated directly from the child into function
Amount of time that was propagated from the child's children to the function

How gprof works

- Instruments program to count calls
- Watches the program running, samples the PC every 0.01 sec
    - Statistical inaccuracy: fast function may take 0 or 1 samples
    - The execution time should be long enough compared with the sampling period

- The output from gprof does not indicate parts of the program that are limited by I/O or swapping bandwidth.
    - This is because samples of the program counter are taken at fixed intervals of <u>run</u> time

- number-of-calls figures are derived by counting, not sampling. They are completely accurate and will not vary from run to run if your program is deterministic
    - Profiling with inlining and other optimizations needs care

Valgrind

- Multi-purpose Linux x86 profiling toolkit

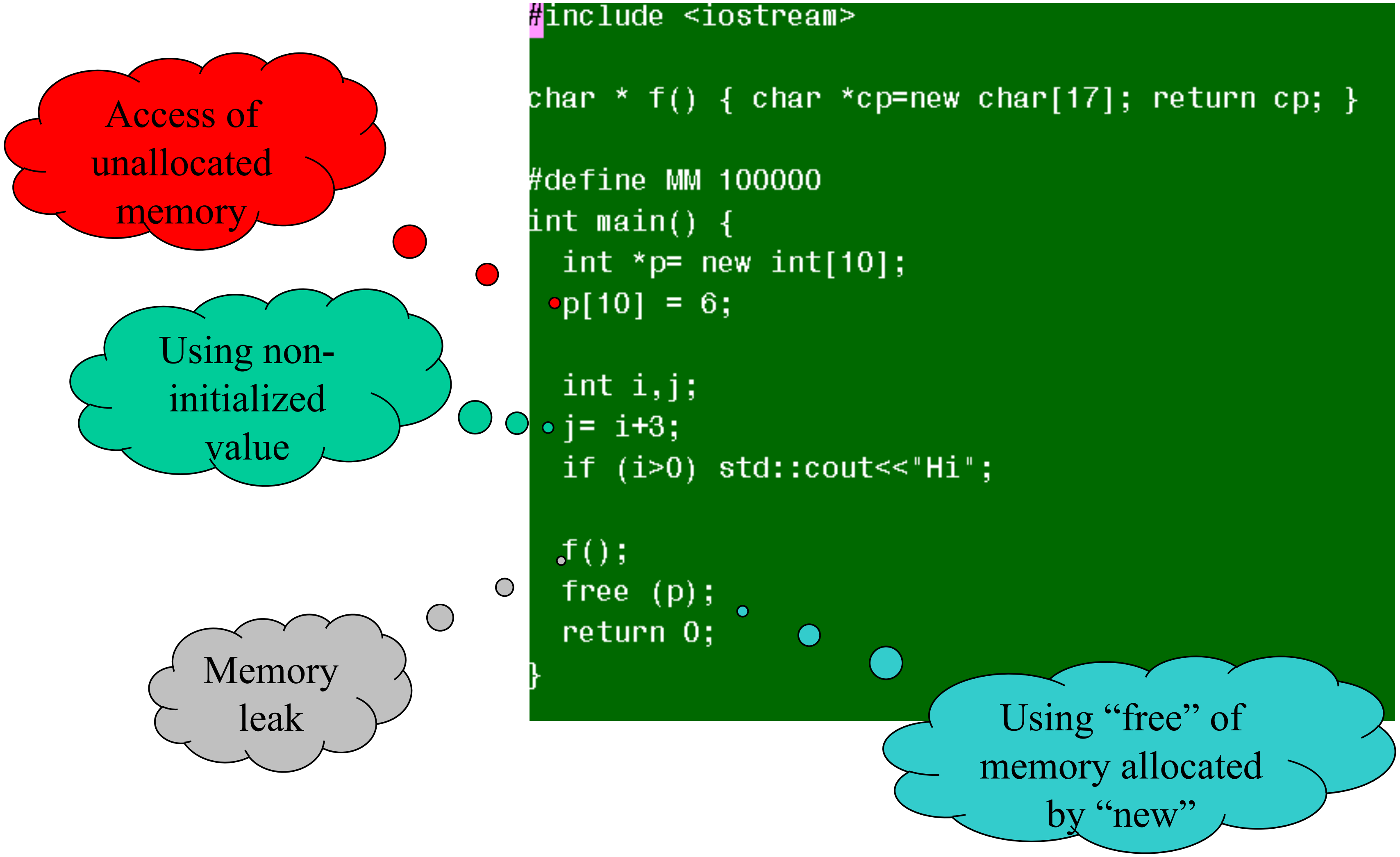
- Memcheck is memory debugger
  - detects memory-management problems
- Cachegrind is a cache profiler
  - performs detailed simulation of the I1, D1 and L2 caches in your CPU
- Massif is a heap profiler
  - performs detailed heap profiling by taking regular snapshots of a program's heap
- Helgrind is a thread debugger
  - finds data races in multithreaded programs

Memcheck Features

- When a program is run under Memcheck's supervision, all reads and writes of memory are checked, and calls to malloc/new/free/delete are intercepted

- Memcheck can detect:
  - Use of uninitialized memory
  - Reading/writing memory after it has been free'd
  - Reading/writing off the end of malloc'd blocks
  - Reading/writing inappropriate areas on the stack
  - Memory leaks -- where pointers to malloc'd blocks are lost forever
  - Passing of uninitialized and/or not addressible memory to system calls
  - Mismatched use of malloc/new/new [] vs free/delete/delete []
  - Overlapping src and dst pointers in memcpy() and related functions
  - Some misuses of the POSIX pthreads API

Memcheck Example



```
#include <iostream>

char * f() { char *cp=new char[17]; return cp; }

#define MM 100000
int main() {
  int *p= new int[10];
  p[10] = 6;

  int i,j;
  j= i+3;
  if (i>0) std::cout<<"Hi";

  f();
  free (p);
  return 0;
}
```

Access of unallocated memory
Using non-initialized value
Memory leak
Using "free" of memory allocated by "new"

Memcheck Example (Cont.)

- Compile the program with –g flag:
  - `g++   -c a.cc  –g   –o a.out`

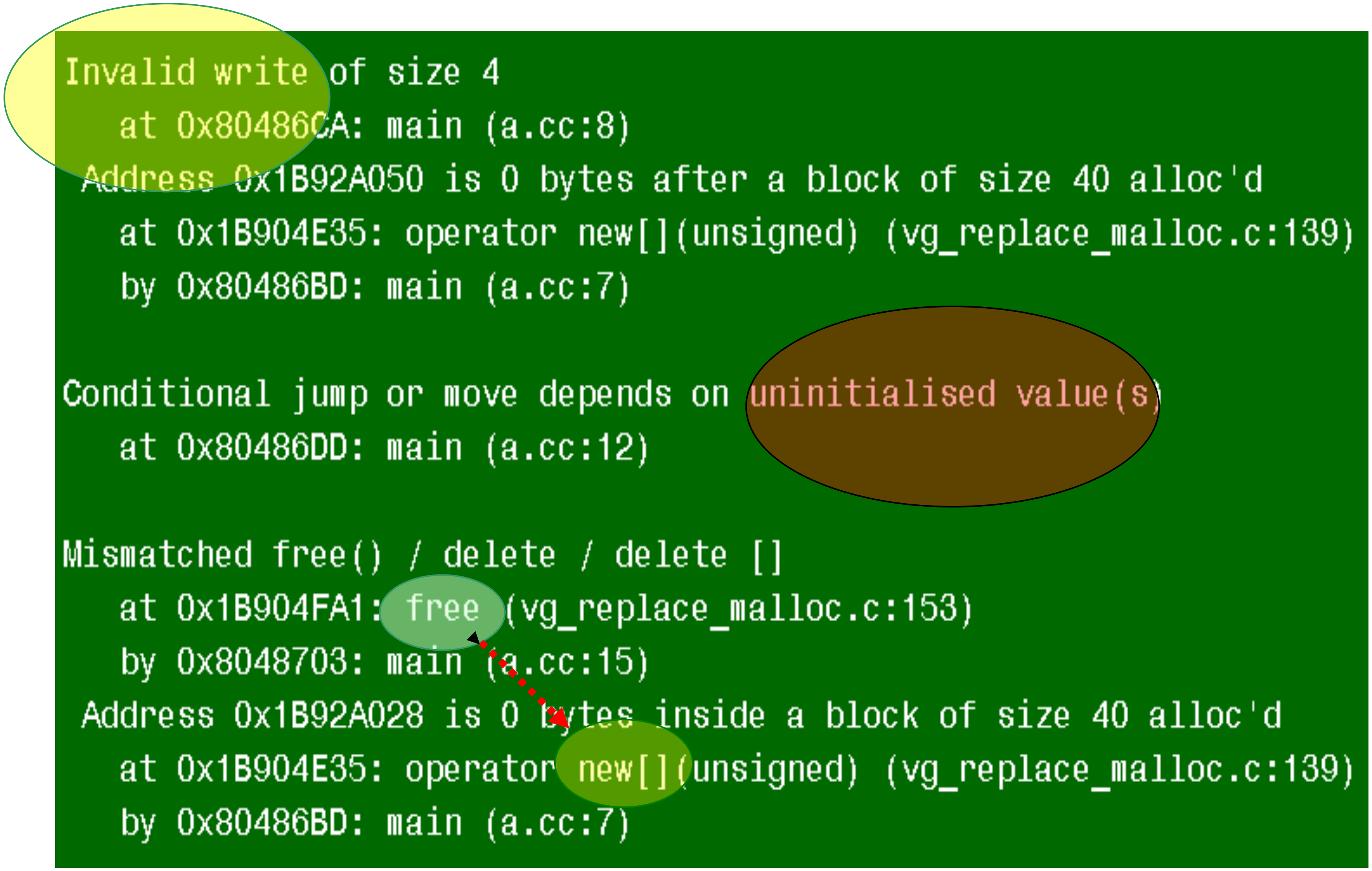
Debug leaks

- Execute valgrind :
  - `valgrind --tool=memcheck   --leak-check=yes a.out >& log.txt`

Executable name

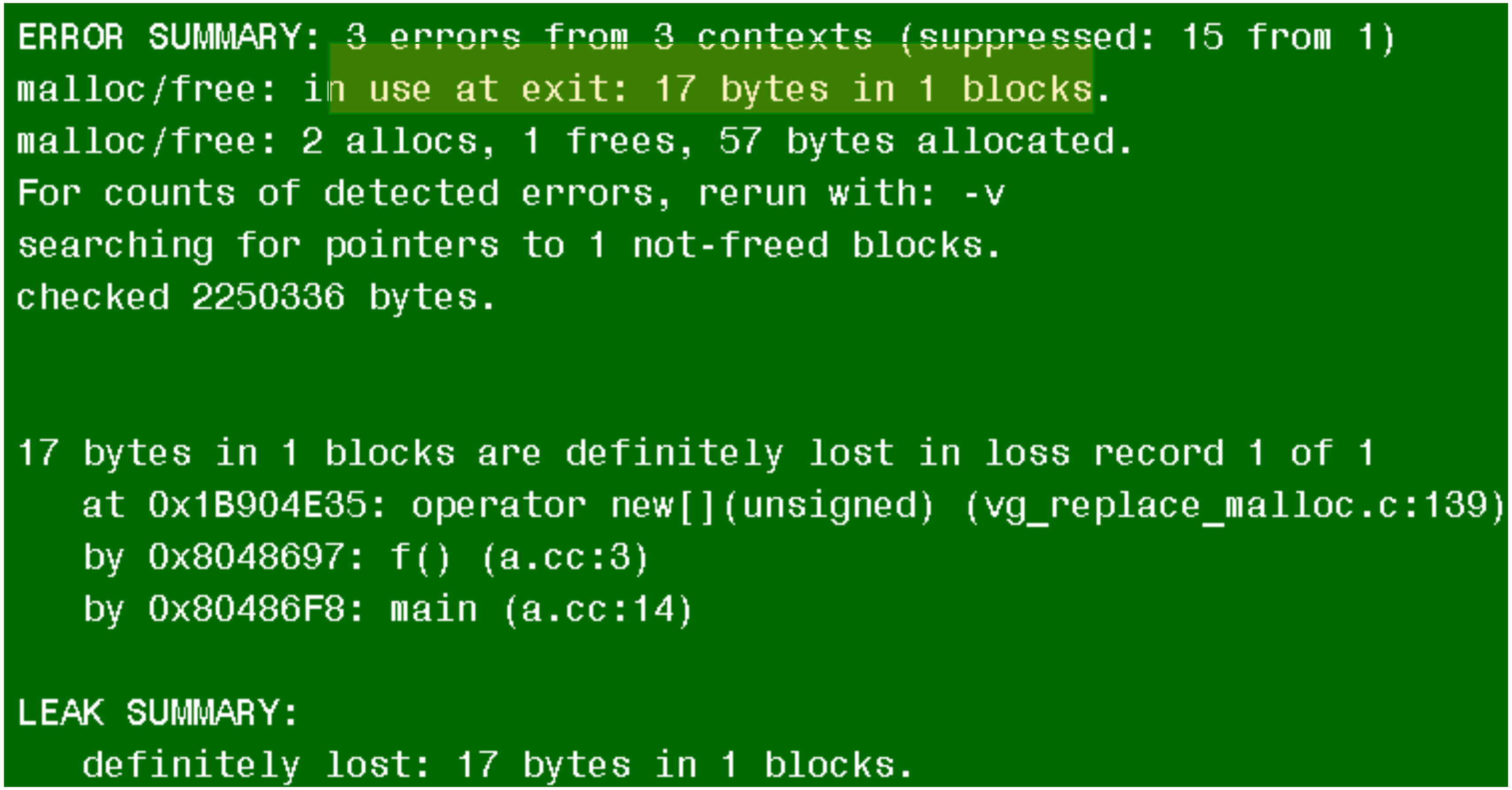
- View log

Memcheck report



```
Invalid write of size 4
   at 0x80486CA: main (a.cc:8)
 Address 0x1B92A050 is 0 bytes after a block of size 40 alloc'd
   at 0x1B904E35: operator new[](unsigned) (vg_replace_malloc.c:139)
   by 0x80486BD: main (a.cc:7)

Conditional jump or move depends on uninitialised value(s)
   at 0x80486DD: main (a.cc:12)

Mismatched free() / delete / delete []
   at 0x1B904FA1: free (vg_replace_malloc.c:153)
   by 0x8048703: main (a.cc:15)
 Address 0x1B92A028 is 0 bytes inside a block of size 40 alloc'd
   at 0x1B904E35: operator new[](unsigned) (vg_replace_malloc.c:139)
   by 0x80486BD: main (a.cc:7)
```

Memcheck report (cont.) Leaks detected:



```
ERROR SUMMARY: 3 errors from 3 contexts (suppressed: 15 from 1)
malloc/free: in use at exit: 17 bytes in 1 blocks.
malloc/free: 2 allocs, 1 frees, 57 bytes allocated.
For counts of detected errors, rerun with: -v
searching for pointers to 1 not-freed blocks.
checked 2250336 bytes.


17 bytes in 1 blocks are definitely lost in loss record 1 of 1
   at 0x1B904E35: operator new[](unsigned) (vg_replace_malloc.c:139)
   by 0x8048697: f() (a.cc:3)
   by 0x80486F8: main (a.cc:14)

LEAK SUMMARY:
   definitely lost: 17 bytes in 1 blocks.
```

S
T
A
C
K

Cachegrind

- Detailed cache profiling can be very useful for improving the performance of the program
  - On a modern x86 machine, an L1 miss will cost around 10 cycles, and an L2 miss can cost as much as 200 cycles
- Cachegrind performs detailed simulation of the I1, D1 and L2 caches in your CPU
- Can accurately pinpoint the sources of cache misses in the code
- Identifies the number of cache misses, memory references, and instructions executed for each line of source code, with per-function, per-module and whole-program summaries
- Cachegrind runs programs about 20--100x slower than normal

How to run

- Run valgrind --tool=cachegrind in front of the normal command line invocation
  - Example : valgrind --tool=cachegrind ls -l

- When the program finishes, Cachegrind will print summary cache statistics. It also collects line-by-line information in a file cachegrind.out.*pid*

- Execute cg_annotate to get annotated source file:

Source files

```
cg_annotate cachegrind.out.7618 a.cc  >  a.cc.annotated
```

PID

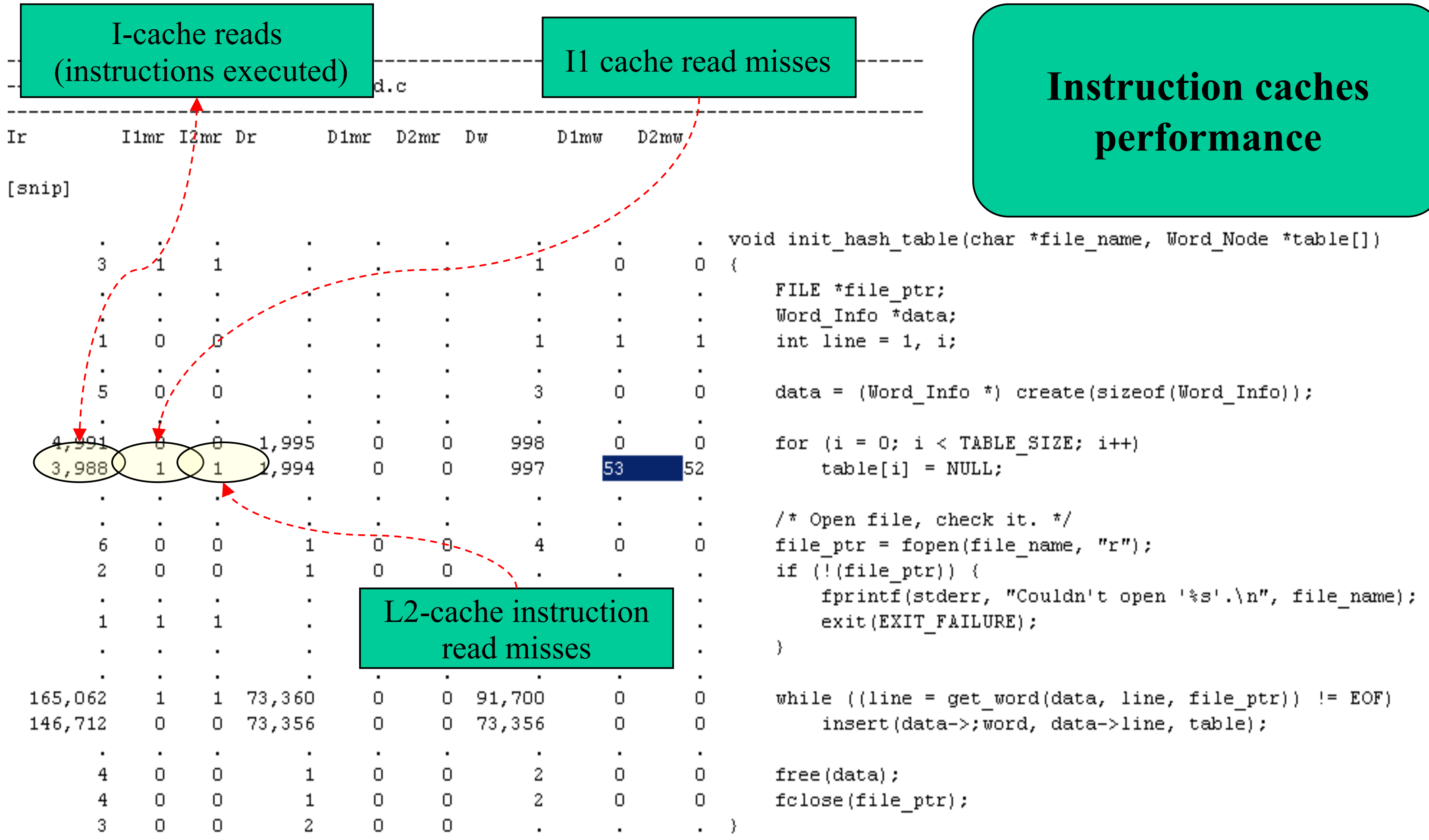
Valgrind

This will output the cache statistics for each line of our program:

Ir	I cache reads (instructions executed)
I1mr	I1 cache read misses (instruction wasn't in I1 cache but was in L2)
I2mr	L2 cache instruction read misses (instruction wasn't in I1 or L2 cache, had to be fetched from memory)
Dr	D cache reads (memory reads)
D1mr	D1 cache read misses (data location not in D1 cache, but in L2)
D2mr	L2 cache data read misses (location not in D1 or L2)
Dw	D cache writes (memory writes)
D1mw	D1 cache write misses (location not in D1 cache, but in L2)
D2mw	L2 cache data write misses (location not in D1 or L2)

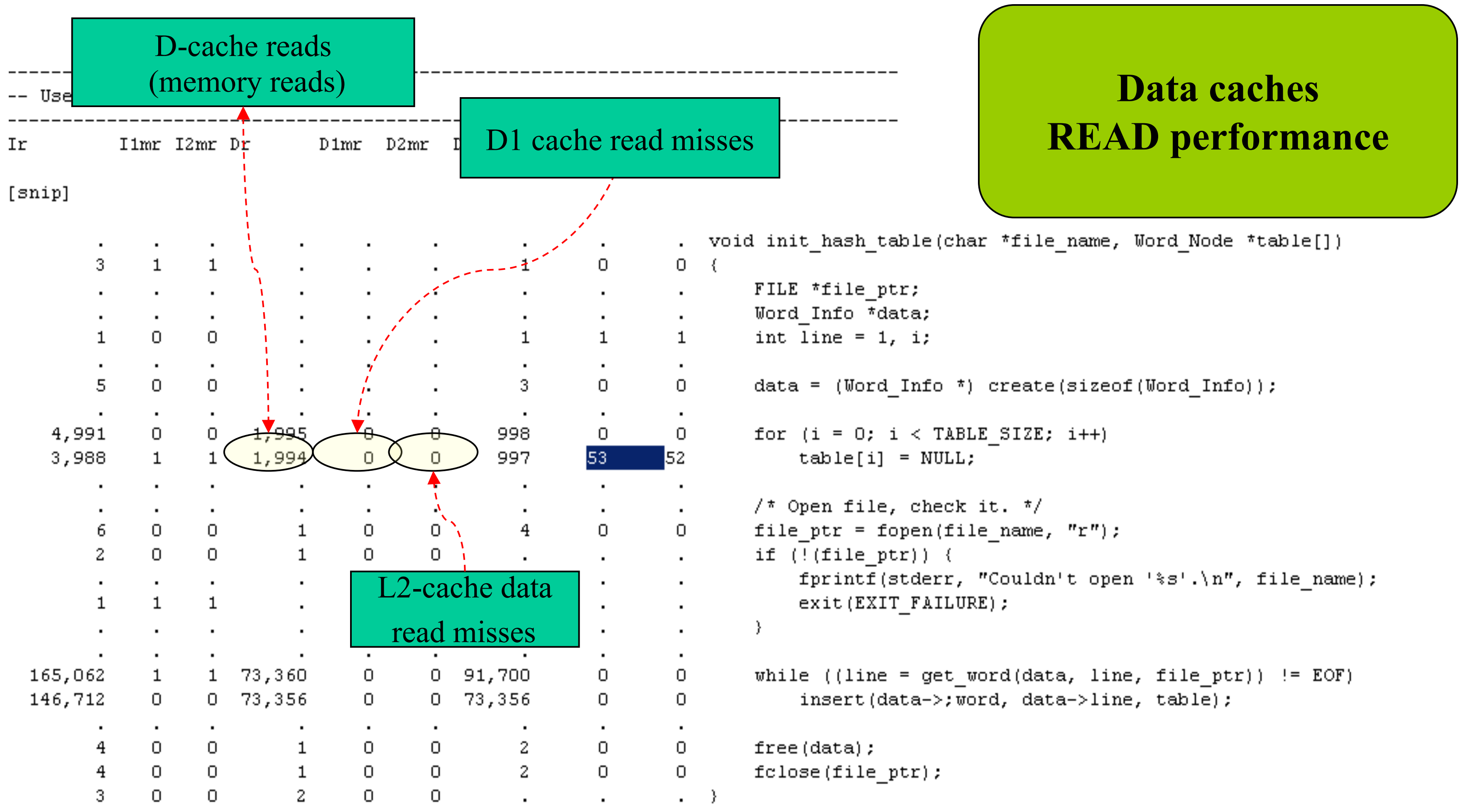
Cachegrind Summary output

Instruction caches performance


I-cache reads (instructions executed)
I1 cache read misses
L2-cache instruction read misses

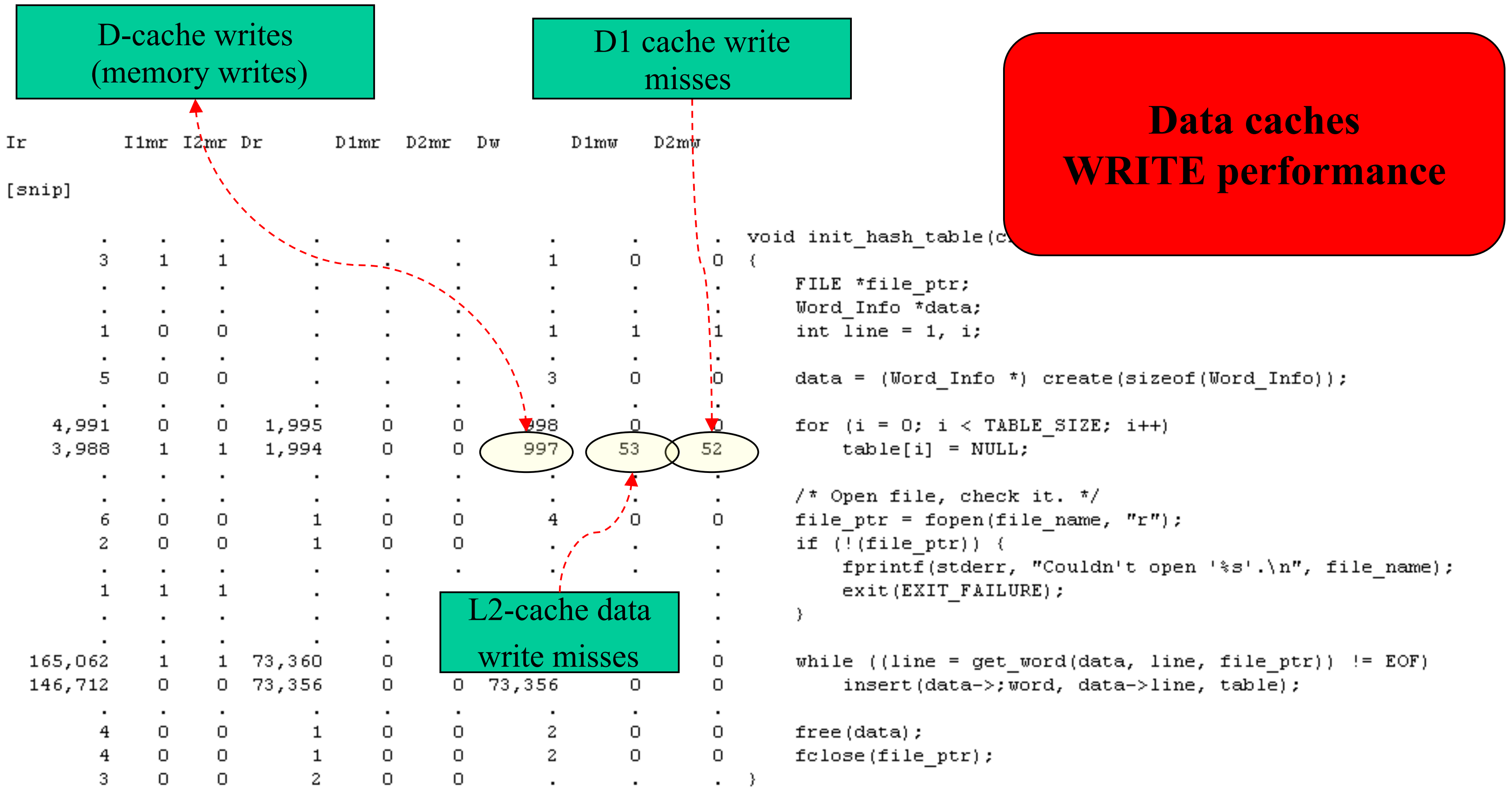
```
--------------------------------------------------------------------------------
-                                  d.c
--------------------------------------------------------------------------------
Ir          I1mr I2mr Dr       D1mr  D2mr  Dw       D1mw    D2mw

[snip]

      .     .    .       .      .     .      .       .       .  void init_hash_table(char *file_name, Word_Node *table[])
      3     1    1       .      .     .      1       0       0  {
      .     .    .       .      .     .      .       .       .       FILE *file_ptr;
      .     .    .       .      .     .      .       .       .       Word_Info *data;
      1     0    0       .      .     .      1       1       1       int line = 1, i;
      .     .    .       .      .     .      .       .       .
      5     0    0       .      .     .      3       0       0       data = (Word_Info *) create(sizeof(Word_Info));
      .     .    .       .      .     .      .       .       .
  4,991     0    0   1,995      0     0    998       0       0       for (i = 0; i < TABLE_SIZE; i++)
  3,988     1    1   1,994      0     0    997      53      52           table[i] = NULL;
      .     .    .       .      .     .      .       .       .
      .     .    .       .      .     .      .       .       .       /* Open file, check it. */
      6     0    0       1      0     0      4       0       0       file_ptr = fopen(file_name, "r");
      2     0    0       1      0     0      .       .       .       if (!(file_ptr)) {
      .     .    .       .                                   .           fprintf(stderr, "Couldn't open '%s'.\n", file_name);
      1     1    1       .                                   .           exit(EXIT_FAILURE);
      .     .    .       .                                   .       }
      .     .    .       .                                   .
165,062     1    1  73,360      0     0 91,700       0       0       while ((line = get_word(data, line, file_ptr)) != EOF)
146,712     0    0  73,356      0     0 73,356       0       0           insert(data->;word, data->line, table);
      .     .    .       .      .     .      .       .       .
      4     0    0       1      0     0      2       0       0       free(data);
      4     0    0       1      0     0      2       0       0       fclose(file_ptr);
      3     0    0       2      0     0      .       .       .  }
```

Cachegrind Summary output



```
-----                                   ------------------------------------------
-- Use
--------------------------------------------------------------------   -----------
Ir         I1mr  I2mr  Dr        D1mr   D2mr   D

[snip]

      .     .     .       .      .      .      .      .      .   void init_hash_table(char *file_name, Word_Node *table[])
      3     1     1       .      .      .      1      0      0   {
      .     .     .       .      .      .      .      .      .       FILE *file_ptr;
      .     .     .       .      .      .      .      .      .       Word_Info *data;
      1     0     0       .      .      .      1      1      1       int line = 1, i;
      .     .     .       .      .      .      .      .      .
      5     0     0       .      .      .      3      0      0       data = (Word_Info *) create(sizeof(Word_Info));
      .     .     .       .      .      .      .      .      .
  4,991     0     0   1,995      0      0    998      0      0       for (i = 0; i < TABLE_SIZE; i++)
  3,988     1     1   1,994      0      0    997     53     52           table[i] = NULL;
      .     .     .       .      .      .      .      .      .
      .     .     .       .      .      .      .      .      .       /* Open file, check it. */
      6     0     0       1      0      0      4      0      0       file_ptr = fopen(file_name, "r");
      2     0     0       1      0      0      .      .      .       if (!(file_ptr)) {
      .     .     .       .                           .      .           fprintf(stderr, "Couldn't open '%s'.\n", file_name);
      1     1     1       .                           .      .           exit(EXIT_FAILURE);
      .     .     .       .                           .      .       }
      .     .     .       .      .      .      .      .      .
165,062     1     1  73,360      0      0 91,700      0      0       while ((line = get_word(data, line, file_ptr)) != EOF)
146,712     0     0  73,356      0      0 73,356      0      0           insert(data->;word, data->line, table);
      .     .     .       .      .      .      .      .      .
      4     0     0       1      0      0      2      0      0       free(data);
      4     0     0       1      0      0      2      0      0       fclose(file_ptr);
      3     0     0       2      0      0      .      .      .   }
```

Cachegrind Summary output



D-cache writes (memory writes)

D1 cache write misses

Data caches WRITE performance

L2-cache data write misses

```
Ir          I1mr I2mr Dr       D1mr  D2mr  Dw       D1mw    D2mw

[snip]

      .       .     .       .     .     .       .     .     .   void init_hash_table(c
      3       1     1       .     .     .       1     0     0   {
      .       .     .       .     .     .       .     .     .       FILE *file_ptr;
      .       .     .       .     .     .       .     .     .       Word_Info *data;
      1       0     0       .     .     .       1     1     1       int line = 1, i;
      .       .     .       .     .     .       .     .     .
      5       0     0       .     .     .       3     0     0       data = (Word_Info *) create(sizeof(Word_Info));
      .       .     .       .     .     .       .     .     .
  4,991       0     0   1,995     0     0     998     0     0       for (i = 0; i < TABLE_SIZE; i++)
  3,988       1     1   1,994     0     0     997    53    52           table[i] = NULL;
      .       .     .       .     .     .       .     .     .
      .       .     .       .     .     .       .     .     .       /* Open file, check it. */
      6       0     0       1     0     0       4     0     0       file_ptr = fopen(file_name, "r");
      2       0     0       1     0     0       .     .     .       if (!(file_ptr)) {
      .       .     .       .     .     .       .     .     .           fprintf(stderr, "Couldn't open '%s'.\n", file_name);
      1       1     1       .     .                         .           exit(EXIT_FAILURE);
      .       .     .       .     .                         .       }
      .       .     .       .     .                         .
165,062       1     1  73,360     0                         0       while ((line = get_word(data, line, file_ptr)) != EOF)
146,712       0     0  73,356     0     0  73,356     0     0           insert(data->;word, data->line, table);
      .       .     .       .     .     .       .     .     .
      4       0     0       1     0     0       2     0     0       free(data);
      4       0     0       1     0     0       2     0     0       fclose(file_ptr);
      3       0     0       2     0     0       .     .     .   }
```

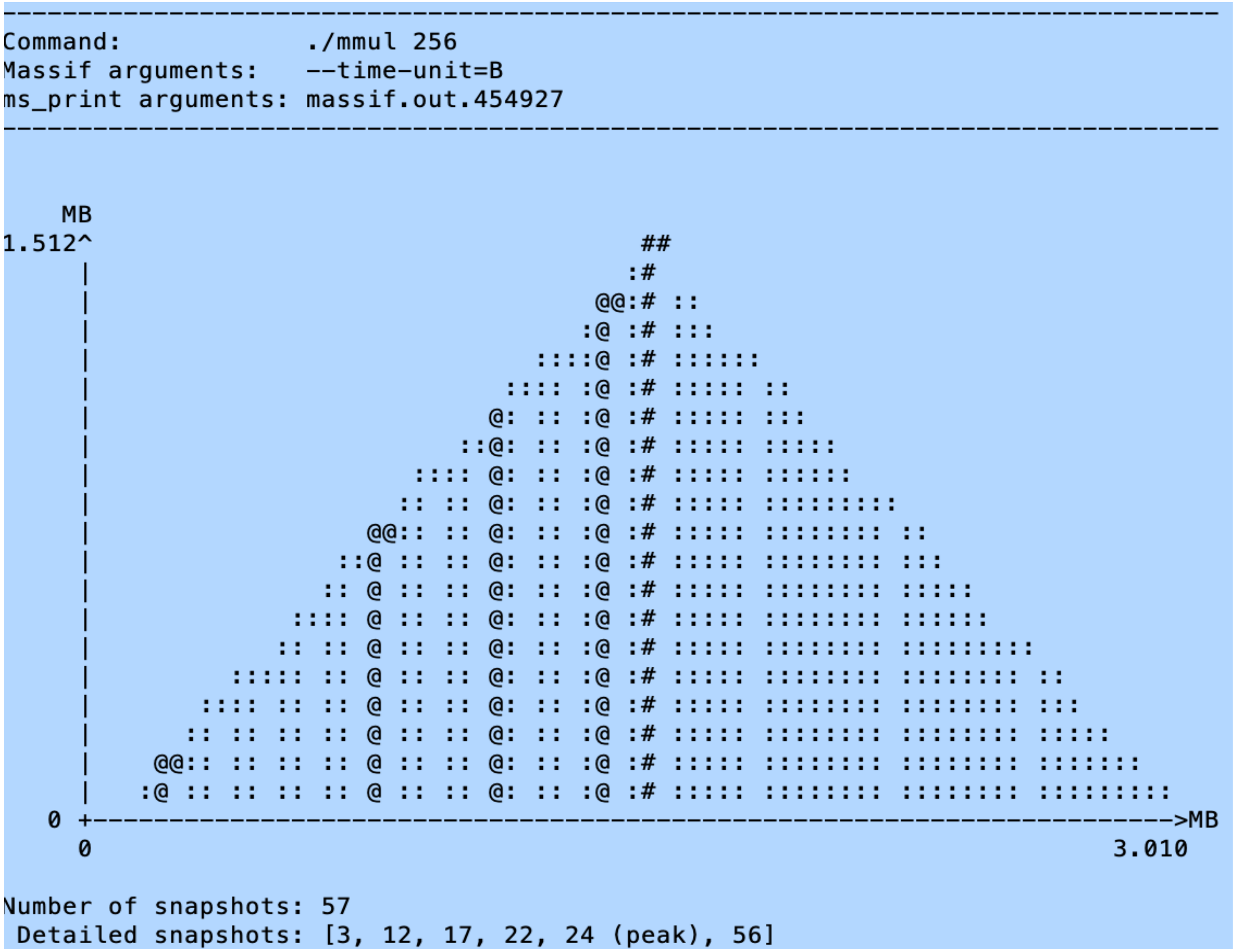
Cachegrind Accuracy

- Valgrind's cache profiling has some shortcomings:
    - It doesn't account for kernel activity -- the effect of system calls on the cache contents is ignored
    - It doesn't account for other process activity (although this is probably desirable when considering a single program)
    - It doesn't account for virtual-to-physical address mappings; hence the entire simulation is not a true representation of what's happening in the cache

Massif tool

- Massif is a heap profiler - it measures how much heap memory programs use. It can give information about:
  - Heap blocks
  - Heap administration blocks
  - Stack sizes
- Help to reduce the amount of memory the program uses
  - smaller program interact better with caches, avoid paging
- Detect leaks that aren't detected by traditional leak-checkers, such as Memcheck
  - That's because the memory isn't ever actually lost - a pointer remains to it - but it's not in use anymore
- Run **valgrind --tool=massif** ***prog***

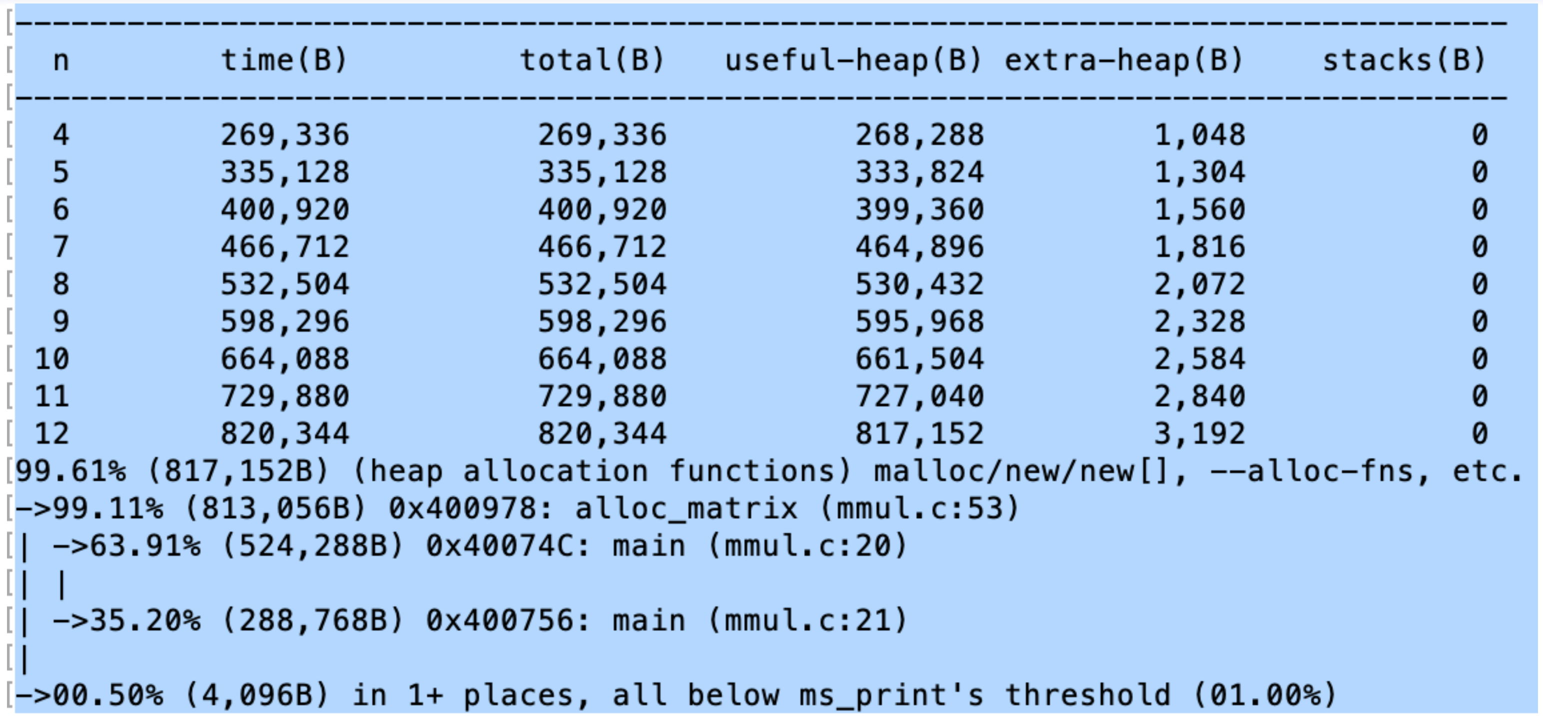
Massif tool



```
--------------------------------------------------------------------------------
Command:            ./mmul 256
Massif arguments:   --time-unit=B
ms_print arguments: massif.out.454927
--------------------------------------------------------------------------------


    MB
1.512^                                    ##
     |                                   :#
     |                                 @@:# ::
     |                                :@ :# :::
     |                             ::::@ :# ::::::
     |                           :::: :@ :# ::::: ::
     |                          @: :: :@ :# ::::: :::
     |                        ::@: :: :@ :# ::::: :::::
     |                     :::: @: :: :@ :# ::::: ::::::
     |                    :: :: @: :: :@ :# ::::: :::::::::
     |                  @@:: :: @: :: :@ :# ::::: :::::::: ::
     |                ::@ :: :: @: :: :@ :# ::::: :::::::: :::
     |               :: @ :: :: @: :: :@ :# ::::: :::::::: :::::
     |             :::: @ :: :: @: :: :@ :# ::::: :::::::: ::::::
     |            :: :: @ :: :: @: :: :@ :# ::::: :::::::: :::::::::
     |         ::::: :: @ :: :: @: :: :@ :# ::::: :::::::: :::::::: ::
     |       :::: :: :: @ :: :: @: :: :@ :# ::::: :::::::: :::::::: :::
     |      :: :: :: :: @ :: :: @: :: :@ :# ::::: :::::::: :::::::: :::::
     |    @@:: :: :: :: @ :: :: @: :: :@ :# ::::: :::::::: :::::::: :::::::
     |   :@ :: :: :: :: @ :: :: @: :: :@ :# ::::: :::::::: :::::::: :::::::::
   0 +----------------------------------------------------------------------->MB
     0                                                                   3.010

Number of snapshots: 57
 Detailed snapshots: [3, 12, 17, 22, 24 (peak), 56]
```

the time unit is bytes, due to the use of --time-unit=B.

Massif tool



```
--------------------------------------------------------------------------------
  n        time(B)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
  4        269,336          269,336          268,288         1,048            0
  5        335,128          335,128          333,824         1,304            0
  6        400,920          400,920          399,360         1,560            0
  7        466,712          466,712          464,896         1,816            0
  8        532,504          532,504          530,432         2,072            0
  9        598,296          598,296          595,968         2,328            0
 10        664,088          664,088          661,504         2,584            0
 11        729,880          729,880          727,040         2,840            0
 12        820,344          820,344          817,152         3,192            0
99.61% (817,152B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->99.11% (813,056B) 0x400978: alloc_matrix (mmul.c:53)
| ->63.91% (524,288B) 0x40074C: main (mmul.c:20)
| |
| ->35.20% (288,768B) 0x400756: main (mmul.c:21)
|
->00.50% (4,096B) in 1+ places, all below ms_print's threshold (01.00%)
```

- **n**: number of snapshot
- **time(B)**: the time unit is bytes, due to the use of --time-unit=B.
- **total(B)**: the total memory consumption at that point.
- **useful-heap(B)**: the number of useful heap bytes allocated at that point.
- **extra-heap(B)**: the number of extra heap bytes allocated at that point.
- **stacks(B)**: the size of the stack(s). By default, stack profiling is off as it slows Massif down greatly.

Valgrind – how it works

- Valgrind is compiled into a shared object, valgrind.so. The shell script valgrind sets the LD_PRELOAD environment variable to point to valgrind.so. This causes the .so to be loaded as an extra library to any subsequently executed dynamically linked ELF binary
- The dynamic linker allows each .so in the process image to have an initialization function which is run before main(). It also allows each .so to have a finalization function run after main() exits
- When valgrind.so's initialization function is called by the dynamic linker, the synthetic CPU starts up. The real CPU remains locked in valgrind.so until the end of the run
- System calls are intercepted; Signal handlers are monitored

Valgrind Summary

- Valgrind will save hours of debugging time
- Valgrind can help speed up your programs
- Valgrind runs on x86-Linux
- Valgrind works with programs written in any language
  - Valgrind is actively maintained
- Valgrind can be used with other tools (gdb)
- Valgrind is easy to use
  - uses dynamic binary translation, so no need to modify, recompile, or re-link applications. Just prefix the command line with valgrind and everything works
- Valgrind is not a toy
  - Used by large projects : 25 millions lines of code
- Valgrind is free

PAPI

- Library that provides a consistent interface (and methodology) for hardware performance counters, found across the system:
    - CPUs, GPUs, on-chip and off-chip Memory, Interconnects, I/O system, File System, Energy/Power, etc.

- PAPI (Performance Application Programming Interface) enables software engineers to see, in near real-time, the relation between SW performance and HW events across the entire system

- Supported Architectures: AMD, ARM, IBM, Intel, NVIDIA
- https://github.com/icl-utk-edu/papi

PAPI Hardware Events

- Countable events are defined in two ways:
  - Platform-neutral Preset Events (e.g., PAPI_TOT_INS)
  - Platform-dependent Native Events (e.g., L3_CACHE_MISS)

- Preset Events can be derived from multiple Native Events
  - PAPI_L1_TCM might be the sum of L1 Data Misses and L1 Instruction Misses on a given platform

- *papi_avail* to see what preset events are available on a given platform
  - Any event countable by the CPU, GPU, network card, parallel file system or others
- *papi_native_avail* utility to see all available native events

PAPI High-Level API

- PAPI_hl_region_begin (const char *region)
  - Read events at the beginning of a region (also start counting the events)
- PAPI_hl_region_end (const char *region)
  - Read events at the end of a region and store the difference from the beginning
- PAPI_hl_read (const char *region)
  - Read events inside a region and store the difference from the beginning
- PAPI_hl_stop ()
  - Stop a running high level event set (optional)

- Environment variable to control events
  - `export PAPI_EVENTS="PAPI_DP_OPS,PAPI_L1_DCM,PAPI_L1_DCA"`

PAPI High-Level API Example

- `% export PAPI_EVENTS="PAPI_TOT_INS,PAPI_TOT_CYC"`

```
#include "papi.h"
int main()
{
  int retval;
  retval= PAPI_hl_region_begin("computation");
  if ( retval!= PAPI_OK )
    handle_error(1);

  /* Do some computation here */
  retval= PAPI_hl_region_end("computation");
  if ( retval!= PAPI_OK )
    handle_error(1);
}
```

Automatic performance report.

```
{
"computation":{
"region_count":"1",
"cycles":"2080863768",
"PAPI_TOT_INS":"2917520595",
"PAPI_TOT_CYC":"2064112930" }
}
```

Perf

- Perf is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command-line interface.
- Perf is based on the perf_events interface exported by recent versions of the Linux kernel.
- The perf tool offers a rich set of commands to collect and analyze performance and trace data.
    - it implements a set of commands: stat, record, report, [...]

```
$ perf
usage: perf [--version] [--help] [OPTIONS] COMMAND [ARGS]
The most commonly used perf commands are:
…
```

Events

- The perf tool supports a list of measurable events. The tool and underlying kernel interface can measure events coming from different sources.
- For instance, some events are pure kernel counters, in this case, they are called software events.
  - Examples: context-switches, minor-faults.
- Another source of events is the processor itself and its Performance Monitoring Unit (PMU). It provides a list of events to measure micro-architectural events
  - Examples: number of cycles, instructions retired, L1 cache misses, and so on. Those events are called PMU hardware events or hardware events for short. They vary with each processor type and model.

```
$ perf list List of pre-defined events (to be used in -e):
…
```

Workflow

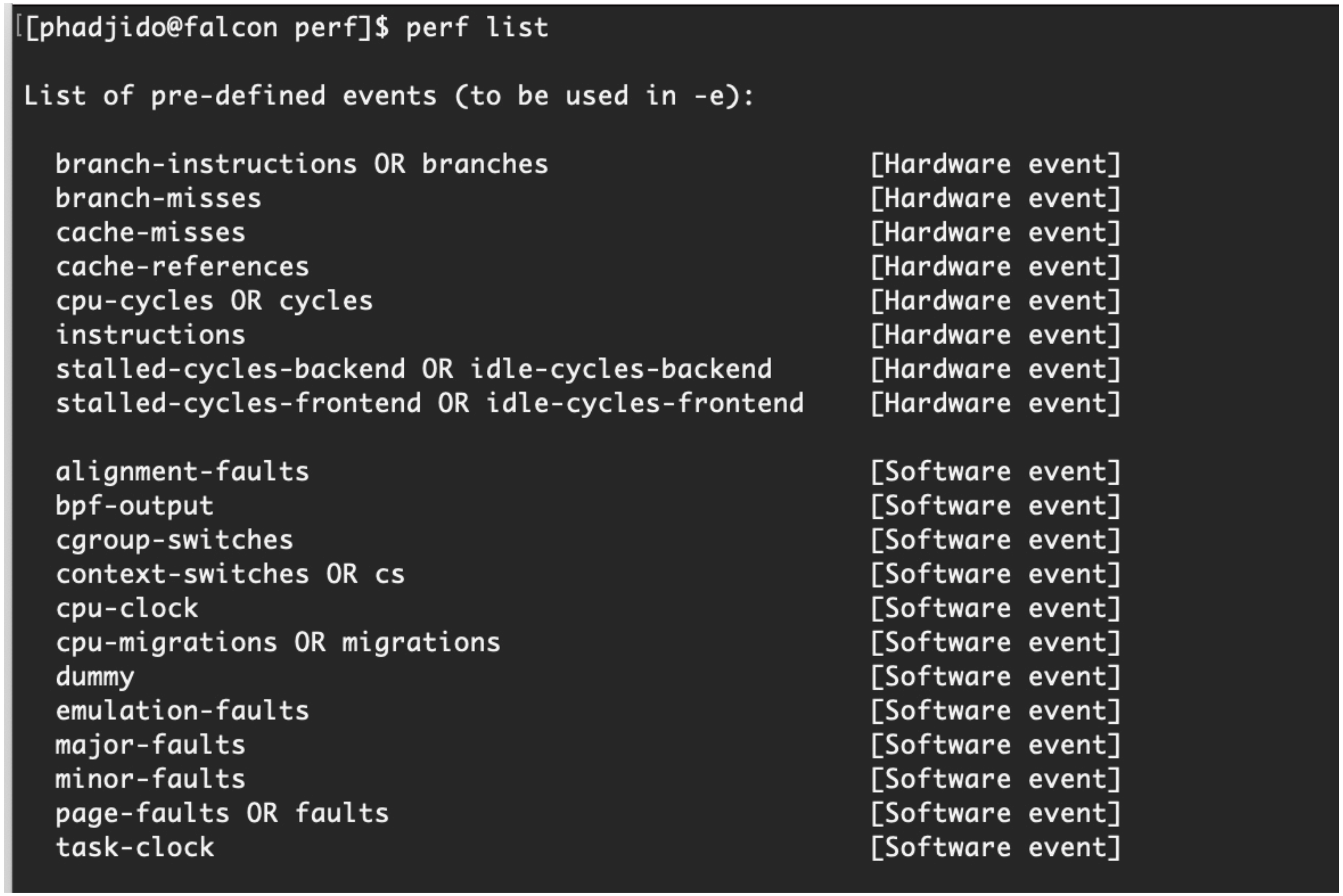
- When profiling a CPU with the perf command, the typical workflow is to use:

1. **perf list**: find events.
2. **perf stat**: count the events.
3. **perf record**: write events to a file.
4. **perf report**: browse the recorded file.
5. **perf script**: dump events after processing.

- The outputs differ based on the system and locally available resources.

perf list

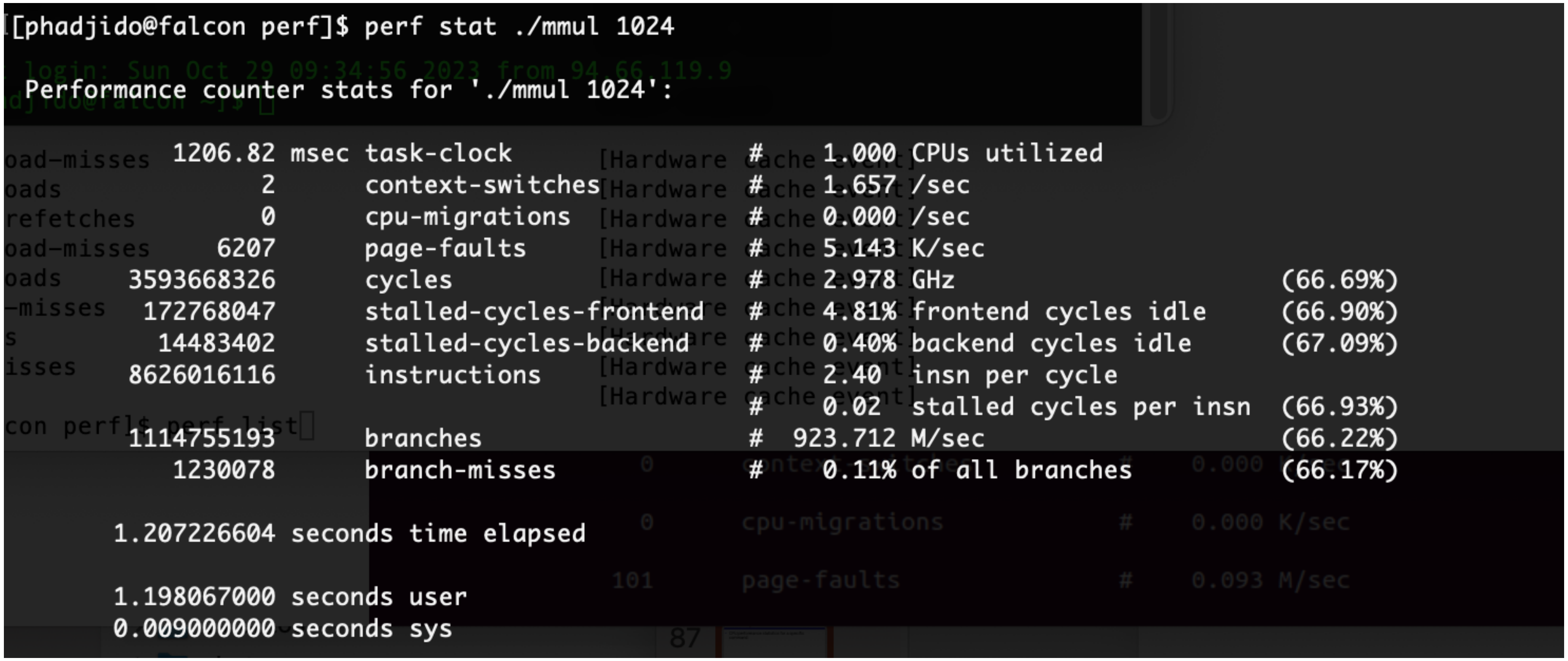
- The output lists all supported events, regardless of type.



```
[[phadjido@falcon perf]$ perf list

List of pre-defined events (to be used in -e):

  branch-instructions OR branches                    [Hardware event]
  branch-misses                                      [Hardware event]
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  stalled-cycles-backend OR idle-cycles-backend      [Hardware event]
  stalled-cycles-frontend OR idle-cycles-frontend    [Hardware event]

  alignment-faults                                   [Software event]
  bpf-output                                         [Software event]
  cgroup-switches                                    [Software event]
  context-switches OR cs                             [Software event]
  cpu-clock                                          [Software event]
  cpu-migrations OR migrations                       [Software event]
  dummy                                              [Software event]
  emulation-faults                                   [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
  task-clock                                         [Software event]
```

perf stat <command>

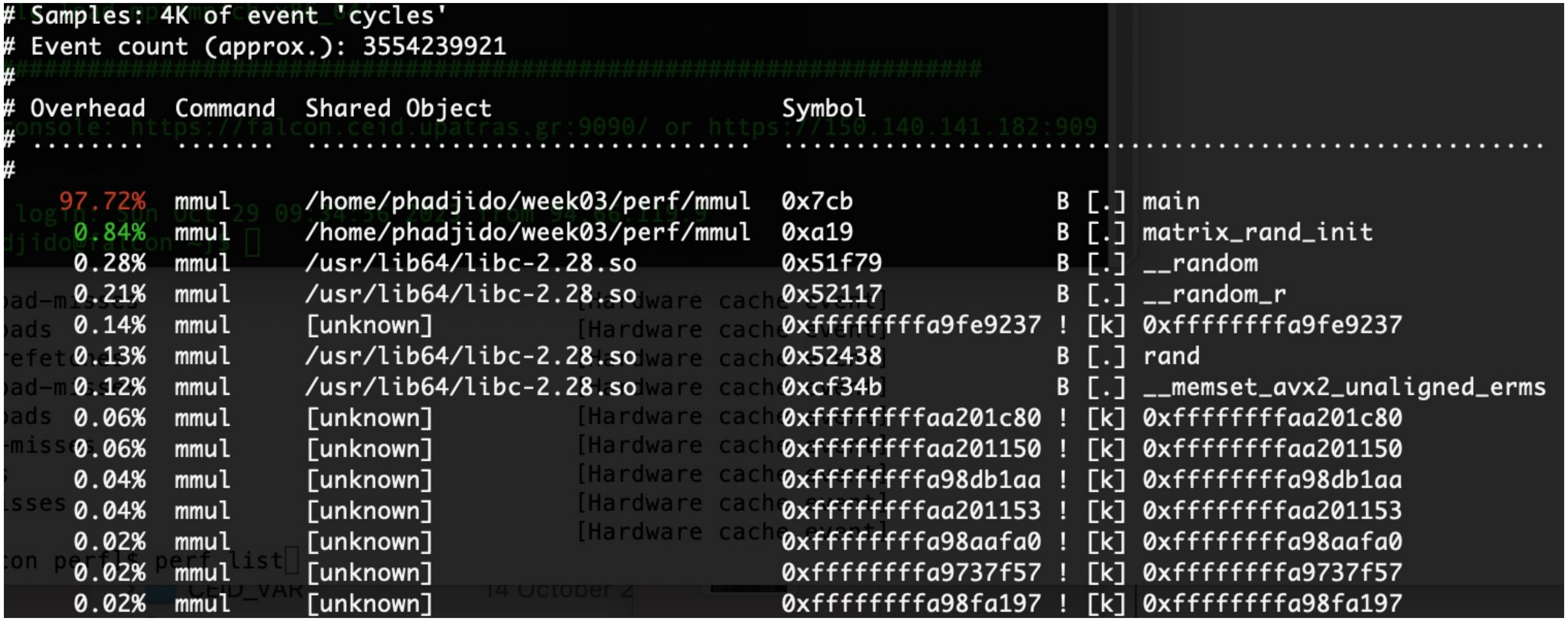
- CPU performance statistics for a specific command.



```
[phadjido@falcon perf]$ perf stat ./mmul 1024

 Performance counter stats for './mmul 1024':

         1206.82 msec task-clock                #    1.000 CPUs utilized
               2      context-switches          #    1.657 /sec
               0      cpu-migrations            #    0.000 /sec
            6207      page-faults               #    5.143 K/sec
      3593668326      cycles                    #    2.978 GHz                      (66.69%)
       172768047      stalled-cycles-frontend   #    4.81% frontend cycles idle     (66.90%)
        14483402      stalled-cycles-backend    #    0.40% backend cycles idle      (67.09%)
      8626016116      instructions              #    2.40  insn per cycle
                                                #    0.02  stalled cycles per insn  (66.93%)
      1114755193      branches                  #  923.712 M/sec                    (66.22%)
         1230078      branch-misses             #    0.11% of all branches          (66.17%)

     1.207226604 seconds time elapsed

     1.198067000 seconds user
     0.009000000 seconds sys
```

perf record & report

- $ perf record ./mmul 1024
- $ perf report --stdio -v



```
# Samples: 4K of event 'cycles'
# Event count (approx.): 3554239921
#
# Overhead  Command  Shared Object                    Symbol
# ........  .......  ...............................  .......................................................
#
    97.72%  mmul     /home/phadjido/week03/perf/mmul  0x7cb                B [.] main
     0.84%  mmul     /home/phadjido/week03/perf/mmul  0xa19                B [.] matrix_rand_init
     0.28%  mmul     /usr/lib64/libc-2.28.so          0x51f79              B [.] __random
     0.21%  mmul     /usr/lib64/libc-2.28.so          0x52117              B [.] __random_r
     0.14%  mmul     [unknown]                        0xffffffffa9fe9237   ! [k] 0xffffffffa9fe9237
     0.13%  mmul     /usr/lib64/libc-2.28.so          0x52438              B [.] rand
     0.12%  mmul     /usr/lib64/libc-2.28.so          0xcf34b              B [.] __memset_avx2_unaligned_erms
     0.06%  mmul     [unknown]                        0xffffffffaa201c80   ! [k] 0xffffffffaa201c80
     0.06%  mmul     [unknown]                        0xffffffffaa201150   ! [k] 0xffffffffaa201150
     0.04%  mmul     [unknown]                        0xffffffffa98db1aa   ! [k] 0xffffffffa98db1aa
     0.04%  mmul     [unknown]                        0xffffffffaa201153   ! [k] 0xffffffffaa201153
     0.02%  mmul     [unknown]                        0xffffffffa98aafa0   ! [k] 0xffffffffa98aafa0
     0.02%  mmul     [unknown]                        0xffffffffa9737f57   ! [k] 0xffffffffa9737f57
     0.02%  mmul     [unknown]                        0xffffffffa98fa197   ! [k] 0xffffffffa98fa197
```