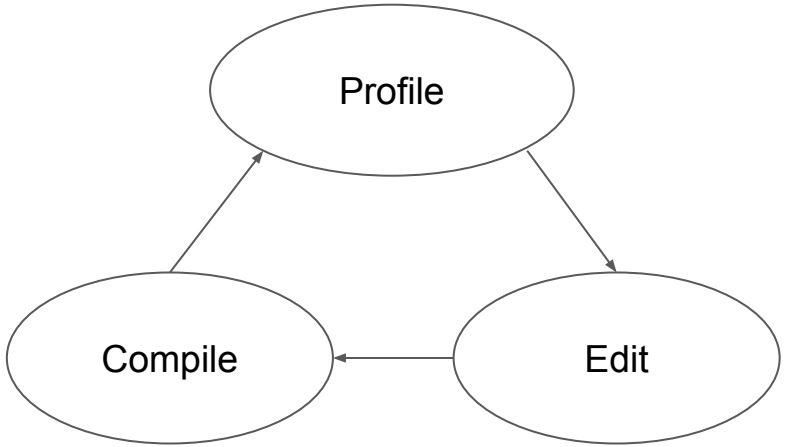
gprof

Presented by Benjamin Black

What do we want out of a profiler?

Profiling

- Compile your code
- Run it on some input, collect data
- Summarize the data so that the user understands how to improve their code



Profiling procedure count

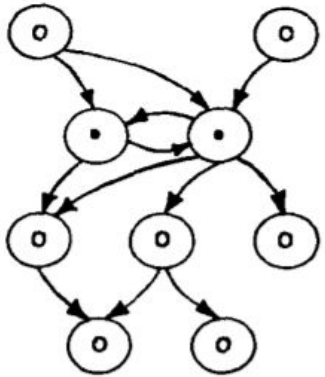
- Is a procedure executed too many times?
  - Is the algorithm choice appropriate?
  - Is the algorithm implemented correctly?
    - n*log(n) vs n^2 quicksort
  - Are there unneeded duplicative calls?
    - 2n vs n calls
- Debugging
  - Is a procedure executed at all?
    - Do you expect that code to be inactive?
    - Do you expect that code to be active?

Profiling timing (two types)

- Time spent inside the procedure itself (not including child calls)
  - "self" time
  - Useful to find vital inner loops in your code
- Duration of the procedure
  - "cumulative" time
  - Useful to see which procedures take up the most total time and should be the focus of your optimization

Call graph

- Your procedure is expensive. Which child procedure that it calls is the most expensive?
- Also useful for implementing profiling techniques efficiently (more on this later)



Implementing gprof

Two types of profiling techniques

- Instrumentation
  - User code logs information as it runs
    - In gprof, logs caller/callee counts are logged this way
- Sampling
  - Monitoring code checks program state at certain intervals
    - In gprof, self time is measured this way

Implementing instrumentation

1. Gprof tells compiler inserts calls to monitoring procedure in function prologue
2. Monitoring procedure investigates return address to find caller function, getting that edge on the graph

```
void foo(){
    monitor(); // inserted by gprof
    …
    // user code
    …
}
void bar(){
     foo();
}
```

Implementing sampling

- Uses an alarm signal to interrupt code at time intervals
- When signal is activated, it increments time counter for the function the program counter is currently in

```
0000000000001e30 <foo>:
    1f27:       mov    %rax,0x28(%rsp)
    1f2c:       xor    %eax,%eax
    1f2e:       mov    0x80(%rsp),%rax

0000000000002100 <bar>:
    2136:       mov    0x78(%rsp),%rcx
    213b:       test   %rax,%rax
 →  213e:       mov    %rcx,0x18(%rsp)
    2143:       mov    %rax,0x10(%rsp)
```

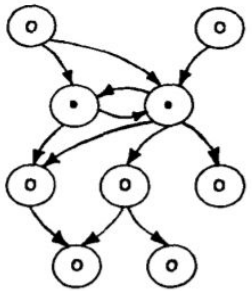
Separate processing from collecting

- Gprof stores collected data from a run in a file
  - Caller-callee count histogram
  - Time samples function histogram
- Gprof's analysis tool can process files from multiple runs to compare or combine them

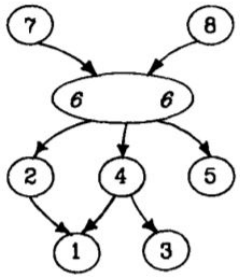
Computing cumulative time from self-time

- Assumes all function calls are the same duration
- Assuming no recursion in program can use formula:
- With recursion in program, can look analyze time spent in strongly connected components as a single node when doing the calculation
- Problem with assumption: What if `foo` makes many quick calls and `bar` makes a few slow calls to a function?

$$T_r = S_r + \sum_{r \text{ CALLS } e} T_e \times \frac{C_e^r}{C_e}$$



Cycle to be collapsed.
Figure 2.



Topological numbering after cycle collapsing.
Figure 3.

Data presentation

Flat profile

- Gets information for each function

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 84.04      0.85     0.85        1   848.84   848.84  yet_another_test
  6.00      0.91     0.06        1    60.63   909.47  test
  1.00      0.92     0.01        1    10.11    10.11  some_other_test
  0.00      0.92     0.00        1     0.00   848.84  another_test
```

Call graph profile

- Which calls are taken within each function?
- Allows you to easily navigate the expensive path through deep call stacks

```
                     Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 1.09% of 0.92 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.92                 main [1]
                0.06    0.85       1/1           test [2]
                0.01    0.00       1/1           some_other_test [5]
-----------------------------------------------
                0.06    0.85       1/1           main [1]
[2]     98.9    0.06    0.85       1         test [2]
                0.00    0.85       1/1           another_test [3]
-----------------------------------------------
                0.00    0.85       1/1           test [2]
[3]     92.3    0.00    0.85       1         another_test [3]
                0.85    0.00       1/1           yet_another_test [4]
-----------------------------------------------
                0.85    0.00       1/1           another_test [3]
[4]     92.3    0.85    0.00       1         yet_another_test [4]
-----------------------------------------------
                0.01    0.00       1/1           main [1]
[5]      1.1    0.01    0.00       1         some_other_test [5]
-----------------------------------------------
```

Stated Limitations

- Accuracy in programs where main computation happens in large recursive cycles is poor (due to assumption stated earlier)

Actual limitations/Obsolescence

- Gprof is not widely used anymore (published in 1982)
  - Significant overhead in monitoring (leads to inaccurate results)
  - Recompilation necessary
- poor man's profiler
  - GDB's interrupt https://poormansprofiler.org/