CSE 303: Concepts and Tools for Software Development

Dan Grossman
Spring 2005
Lecture 17—Profilers, e.g., gprof
Profilers

A profiler monitors and reports (performance) information about a program execution.

They are useful for "debugging correct programs" by learning where programs consume most time and/or space.

"90/10 rule of programs" (and often worse for new programs) – a profiler helps you "find the 10".

But: The tool can be misused and misleading.
What profilers tell you

Different profilers profile different things.

gprof, a profiler for code produced by gcc is widely available and pretty typical:

- **Call counts**: # of times each function $a$ calls each function $b$
  - And the simpler fact: # of times $a$ was called

- **Time samples**: # of times the program was executing $a$ when “the profiler woke up to check where the program was”.

Neither is quite what you want (as we’ll see later), but they’re semi-easy and semi-quick to do:

- **Call counts**: Add code to every function call to update a table indexed by function pairs.

- **Time samples**: Use the processor’s timer; wake up and see where the program is.
Using gprof

• Compile with -pg
  – When you create the .o (for call counts)
  – When you create the executable (for time samples)
• Run the program (creates (overwrites) gmon.out)
• Run gprof (on gmon.out) to get human-readable results.
• Read the results (takes a little getting used to).
Getting useful info

• The information depends on your inputs! (Always know what you’re profiling)

• Statistical sampling requires a reasonable number of samples
  – Probably want at very least a few thousand
  – Can run a program over and over and use gprof -s (learn on your own; write a shell-script)

• Make sure performance matters
  – Is 10% faster worth uglier or buggier code?
  – Do you have better things to do (documentation, testing, ...)?
Performance tuning

• Never tune until you know the bottleneck (that’s what gprof is for, but it doesn’t tell you how to tune).

• Never overtune to some inputs at the expense of others.

• Always focus on the overall algorithm first.

• Think doubly-hard about making non-modular changes.

• Focus on low-level tricks only if you really need to ($< 5$ times in your career?)

Note: Performance tuning a library is harder because you want to do well for “unknown programs and inputs”.
Our example

- Different bottlenecks for large array-size and large max-number!!
  - If you knew max-number could never be more than 10, would you optimize is_prime?

- Optimal algorithm for is_prime is slower than for find_largest, but we did not write the optimal algorithms!

- After fixing time for find_largest, we still had a stack overflow.

- Changing the is_prime algorithm helped a lot.

- Little things (e.g., setting the largest prime to “not prime”) generally “lost in the noise”.

- Output affects wall-clock time.

Note: For more rigorous comparisons, we should not randomly seed the random-number generator.
Misleading Fact #1

Cumulative times are based on call estimation. They can be really, really wrong, but usually aren’t.

```c
int g = 0;
void c(int i) {
    if(i) return;
    for(; i < 100000000; ++i)
        ++g;
}
void a() { c(0); }
void b() { c(1); }
int main(int argc, char**argv) { a(); b(); return 0; }
```

Conclusion: You must understand what your profiler measures and what it presents to you. gprof doesn’t lie (if you read the manual)
Misleading Fact #2

*Sampling errors* (for time samples) can be caused by too few samples, or by *periodic sampling*

```c
void a() { /* takes 0.09 s */ }
void b() { /* takes 0.01 s */ }
int main(int argc, char**argv) {
    for(; i < 10000; ++i) {
        a();
        b();
    }
}
```

This probably doesn’t happen much and better profilers can use *random intervals* to avoid it.

Related fact: Measurement code changes timing (an uncertainty principle).
Poor man’s profiling

The time command is more useful because no measurement overhead, but less useful because you get only whole-program numbers.

- real: roughly “wall-clock”
- user: time spent running the code in the program
- system: time the O/S spent doing things on behalf of the program

Not precise for small numbers

Misleading Fact #3: gprof does not measure system time?

Effects on real time: Machine load, disk access, I/O

Effects on system time: I/O to screen, file, or /dev/null