Profiling

CMSC 257: Computer Systems

Instructor:

Preetam Ghosh (pghosh@vcu.edu)

Program Performance

- Programs run only as well as the code you write
- Poor code often runs poorly
    - Crashes or generates incorrect output (bugs)
    - Is laggy, jittery or slow (inefficient code)
        - Too slow on real inputs (data processing)
        - Not-reactive enough to be usable (interfaces)

Optimization

- Optimization is the process where you take an existing program and alter it to remove inefficiencies.

    - Change algorithms
    - Restructure code
    - Redesign data structures
    - Refactor code

Example Inefficient Program

```c
/* Program: profiling */
#include <stdio.h>

int add_example_one(int a, int b) {
  int out, i, j;
  out = 0;
  for (i=0; i<a; i++) {
    out ++;
  }
  for (j=0; j<b; j++) {
    out ++;
  }
  return(out);
}

int add_example_two(int a, int b) {
  return( a+b );
}

int main(void) {
  int i, x, y;
  for (i=0; i<10000; i++) {
    x = add_example_one(10000, 20000);
    y = add_example_two(10000, 20000);
  }
  return(0);
}
```

- Q: Which function is going to run faster and why?

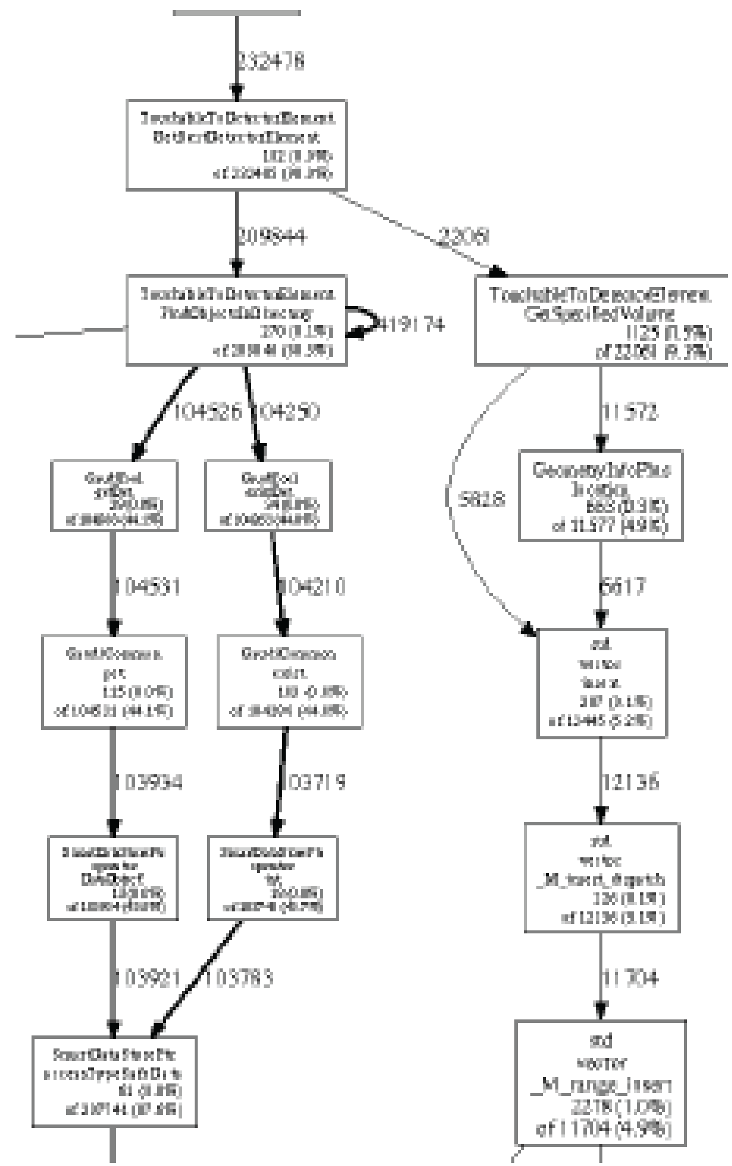
Profiling

- Debugging helps the programmer find and fix bugs …
- *Profiling helps the programmer find inefficiencies*
  - Profiling involves running a version of the program instrumented with code to measure how much time is spent in certain areas of the code.
  - How much time in each of the modules of the program?

gprof

- gprof is a utility that measures a program's performance and behavior.

- This produces *non-interactive* profile output
- The output provides detail
  - The time that the program ran, and time for each function
    - Statistics and detail on "*performance*"
  - Which functions called other functions and how many times
    - Statistics and detail on the "*call graph*"



Running gprof

1. First compile program using the “-pg” flag:

   $ gcc -pg profiling.c –o profiling

2. Run the program (will generate file gmon.out):

   $ ./profiling

3. Run gprof with the named program

   $ gprof profiling | less

4. Review the output

5. Optimize the program, re-profile
6. GOTO step#1

Gprof (flat profile)

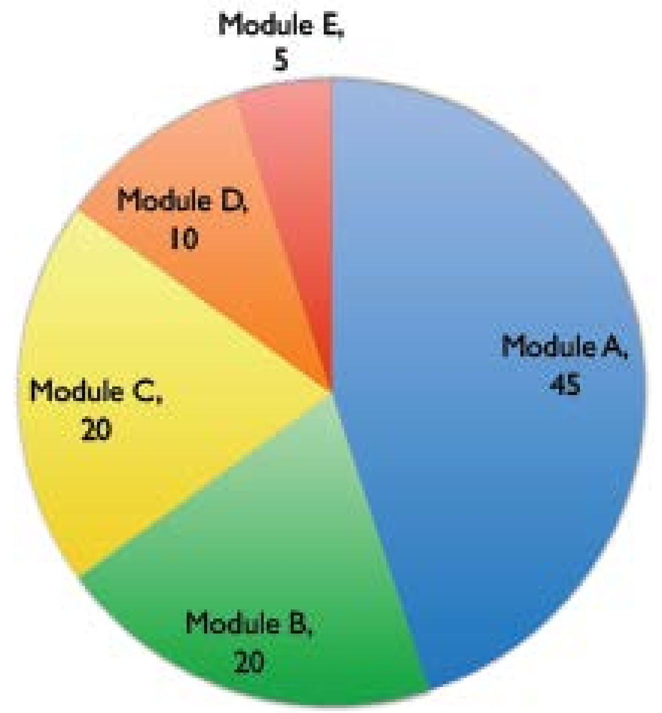
```
$ gprof profiling
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
101.15      0.47     0.47    10000    46.53    46.53  add_example_one
  0.00      0.47     0.00    10000     0.00     0.00  add_example_two

 %          the percentage of the total running time of the
time        program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self       the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
           else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
           function is profiled, else blank.
...
```

Gprof (Call Graph)

```
...
Call graph (explanation follows)

index % time    self  children    called     name
                0.47    0.00   10000/10000       main [2]
[1]    100.0    0.47    0.00   10000         add_example_one [1]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00    0.47                 main [2]
                0.47    0.00   10000/10000       add_example_one [1]
                0.00    0.00   10000/10000       add_example_two [3]
-----------------------------------------------
                0.00    0.00   10000/10000       main [2]
[3]      0.0    0.00    0.00   10000         add_example_two [3]
-----------------------------------------------

 This table describes the call tree of the program, and was sorted by
 the total amount of time spent in each function and its children.

 Each line lists:

     % time     This is the percentage of the `total' time that was spent
                in this function and its children.
     self       This is the total amount of time spent in this function.

     children   This is the total amount of time propagated into this
                function by its children.

     called     This is the number of times the function was called.
                If the function called itself recursively, the number
                only includes non-recursive calls, and is followed by
                a `+' and the number of recursive calls.
```

Optimization revisited …

- When optimizing, you focus on modules of the program which implement the features and processing of the program.
  - Which parts of the program you select depends on what parts are running the most.
    - then, focus on those parts which take up the most time.
- Profiling tells us where to spend our time.



Amdahl's Law

- Amdahl's law models the maximum performance gain that can be expected by improving part of the system, i.e., what can we expect in terms of improvement.
- Consider
  - k is the percentage of total execution time spent in the optimized module(s).
  - s is the execution time expressed in terms of a n-factor speedup (2X, 3X…), which can be found as

$$\frac{original\ execution\ time}{improved\ execution\ time}$$

Amdahl's Law (cont.)

- The overall speedup T of the program is expressed :

$$T = \frac{1}{(1-k) + \frac{k}{s}}$$

- Intuition:
  - $1 - k$ is the part of the program taht unchanged
  - $\frac{k}{s}$ is the ratio of altered program size to speedup

Amdahl's Law (example)

- Assume that a module A of a program is optimized.
  - A represents 45% of the run time of the program.
  - The optimization reduces the runtime of module from 750ms to 50ms.

- What is the program speedup?

Amdahl's Law (example)

- Assume that a module A of a program is optimized.
  - A represents 45% of the run time of the program.
  - The optimization reduces the runtime of module from 750ms to 50ms.

$$s = 750/50 = 15$$

$$k = .45$$

$$T = \frac{1}{(1-.45)+\frac{.45}{15}} = \frac{1}{.55+.03} = 1.724$$

- What is the program speedup? (A: *1.724X*)