Some measures of behavior

- Profiling
- Code coverage

What is profiling?

- Allows you to learn:
  - where your program is spending its time
  - what function are called, and what other functions are doing the calling
- This can show you which pieces of your program are slower than you expected
  - Such piece might be candidates for rewriting
- Are functions called more or less often than expected?
  - This may help you spot bugs that had otherwise been unnoticed

Profilers

- Profilers use information collected during the **actual execution** of a program
- Can be used on programs that are too large or too complex to analyze by reading the source
- How your program is executed affects the information that shows up in the profile data
  - If you do not use some feature of your program while it is being profiled, then no profile information will be generated for that feature!
  - Some insights gleaned from white-box testing (i.e., testing informed by tested code's structure) are helpful here.

Available Profilers

- **gprof**
- PGI pgprof
- TAU
- Allinea OPT
- profilers built into IDEs
- research tools
- others.....

Profiler steps

- The exact steps may very from program to program
- We'll concentrate on **gprof**
- Steps:
  1. Compile and link your program with profiling enabled
  2. Execute your program to generate a profile data file
  3. Run gprof to analyze the profile data
- Output produce (beside regular program side effects):
  a) Flat profile
  b) Call graph

a) Flat Profile

- Time your program spent in each function
- How many times that function was called
- Clearly indicated here is information on which functions burn most of the CPU cycles

b) Call Graph

- Shows, for each function **foo()**:
  - which functions called foo(),
  - which other functions foo() called,
  - how many times foo() was called.
- Also: An estimate of how much time was spent in the subroutines of each function
- Bottom-line: Suggests places where you might try to reduce time overhead in function calls that appear to use a lot of time.

Compiling Applications

- Need to recompile application with profile support enabled
- Often this option is '-pg'
  - But check compiler documentation to be sure!
  - Sometimes the IDE takes care of this detail for you
- All modules in the program to be profiled must be compiled with the appropriate option

Running Applications

- After recompiling with profile support enabled, need to run the (newly instrumented) program
- Application is run as usual, with usual arguments and data
  - Choice of arguments and data can have a dramatic effect on profile information
  - Therefore choice must be carefully made
- Program will run somewhat slower than normal
  - time spent collecting and the writing the profile data

Post Execution

- Application on normal exit will write the profile data into a metadata file
  - In the case of gprof, the file called 'gmon.out'
  - If there is already a file called 'gmon.out' its contents are overwritten
- Gprof gotcha: Cannot specify any other file name at run time
  - But you can rename the file afterward if you are concerned that it may be overwritten
  - This would normally happen if you want to obtain different profiles based on different arguments & data
- 'gmon.out' written to current directory

Analysis Phase

- Again focusing on gprof
  - After 'gmon.out' is generated, execute **gprof** to interpret the information
  - gprof prints both the flat profile and the call graph to the screen
  - redirect the output of gprof into a file with the shell redirection command`>'

gprof Command Summary

- gprof options [executable-file [profile-data-files...]] [> outfile]

- 'man gprof' for a list of options

Sample Flat Profile Output

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 14.60      4.30     4.30     1024     0.00     0.00  savfz_
 13.82      8.37     4.07     1024     0.00     0.00  eysfld_
 13.58     12.37     4.00     1024     0.00     0.00  liaobc_
 13.10     16.23     3.86     1024     0.00     0.00  ezsfld_
 12.76     19.99     3.76     1024     0.00     0.00  exsfld_
 10.32     23.03     3.04     1024     0.00     0.00  hzsfld_
 10.22     26.04     3.01     1024     0.00     0.00  hxsfld_
 10.18     29.04     3.00     1024     0.00     0.00  hysfld_
  1.29     29.42     0.38                             __fmth_i_exp
  0.07     29.44     0.02        1     0.02     0.02  build_
  0.04     29.45     0.01                             __fvdexp
  0.03     29.46     0.01                0.01     0.01  zero_
  0.00     29.46     0.00    30976     0.00     0.00  dcube_
  0.00     29.46     0.00     5120     0.00     0.00  finc_
  0.00     29.46     0.00        1     0.00    29.07  MAIN_
  0.00     29.46     0.00        1     0.00     0.00  farout_
  0.00     29.46     0.00        1     0.00     0.00  setfz_
  0.00     29.46     0.00        1     0.00     0.00  setup_
  0.00     29.46     0.00        1     0.00     0.00  stliao_
```

Interpreting the Output

- functions are sorted by decreasing run-time
- sampling period estimates the margin of error in each of the time figures
- Note:
  - A time figure that is not much larger than the sampling period is not necessarily reliable
  - function zero_ in previous slide
  - If elapsed time is zero or close to zero, then we can only profitably use the "calls" column

Column definitions

- % time
  - percentage of the total execution time program spent in this function
  - all functions combined should add up to 100%!
- cumulative seconds
  - This is the cumulative total number of seconds spent executing this function plus the time spent in all the functions above this one in this table
- self seconds
  - number of seconds accounted for by this function alone
  - flat profile listing is sorted first by this number
- calls
  - total number of times the function was called
  - if the function was never called, or the number of times it was called cannot be determined (probably because the function was not compiled with profiling enabled), the calls field is blank

Column definitions

- self ms/call
  - represents the average number of milliseconds spent in this function per call
  - if this function is not profiled this field is blank
- total ms/call
  - represents the average number of milliseconds spent in this function and its descendants per call
  - if this function is not profiled this field is blank
- name
  - name of the function
  - flat profile is sorted by this field alphabetically after the self seconds field is sorted

Call Graph

- This is another view of the program's execution
  - shows how much time was spent in each function and its children
  - can find functions that themselves may not use much time but call other functions that do use unusual amounts of time.
- Helps us when reasoning about the amount of time spent in a function
  - Is the time spent in the function's statements?
  - Or is the time spent in the subfunctions called by the function?

Call Graph Sample

```
granularity: each sample hit covers 64 byte(s) for 0.03% of 29.46 seconds

index % time    self  children    called     name
                0.00   29.07       1/1          main [2]
[1]     98.7    0.00   29.07       1          MAIN_ [1]
                4.30    0.00    1024/1024        savfz_ [3]
                4.07    0.00    1024/1024        eysfld_ [4]
                4.00    0.00    1024/1024        liaobc_ [5]
                3.86    0.00    1024/1024        ezsfld_ [6]
                3.76    0.00    1024/1024        exsfld_ [7]
                3.04    0.00    1024/1024        hzsfld_ [8]
                3.01    0.00    1024/1024        hxsfld_ [9]
                3.00    0.00    1024/1024        hysfld_ [10]
                0.02    0.00       1/1           build_ [12]
                0.01    0.00       1/1           zero_ [14]
                0.00    0.00       1/1           setup_ [19]
                0.00    0.00       1/1           setfz_ [18]
                0.00    0.00       1/1           farout_ [17]
                0.00    0.00       1/1           stliao_ [20]
-----------------------------------------------
```

Reading the Call Graph

- the lines of full dashes divide the table into entries
  - one for each function
  - each entry has one or more lines
- for each entry, the primary line is the one that starts with an index number in square brackets
  - the end of this line says which function the entry is for
  - preceding lines in the entry describe the callers of this function
  - the following lines describe its subroutines (children)
- entries are sorted by time spent in the function and its subroutines

How a profiler works...

- The profiler changes how every function in your program is compiled
  - when the function is called...
  - ... there is a little bit of code inserted by the profiler that stashes away some information about where the function was called
- profiler can figure out what function called it, and can count how many times it was called.
  - '-pg' compiler option.
- regular program snapshots
  - maintains histogram of where the program counter happens to be every now and then
  - typically 100 times per second of run time
  - but the exact frequency may vary from system to system

Limits to profiling...

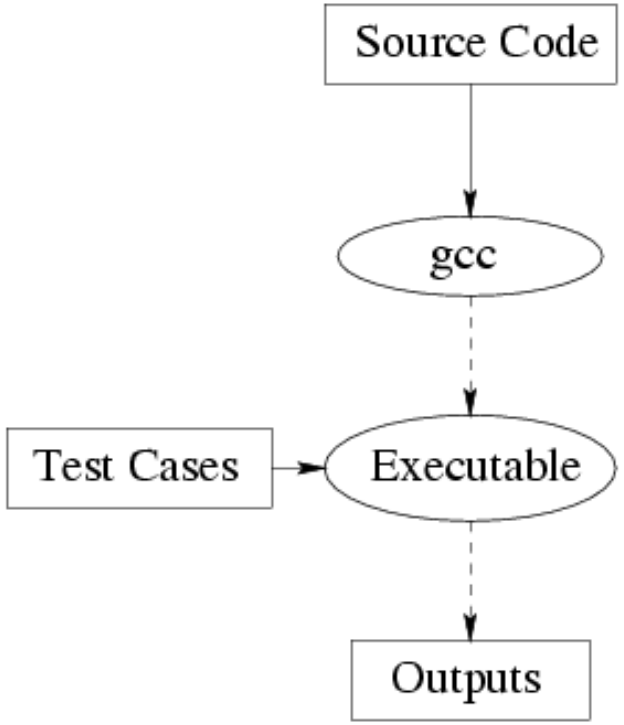
- gprof gives no indication of parts of your program that are limited by I/O or swapping bandwidth
  - samples of the program counter are taken at fixed intervals of run time
  - the time measurements in gprof output say nothing about time that your program was not running
- Example:
  - application runs slowly due to memory swapping…
  - …but gprof will say it uses little time, since time spent in swap routines is not counted

gcov: A Structural Testing Tool

- gcov: GNU Project coverage utility
  - Based on earlier utility tcov
  - GNU Project:
    - Supplies free software
    - Created gcc compiler for C
    - Most Linux utility programs come from GNU Project
- To use gcov (overview):
  - Compile your C program using gcc with special switches
  - Run your program normally
  - While running, your program will write information to special “log files”
  - After running test cases, run gcov
  - gcov will generate a coverage report listing which lines have been executed
- We will look at each of these steps in more detail

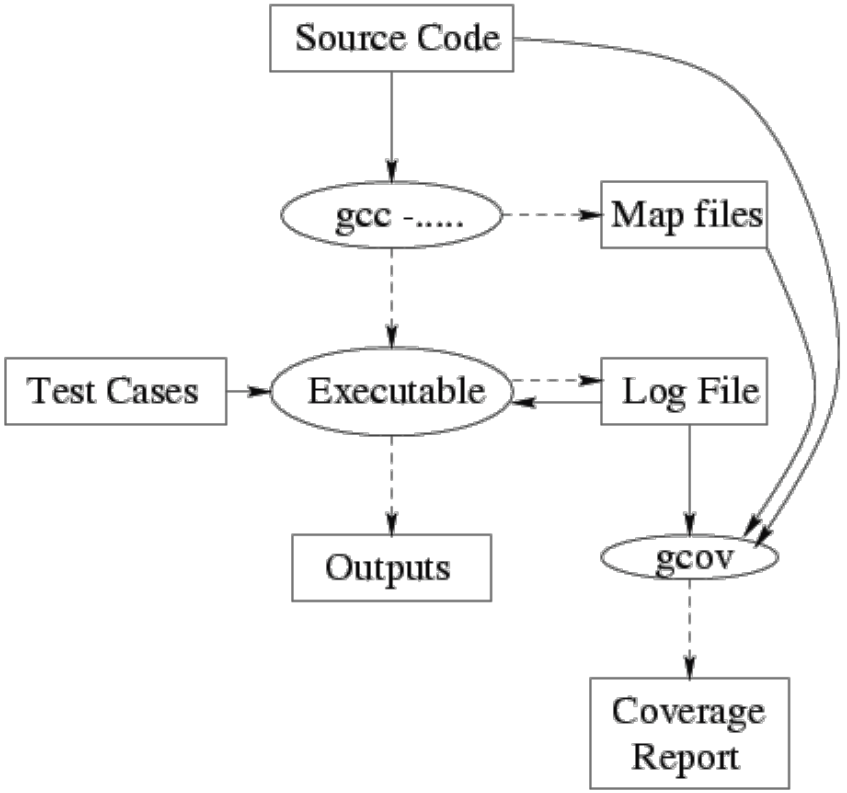
Normal Compilation of a Program

- In the above diagram,
  - Solid lines = inputs
  - Dashed lines = outputs
- Normally, we just:
  - Compile a program from source code
  - Execute it on some selected test cases
  - Look at the output for correctness



Compilation Using gcov

- Compilation / execution with gcov is basically the same
- However, extra switches to gcc cause extra things to happen
- Switches: -fprofile-arcs -ftest-coverage



Compilation Using gcov continued

- When we compile with the extra switches, gcc does the following
- extra things:
  - Generates “map file” of the blocks of code in our source files
  - Generates object code which counts the number of times each block of code has been executed
- “Block of code”:
  - Any sequence of statements such that executing one of them guarantees we must execute the next one
- Example (program wordcount.c):
  - Instead of just “gcc wordcount.c”, we say
    gcc -fprofile-arcs -ftest-coverage wordcount.c
  - gcc generates map file wordcount.gcno
  - Exact format of the “map file” is not important to us

Running a gcov-Compiled Program

- Run program as normal on test cases
- First time program is run, a coverage data file appears (with extension .gcda)
  - Coverage data file contains information about how many times each line of code has been executed
- Example:
  - After running wordcount, file wordcount.gcda is created
- Coverage data is updated every subsequent time program is run
- Coverage data is cumulative; e.g.
  - Line 42 executed 5 times on first run;
  - Line 42 executed 10 times on second run;
  - Therefore, coverage data stored in .gcda file shows 15 executions of line 42

Getting Information from gcov

- Run gcov with source file name as argument (e.g. gcov wordcount.c)
- gcov writes some statistics to screen, creates a file with extension ".gcov" (e.g. wordcount.c.gcov)
- .gcov file contains:
  - On right: source code
  - On left: number of times each line has been executed
  - If lines have never been executed, .gcov file shows ##### to highlight it

Example wordcount

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
main() {
  int nl=0, nw=0, nc=0;
  int inword = FALSE;
  char c;
  c = getchar();
  while (c != EOF) {
    ++nc;
    if (c == '\n')
      {++nl;}
    if (c==' ' || c=='\n' || c =='\t')
      {inword = FALSE;}
    else if (!inword)
      {inword = TRUE; ++nw;}
    c = getchar();
  }
  printf("%d lines, %d words, %d chars\n", nl, nw, nc);
}
```

wordcount Example:

- We compile with:
  - gcc -fprofile-arcs -ftest-coverage -o wordcount wordcount.c
- We get:
  - Executable in wordcount
  - System files wordcount.gcno
- We run wordcount
  - We give it as input two empty lines (two carriage returns)
- File wordcount.da is created
- We now run gcov wordcount.c
- gcov tells us:
  - 86.67% of 15 source lines executed in file wordcount.c
    Creating wordcount.c.gcov.

```
-:    0:Source:wordcount.c
        -:    0:Graph:wordcount.gcno
        -:    0:Data:wordcount.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:#define TRUE 1
        -:    3:#define FALSE 0
        1:    4:main() {
        1:    5:    int nl=0, nw=0, nc=0;
        1:    6:    int inword = FALSE;
        -:    7:    char c;
        1:    8:    c = getchar();
        -:    9:
        5:   10:    while (c != EOF) {
        3:   11:        ++nc;
        3:   12:        if (c == '\n') {
        3:   13:            ++nl;
        -:   14:        }
        -:   15:
        6:   16:        if (c==' ' || c=='\n' || c =='\t') {
        3:   17:            inword = FALSE;
    #####:   18:        } else if (!inword) {
    #####:   19:            inword = TRUE;
    #####:   20:            ++nw;
        -:   21:        }
        3:   22:        c = getchar();
        -:   23:    }
        1:   24:    printf("%d lines, %d words, %d chars\n", nl, nw, nc);
        -:   25:}
```

wordcount: The Output

- The two lines marked with # have never been executed
- The other lines have been executed
- For instance:
  - The initial c = getchar() has been executed once (1 in left margin)
  - The ++nc has been executed twice (2 in left margin)
    - This is because each carriage return counts as 1 character
- The declaration lines:
  - int nl=0, nw=0, nc=0;
  - int inword = FALSE;
- are marked as executed because they contain initialization code
- The declaration line
- char c;
- is not marked at all because it is just a declaration

wordcount Example: Carrying On

- We now run wordcount again, this time giving just the line “Zippy” as input
- We now run gcov wordcount.c
- gcov tells us:

  100.00% of 15 source lines executed in file wordcount.c

  Creating wordcount.c.gcov.

```
-:    0:Source:wordcount.c
        -:    0:Graph:wordcount.gcno
        -:    0:Data:wordcount.gcda
        -:    0:Runs:2
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:#define TRUE 1
        -:    3:#define FALSE 0
        2:    4:main() {
        2:    5:    int nl=0, nw=0, nc=0;
        2:    6:    int inword = FALSE;
        -:    7:    char c;
        2:    8:    c = getchar();
        -:    9:
       13:   10:    while (c != EOF) {
        9:   11:        ++nc;
        9:   12:        if (c == '\n') {
        4:   13:            ++nl;
        -:   14:        }
        -:   15:
       13:   16:        if (c==' ' || c=='\n' || c =='\t') {
        4:   17:            inword = FALSE;
        5:   18:        } else if (!inword) {
        1:   19:            inword = TRUE;
        1:   20:            ++nw;
        -:   21:        }
        9:   22:        c = getchar();
        -:   23:    }
        2:   24:    printf("%d lines, %d words, %d chars\n", nl, nw, nc);
        -:   25:}
```

wordcount Example: New Output

- We have given the system an input file of 6 characters on this run (5 letters + carriage return)
- Along with the 2 characters we gave on the previous run, the runs of wordcount have read 8 characters
- Hence, ++nc line has been executed 8 times in total
  - Counts on other lines have similarly been updated
- We have now executed the case where we have a non-whitespace
- character
  - Therefore, the lines
    else if (!inword)
      { inword = TRUE; ++nw; }
  - have been executed
  - Therefore, 100% of the lines of the program have now been executed

Colophon

- Lectures based on material in:
  - "Application Profiling" based on notes by Jeffrey Osier
  - "Structural Testing", CS 212a/b (University of Waterloo"