Lecture 26: Profiling and gprof

Kenneth M. Anderson
Software Methods and Tools
CSCI 3308 - Fall Semester, 2004

Two Problems

- Software often has performance problems
  - especially when handling a lot of data
  - or when placed in “real-time” situations
- Developers often think they know the cause of such problems
  - Without measuring the system at run-time, they make changes, recompile, and discover that the performance problems are still there
  - They often get caught up in “little” optimizations
    - making private data public; forcing the inlining of functions, decreasing the modularity of code, etc.

80/20 Rule

- These “little” optimizations often fail to have an effect due to the 80/20 rule
  - 80% of run-time is spent in about 20% of the code
  - So, you first need to find that 20% and focus your optimization efforts there
    - optimizing the other 80% of the code, will not provide much overall benefit (because that code is only rarely executed!)

Profiling

- In order to do this, we need some way of measuring our program’s execution
  - in particular we need to know how long each part of a program takes to execute
- Performance profiling offers two techniques for accomplishing this
  - Software Profiling
  - Hardware Profiling

Software Profiling

- A compiler adds statements to a program that take time measurements as it is running
  - Add statements to capture the current time at the beginning and end of a function
  - Subtract to calculate the time spent in the function
  - Add the time spent to a running total
  - At the end of the program, calculate the percentage of program time spent in the function by dividing its total time by the total execution time of the program
- Software profiling is less accurate because you are changing the program you are trying to measure, but it is easier to do

Hardware Profiling

- Measurements are taken with hardware
- Components are attached to the motherboard and take timing measurements without changing how the program is run

gprof

- gprof is an example of a software profiler. Its output is divided into two sections
  - Flat Profile
    - The total time taken by each function
  - Call Graph
    - describes the call graph of the program
    - It shows what functions were called by other functions, and how much time was taken by the children of a function
      - You can subtract the time taken by a function's children from its total time, to get its true time

Using gprof

- Using gprof is a three step process
- First, you must compile and load the program with the "-pg" command flag
- Second, you run the program…this generates a file called gmon.out
- Third, you invoke gprof with the command
  - gprof program gmon.out
- gprof prints the flat profile and call graph information to standard out; to save it use:
  - gprof program gmon.out > profiling-results

Example Flat Profile

```
granularity: each sample hit covers 4 byte(s) for 5.56% of 0.18 seconds

  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 83.3      0.15     0.15      203     0.74     0.88  _find_alphabetical_location
 16.7      0.18     0.03    15925     0.00     0.00  _stringToUpper
  0.0      0.18     0.00      204     0.00     0.00  _format_currency
  0.0      0.18     0.00      203     0.00     0.00  _calculate_wages
  0.0      0.18     0.00      203     0.00     0.89  _insert_alphabetical
  0.0      0.18     0.00        1     0.00     0.00  _free_queue
  0.0      0.18     0.00        1     0.00   180.00  _main
  0.0      0.18     0.00        1     0.00     0.00  _print_paycheck_summary
  0.0      0.18     0.00        1     0.00   180.00  _read_employees
```


Flat Profile Columns

- **% time** - the percentage of the total running time of the program used by this function.
- **cumulative seconds** - a running sum of the number of seconds accounted for by this function and those listed above it.
- **self seconds** - the number of seconds accounted for by this 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.

Flat Profile Columns, continued

- **self ms/call** - the average number of milliseconds spent in this function per call, if this function is profiled, else blank.
- **total ms/call** - the average number of milliseconds spent in this function and its descendents per call, if this function is profiled, else blank.
- **name** - the name of the function. This is the minor sort for this listing.

Call Graph (modified)

```
granularity: each sample hit covers 4 byte(s) for 5.56% of 0.18 seconds

                                  called/total       parents
index  %time    self descendents  called+self    name           index
                                  called/total       children

                0.00        0.18     203/203         _read_employees [3]
[1]    100.0    0.00        0.18     203         _insert_alphabetical [1]
                0.15        0.03     203/203         _find_alphabetical_locat
                0.00        0.00     808/15925       _stringToUpper [6]

                0.15        0.03     203/203         _insert_alphabetical [1]
[5]     99.2    0.15        0.03     203         _find_alphabetical_location
                0.03        0.00   15117/15925       _stringToUpper [6]

                0.00        0.00     808/15925       _insert_alphabetical [1]
                0.03        0.00   15117/15925       _find_alphabetical_locat
[6]     16.7    0.03        0.00   15925         _stringToUpper [6]
```

Call Graph Description

- Each entry in this table consists of several lines.
- The line with the index number at the left hand margin lists the current function.
- The lines above it list the functions that called this function, and the lines below it list the functions this one called.

Call Graph Columns (Function)

- **index** - A unique number given to each element of the table.
- **% time** - percentage of the “total” time that was spent in this function and its children.
- **self** - total amount of time spent in this function.
- **children** - total amount of time propagated into this function by its children.
- **called** - number of times the function was called (plus recursive calls)

Call Graph Columns (Parents)

- **self** - amount of time that was propagated directly from the function into this parent.
- **children** - amount of time propagated from the function's children into this parent.
- **called** - number of times this parent called the function / total number of times called.
- **name** - This is the name of the parent.
- If the parents of a function cannot be determined, the word `<spontaneous>' is printed in the `name' field

Call Graph Columns (Children)

- **self** - amount of time propagated directly from the child into the function.
- **children** - amount of time propagated from the child's children to the function.
- **called** - number of times the function called this child / total number of times the child was called.
- **name** - This is the name of the child.

Improving Performance

- When you have measured your code, how do you make it go faster?
- There are several ways to optimize
  - Changing algorithms
  - Caching values (especially strings!)
  - For graphics applications, reducing the amount of drawing per frame
  - and so on…

Demo: Graphics Application and gprof

- First Demo: consists of two versions of a simple graphics applications; the first version makes use of a dumb algorithm for updating the screen; the second is much smarter and much faster
- gprof - MacOS X (and other Unix variants) have gprof; I'll step through a quick demo of applying gprof to a C-based version of ezpay