>>

Performance Analysis

∧ >>

❖ Performance

Why do we care about performance?

Good performance → less hardware, happy users.

Bad performance → more hardware, unhappy users.

Generally: performance = execution time

Other measures: memory/disk space, network traffic, disk i/o.

Execution time can be measured in two ways:

- cpu ... time your program spends in the processor
- elapsed ... wall-clock time between start and finish

❖ ... Performance

In the (distant) past, performance was a significant problem

- much programming effort was spent on efficiency "tricks"

Unfortunately, there is usually a trade-off between ...

- execution efficiency achieved by "tweaking" code
- the understandability of the code

Knuth: "Premature optimization is the root of all evil"

<< ∧ >>

❖ Development Strategy

A pragmatic approach to efficiency:

- first, make the program simple, clear, robust and correct
- then, worry about efficiency ... if it's a problem at all

Points to note:

- good design is always critical
  (at design time, make sensible choice of data structures, algorithms)
- can handle efficiency (somewhat) at system level
  (e.g. buy a bigger machine, use compiler optimisation, ...)

Pike: "A fast program that gets the wrong answer saves no time."

<< ^ >>

❖ ... Development Strategy

Strategy for developing efficient programs:

1. Design the program well
2. Implement the program well (choose good algorithms)
3. Test the program well
4. Only after you're sure it's working, measure performance
5. If (and only if) performance is inadequate, find the "hot spots"
6. Tune the code to fix these
7. Repeat measure-analyse-tune cycle until performance ok

<< ^ >>

❖ Performance Analysis

Complexity/estimates give some idea of performance in advance.

Often, however ..

- assumptions made in estimating performance are invalid
- we overlook some frequent and/or expensive operation

Best way to evaluate performance: measure program execution.

Performance analysis can be:

- coarse-grained ... overview of performance characteristics
- fine-grained ... detailed description of performance

<< ^ >>

❖ ... Performance Analysis

Coarse-grained performance analysis

- devise a range of *representative* inputs
- measure execution time of program on each input
- can conveniently be combined with testing
  (but we only care about timing if correct result produced)

The Unix **`time`** command provides a suitable mechanism

```
$ time ./myProg < LargeInput > /dev/null
real   0m5.064s
user   0m4.113s
sys    0m0.802s
```

<< ^ >>

❖ ... Performance Analysis

Components of Unix **`time`** output:

`real` time

- elapsed time from when program starts to when it finishes
- affected by load on the system, not just the time spent by your program

`user` time

- time spent executing the code of your program (and std libraries)
- this is the important measure of your code's efficiency

`sys` time

- time spent doing system calls on behalf of your program
- could be large if e.g. substantial i/o, network usage

<< ∧ >>

❖ ... Performance Analysis

Using **`time`** for performance analysis

- run the program multiple times on the same data set
- take an average of just the **`user`** time
- include **`sys`** time as well, if it's significant
- repeat for several data sets of significantly different size

Note: on a very-lightly-loaded system **`user+sys`** ≅ **`real`**

Note also: some versions of **`time`** have a different output format

- but it's always(?) possible to identify **`user`** and **`sys`**

❖ ... Performance Analysis

Decades of empirical study of program execution have shown ...

The 90/10 rule generally holds (or 80/20 rule or ...):

- "90% of the execution time is spent in 10% of the code"

Implications:

- most of the code has little impact on overall performance
- small regions of the code are bottlenecks (aka hot-spots)

To significantly improve performance: make bottlenecks faster.

<< ^ >>

❖ Profiles

Need a method for locating hot spots

An *execution profile* for a program is

- the total cost of performing each code block
- for one execution of the program

Cost may be measured via

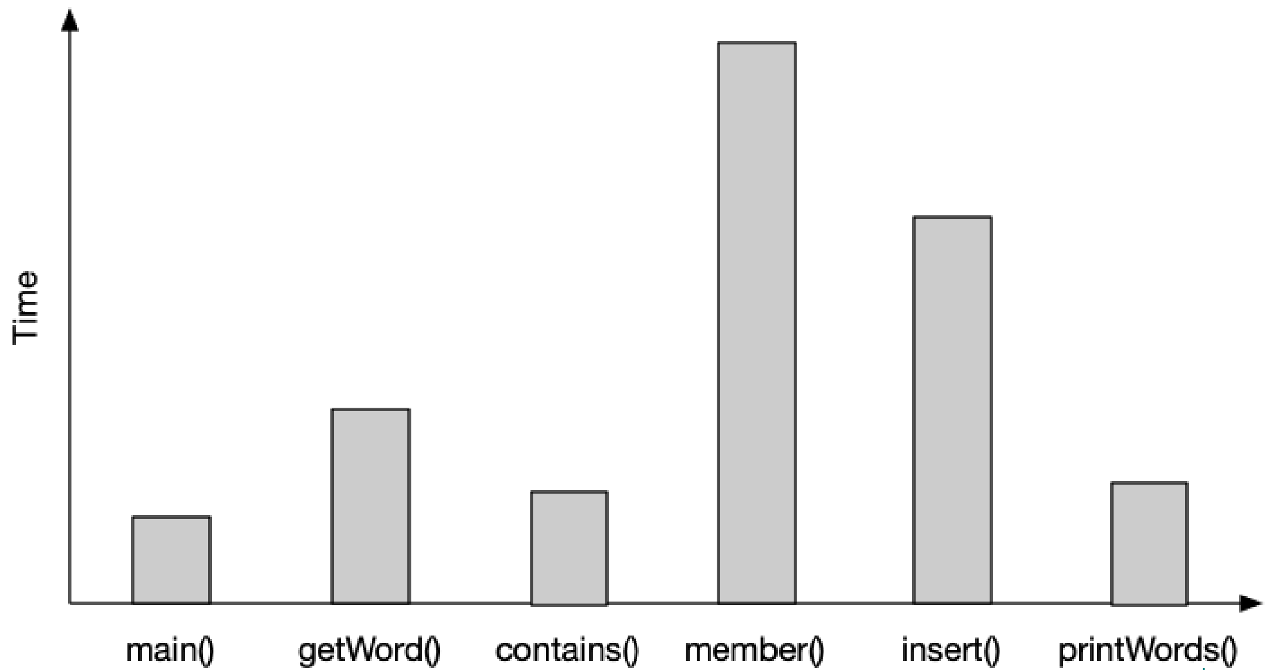
- a count of the number of times the block is executed
- the total execution time spent within that block

Profiles typically collected at function level (i.e. code block = function).

<< ∧ >>

❖ ... Profiles

A profile shows how much time is spent in each code block ...



Software tools can generate profiles of program execution.

<< ∧ >>

❖ gprof: A Profiler

The **gprof** command displays execution profiles ...

- must compile and link program with the **-pg** flag
- executing program creates an new **gmon.out** file
- **gprof** reads **gmon.out** and prints profile on stdout

Example of use:

```
$ gcc -pg -o xyz xyz.c
$ xyz < data > /dev/null
$ gprof xyz | less
```

For further usage details, **man gprof**.

<< ∧ >>

❖ ... gprof: A Profiler

The **gprof** command works at the function level.

It gives a flat profile containing:

- number of times each function was called
- % of total execution time spent in the function
- average execution time per call to that function
- execution time for this function and its children

It also gives a call graph, containing:

- which functions called each function
- which functions were called by each function

<< ∧ >>

❖ Profile Example

Consider the following program ...

```
// search for words in text containing a given substring
// display each such word once (in alphabetical order)

int main(int argc, char*argv[])
{
    char  word[MAXWORD];  // current word
    List  matches;        // list of matched words
    char  *substring;     // string to look for
    FILE  *input;         // the input file

    // ... Check command-line args, open input file ...

    // Process the file - find the matching words
    matches = newList();
    while (getWord(input, word) != NULL) {
        if (contains(word,substring)
                 && !member(matches,word))
            matches = insert(matches,word);
    }
    printWords(matches);
    return 0;
}
```

<< ∧ >>

❖ ... Profile Example

Flat profile for this program (**`xwords et data3`**):

```
 %   cumulative    self              self     total
time   seconds   seconds   calls   us/call   us/call   name
75.00      0.03      0.03  30212      0.99      0.99   getWord
25.00      0.04      0.01  30211      0.33      0.33   contains
 0.00      0.04      0.00    489      0.00      0.00   member
 0.00      0.04      0.00    267      0.00      0.00   insert
 0.00      0.04      0.00      1      0.00  40000.00   main
 0.00      0.04      0.00      1      0.00      0.00   printWords
```

The flat profile shows which functions contribute most to the execution cost.

For more info on how to interpret **`gprof`** flat profiles, look at:

- The GNU Profiler: How to Understand the Flat Profile

Note: **`wc data3`** → **`7439 30211 188259`**.

<< ^ >>

❖ ... Profile Example

```
 %   cumulative    self              self     total
time   seconds   seconds   calls  us/call   us/call  name
75.00      0.03     0.03   30212     0.99      0.99  getWord
25.00      0.04     0.01   30211     0.33      0.33  contains
 0.00      0.04     0.00     489     0.00      0.00  member
 0.00      0.04     0.00     267     0.00      0.00  insert
 0.00      0.04     0.00       1     0.00  40000.00  main
 0.00      0.04     0.00       1     0.00      0.00  printWords
```

We can learn quite a lot from this:

- ~ 75% of the execution time is spent reading words
- ~ 25% of the execution time is spent checking for the substring
- because most words don't contain substring, few calls to **`member()`**
- there are 30211 words in the input file (+1 **`getword()`** call to find EOF)
- there are 489 total incl. 267 distinct words containing the substring

<< ∧ >>

❖ ... Profile Example

Call graph for the same execution (**`xwords et data3`**):

```
index %time    self  children    called     name
               0.00    0.04       1/1           _start [2]
[1]   100.0    0.00    0.04       1         main [1]
               0.03    0.00   30212/30212       getWord [3]
               0.01    0.00   30211/30211       contains [4]
               0.00    0.00     489/489         member [5]
               0.00    0.00     267/267         insert [6]
               0.00    0.00       1/1           printWords [7]
-----------------------------------------------
[2]   100.0    0.00    0.04                 _start [2]
               0.00    0.04       1/1           main [1]
-----------------------------------------------
               0.03    0.00   30212/30212       main [1]
[3]    75.0    0.03    0.00   30212         getWord [3]
-----------------------------------------------
               0.01    0.00   30211/30211       main [1]
[4]    25.0    0.01    0.00   30211         contains [4]
----------------------------------------------
               0.00    0.00     489/489         main [1]
[5]     0.0    0.00    0.00     489         member [5]
-----------------------------------------------
               0.00    0.00     267/267         main [1]
[6]     0.0    0.00    0.00     267         insert [6]
-----------------------------------------------
               0.00    0.00       1/1           main [1]
[7]     0.0    0.00    0.00       1         printWords [7]
```

❖ ... Profile Example

What does each entry mean?

```
index %time    self  children    called     name
               0.00    0.04       1/1           _start [2]
[1]   100.0    0.00    0.04       1         main [1]
               0.03    0.00   30212/30212       getWord [3]
               0.01    0.00   30211/30211       contains [4]
               0.00    0.00     489/489         member [5]
               0.00    0.00     267/267         insert [6]
               0.00    0.00       1/1           printWords [7]
```

This entry shows info about the **`main()`** function

- **`main()`** is called once, by a **`_start`** "function"
- **`main()`** and its called functions account for 100% of the exection time
- **`main()`** calls five functions (**`getWord`**, **`contains()`**, etc.)
- all calls to the functions come from **`main()`**
  - of the 489 calls to **`member()`**, all of them are made by **`main()`**

<< ^

❖ ... Profile Example

The above call graph is unusual

- there is only one level of function calls
- none of the functions call other functions, except **`main()`**

Normally, each function would call other functions

- we can learn where each function is called from
- we can learn which function makes the majority of those calls

For more info on how to interpret **`gprof`** call graphs, look at:

- The GNU Profiler: How to Read the Call Graph

Produced: 11 Jul 2020