PennState

CMPSC 311 - Introduction to Systems Programming

Introduction to Profiling

Suman Saha

(Slides are mostly by Professors Patrick McDaniel and Abutalib Aghayev)

Program Performance

PennState

- 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 (
    - Too slow on real inputs (dat
    - Not-reactive enough to be usable (interf

Optimization

PennState

- 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

Career Notes:

1. Learning to optimize your code is essential to becoming a professional programmer.
2. Optimizing code is a phase of development you don't experience in school.
3. You will spend a good deal of your professional life optimizing existing code without changing its function.

Don’t optimize prematurely

PennState

- The first rule of optimization is:
  - Don’t do it.
- The second rule of optimization (for experts only):
  - Don’t do it yet. Measur

- It is far, far easier to make a correct pro

  than it is to make a fast program correct

Premature optimization is the root of all..

PennState

- "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at have a strong negative debugging and maintenance are consid We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil (quoting Tony Hoare). Yet we should not pass up our opportunities in that critical 3%."

Donald Knuth

Example Inefficient Code

PennState

```c
uint64_t prod_one(uint64_t a, uint64_t b) {
  uint64_t out = 0;
  for (uint64_t i = 0; i < a; ++i)
    out += b;
  return out;
}

uint64_t prod_two(uint64_t a, uint64_t b) {
  return a * b;
}

int main(void) {
  uint64_t a = 10000000000, b = 100000000, sum = 0;
  clock_t start;
  double elapsed;

  start = clock();
  sum = prod_one(a, b);
  elapsed = (clock() - start) / CLOCKS_PER_SEC;
  printf("%ld * %ld = %ld (took %.3f seconds) \n", a, b, sum, elapsed);

  start = clock();
  sum = prod_two(a, b);
  elapsed = (clock() - start) / CLOCKS_PER_SEC;
  printf("%ld * %ld = %ld (took %.3f seconds) \n", a, b, su

  return 0;
}
```

- Try it with
  - gcc -O0
  - gcc -O1
  - gcc -O2


```
$ gcc -O2 -Wall opt.c -o opt
[0s euclid:~/tmp (master)]
$ ./opt
10000000000 * 100000000 = 1000000000000000000 (took 0.000 seconds)
10000000000 * 100000000 = 1000000000000000000 (took 0.000 seconds)
[0s euclid:~/tmp (master)]
```

Profiling

PennState

- 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 spen
  - How much time in each modules of the program?

gprof

PennState

- `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 progr
    and time for each func
    - Statistics and detail on "performance"
  - Which functions called other
    function and how many times
    - Statistics and detail on the "call graph"

Running gprof

PennState

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

```
$ gcc –pg profiling.c –o profiling
```

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

```
$ ./pr
```

3. Run `gprof` with th

```
$ gprof profiling |
```

4. Review the output

```
$ gprof profiling
Flat profile:
…
```

5. Optimize the program, re-profile

6. GOTO step#1

Gprof (flat profile)


```
$ gprof opt
Flat profile:

Each sample counts as
  %   cumulative   sel
 time   seconds   seco
101.31     21.87    21.87        1    21.87    21.87  prod_one
  0.00     21.87     0.00        1     0.00     0.00

 %         the percentage of the total running time o
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.
```

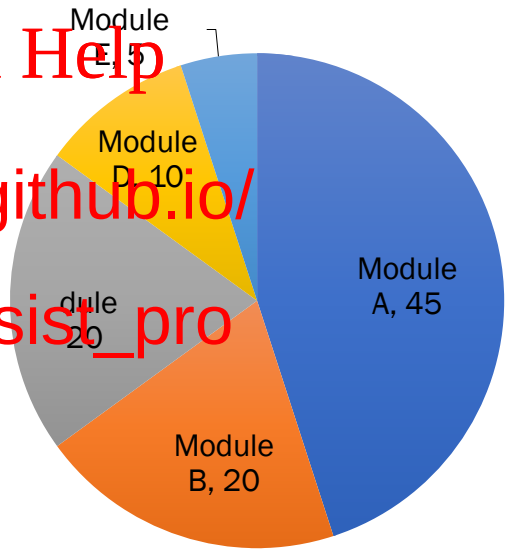
Gprof (Call Graph)


```
                     Call graph (explanation follows)


granularity: each sample

index % time    self  ch                 
                21.87    0.00       1/1           main [2]
[1]    100.0    21.87    0.00       1         prod_one
-----------------------------------------------
                                              <spon
[2]    100.0     0.00   21.87                 main [2]
                21.87    0.00       1/1           prod_one [1]
                 0.00    0.00       1/1           prod_two [3]
-----------------------------------------------
                 0.00    0.00       1/1           main [2]
[3]      0.0     0.00    0.00       1         prod_two [3]
-----------------------------------------------
```

Optimization revisited ...

PennState

- When optimizing, you focus on modules of the program which implement the features and processing of the program.
  - Which parts of the prog depends on what parts
  - then, focus on those parts which take up th most time.



- Profiling tells us where to spend our time.

Amdahl's Law

PennState

- 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 t optimized module(s).
  - s is the execution time expressed in terms speedup (2X, 3X…), which can be found as

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

Amdahl's Law (cont.)

PennState

- 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
  - $\frac{k}{s}$ is the ratio of altered program size to sp

Amdahl's Law (example)

PennState

- 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)

PennState

- 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.

$k = .45$

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

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

A more complex example

PennState

A even more complex example:

PennState

- Assume another system: we have 4 modules each being measured before and after optimization

Module	Before Optimization (usec)	After Optimization (usec)
		60
		11
		600
D		1

- Now suppose that the runtime of the original execution is 2000 usec, what is the speedup?

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

What is going on?

PennState

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define SIZE 35000

int main(void) {
  uint64_t **a = malloc(SIZE * sizeof(uint64_t *));
  for (int i = 0; i < SIZE; ++i)
    a[i] = malloc(SIZE * sizeof(uint64_t

  uint64_t sum = 0;
  for (int i = 0; i < SIZE; ++i)
    for (int j = 0; j < SIZE; ++j)
      sum += a[i][j];

  printf("sum is %ld\n", sum);

  return 0;
}
```



```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define SIZE 35000
int main(void) {
  uint64_t **a = malloc(SIZE * sizeof(uint64_t *));
           i = 0; i < SIZE; ++i)
    malloc(SIZE *       sizeof(uint64_t));
sum = 0;
  for (int i = 0; i < SIZE; ++i)
             = 0; j < SIZE; ++j)
             j][i];
             s %ld\n", sum);

  return 0;
}
```

$ gcc –Wall –O2 c.c –o c
$ time ./c
sum is 0

real	0m1.560s
user	0m1.060s
sys	0m0.500s

$ gcc -Wall -O2 c.c -o c
$ time ./c
sum is 0

real	0m14.543s
user	0m13.597s
sys	0m0.948s