PennState

CMPSC 311 - Introduction to Systems Programming

Introduction to Profiling

Suman Saha

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

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)

WHEN THE PROGRAM YOU WROTE NOW ONLY NEEDS 6MB MEMORY INSTEAD OF 200MB
HACKERMAN
memegenerator.net

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

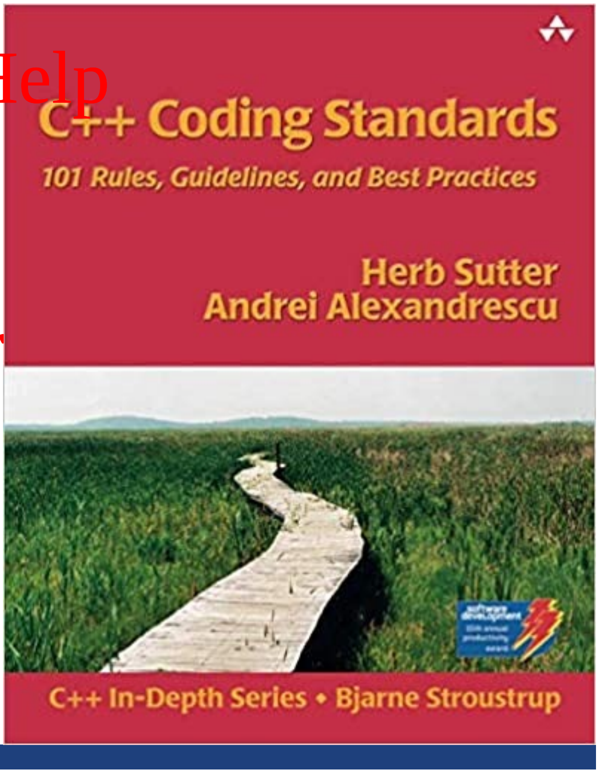
Career Notes:

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

- The first rule of optimization is:
  - Don’t do it.
- The second rule of optimization (for experts only):
  - Don’t do it yet. Measure twice, optimize once.

- It is far, far easier to make a correct program fast than it is to make a fast program correct



C++ Coding Standards

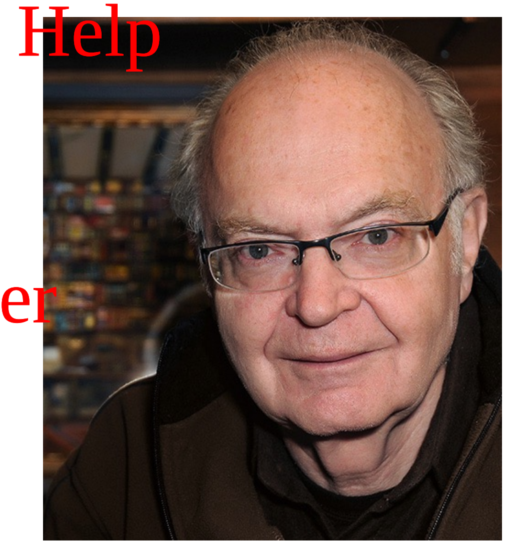
101 Rules, Guidelines, and Best Practices

Herb Sutter
Andrei Alexandrescu

C++ In-Depth Series • Bjarne Stroustrup

Premature optimization is the root of all…

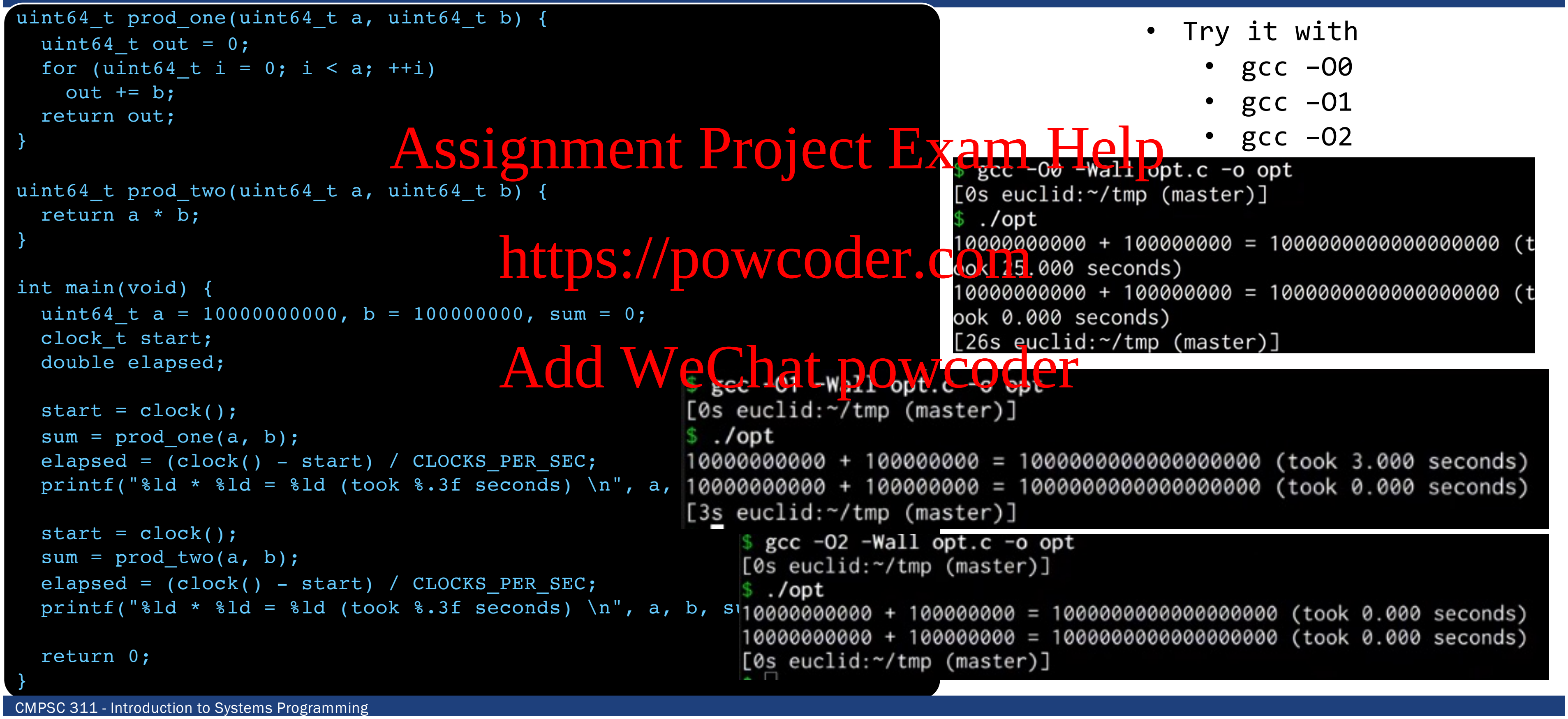
- "Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 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

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



```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,

  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;
}
```

```
$ gcc -O0 -Wall opt.c -o opt
[0s euclid:~/tmp (master)]
$ ./opt
10000000000 + 100000000 = 1000000000000000000 (t
ook 25.000 seconds)
10000000000 + 100000000 = 1000000000000000000 (t
ook 0.000 seconds)
[26s euclid:~/tmp (master)]
```

```
$ gcc -O1 -Wall opt.c -o opt
[0s euclid:~/tmp (master)]
$ ./opt
10000000000 + 100000000 = 1000000000000000000 (took 3.000 seconds)
10000000000 + 100000000 = 1000000000000000000 (took 0.000 seconds)
[3s euclid:~/tmp (master)]
```

```
$ 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)]
```

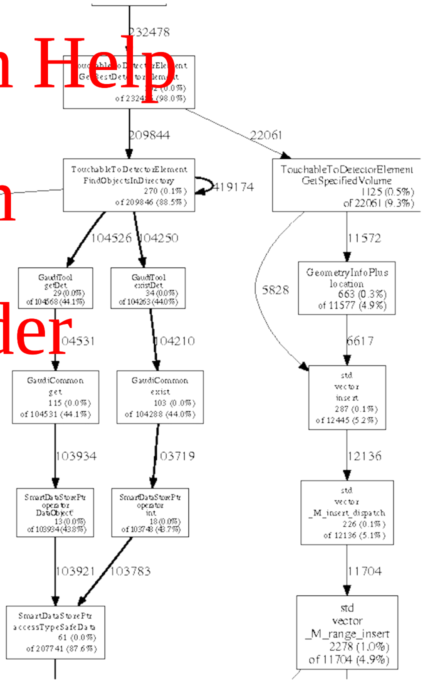
Profiling

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 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 function 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
```

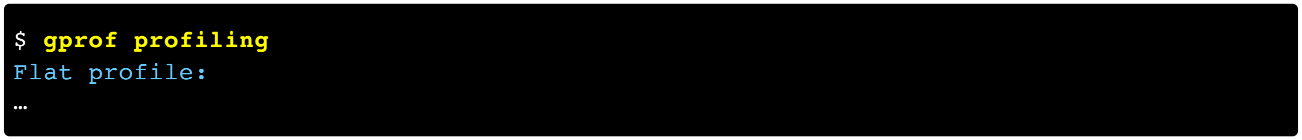
2. Run the program (will generate file gmon.out):

```
$ ./profiling
```

3. Run `gprof` with the named program

```
$ gprof profiling | less
```

4. Review the output

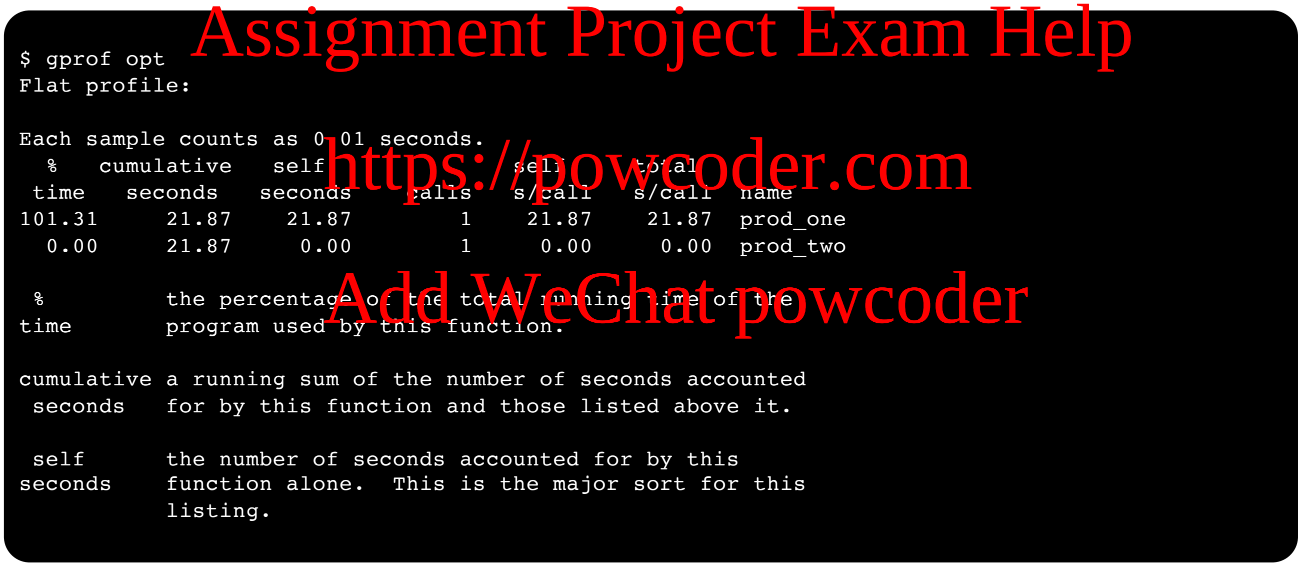


```
$ gprof profiling
Flat profile:
…
```

5. Optimize the program, re-profile

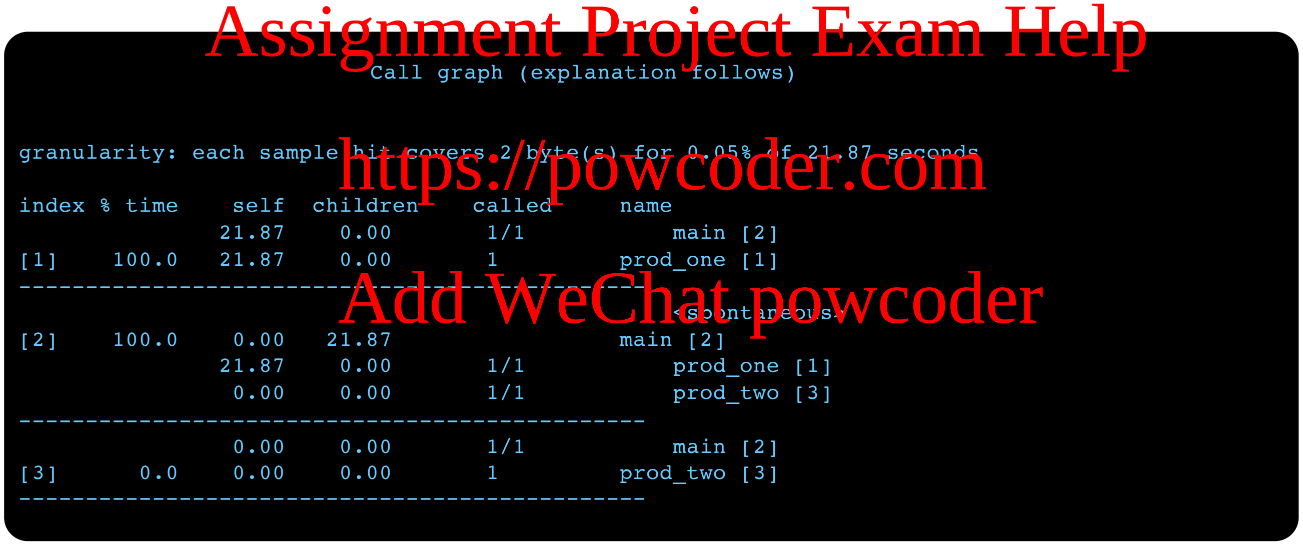
6. GOTO step#1

Gprof (flat profile)



```
$ gprof opt
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
101.31     21.87    21.87        1    21.87    21.87  prod_one
  0.00     21.87     0.00        1     0.00     0.00  prod_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.
```

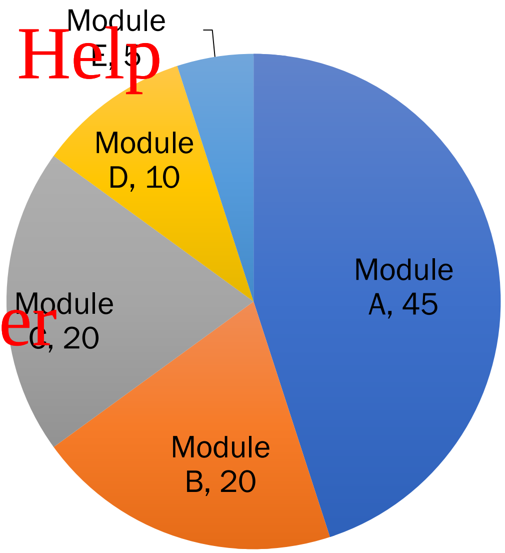
Gprof (Call Graph)



```
                          Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) for 0.05% of 21.87 seconds

index % time    self  children    called     name
               21.87    0.00       1/1           main [2]
[1]    100.0   21.87    0.00       1         prod_one [1]
-----------------------------------------------
                                                 <spontaneous>
[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 …

- 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

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

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 that unchanged
  - $\dfrac{k}{s}$ 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.

$$s = 750/50 = 15$$

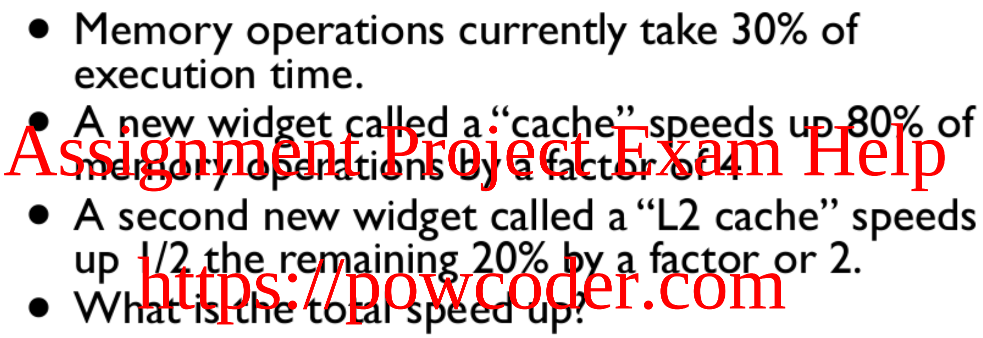
$$k = .45$$

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

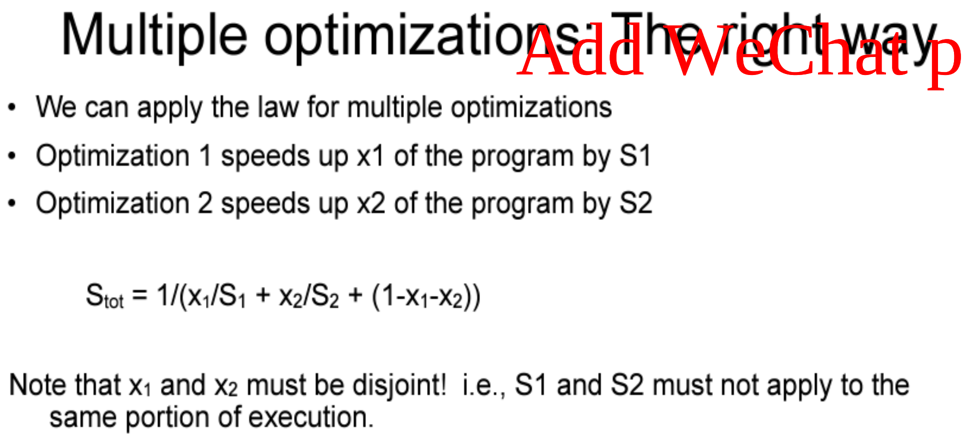
- What is the program speedup? (A: *1.724X*)

A more complex example

PennState



- Memory operations currently take 30% of execution time.
- A new widget called a "cache" speeds up 80% of memory operations by a factor of 4
- A second new widget called a "L2 cache" speeds up 1/2 the remaining 20% by a factor or 2.
- What is the total speed up?

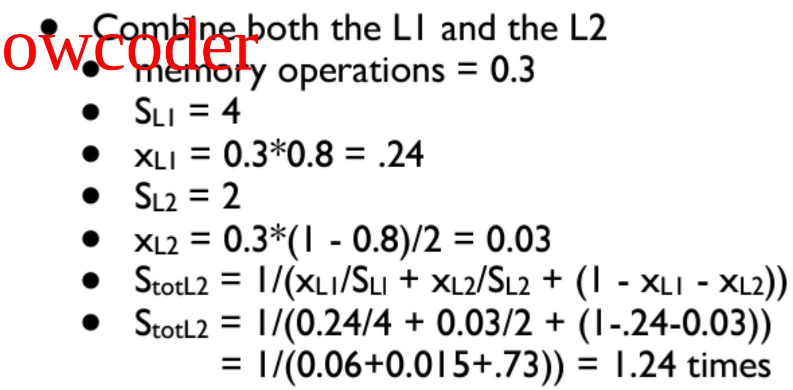


Multiple optimizations: The right way

- We can apply the law for multiple optimizations
- Optimization 1 speeds up x1 of the program by S1
- Optimization 2 speeds up x2 of the program by S2

$$S_{tot} = 1/(x_1/S_1 + x_2/S_2 + (1-x_1-x_2))$$

Note that x_1 and x_2 must be disjoint! i.e., S1 and S2 must not apply to the same portion of execution.



- Combine both the L1 and the L2
- memory operations = 0.3
- $S_{L1} = 4$
- $x_{L1} = 0.3*0.8 = .24$
- $S_{L2} = 2$
- $x_{L2} = 0.3*(1 - 0.8)/2 = 0.03$
- $S_{totL2} = 1/(x_{L1}/S_{L1} + x_{L2}/S_{L2} + (1 - x_{L1} - x_{L2}))$
- $S_{totL2} = 1/(0.24/4 + 0.03/2 + (1-.24-0.03))$
  $= 1/(0.06+0.015+.73)) = 1.24$ times

A even more complex example:

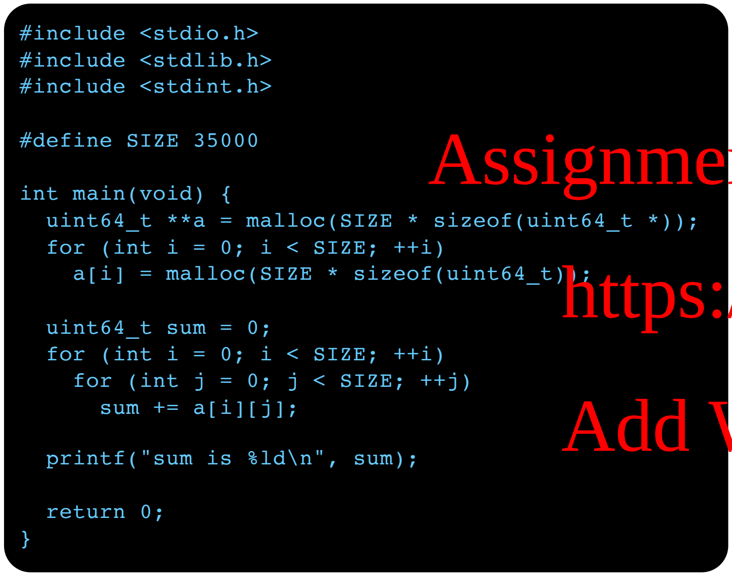
- Assume another system: we have 4 modules each being measured before and after optimization

Module	Before Optimization (usec)	After Optimization (usec)
A	200	60
B	450	11
C	1000	600
D	125	1

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

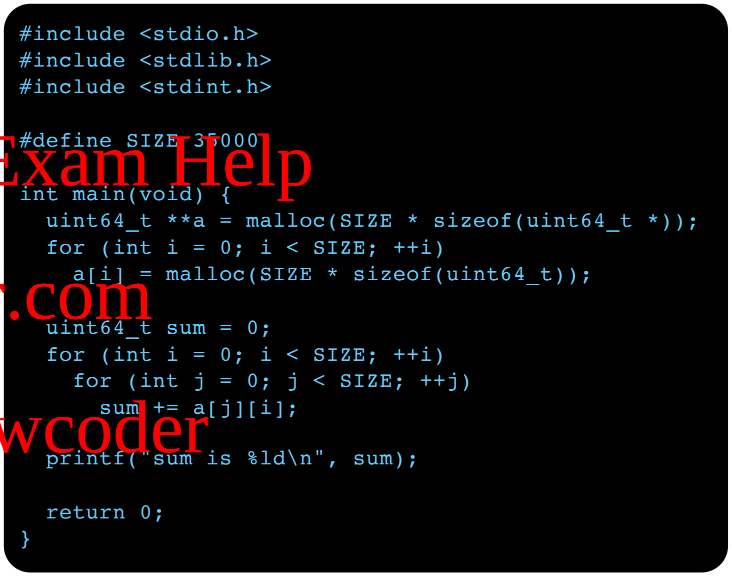
$$T = \frac{1}{(1-k) + \frac{k}{s}}$$

What is going on?



```c
#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;
}
```

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

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



```c
#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[j][i];

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

  return 0;
}
```

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

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