# COS 217: Introduction to Programming Systems

# Performance Improvement

"Premature optimization is the root of all evil." – Donald Knuth

"Rules of Optimization:

Rule 1: Don't do it.

Rule 2 (for experts only): Don't do it yet."

- Michael A. Jackson

# **PRINCETON** UNIVERSITY

# "Programming in the Large"

### Design & Implement

- Program & programming style (done)
- Common data structures and algorithms (done)
- Modularity (done)
- Building techniques & tools (done)

### Debug

• Debugging techniques & tools (done)

### Test

• Testing techniques (done)

### Maintain

• Performance improvement techniques & tools  $\leftarrow$  we are here

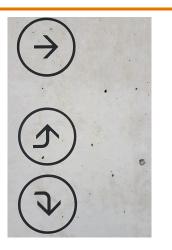


## Goals of this Lecture

#### Help you learn about:

- How to use profilers to identify code hot-spots
- How to make your programs run faster

### Why?



- In a large program, typically a small fragment of the code consumes most of the CPU time
- Identifying that fragment is likely to identify the source of inadequate performance
- Part of "programming maturity" is being able to recognize common approaches for improving the performance of such code fragments
- Part of "programming maturity" is also being able to recognize what is worth your time to improve and what is already "good enough"



4



Should you optimize?

What should you optimize?

**Optimization techniques** 

# Performance Improvement Pros

#### Techniques described in this lecture can answer:



• How can I make my program use less memory?

5

# Performance Improvement Cons

### Techniques described in this lecture can yield code that:

- Is less clear/maintainable
- Might confuse debuggers
- Might contain bugs

So...

6

• Requires regression testing





https://www.linusakesson.net/programming/kernighans-lever/; https://plauger.com/

When to Improve Performance



"The first principle of optimization is

don't.

Is the program good enough already? Knowing how a program will be used and the environment it runs in, is there any benefit to making it faster?"

-- Kernighan & Pike



Run a tool to time program execution

• E.g., Unix time command

\$ time sort < bigfile.txt > output.txt
real 0m12.977s
user 0m12.860s
sys 0m0.010s

### Output:

- Real: Wall-clock time between program invocation and termination
- User: CPU time spent executing the program
- System: CPU time spent within the OS on the program's behalf

# **Enabling Compiler Optimization**

### Enable compiler speed optimization

#### gcc217 -Ox mysort.c -o mysort

- Compiler looks for ways to transform your code so that result is the same but it runs faster
- X controls how many transformations the compiler tries see "man gcc" for details
  - -OO: do not optimize (default if -O not specified)
  - -O1: optimize (default if -O but no number is specified)
  - -02: optimize more (longer compile time)
  - -O3: optimize yet more (including inlining)

### Warning: Speed optimization can affect debugging

• e.g., Optimization eliminates variable  $\Rightarrow$  GDB cannot print value of variable

9





10



# So you've determined that your program is taking too long, even with compiler optimization enabled (and NDEBUG defined, etc.)

### Is it time to completely rewrite the program?





11



Should you optimize?

What should you optimize?

**Optimization techniques** 



Spend time optimizing only the parts of the program that will make a difference!

Gather statistics about your program's execution

- Coarse-grained: how much time did execution of a particular function call take?
  - Time individual function calls or blocks of code
- *Fine-grained:* how many times was a particular function called? How much time was taken by all calls to that function?
  - Use an execution profiler such as gprof

# Timing Parts of a Program



Call a function to compute wall-clock time consumed

• Unix gettimeofday() returns time in seconds + microseconds

#include <sys/time.h>

struct timeval startTime; struct timeval endTime; double wallClockSecondsConsumed;

gettimeofday(&startTime, NULL); <execute some code here> gettimeofday(&endTime, NULL); wallClockSecondsConsumed = endTime.tv\_sec - startTime.tv\_sec + 1.0E-6 \* (endTime.tv\_usec - startTime.tv\_usec);

• Not defined by C90 standard

# Timing Parts of a Program (cont.)

Call a function to compute CPU time consumed

clock() returns CPU times in CLOCKS\_PER\_SEC units

```
#include <time.h>
```

```
clock_t startClock;
clock_t endClock;
double cpuSecondsConsumed;
```

```
startClock = clock();
<execute some code here>
endClock = clock();
cpuSecondsConsumed =
  ((double)(endClock - startClock)) / CLOCKS_PER_SEC;
```

Defined by C90 standard

14



Spend time optimizing only the parts of the program that will make a difference!

Gather statistics about your program's execution

- Coarse-grained: how much time did execution of a particular function call take?
  - Time individual function calls or blocks of code
- *Fine-grained:* how many times was a particular function called? How much time was taken by all calls to that function?
  - Use an execution profiler such as gprof

# **GPROF Example Program**

### Example program for GPROF analysis

- Sort an array of 10 million random integers
- Artificial: consumes lots of CPU time, generates no output

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
enum {MAX_SIZE = 10000000};
int a[MAX_SIZE];
void fillArray(int a[], int size)
{ int i;
   for (i = 0; i < size; i++)</pre>
      a[i] = rand();
void swap(int a[], int i, int j)
{ int temp = a[i];
   a[i] = a[j];
   a[j] = temp;
```

16

```
int part(int a[], int left, int right)
  int first = left-1;
{
   int last = right;
   for (;;)
   { while (a[++first] < a[right]);</pre>
      while (a[right] < a[--last])</pre>
         if (last == left)
             break;
      if (first >= last)
         break;
      swap(a, first, last);
   swap(a, first, right);
   return first;
}
...
```



# GPROF Example Program (cont.)

Example program for GPROF analysis

- Sort an array of 10 million random integers
- Artificial: consumes lots of CPU time, generates no output

```
"
"
void quicksort(int a[], int left, int right)
{
    if (right > left)
        { int mid = part(a, left, right);
        quicksort(a, left, mid - 1);
        quicksort(a, mid + 1, right);
    }
}
int main(void)
{
    fillArray(a, MAX_SIZE);
        quicksort(a, 0, MAX_SIZE - 1);
        return 0;
}
```



# Using GPROF

#### Step 1: Instrument the program

#### gcc217 -pg mysort.c -o mysort

- Adds profiling code to mysort, that is...
- "Instruments" mysort

#### Step 2: Run the program

#### ./mysort

• Creates file gmon.out containing statistics

Step 3: Create a report

#### gprof mysort > myreport

• Uses mysort and gmon.out to create textual report

Step 4: Examine the report

#### cat myreport

18

### The GPROF Report



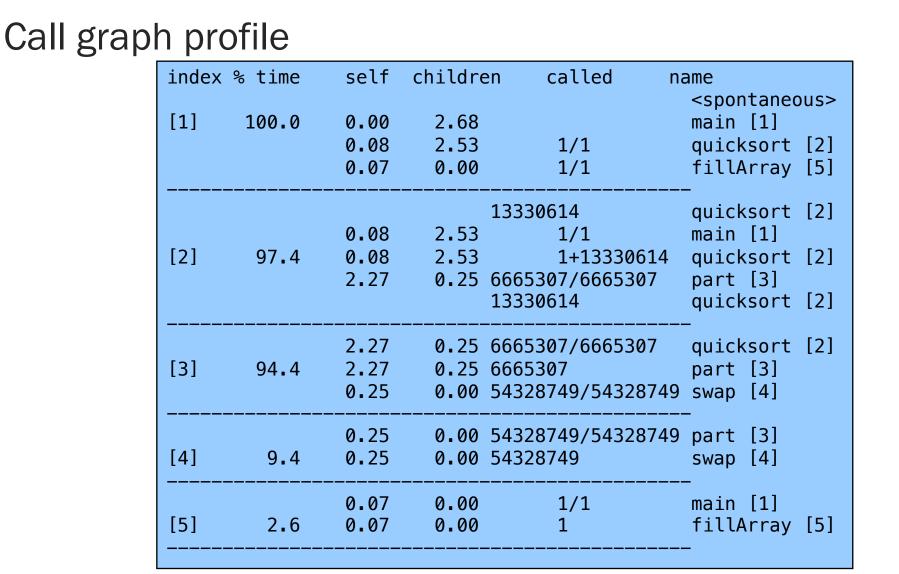
%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
84.5	4 2.27	2.27	6665307	0.00	0.00	part
9.3	3 2.53	0.25	54328749	0.00	0.00	swap
2.9	9 2.61	0.08	1	0.08	2.61	quicksort
2.6	1 2.68	0.07	1	0.07	0.07	fillArray

#### • Each line describes one function

- name: name of the function
- %time: percentage of time spent executing this function
- cumulative seconds: [skipping, as this isn't all that useful]
- **self seconds**: time spent executing this function
- calls: number of times function was called (excluding recursive)
- self s/call: average time per execution (excluding descendants)
- total s/call: average time per execution (including descendants)

### The GPROF Report (cont.)

21



### The GPROF Report



% C	umulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
84.54	2.27	2.27	6665307	0.00	0.00	part
9.33	2.53	0.25	54328749	0.00	0.00	swap
2.99	2.61	0.08	1	0.08	2.61	quicksort
2.61	2.68	0.07	1	0.07	0.07	fillArray

#### • Each line describes one function

- name: name of the function
- %time: percentage of time spent executing this function
- cumulative seconds: [skipping, as this isn't all that useful]
- **self seconds**: time spent executing this function
- calls: number of times function was called (excluding recursive)
- self s/call: average time per execution (excluding descendants)
- total s/call: average time per execution (including descendants)





Should you optimize?

What should you optimize?

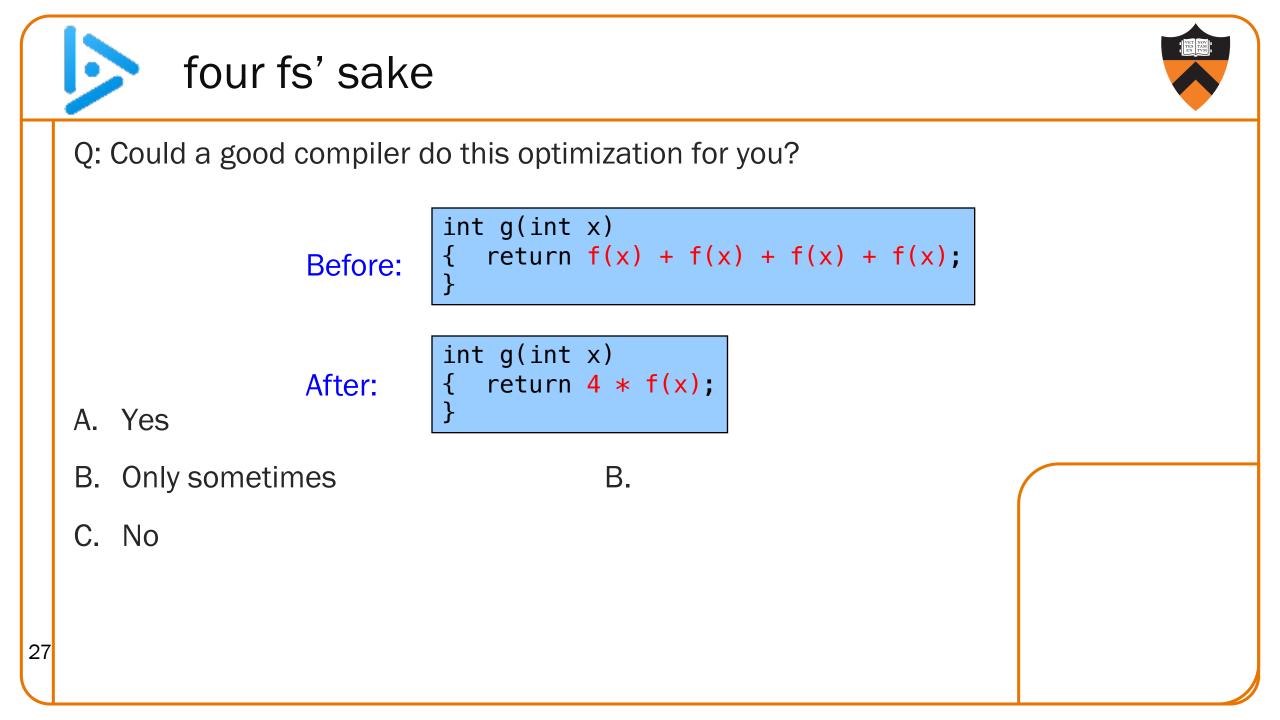
**Optimization techniques** 

#### Use a better algorithm or data structure

• e.g., would a different sorting algorithm work better?

### See COS 226 ...

 But only where it would really help!
 Not worth using asymptotically efficient algorithms and data structures that are {complex, hard-to-understand, hard-to-maintain, ...} if they will not make any difference anyway!



### Aside: Side Effects as Blockers

int g(int x) { return f(x) + f(x) + f(x) + f(x);

### Suppose f() has side effects?

```
int counter = 0;
...
int f(int x)
{ return counter++;
```

And f () might be defined in another file known only at link time!

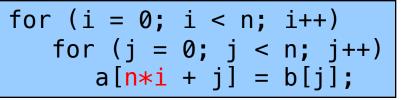




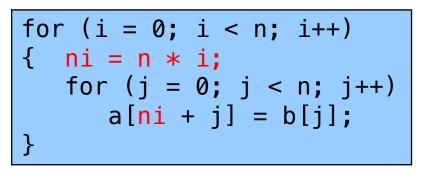


Q: Could a good compiler do this optimization for you?

Before:



After:



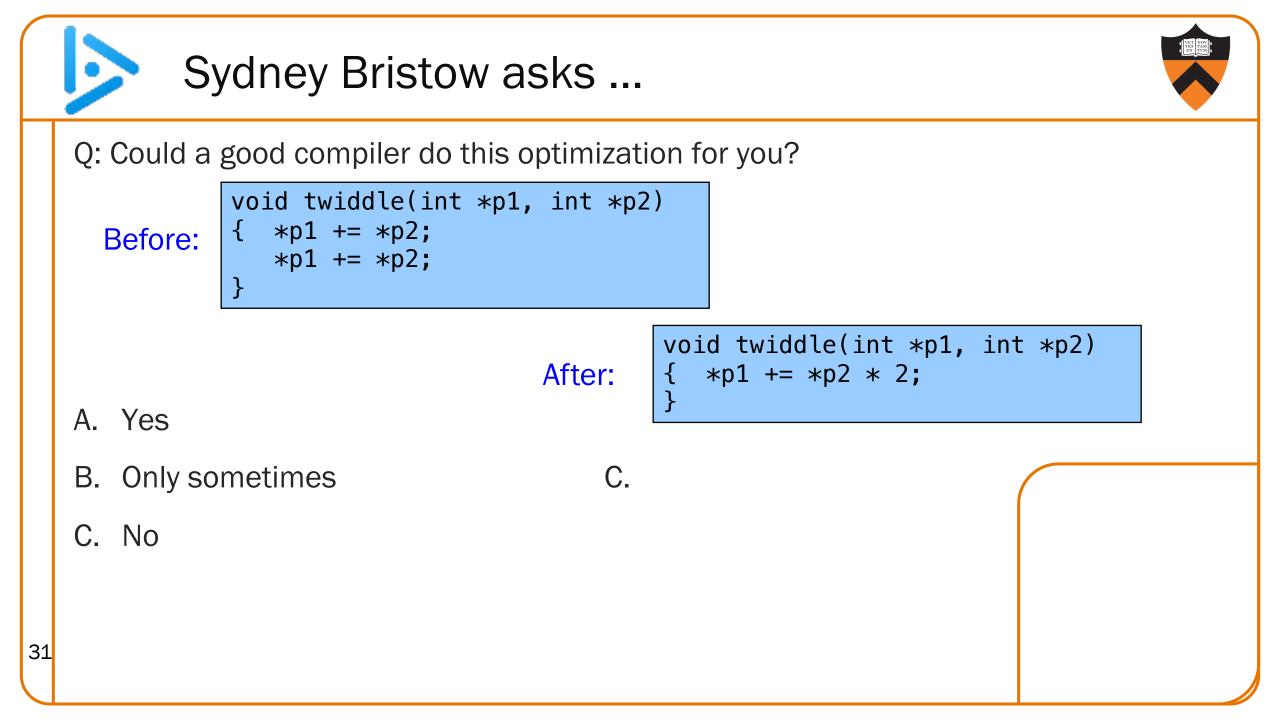
A. Yes

B. Only sometimes

Probably A.

C. No

# Avoiding Repeated Computation for (i = 0; i < strlen(s); i++)</pre> Before: /\* Do something with s[i] \*/ { length = strlen(s); After: for (i = 0; i < length; i++)</pre> { /\* Do something with s[i] \*/ Could a good compiler do that for you? 30





void twiddle(int \*p1, int \*p2)
{ \*p1 += \*p2;
}
void twiddle(int \*p1, int \*p2)
{ \*p1 += \*p2;
}

### What if p1 and p2 are aliases?

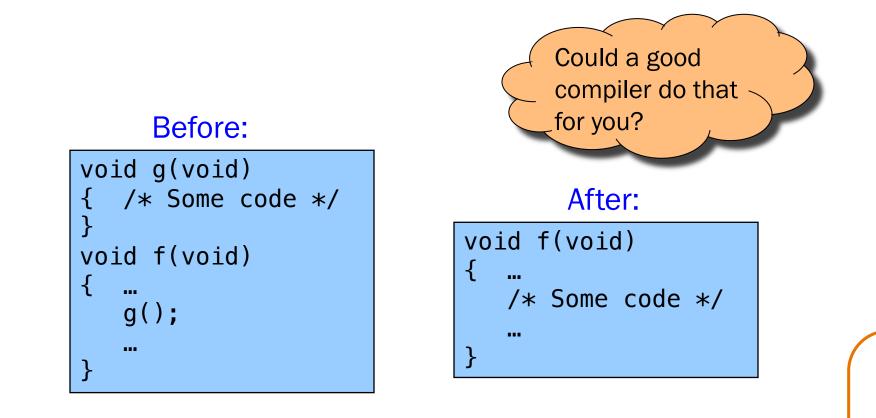
- What if p1 and p2 point to the same integer?
- First version: result is 4 times **\*p1**
- Second version: result is 3 times \*p1

### C99 supports the restrict keyword

•e.g., int \* restrict p1

# **Inlining Function Calls**

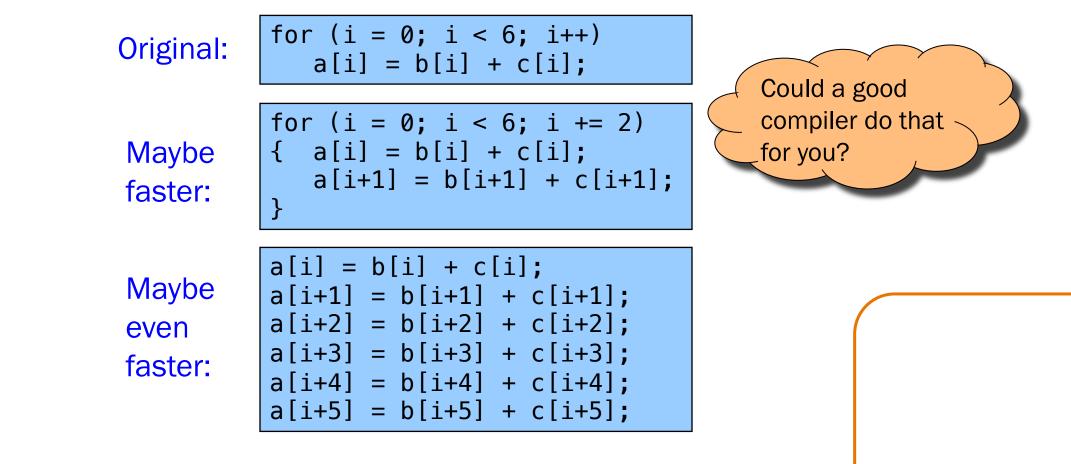




Beware: Can introduce redundant/cloned code Some compilers support inline keyword Unrolling Loops

34





Some compilers provide option, e.g. -funroll-loops

# Using a Lower-Level Language

### Rewrite code in a lower-level language

- As described in this module of the course ...
- Compose key functions in assembly language instead of C
  - Use registers instead of memory
  - Use instructions (e.g. adc) that compiler doesn't know

### Beware!

Modern optimizing compilers generate fast code

• Hand-written assembly language code could be slower!

### Summary



### Steps to improve execution (time) efficiency:

- Don't do it.
- Don't do it yet.
- Time the code to make sure it's necessary
- Enable compiler optimizations
- Identify hot spots using profiling
- Use a better algorithm or data structure
- Identify common inefficiencies and bad idioms
- Fine-tune the code