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 ← 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"

Agenda

Should you optimize?

What should you optimize?

Optimization techniques

Performance Improvement Pros

Techniques described in this lecture can answer:

How slow is my code?

Where is it slow?

Why is it slow?

Similar techniques (not discussed) can address:

- How can I make my program use less memory?

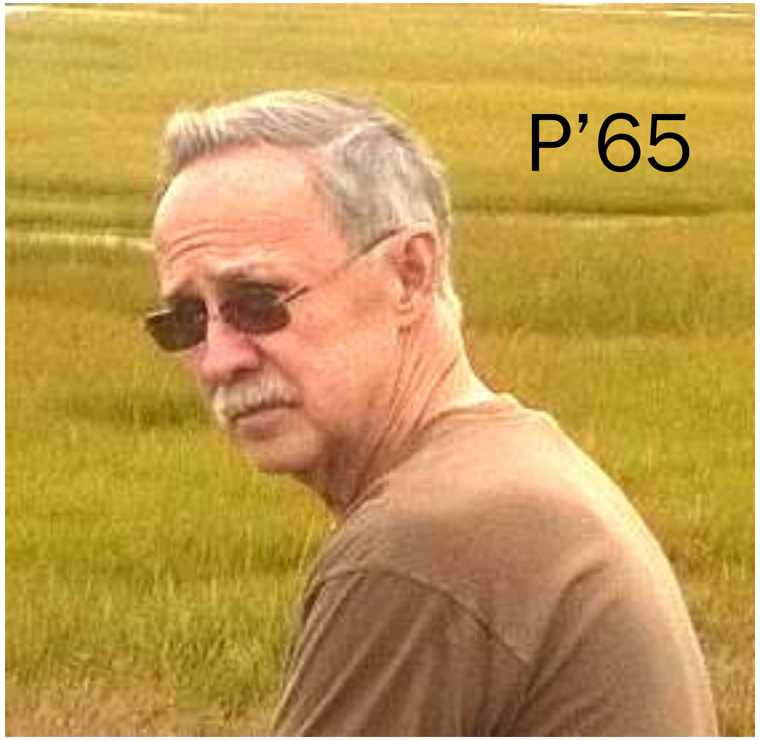
@chuttersnap @markuswinkler @emilymorter

Performance Improvement Cons

Techniques described in this lecture can yield code that:

- Is less clear/maintainable
- Might confuse debuggers
- Might contain bugs
  - Requires regression testing

So...



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

Timing a Program

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 details with `man gcc`
    - -O0: do not optimize (default if –O not specified)
    - -O1: optimize (default if –O but no number is specified)
    - -O2: optimize more (longer compile time)
    - -O3: optimize yet more (including inlining)

Warning: Speed optimization can affect debugging

- e.g., Optimization eliminates variable ⇒ GDB cannot print value of variable

Now What?

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?

Agenda

Should you optimize?

What should you optimize?

Optimization techniques

Identifying Hot Spots

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

Identifying Hot Spots

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`

Opti{mal,mization,on}

You can optimize function A to save 1 second per call. It runs twice.

You can optimize function B to save 1 millisecond per call. It runs 100k times.

Which optimization should you prioritize?

A. A

B. B

C. Aren't you glad I didn't put function A as option B and function B as option A?

D. Well, it depends ...

D is right (Of course! The answer is **always** "it depends"!), because the options aren't well-specified: "you can optimize" ... but at what programmer cost /dev time cost?

B is the better bang for your buck if looking only at program runtimes (2 vs 100 seconds)

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++)
      a[i] = rand();
}

void swap(int a[], int i, int j)
{
   int temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
```

```
int part(int a[], int left, int right)
{
   int first = left-1;
   int last = right;
   for (;;) {
      while (a[++first] < a[right])
         ;
      while (a[right] < a[--last])
         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

```
more myreport
```

`gprof` Design

What's going on behind the scenes?

- `–pg` generates code to interrupt program many times per second
- Each time, records *where* the code was when it was interrupted
  - `gprof` uses symbol table to map back to function name

The GPROF Report

% time	cumulative seconds	self seconds	calls	self s/call	total 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)

GPROF Report Analysis

```
  %   cumulative   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
```

Observations:

- `swap()` is called many times; each call consumes little time; in all, `swap()` consumes only 9% of the time overall
- `part()` is called fewer times; each call consumes little time, but clearly more than `swap()`, since `part()` consumes 85% of the time overall

Conclusions:

- To improve performance, try to make `part()` faster
- Don’t even think about trying to make `fillArray()` or `quicksort()` faster

Agenda

Should you optimize?

What should you optimize?

Optimization techniques

Using Better Algs and DSs

Use a better algorithm or data structure

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

#include COS 226 (or should we say, import 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 debug, or hard to maintain if they will not make any difference anyway!

Optimization Strategy: Avoid Repeated Computation

Before:

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

After:

```
int g(int x)
{
   return 4 * f(x);
}
```

four fs' sake

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

Before:

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

After:

```
int g(int x)
{
    return 4 * f(x);
}
```

A. Yes

B. Only sometimes

C. No

Answer: only sometimes!

Side Effects as Blockers

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

```
int g(int x)
{
   return 4 * 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!

Lift Your nis

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

Before:

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
      a[n*i + j] = b[j];
```

After:

```
for (i = 0; i < n; i++) {
   int ni = n * i;
   for (j = 0; j < n; j++)
      a[ni + j] = b[j];
}
```

A. Yes

B. Only sometimes

C. No

D. If you bring it a shrubbery

Probably A.

Wasn’t this in the A2 grading rubric?

Before:

```
for (i = 0; i < strlen(s); i++) {
   /* Do something with s[i] */
}
```

After:

```
length = strlen(s);
for (i = 0; i < length; i++) {
   /* Do something with s[i] */
}
```

Could a good compiler do that for you?

Sydney Bristow asks …

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

Before:

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

A. Yes

B. Only sometimes

C. No

D. Slide title hints referencing character names from shows that went off the air before we started kindergarten aren't that helpful

After:

```
void twiddle(int *p1, int *p2)
{
  *p1 += *p2 * 2;
}
```

C! … in fact, this "optimization" might not even get the right answer!

ALIASes as Blockers

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

```
void twiddle(int *p1, int *p2)
{
   *p1 += *p2 * 2;
}
```

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

Could a good compiler do that for you?

Before:

```
void g(void)
{
  /* Some code */
}
void f(void)
{
  ...
   g();
   ...
}
```

After:

```
void f(void)
{
  ...
   /* Some code */
   ...
}
```

Beware: Can introduce redundant/cloned code, making maintenance more difficult.

Some compilers support `inline` keyword in C99 and beyond

Unrolling Loops

Original:

```
for (i = 0; i < 6; i++)
   a[i] = b[i] + c[i];
```

Maybe faster:

```
for (i = 0; i < 6; i += 2) {
   a[i] = b[i] + c[i];
   a[i+1] = b[i+1] + c[i+1];
}
```

Maybe even faster:

```
a[i] = b[i] + c[i];
a[i+1] = b[i+1] + c[i+1];
a[i+2] = b[i+2] + c[i+2];
a[i+3] = b[i+3] + c[i+3];
a[i+4] = b[i+4] + c[i+4];
a[i+5] = b[i+5] + c[i+5];
```

Could a good compiler do that for you?

Some compilers provide option, e.g. `–funroll–loops`

Using a Lower-Level Language

Rewrite code in a lower-level language

- Use registers instead of memory
- Use instructions (e.g. `adc`) that compiler doesn't know
- Gee, where have I seen this before...?

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

Final Exam Info

(No egg puns this time, I promise.)

What: Final Exam!

When: 1 month from today
Wednesday, May 17
7:30pm – 10:30 pm

Where: McCosh 10

How: On paper. Closed book, but 1 two-sided study sheet allowed.

Why: Cumulative assessment. You've learned a lot, so show us!

Info: https://www.cs.princeton.edu/courses/archive/spr23/cos217/exam2.html