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

“Programming in the Large” Steps

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
- A power programmer knows how to identify such code fragments
- A power programmer knows techniques for improving the performance of such code fragments

Agenda

Should you optimize?

What should you optimize?

Optimization techniques

Performance Improvement Pros

Techniques described in this lecture can yield answers to questions such as:

- How slow is my program?
- Where is my program slow?
- Why is my program slow?
- How can I make my program run faster?
- How can I make my program use less memory?

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…

When to Improve Performance

"The first principle of optimization is

<u>don't</u>.

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

But, which *parts* of the code are the most time consuming?

Timing Parts of a Program

Call a function to compute **wall-clock time** consumed

- E.g., Unix **`gettimeofday()`** function (time since Jan 1, 1970)

```
#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

- E.g. **`clock()`** function

```
#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

Enabling Compiler Optimization

Enable compiler speed optimization

```
gcc217 –Ox mysort.c –o mysort
```

- Compiler spends more time compiling your code so…
- Your code spends less time executing
- **x** can be:
    - **1**: optimize (default if no number is specified)
    - **2**: optimize more (longer compile time)
    - **3**: optimize yet more (including inlining)
- See "man gcc" for details

Beware: 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 rewrite the program?

Agenda

Should you optimize?

What should you optimize?

Optimization techniques

Identifying Hot Spots

Gather statistics about your program's execution

- How much time did execution of a particular function take?
- How many times was a particular function called?
- How many times was a particular line of code executed?
- Which lines of code used the most time?
- Etc.

How? Use an **execution profiler**

- Example: **`oprofile`**

Example Program

Example program for profiler analysis

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

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

enum {MAX_SIZE = 10000000};
int a[MAX_SIZE]; /* Too big to fit in stack! */

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;
}
…
```

Example Program (cont.)

Example program for profiler analysis (cont.)

```
…
int partition(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;
}
…
```

Example Program (cont.)

Example program for profiler analysis (cont.)

```
…
void quicksort(int a[], int left, int right)
{  if (right > left)
   {  int mid = partition(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 `oprofile`

Step 1: Compile the program with –g and –O

```
gcc –g –O mysort.c –o mysort
```

-g adds "symbol table" (also necessary for debugging)

-O says "compile with optimizations." If you're worried enough about performance to want to profile, then measure the compiled-for-speed version of the program.

Step 2: Run the program

```
operf ./mysort
```

- Creates subdirectory **oprofile_data** containing statistics

Step 3: Create a report

```
opreport -l > myreport
```

- Uses **oprofile_data** and **`mysort`**'s symbol table to create textual report

Step 4: Examine the report

```
more myreport
```

`oprofile` Design

What's going on behind the scenes?

- **`operf`** interrupts program many times per second
- Each time, sees where the code was interrupted
- **`opreport`** uses symbol table to map back to function name

The `oprofile` report

% of execution time spent per function

Executable / library containing the function

Name of the function

```
samples  %        image name               symbol name
30640    74.6062  mysort                   partition
5998     14.6047  mysort                   swap
1462      3.5599  libc-2.17.so             random_r
1408      3.4284  mysort                   quicksort
700       1.7044  libc-2.17.so             random
444       1.0811  no-vmlinux               /no-vmlinux
254       0.6185  libc-2.17.so             rand
160       0.3896  mysort                   fillArray
1         0.0024  ld-2.17.so               _dl_lookup_symbol_x
1         0.0024  ld-2.17.so               _dl_map_object_from_fd
1         0.0024  libc-2.17.so             intel_02_known_compare
```

Standard C library

System "overhead" functions

Report Analysis

Observations

- **`partition()`** consumes 75% of the time overall
- **`swap()`** consumes only 15% of the time overall (even though it's called a lot)

Conclusions

- To improve performance, try to make **`partition()`** 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

Example:

- Would a different sorting algorithm work better?

See COS 226…

- But only where it would help! Not worth using asymptotically efficient (but complex, hard-to-understand, and hard-to-maintain) algorithms and data structures in parts of code that don't matter!

Avoiding 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);
}
```

Could a good compiler do that for you?

Aside: 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);
}
```

Q: Could a good compiler do that for you?

A: Only sometimes…

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!

Avoiding Repeated Computation

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?

Avoiding Repeated Computation

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

Could a good compiler do that for you?

Avoiding Repeated Computation

Before:

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

After:

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

Could a good compiler do that for you?

Aside: Aliases as Blockers

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

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

Q: Could a good compiler do that for you?

A: Not necessarily

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`**

Some compilers support **`restrict`** keyword

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
Some compilers support **`inline`** keyword

Unrolling Loops

Could a good compiler do that for you?

Original:

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

Maybe faster:

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

Maybe even faster:

```
a[i+0] = b[i+0] + c[i+0];
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];
```

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

Using a Lower-Level Language

Rewrite code in a lower-level language

- As described in second half of 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
- Tune the code