



Performance Improvement

The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 7



For Your Amusement

“Optimization hinders evolution.”

-- Alan Perlis

“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:

- Techniques for improving program performance
 - How to make your programs run faster and/or use less memory
- The GPROF execution profiler

Why?

- In a large program, typically a small fragment of the code consumes most of the CPU time and/or memory
- A power programmer knows how to identify such code fragments
- A power programmer knows techniques for improving the performance of such code fragments

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

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



Agenda

Execution (time) efficiency

- **Do timing studies**
- Identify hot spots
- Use a better algorithm or data structure
- Enable compiler speed optimization
- Tune the code

Memory (space) efficiency



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



Agenda

Execution (time) efficiency

- Do timing studies
- **Identify hot spots**
- Use a better algorithm or data structure
- Enable compiler speed optimization
- Tune the code

Memory (space) efficiency



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: `gprof` (GNU Performance Profiler)



GPROF Example Program

Example program for GPROF 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;
}
...
```

GPROF Example Program (cont.)



Example program for GPROF 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;
}
...
```

GPROF Example Program (cont.)



Example program for GPROF 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 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
```



The GPROF Report

Flat profile

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
84.54	2.27	2.27	6665307	0.00	0.00	partition
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 descendents)
 - **total s/call**: average time per execution (including descendents)

The GPROF Report (cont.)



Call graph profile

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.00	2.68		main [1]
		0.08	2.53	1/1	quicksort [2]
		0.07	0.00	1/1	fillArray [5]

			13330614		quicksort [2]
		0.08	2.53	1/1	main [1]
[2]	97.4	0.08	2.53	1+13330614	quicksort [2]
		2.27	0.25	6665307/6665307	partition [3]
			13330614		quicksort [2]

		2.27	0.25	6665307/6665307	quicksort [2]
[3]	94.4	2.27	0.25	6665307	partition [3]
		0.25	0.00	54328749/54328749	swap [4]

		0.25	0.00	54328749/54328749	partition [3]
[4]	9.4	0.25	0.00	54328749	swap [4]

		0.07	0.00	1/1	main [1]
[5]	2.6	0.07	0.00	1	fillArray [5]



The GPROF Report (cont.)

Call graph profile (cont.)

- Each section describes one function
 - Which functions called it, and how much time was consumed?
 - Which functions it calls, how many times, and for how long?
- Usually overkill; we won't look at this output in any detail



GPROF Report Analysis

Observations

- `swap ()` is called very many times; each call consumes little time; `swap ()` consumes only 9% of the time overall
- `partition ()` is called many times; each call consumes little time; but `partition ()` consumes 85% of the time overall

Conclusions

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



Aside: GPROF Design

Incidentally...

How does GPROF work?

- Good question!
- Essentially, by randomly sampling the code as it runs
- ... and seeing what line is running, & what function it's in



Agenda

Execution (time) efficiency

- Do timing studies
- Identify hot spots
- **Use a better algorithm or data structure**
- Enable compiler speed optimization
- Tune the code

Memory (space) efficiency



Using Better Algs and DSs

Use a better algorithm or data structure

Example:

- For mysort, would mergesort work better than quicksort?

See COS 226!



Agenda

Execution (time) efficiency

- Do timing studies
- Identify hot spots
- Use a better algorithm or data structure
- **Enable compiler speed optimization**
- Tune the code

Memory (space) efficiency



Enabling Speed 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
 - **2**: optimize more
 - **3**: optimize yet more
- See “man gcc” for details

Beware: Speed optimization can affect debugging

- E.g. Optimization eliminates variable => GDB cannot print value of variable



Agenda

Execution (time) efficiency

- Do timing studies
- Identify hot spots
- Use a better algorithm or data structure
- Enable compiler speed optimization
- **Tune the code**

Memory (space) efficiency



Avoiding Repeated Computation

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);
}
```

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: Probably not

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

Avoid 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



Avoid repeated computation

Before:

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

After:

```
int ni;  
...  
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?



Tune the Code

Avoid 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

Inline function calls

Before:

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

Could a good compiler do that for you?

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?

Unroll 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+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!



Agenda

Execution (time) efficiency

- Do timing studies
- Identify hot spots
- Use a better algorithm or data structure
- Enable compiler speed optimization
- Tune the code

Memory (space) efficiency



Improving Memory Efficiency

These days memory is cheap, so...

Memory (space) efficiency typically is less important than **execution (time)** efficiency

Techniques to improve memory (space) efficiency...



Improving Memory Efficiency

Use a smaller data type

- E.g. `short` instead of `int`

Compute instead of storing

- E.g. To determine linked list length, traverse nodes instead of storing node count

Enable compiler size optimization

- `gcc217 -Os mysort.c -o mysort`



Summary

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

- Do timing studies
- Identify hot spots (using GPROF)
- Use a better algorithm or data structure
- Enable compiler speed optimization
- Tune the code

Techniques to improve **memory (space)** efficiency:

- Use a smaller data type
- Compute instead of storing
- Enable compiler size optimization

And, most importantly...



Summary (cont.)

Clarity supersedes performance

**Don't improve
performance unless
you must!!!**