Performance Improvement

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

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

Execution Efficiency

• We propose 5 steps to improve execution time efficiency
  (1) Do timing studies
  (2) Identify hot spots
  (3) Use a better algorithm or data structure
  (4) Enable compiler speed optimization
  (5) Tune the code

• Let’s consider one at a time…
Timing Studies

(1) Do timing studies

• To time 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 Studies (cont.)

• To time 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 Studies (cont.)

- To time parts of a program... Call a function to compute CPU time consumed
  - E.g. `clock()` function
    ```c
    #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

Identify Hot Spots

(2) Identify hot spots

- Gather statistics about your program’s execution
  - How much time did execution of a 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

```c
#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;
}

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.)

```c
```
GPROF Example Program (cont.)

• Example program for GPROF analysis (cont.)

```c
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

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self seconds</th>
<th>calls</th>
<th>self s/call</th>
<th>total s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>84.54%</td>
<td>2.27</td>
<td>2.27</td>
<td>6665307</td>
<td>0.00</td>
<td>0.00 partition</td>
</tr>
<tr>
<td>9.33%</td>
<td>2.53</td>
<td>0.25</td>
<td>54328749</td>
<td>0.00</td>
<td>0.00 swap</td>
</tr>
<tr>
<td>2.99%</td>
<td>2.61</td>
<td>0.08</td>
<td>1</td>
<td>0.08</td>
<td>2.61 quicksort</td>
</tr>
<tr>
<td>2.61%</td>
<td>2.68</td>
<td>0.07</td>
<td>1</td>
<td>0.07</td>
<td>0.07 fillArray</td>
</tr>
</tbody>
</table>

• 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

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] 100.0</td>
<td>0.00</td>
<td>2.68</td>
<td>1/1</td>
<td>main [1]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.08</td>
<td>2.53</td>
<td></td>
<td>quicksort [2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.07</td>
<td>0.00</td>
<td>1/1</td>
<td>fillArray [5]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>13330614</td>
<td></td>
<td></td>
<td>quicksort [2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.08</td>
<td>2.53</td>
<td>1/1</td>
<td>main [1]</td>
<td></td>
</tr>
<tr>
<td>[2] 97.4</td>
<td>0.08</td>
<td>2.53</td>
<td>1+13330614</td>
<td>quicksort [2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.27</td>
<td>0.25</td>
<td>6665307/6665307</td>
<td>partition [3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>13330614</td>
<td></td>
<td></td>
<td>quicksort [2]</td>
<td></td>
</tr>
<tr>
<td>[3] 94.4</td>
<td>2.27</td>
<td>0.25</td>
<td>6665307</td>
<td>partition [3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>0.00</td>
<td>54328749/54328749</td>
<td>swap [4]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>0.00</td>
<td>54328749/54328749</td>
<td>swap [4]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>0.00</td>
<td>54328749/54328749</td>
<td>partition [3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.25</td>
<td>0.00</td>
<td>54328749/54328749</td>
<td>swap [4]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.07</td>
<td>0.00</td>
<td>1/1</td>
<td>main [1]</td>
<td></td>
</tr>
<tr>
<td>[5] 2.6</td>
<td>0.07</td>
<td>0.00</td>
<td>1</td>
<td>fillArray [5]</td>
<td></td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self time</th>
<th>self seconds</th>
<th>calls</th>
<th>s/call self</th>
<th>s/call total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>84.54</td>
<td>2.27</td>
<td>2.27</td>
<td>6665307</td>
<td>0.00</td>
<td>0.00</td>
<td>partition</td>
</tr>
<tr>
<td>9.33</td>
<td>2.53</td>
<td>0.25</td>
<td>54328749</td>
<td>0.00</td>
<td>0.00</td>
<td>swap</td>
</tr>
<tr>
<td>2.99</td>
<td>2.61</td>
<td>0.08</td>
<td>1</td>
<td>0.08</td>
<td>2.61</td>
<td>quicksort</td>
</tr>
<tr>
<td>2.61</td>
<td>2.68</td>
<td>0.07</td>
<td>1</td>
<td>0.07</td>
<td>0.07</td>
<td>fillArray</td>
</tr>
</tbody>
</table>

- **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
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

Execution Efficiency Summary

• Steps to improve execution (time) efficiency:
  (1) Do timing studies
  (2) Identify hot spots
  (3) Use a better algorithm or data structure
  (4) Enable compiler speed optimization
  (5) Tune the code
(3) Use a better algorithm or data structure

- Example:
  - For mysort, would mergesort work better than quicksort?

- Depends upon:
  - Data
  - Hardware
  - Operating system
  - …

(4) Enable compiler speed optimization

```bash
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
(5) Tune the code

- Some common techniques
  - Factor computation out of loops

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

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

Tune the Code (cont.)

- Some common techniques (cont.)
  - Inline function calls

  - Example:
    ```c
    void g(void) {
      /* Some code */
    }
    
    void f(void) {
      ...
      g();
      ...
    }
    ```

  - Maybe faster:
    ```c
    void f(void) {
      ...
      /* Some code */
      ...
    }
    ```

- Beware: Can introduce redundant/cloned code
- Some compilers support `inline` keyword
Tune the Code (cont.)

• Some common techniques (cont.)
  • Unroll loops

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

  • Maybe faster:
    ```c
    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:
    ```c
    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`

Tune the Code (cont.)

• Some common techniques (cont.):
  • Rewrite in a lower-level language
    • Write 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 than compiler-generated code, especially when compiled with speed optimization
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…

(1) Use a smaller data type
• E.g. short instead of int

(2) Compute instead of storing
• E.g. To determine linked list length, traverse nodes instead of storing node count

(3) Enable compiler size optimization
   gcc217 -Os mysort.c -o mysort
### Summary

- **Steps to improve execution (time) efficiency:**
  1. Do timing studies
  2. Identify hot spots *
  3. Use a better algorithm or data structure
  4. Enable compiler speed optimization
  5. Tune the code
     * Use GPROF

- **Techniques to improve memory (space) efficiency:**
  1. Use a smaller data type
  2. Compute instead of storing
  3. Enable compiler size optimization

- And, most importantly…

### Summary (cont.)

Clarity supersedes performance

*Don’t improve performance unless you must.*