Performance Improvement

Background reading: The Practice of Programming (Kernighan & Pike) Chapter 7

“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

Case study: 25 most common words

Find the 25 most common words in a text file, print their frequencies in decreasing order

Hint:
- No googling for trivia question:
- What work of literature is this?
- Hint: Project Gutenberg’s #1 downloaded book

A program, “buzz.c”

Reading the input

```c
#define MAX_LEN 1000
int readWord(char *buffer, int buflen) {
    int c; /* Skip non-alphabetic characters */
    do {
        c = getchar();
        if (c == EOF) return 0;
        while (! isalpha(c))
            buffer[0] = 0;
        if (c == EOF) return 0;
        if (strlen(buffer) < buflen - 1) {
            buffer[strlen(buffer)] = ' ';
        }
        buffer[strlen(buffer)] = 0;
        return 1;
    }

    /* Enter every word from stdin into a SymTable, bound to its # of occurrences */
    void readInput(SymTable_T table) {
        char word[MAX_LEN+1];
        while (readWord(word, MAX_LEN+1)) {
            int *p = (int*)SymTable_get(table, word);
            if (p == NULL) {
                p = (int*)malloc(sizeof(int));
                *p = 0;
                SymTable_put(table, word, p);
            }
            (*p)++;
        }
    }
```

Extracting the counts

```c
struct word_and_count {
    const char *word;
    int count;
};
struct counts {
    int filled;
    int max;
    struct word_and_count *array;
}
struct counts *makeCounts(int max) {
    struct counts *p = (struct counts *) malloc(sizeof(struct counts));
    assert(p);
    p->filled = 0;
    p->max = max;
    p->array = (struct word_and_count *) malloc(sizeof(struct word_and_count)*max);
    assert(p->array);
    return p;
}
```
Sorting and printing the counts

```c
void swap (struct word_and_count *a, struct word_and_count *b) {
    struct word_and_count t;
    t=*a; *a=*b; *b=t;
}

void sortCounts(struct counts *counts) {
    /* insertion sort */
    int i, j;
    int n = counts->filled;
    struct word_and_count *a = counts->array;
    for (i=1; i<n; i++) {
        for (j = i; j>0 && a[j-1].count<a[j].count; j--)
            swap(a+j, a+j-1);
    }
}
```

Timing a Program

Run a tool to time program execution
- E.g., Unix command
  ```bash
  $ time ./buzz < corpus.txt > output.txt
  3.58user 0.00system 0:03.59elapsed 99%CPU
  ```

Output:
- Real (or "elapsed"): 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

In summary: takes 3.58 seconds to process 703,549 characters of input. That’s really slow!
(especially if we want to process a whole library of books)

What should you do?

The COS 226 answer:
Use asymptotically efficient algorithms and data structures everywhere.

WRONG!
(and, to be fair, that was a caricature of the COS 226 answer)

What should you do?

Caricature of the COS 226 answer:
Use asymptotically efficient algorithms and data structures everywhere.

Most parts of your program won’t run on “big data!”
Simplicity, maintainability, correctness, easy algorithms and data structures are most important.

Words of the sages

"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

*The MIT professor, not the pop singer.

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
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?

“...The only reason we’re even allowed to be here (as good software engineers) is because we did the performance measurement (700k characters in 3.58 seconds) and found it unacceptable.”

-- Kernighan & Pike

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 oprofile 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?

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 Parts of a Program

Call a function to compute wall-clock time consumed
• E.g. Use gettimeofday() function (time since Jan 1, 1970)

```c
#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);
```

Timing Parts of a Program (cont.)

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;
```
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)
- Reports how many seconds spent in each of your programs’ functions, to the nearest millisecond.

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)
- Reports how many seconds spent in each of your programs’ functions, to the nearest millisecond.

The 1980s just called, they want their profiler back . . .

For some reason, between 1982 and 2016 while computers got 1000x faster, nobody thought to tweak gprof to make it report to the nearest microsecond instead of millisecond.

The 1980s just called, they want their profiler back . . .

So we will use oprofile, a 21st-century profiling tool. But gprof is still available and convenient: what I show here (with oprofile) can be done with gprof.

Read the man pages:

$ man gprof
$ man oprofile

Using oprofile

Step 1: Compile the program with –g and –O2

gcc –g –O2 buzz.c; gcc buzz.o symtable.o –o buzz1
-02 says “compile with optimizations.”

Step 2: Run the program

operf ./buzz1 < corpus.txt > output
- Creates subdirectory oprofile_data containing statistics

Step 3: Create a report

oprofile –l buzz1 > myreport
- Uses oprofile’s demand buzz’s symbol table to create textual report

Step 4: Examine the report

cat myreport
The oprofile report

The use of insertion sort instead of quicksort doesn’t actually seem to be
3.58 seconds to 0.06 seconds
symtablehash.c instead of symtablelist.c
gcc

executable engineering, not just hacking.

The file

I’ve left out the ≈ 1 here; otherwise it would leave out any line whose % is less than 1

Use better algorithms and data structures

Improve the “buzz” program by using
symtablehash.c instead of symtablelist.c

gcc -g -o2 -c buzz.c; gcc buzz.o symtablehash.o -o buzz1

gcc -g -o2 -c buzz.c; gcc buzz.o symtablelist.o -o buzz2

Result: execution time decreases from 3.58 seconds to 0.06 seconds

The use of insertion sort instead of quicksort doesn’t actually seem to be
a problem! That’s what we learned from doing the oprofile. This is
engineering, not just hacking.

Line-by-line view in oprofile

operf ./buzz2 < corpus.txt > output2
operf -l -t 1 myo profile

What if 0.06 seconds isn’t fast enough?

The function

void swap (struct

struct word_and_count *a,  
struct word_and_count *b) {
  struct word_and_count *const
  p = a, *q = b;

  while (p < q) {
    if (strcmp(p, q) < 0) { ... }
  }

  return p;
}

Insertion Sort

Quicksort

Use the qsort function from the standard library
(covered in precept last week)

int compare (const void *p, const void *q) {
  int i;
  int n = count;  //count
  int x = *count;  //x
  int array = i;  //for i=1; i<n; i++)
  { for (p; 
    if (p) { ... }
  }

  return x;
}
Use quicksort instead of insertion sort

Result: execution time decreases from 0.06 seconds to 0.04 seconds

We could have predicted this! If 40% of the time was in the sort function, and we practically eliminate all of that, then it’ll be 40% faster.

Is that fast enough? Well, yes.

But just for fun, let’s run the profiler again.

What if 0.04 seconds isn’t fast enough?

<table>
<thead>
<tr>
<th>function</th>
<th>image name</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>34%</td>
<td>libc</td>
<td>buzz3</td>
<td>__strlen_sse2_pminub</td>
</tr>
<tr>
<td>27%</td>
<td>buzz3</td>
<td>buzz3</td>
<td>readWord</td>
</tr>
<tr>
<td>17%</td>
<td>buzz3</td>
<td>buzz3</td>
<td>SymTable_hash</td>
</tr>
<tr>
<td>15%</td>
<td>libc</td>
<td>buzz3</td>
<td>__strcmp_sse42</td>
</tr>
<tr>
<td>10%</td>
<td>no-vmlinux</td>
<td>no-vmlinux</td>
<td></td>
</tr>
<tr>
<td>9%</td>
<td>libc</td>
<td>buzz3</td>
<td>_int_malloc</td>
</tr>
<tr>
<td>8%</td>
<td>buzz3</td>
<td>buzz3</td>
<td>msort_with_t</td>
</tr>
</tbody>
</table>

27% of execution time in strlen(). Who’s calling strlen()?

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:
  - 0: don’t optimize
  - 1: optimize (this is the default)
  - 2: optimize more
  - 3: optimize across .c files
- See "man gcc" for details

Beware: Speed optimization can affect debugging

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

Reading the input

```c
enum {MAX_LEN = 1000};

int readWord(char *buffer, int buflen) {
    int c;
    /* Skip non-alphabetic characters */
    do {
        c = getchar();
        if (c == EOF) return 0;
        if (!isalpha(c)) {
            buffer[0] = '\0';
        } else {
            while (isalpha(c)) {
                buffer[strlen(buffer)+1] = '\0';
                buffer[strlen(buffer)] = tolower(c);
                c = getchar();
            }
            buffer[strlen(buffer)] = '\0';
            return 1;
        }
    } while (true);
}
```

This is just silly. We could keep track of the length of the buffer in an integer variable, instead of recomputing each time.

How much faster would the program become?

27% faster; from 0.04 sec to 0.03 sec.

Is it worth it? Perhaps, especially if the program doesn’t become harder to read and maintain.
## Avoiding Repeated Computation

Avoid repeated computation

**Before:**
```c
int g(int x)
{
 return f(x) + f(x) + f(x) + f(x);
}
```

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

Could a good compiler do that for you?

## Aside: Side Effects as Blockers

**Q:** Could a good compiler do that for you?

**A:** Probably not

Suppose `f()` has side effects?

```
to do
```

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

## Avoiding Repeated Computation

Avoid repeated computation

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

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

## Tune the Code

Avoid repeated computation

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

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

Could a good compiler do that for you?

## Aside: Aliases as Blockers

**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 the `restrict` keyword.
### Inlining Function Calls

Inline function calls

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

**After:**
```c
void g(void) {
    /* Some code */
}
void f(void) {
    ...
}
```

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

Could a good compiler do that for you?

### Unrolling Loops

Unroll loops

**Original:**
```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+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 oprofile)
- Use a better algorithm or data structure
- Enable compiler speed optimization
- Tune the code

Techniques to improve memory (space) efficiency:
- Profile using valgrind
- Use a more efficient data structure (based on evidence from profile)
- Or (in some cases) recompute instead of storing

And, most importantly…

Clarity supersedes performance

Don’t improve performance unless you must!!!