**Performance Improvement**

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

Princeton University
Computer Science 217: Introduction to Programming Systems

---

**“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 this trivia question. What work of literature is this?

Hint 2

**Project Gutenberg’s #1 downloaded book**

---

**A program, “buzz.c”**

/* Enter every word from stdin into a SymTable, bound to its # of occurrences */
void readInput (SymTable_T table);

/* Make an array of (word, #occ) from the contents of the SymTable */
struct counts *extractCounts(SymTable_T table);

/* Sort the “counts” array in descending order, and print the first 25 entries */
void analyzeData(struct counts *p);

/* The main program */
int main(void) {
    SymTable_T table = SymTable_new();
    readInput(table);
    analyzeData(extractCounts(table));
    return 0;
}

---

**Reading the input**

```c
enum {MAX_LEN = 1000};
int readWord(char *buffer, int buflen) {
    int c;
    /* Skip nonalphabetic characters */
    do {
        c = getchar();
        if (c==EOF) return 0;
    } while (!isalpha(c));
    buffer[0]='\0';
    /* Process alphabetic characters */
    while (isalpha(c)) { 
        if (strlen(buffer)<buflen-1) {
            buffer[strlen(buffer)+1]='\0';
            buffer[strlen(buffer)]=tolower(c);
        } else break;
        c=getchar();
    } buffer[strlen(buffer)]='\0';
    return 1;
}
```

---

**Extracting the counts**

```c
struct word_and_count {
    const char *word;
    int count;
};
```

```c
void handleBinding(const char *key, void *value, void *extra) {
    struct counts *c = (struct counts *) extra;
    assert (c->filled < c->max);
    c->array[c->filled].word = key;
    c->array[c->filled].count = *((int*)value);
    c->filled += 1;
}
```

```c
struct counts *extractCounts(SymTable_T table) {
    struct counts *p = makeCounts(SymTable_getLength(table));
    SymTable_map(table, handleBinding, (void*)p);
    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) {
    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);
    }
}

/* Sort the "counts" array in descending order, and print the first 25 entries */
void analyzeData(struct counts *p) {
    int i, n;
    assert (p->filled == p->max);
    sortCounts(p);
    n = 25<p->max ? 25 : p->max;
    for (i=0; i<n; i++)
        printf("%10d %s\n", p->array[i].count, p->array[i].word);
}
```

### Timing a Program

Run a tool to time program execution
- E.g., Unix `time` 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!
(even 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...”

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 good software engineer knows how to identify such fragments, and knows how to improving their performance

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?

Timing Parts of a Program

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

```c
#include <sys/time.h>

struct timeval startTime;
struct timeval endTime;
double wallClockSecondsConsumed;

gmtimeofday(&startTime, NULL);
<execute some code here>
gmtimeofday(&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;
```

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)

Milliseconds? Really? My whole program runs in a couple of milliseconds! What century do you think we’re in?

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.

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

The oprofile report

<table>
<thead>
<tr>
<th>samples %</th>
<th>image name</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>20871</td>
<td>libc-2.17.so</td>
<td>buzz1</td>
<td>_strcmp_sse42</td>
</tr>
<tr>
<td>5732</td>
<td>20.8398 buzz1</td>
<td>Buzz1</td>
<td>SymTable_get</td>
</tr>
<tr>
<td>257</td>
<td>0.9344 buzz1</td>
<td>Buzz1</td>
<td>SymTable_put</td>
</tr>
<tr>
<td>256</td>
<td>0.9307 buzz1</td>
<td>Buzz1</td>
<td>sortCounts</td>
</tr>
<tr>
<td>105</td>
<td>0.3817 buzz1</td>
<td>Buzz1</td>
<td>readL</td>
</tr>
<tr>
<td>92</td>
<td>0.3345 no-vmlinux</td>
<td>Buzz1</td>
<td>/no-vmlinux</td>
</tr>
<tr>
<td>75</td>
<td>0.2727 libc-2.17.so</td>
<td>Buzz1</td>
<td>fgets</td>
</tr>
<tr>
<td>73</td>
<td>0.2654 libc-2.17.so</td>
<td>Buzz1</td>
<td>_strlenn_sse2_pminub</td>
</tr>
<tr>
<td>92</td>
<td>0.3345 no-vmlinux</td>
<td>Buzz1</td>
<td>/no-vmlinux</td>
</tr>
<tr>
<td>75</td>
<td>0.2727 libc-2.17.so</td>
<td>Buzz1</td>
<td>fgets</td>
</tr>
<tr>
<td>73</td>
<td>0.2654 libc-2.17.so</td>
<td>Buzz1</td>
<td>_strlenn_sse2_pminub</td>
</tr>
<tr>
<td>105</td>
<td>0.3817 buzz1</td>
<td>Buzz1</td>
<td>readL</td>
</tr>
<tr>
<td>92</td>
<td>0.3345 no-vmlinux</td>
<td>Buzz1</td>
<td>/no-vmlinux</td>
</tr>
<tr>
<td>75</td>
<td>0.2727 libc-2.17.so</td>
<td>Buzz1</td>
<td>fgets</td>
</tr>
<tr>
<td>73</td>
<td>0.2654 libc-2.17.so</td>
<td>Buzz1</td>
<td>_strlenn_sse2_pminub</td>
</tr>
<tr>
<td>92</td>
<td>0.3345 no-vmlinux</td>
<td>Buzz1</td>
<td>/no-vmlinux</td>
</tr>
</tbody>
</table>

I’ve left out the -t 1 here; otherwise it would leave out any line whose % is less than 1

What do we learn from this?

% of execution time spent in this function

Name of the function

Shell command that ran this

96% of execution time is in strcmp() and in SymTable_get()
**Use better algorithms and data structures**

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

```bash
gcc -g -O2 -c buzz.c; gcc buzz.o symtablelist.o -o buzz1
```

```bash
gcc -g -O2 -c buzz.c; gcc buzz.o symtablehash.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.*

---

**What if 0.06 seconds isn’t fast enough?**

```bash
oprofile ./buzz2 < corpus.txt >output
opreport -l -t 1 > myreport
```

<table>
<thead>
<tr>
<th>samples %</th>
<th>image name</th>
<th>app name</th>
<th>symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>221</td>
<td>39.6057 buzz2</td>
<td>buzz2</td>
<td>sortCounts</td>
</tr>
<tr>
<td>66</td>
<td>11.8280 buzz2</td>
<td>buzz2</td>
<td>_strlen_sse2_pminub</td>
</tr>
<tr>
<td>50</td>
<td>8.9606 buzz2</td>
<td>buzz2</td>
<td>SymTable_hash</td>
</tr>
<tr>
<td>45</td>
<td>8.0645 buzz2</td>
<td>buzz2</td>
<td>fgets</td>
</tr>
<tr>
<td>37</td>
<td>6.6308 buzz2</td>
<td>buzz2</td>
<td>readWord</td>
</tr>
<tr>
<td>20</td>
<td>3.5842 buzz2</td>
<td>buzz2</td>
<td>_strcmp_sse42</td>
</tr>
<tr>
<td>20</td>
<td>3.5842 no-vmlinux</td>
<td>buzz2</td>
<td>/no-vmlinux</td>
</tr>
</tbody>
</table>

40% of execution time in `sortCounts`. Let’s make it faster.

---

**Line-by-line view in oprofile**

```bash
operf ./buzz2 < corpus.txt >output2
opannotate -s > annotated-source2
```

**The file `annotated-source2`**:

```c
/*----------- Sort the counts ----------------*/

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;
  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);
  }
}

int compare_count( const void *p, const void *q) {
  return ((struct word_and_count *)q)->count
    - (struct word_and_count *)p)->count;
}

void sortCounts (struct counts *counts) {
  qsort(counts->array, counts->filled, sizeof(struct word_and_count), compare_count);
}
```

---

**Insertion Sort**

**Quicksort**

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

```c
int compare_count( const void *p, const void *q) {
  return ((struct word_and_count *)q)->count
    - (struct word_and_count *)p)->count;
}
```

```c
void sortCounts (struct counts *counts) {
  qsort(counts->array, counts->filled, sizeof(struct word_and_count), compare_count);
}
```

---

**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?**

```bash
samples % | image name    | app name    | symbol name                  |
-----------|---------------|-------------|------------------------------|
 73        | 27.3408 libc-2.17.so buzz3 | __strlen_sse2_pminub |
 48        | 17.9775 buzz3    | buzz3       | readWord                     |
 36        | 13.4831 buzz3    | buzz3       | SymTable_hash               |
 33        | 12.3596 libc-2.17.so buzz3 | fgets    |
 27        | 10.1124 buzz3    | buzz3       | SymTable_get                |
 15        | 5.6180 no-vmlinux buzz3 | /no-vmlinux |
 11        | 4.1199 libc-2.17.so buzz3 | __strcmp_sse42 |
 4         | 1.4981 libc-2.17.so buzz3 | _int_malloc |
 3         | 1.1236 libc-2.17.so buzz3 | _malloc_with_t |
```

27% of execution time in `strlen()`. Who’s calling `strlen()`?
**Reading the input**

```c
enum {MAX_LEN = 1000};

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

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.

---

**Enabling Speed Optimization**

Enable compiler speed optimization

```
gcc -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 `x` => 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:**

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

**After:**

```c
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**?

```c
int counter = 0;
...
int f(int x) {
    return counter++;
}
```

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

---

**Aside: Side Effects as Blockers**

```c
int g(int x) {
    return f(x) + f(x) + f(x) + f(x);
}
```
Avoiding Repeated Computation

Avoid repeated computation

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

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

Could a good compiler do that for you?

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 restrict keyword

Inlining Function Calls

Inline function calls

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

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

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

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+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

Could a good compiler do that for you?
Rewrite code in a lower-level language
- As described in the second half of course...
- Compose key functions in assembly language instead of C
  - Use registers instead of memory
  - Use instructions (e.g. `adc`) that the compiler doesn’t know

Beware: Modern optimizing compilers generate fast code
- Hand-written assembly language code could be slower!

Clarity supersedes performance

Don’t improve performance unless you must!!!