Profiling 2

Motivation

- We’ve used a profiler and located the code’s hot spots
- We’ve applied reasoning to underlying math/algorithms
- How can we make things better?

Case Study

- We’ve seen raytracing as a motivating example
- This time, we’ll have a different case study
  - But we need some background first

History

- Stone age: Characters represented by bytes
- Byte = 28 combinations
- Thus, 256 possible glyphs

ASCII

- ASCII: American Standard Code for Information Interchange
  - Defined 0-31 as control characters
    - Newline, tab, form feed, bell, etc.
  - 32-126 = printable characters
  - 127 = DEL (rubout)
  - Did not specify what 128-255 represented

OEM

- Many people began using codes 128-255 for their own purposes
- IBM used them for Text-GUI symbols
  - Could draw boxes, etc. by putting symbols together:
    - ⌈ ⌉ ⌋ ⌊
- Other cultures used them for their alphabet symbols
  - Cyrillic, Arabic, Hebrew, Greek, Japanese, etc.

Problem

- If you get a document, how do you know what symbols to display it in?
- Microsoft used *code pages*
  - Collection of glyphs for a particular language
    - Ex: Code page 1252 = Western European; 950 (“big5”) used for Chinese, 21866 (“koi8-u”) for Cyrillic, ...
  - You have to tell system what code page a document uses
  - Can’t mix pages within a document

CJK

- What about CJK languages?
  - More than 256 glyphs!
  - Solution: Double-byte code pages
  - So a code page consists of set of symbols + information on whether each glyph in the set is represented by one byte or two
  - Makes going backward in document very challenging!
    - Ex: Suppose you see previous byte is 20 and next-previous is 82.
    - Do we have two characters (82, 20) or one character (21012 little endian) or one character (5202 big endian)?
    - In the limit, we might have to go *all the way back to the beginning* and parse *the entire document*
    - Just to go back one character!

Unicode

- Enter Unicode
- Unicode consortium standardized a way of representing characters
- Each character corresponds to a *code point* (a number)
  - This isn't always trivial to decide

ss vs. ß

- Ex: In German: There's the sequence ss and there's also ß
  - ß is pronounced as s but appears when next to a dipthong long vowel
    - Ex: Straße (street)
    - Is this a separate character? German alphabet says no, it's collated as if it's 'ss'

Storage

- How to store Unicode?
- One early idea: Use 16 bits per character
- Problem: Big vs little endian
- Convention: First 16 bits of file are 0xfeff
  - If read in as 0xfffe, need to swap bytes to get actual values
- Some unicode characters require several 16-bit chunks
  - But most characters are below the 65536 code point

Problem

- Storing every character as 16 bits is wasteful if most (or all) of them have code points below 256
- And: Many existing documents; stored as 8 bit ASCII
- And: C routines assume 0 terminates strings
  - If a UTF-16 string contains any zero bytes within a character: It won't play well with C library routines (strcmp, strcpy, strlen, ...)
- Enter UTF-8

UTF-8

- Code points 0-127 are encoded as-is
- Code points 128-2047 are encoded in two bytes like so:
  - First byte is 110xxxxx
  - Second byte is 10xxxxxx
  - The x's represent the bits of the code point
  - lsb of code point goes in the lsb of the second byte; msb of code point goes next to the 0 in the first byte
- Code points 2048-65535 encoded as 1110xxxx 10xxxxxx 10xxxxxx
- Code points 65536-1114111 encoded as 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Note

- UTF-8 has some nice properties:
- No character translates to have an embedded 0 byte (preserving compatibility with C)
- ASCII text “just works”
- Stepping forward in text is straightforward
  - Can look at first byte’s upper bits to tell how many bytes to skip
    - Let B = byte & 11100000
    - If B == 0: +1
    - If B == 0xc0: +2
    - If B == 0xe0: +3 r
    - If B == 0xf0: +4
    - Else, illegal

Note

- Stepping backwards isn't too difficult
  - If upper bit is zero, go back one byte
  - If upper bit is one, keep going back until you see character with upper two bits both one
    - Then step back one more byte

Note

- In theory, each code point has exactly one encoding
- But invalid codings can also appear
- Ex: Suppose we have code point 1
- Should encode as: 00000001
- But what if we see sequence 11000000 10000001 ?
  - Parsers are expected to reject second sequence as invalid

Code

- Example (based on one from Guntheroth [see references])
  - Suppose we want to strip out invalid UTF characters from a string
  - We might do so with C code like so...utf.h

Timing

- We need a baseline to time this code
- Here's the mktest.py and driver.cpp files
- Inputs: Small: testfile.txt and large: testfile2.txt

Running It

- It's important that results can be reproduced
- So we carefully note compiler options and runtime arguments
  - Compile: g++ -g -O3 driver.cpp -include utf.h
  - Run: valgrind --tool=callgrind ./a.out testfile.txt 1
  - Analyze: callgrind_annotate --show-percs=yes --auto=yes callgrind.out.14054
  - Output:
    Time per function call: 3,685,596 μsec
- Analyze: Where are the hot spots?

Analysis

- None of the lines of code in my program accounted for significant time
- But...

Profile

- Here's what I got from the function call analysis
- Notice most of the time is spent in library functions

```
145,911,466 (89.97%)  ???:__memcpy_avx_unaligned_erms [/usr/lib64/libc-2.29.so]
  5,378,363 ( 3.32%)  ???:_int_malloc [/usr/lib64/libc-2.29.so]
  2,632,631 ( 1.62%)  ???:_int_free [/usr/lib64/libc-2.29.so]
  1,338,078 ( 0.83%)  ???:malloc [/usr/lib64/ld-2.29.so]
    970,581 ( 0.60%)  ???:_dl_lookup_symbol_x [/usr/lib64/ld-2.29.so]
    835,542 ( 0.52%)  ???:std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::
         _M_mutate(unsigned long, unsigned long, char const*, unsigned long) [/usr/lib64/libstdc++.so
         .6.0.26]
    633,306 ( 0.39%)  ???:std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::
         _M_replace_aux(unsigned long, unsigned long, unsigned long, char) [/usr/lib64/libstdc++.so.6.0.26]
    633,040 ( 0.39%)  ???:do_lookup_x [/usr/lib64/ld-2.29.so]
    532,427 ( 0.33%)  ???:free [/usr/lib64/ld-2.29.so]
    526,952 ( 0.32%)  ???:unlink_chunk.isra.0 [/usr/lib64/libc-2.29.so]
    495,344 ( 0.31%)  /usr/include/c++/9/bits/basic_string.h:removeInvalid(std::__cxx11::basic_string<char,
         std::char_traits<char>, std::allocator<char> >)
    357,005 ( 0.22%)  ???:_dl_relocate_object [/usr/lib64/ld-2.29.so]
    354,816 ( 0.22%)  ???:std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::
         _M_create(unsigned long&, unsigned long) [/usr/lib64/libstdc++.so.6.0.26]
```

Question

- Where in the code are we copying memory?

Observe

- One place where a memory copy would occur is when we do work with strings
  - Any place we have string1 = string2 is a candidate for a memcopy
- Example:
  res = res + v1
  - Makes a copy of data in res & in v1
- How can we improve our code?

Strings

- String has built-in function: `+=`
  - Might be more intelligent
- Let's try it: utf-2.h

Results

- Time per function call: 1130 μsec!
- Wow!
- Speedup = 3262x!!!

Note

- When I ran the code, I initially got a time of 3,651 μsec
- But: When times get very small: “Noise” becomes an issue
  - OS interrupts, etc.
- So I re-ran with 1000 iterations
  - That gave average time per iteration of 1130 μsec
  - Of course, some of the speedup could have been due to cache preloading!
  - How could we determine if that was the case?

Principle

- Memory access is often a bottleneck
  - If we can minimize the amount of memory copies, we often see a win
  - C++ STL containers often have functions that can minimize copying...If we use them

Example

- Another case where a similar issue arises: Suppose we have two vectors:

```
vector<Foo> v1;
...
vector<Foo> v2;
v2 = v1;
```

- This involves a copy

Improvement

- vector (like most STL containers) has a swap() operation
  - Very fast: Just shuffles a few pointers/integers around
- If you know you won't need v1 again, use v1.swap(v2)

push_back

- How does vector work internally? Common strategy:
  - When you push_back to empty vector, it allocates one item's worth of memory
  - If you insert again, vector allocates two items' worth of space
  - Another insert: Vector allocates four items' worth of space
  - On next insert: No allocation required (capacity = 4, size=3)
  - Next insert: Still no allocation required

push_back

- Every time free space exhausted, vector doubles storage space
  - STL calls this *capacity*
- As more elements put into vector, reallocations more costly but also less frequent
  - Reallocation = Need to copy data from old buffer to new one

Improvement

- Call vector's reserve() function to preallocate enough space
  - Sets capacity directly
  - Avoids reallocation & copies

Further Work

- We can do better.
- Where are other potential bottlenecks?

Parameters

- When we pass parameters to functions, we must copy arguments
- If we only call function once or twice, no big deal
- But as number of calls adds up, so does overhead
- So let's see if we change parameter to a reference parameter
  - Can make it const so we don't accidentally change caller

- utf-3.h

Results

- Time per function call: 952 μsec
- Speedup = 1.19x

Change

- Using the []'s involves pointer arithmetic
- Can we use iterators to speed things up?

- utf-4.h

Results

- Time = 916 μsec
  - *Teeny* improvement!
- Is the call to s.end() time consuming?

- utf-5.h

Results

- 891 μsec
- Hardly any improvement

Change

- What if we preallocate space? Strings are probably implemented a lot like vectors internally...
- [utf-6.h]

Result

- Time per function call: 748 μsec

Summary

- So here's what we end up with (using the big input file):
  - Original: 3,685,596 μsec
  - Using +=: 1130 μsec
  - Reference parameters: 952 μsec
  - Iterators: 916 μsec
  - Using end() instead of length(): 891 μsec
  - preallocate: 748 μsec

Principle

- Often, we see a pattern like this:
  - Low hanging fruit: First optimizations are “obvious” and easy, give big improvement
  - As we apply additional changes, we find diminishing returns
  - At some point, we declare victory and move on

Idea

- What kind of transformations should we be trying for?
- There are several that appear repeatedly in different contexts
- It’s worth looking for places where we can apply them.

Precomputing

- Some values can be computed at compile time
  - Smart compilers will do the work at that point instead of at runtime
  - Ex: x = sin(3.14)
  - How do we know if our compiler does this?
  - Only real way is to look at the generated code

Example

- Using gcc: Code:

```
double foo(double y){
    double x = sin(1.23);
    return y+x;
}
```

objdump

- objdump -C -M intel-mnemonic -S a.out

```
double foo(double y){
400737:       55 push   rbp
400738:       48 89 e5                mov    rbp,rsp
40073b:       f2 0f 11 45 e8          movsd  QWORD PTR [rbp-0x18],xmm0
double x = sin(1.23);
400740:       f2 0f 10 05 50 01 00    movsd  xmm0,QWORD PTR [rip+0x150]
400747:       00
400748:       f2 0f 11 45 f8          movsd  QWORD PTR [rbp-0x8],xmm0
return y+x;
40074d:       f2 0f 10 45 e8          movsd  xmm0,QWORD PTR [rbp-0x18]
400752:       f2 0f 58 45 f8          addsd  xmm0,QWORD PTR [rbp-0x8]
400757:       5d                      pop    rbp
400758:       c3                      ret
```

- It’s not obvious, but the compiler cached the sine at [rip+0x150]
  - Look at 0x400740

Comparison

- If we change the sine to: double x = sin(y+1.23) we know there can't be precomputation, and an assembly dump confirms it:

```
double foo(double y){
4007a7:       55                         push   rbp
4007a8:       48 89 e5                   mov    rbp,rsp
4007ab:       48 83 ec 20                sub    rsp,0x20
4007af:       f2 0f 11 45 e8             movsd  QWORD PTR [rbp-0x18],xmm0
double x = sin(y+1.23);
4007b4:       f2 0f 10 4d e8             movsd  xmm1,QWORD PTR [rbp-0x18]
4007b9:       f2 0f 10 05 57 01 00       movsd  xmm0,QWORD PTR [rip+0x157]
4007c0:       00
4007c1:       f2 0f 58 c1                addsd  xmm0,xmm1
4007c5:       e8 f6 fe ff ff             call   4006c0 <sin@plt>
4007ca:       66 48 0f 7e c0             movq   rax,xmm0
4007cf:       48 89 45 f8                mov    QWORD PTR [rbp-0x8],rax
return y+x;
4007d3:       f2 0f 10 45 e8             movsd  xmm0,QWORD PTR [rbp-0x18]
4007d8:       f2 0f 58 45 f8             addsd  xmm0,QWORD PTR [rbp-0x8]
4007dd:       c9                         leave
4007de:       c3                         ret
```

Lazy Initialization

- Sometimes, it’s costly to initialize a variable
  - Lengthy computation
  - Reading lots of data from disk
  - Etc.
- Pattern: Lazy initialization
  - When creating object, set a flag that says “needs initialization”
  - When we go to use the object, check the flag

Example

```
class Foo{
    bool initialized;
    vector<string> bigDataField;
    map<int,string> otherDataField;
public:
    Foo(){
        initialized=false;
    }
    bool isPresent(int value){
        if( !initialized ) init();
        return  otherDataField.find(value) !=  otherDataField.end();
    }
    void addItem(string s){
        if( !initialized) init();
        bigDataField.push_back(s);
    }
    void init(){
        ...
        initialized=true;
    }
};
```

Analysis

- Can save runtime
- But: Introduces additional branches
  - Could be an issue if member functions are called in tight loop
- And: Easy to forget to add the if-check
  - As class is modified over the years by other developers
- Makes concurrency more difficult

Caching

- If values are expensive to compute, save them once produced
- Ex:

```
double foo(double x){
    static map<double,double> M;
    auto it = M.find(x);
    if(it == M.end() ){
        double tmp = computeStuff(x);
        M[x] = tmp;
        return tmp;
    }
    return it->second;
}
```

Analysis

- Also potential time saver
- But: Need extra memory for cache
- Problem: How to limit overall memory usage?
  - Need to know when to delete data from cache
- Need to do searching on cache items
  - Needs to be fast

Batching

- Consider code to copy one file to another: copy.cpp
- But what if we do a chunk of characters at a time: copy2.cpp
- Results on an ~8MB file:

```
$ time ./copy q q2
real 0m0.153s
user 0m0.142s
sys 0m0.010s
$ time ./copy2 q q2
real 0m0.020s
user 0m0.010s
sys 0m0.008s
```

Code Arrangement

- Which is faster:

```
if(x == 0 ){
...
} else if( x == 1 ){
...
} else if( x == 2 ){
...
} else if( x == 3 ){
...
} else {
...
}
```

```
switch(x){
case 0:
...
case 1:
...
case 2:
...
case 3:
...
default:
...
}
```

Code

- Switch can be faster
- Compiler can produce a *jump table*
- Look up index in table, transfer control to that location
- Compiler doesn’t always use a jump table, but code should never be *worse* in switch compared to if-else

Sources

- Microsoft. https://msdn.microsoft.com/en-us/library/windows/desktop/dd317756(v=vs.85).aspx
- Joel Spolsky. The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!). https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/
- https://www.germanveryeasy.com/eszett
- Wikipedia (German)
- F. Yergeau. UTF-8, A Transformation Format of ISO 10646. http://www.ietf.org/rfc/rfc3629.txt
- Kurt Guntheroth. Optimized C++. O'Reilly Media.

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org