Writing Optimized Code

ITP 435
Week 1, Lecture 2

On Performance and Big-O

- A fun musing based on a talk given by Bjarne Stroustrup several years ago…you can find his talk on YouTube

Here's a Problem

- Given ***N*** random integers, write code that inserts them into a collection in ascending order.
- (After each insertion, the collection must remain sorted – you can't just insert and then sort at the end)

- Eg. if the numbers are 5, 1, 4, 2 they should appear in the collection as:
  { 1, 2, 4, 5 }

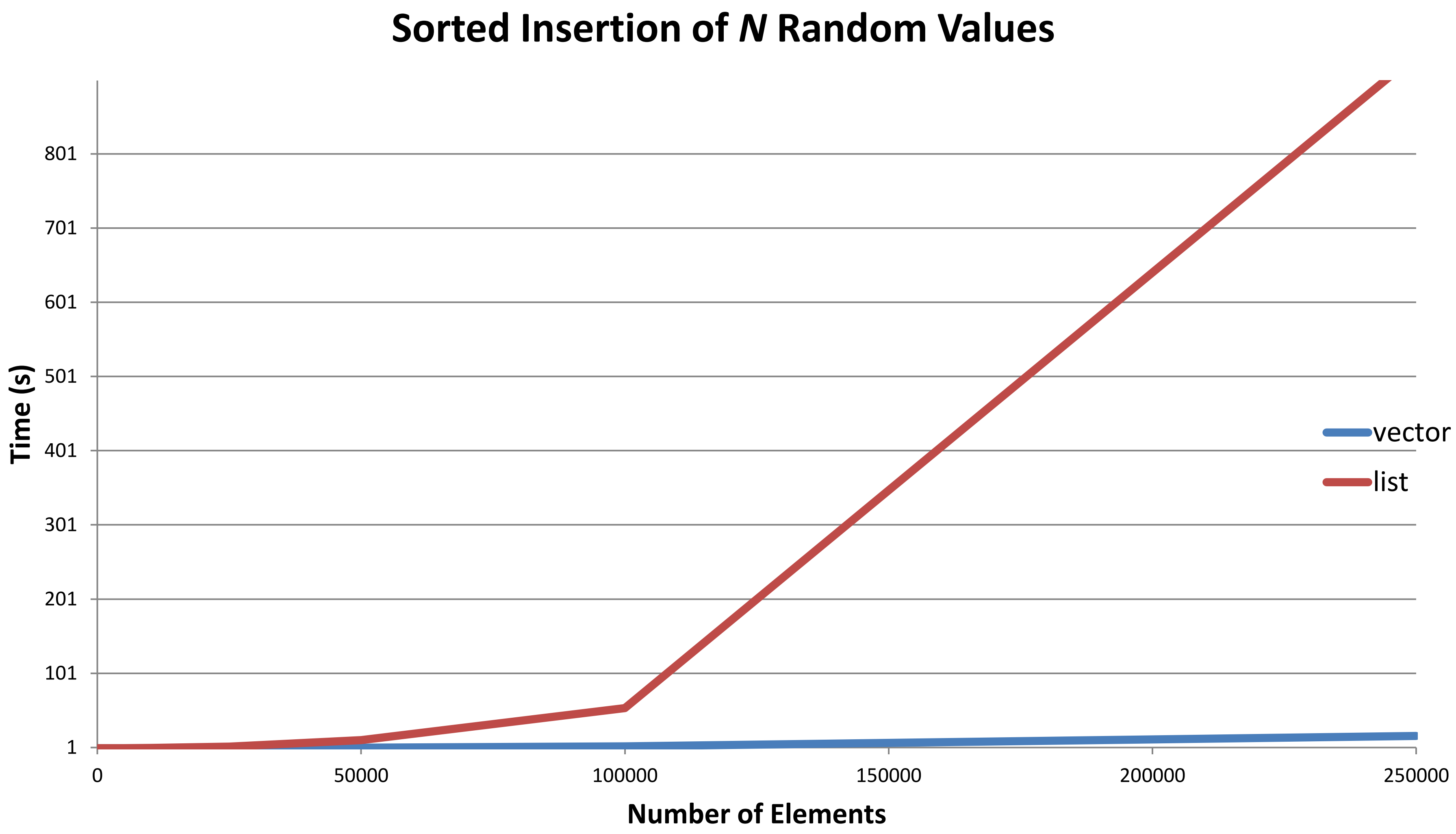
- ***Q***: For which ***N*** is it better to use a linked list instead of a vector?

Code

- The code can be something like this:

```
Timer t;
srand(123456); // Use a set seed
std::vector<int> v;
t.start();
for (size_t i = 0; i < count; i++) {
   int number = rand() % 10000;
   auto iter = v.begin();
   auto end = v.end();
   while (iter != end) {
       if (*iter > number) {
         break;
      }
      ++iter;
   }
   v.insert(iter, number);
}
double elapsed = t.getElapsedMs();
```

Results



Sorted Insertion of *N* Random Values
Time (s)
801
701
601
501
401
301
201
101
1
0
50000
100000
150000
200000
250000
Number of Elements
vector
list

But wait...

- Linear search is ***O(n)***, whether it's a vector or list

- Insertion at an arbitrary location in a list is ***O(1)***
- Insertion into an arbitrary location in a vector is ***O(n)***

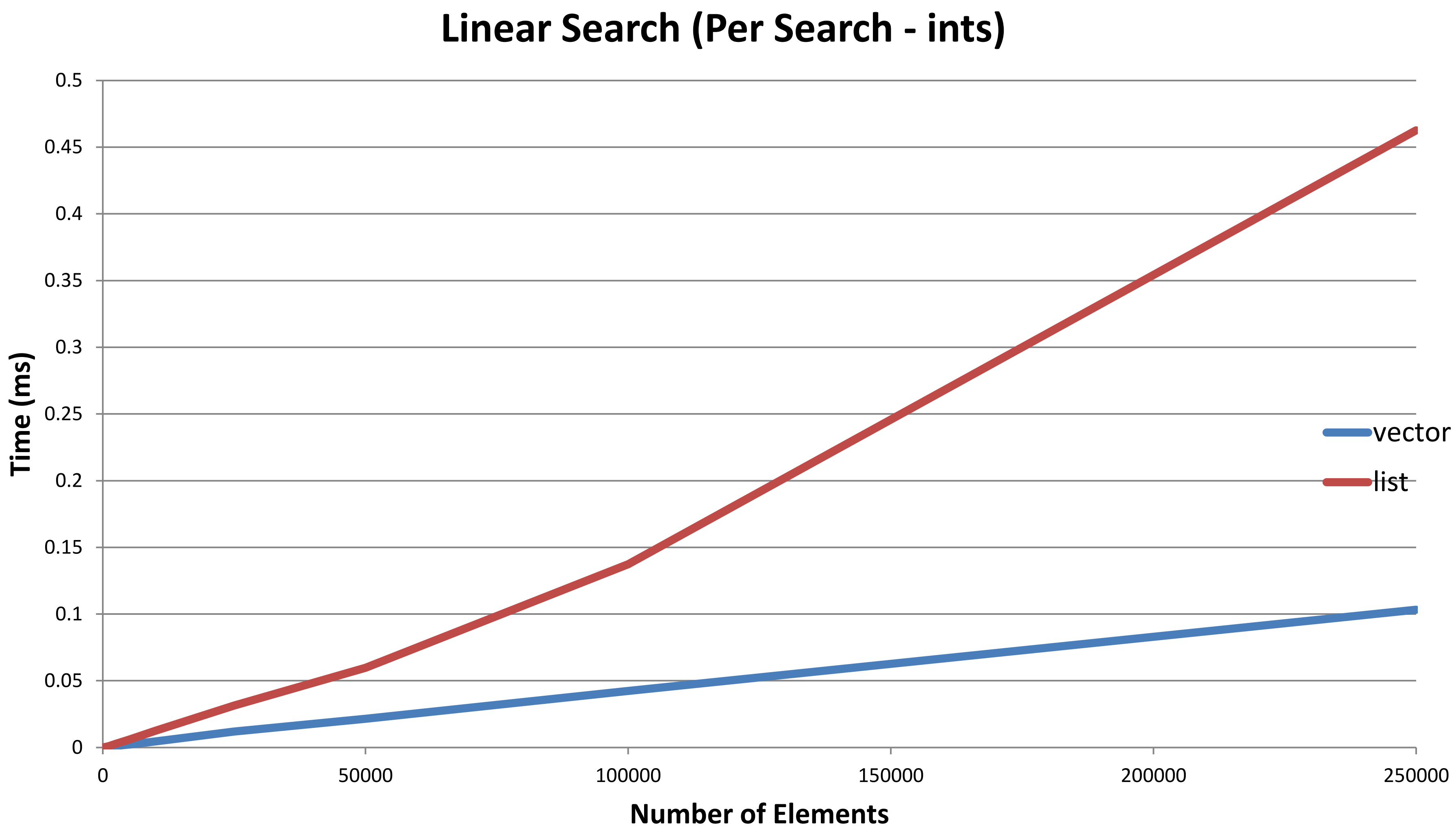
- This would lead you to believe that the list should perform better!

Linear Search of Vector/List

- A linear search over a vector or a list should be an ***O(n)*** operation

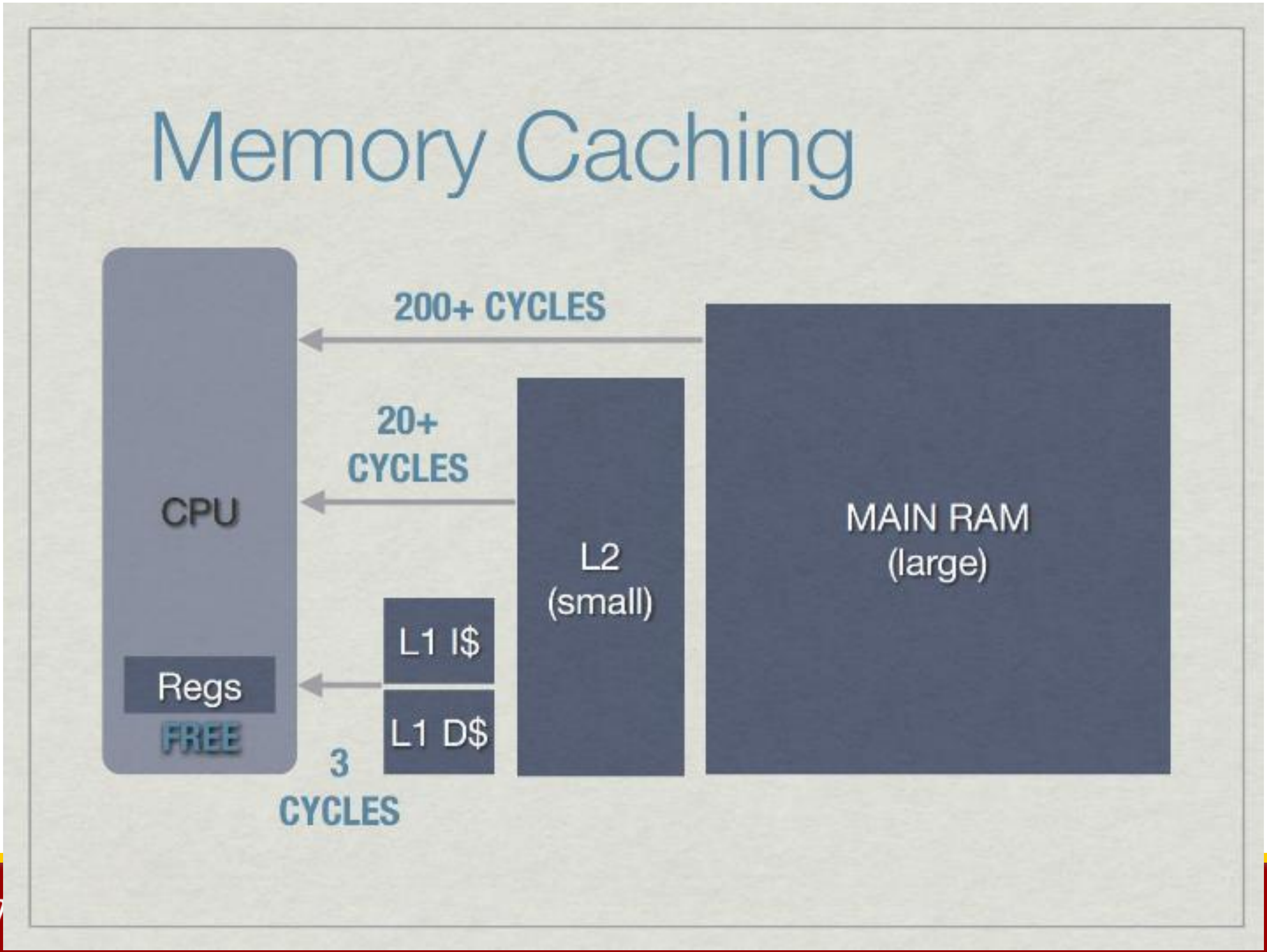
- However, we it turns out that all ***O(n)***s are not created equal!

Linear Search Per Search – ints



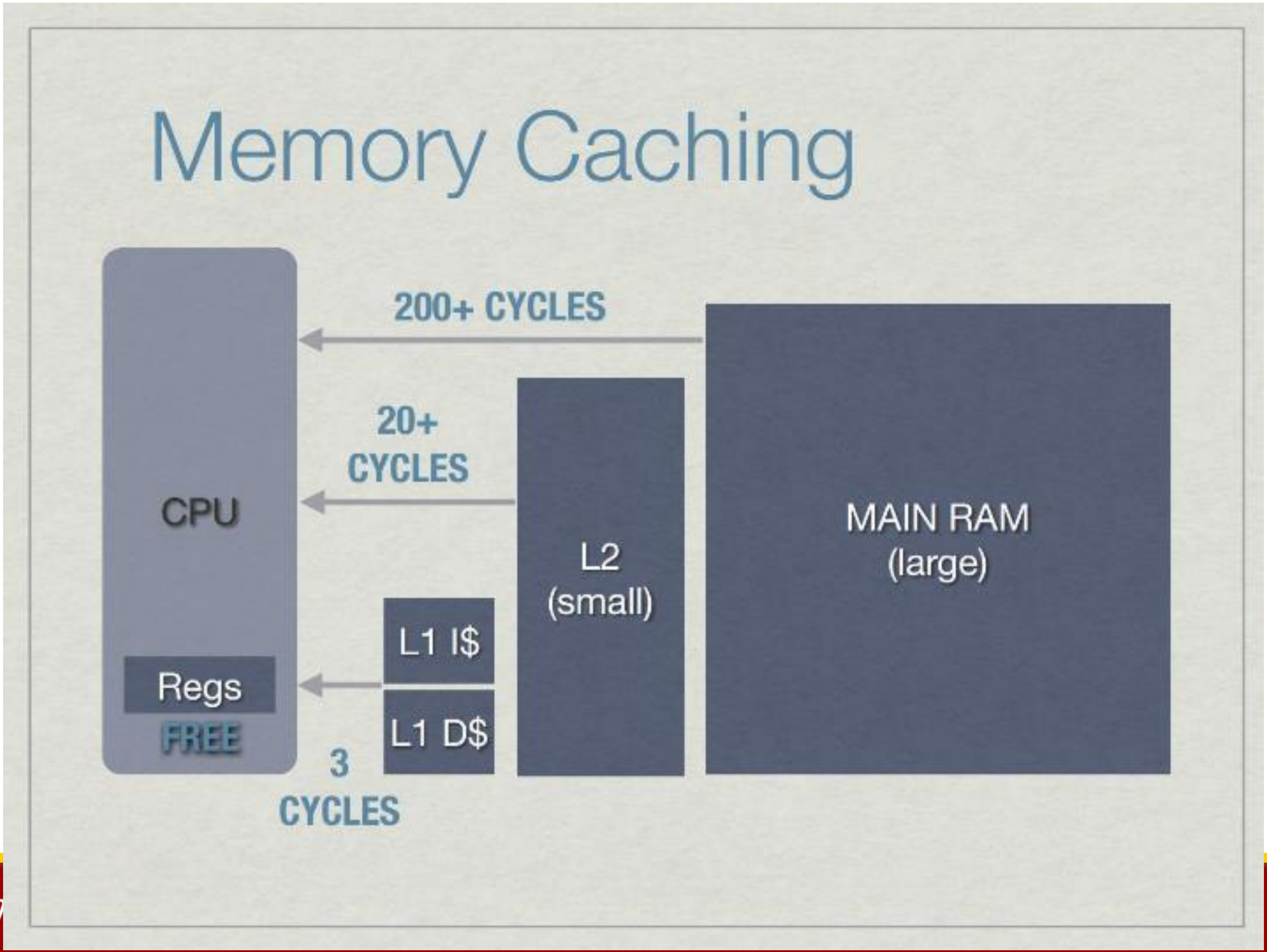
Why?

- So why are we getting these results?



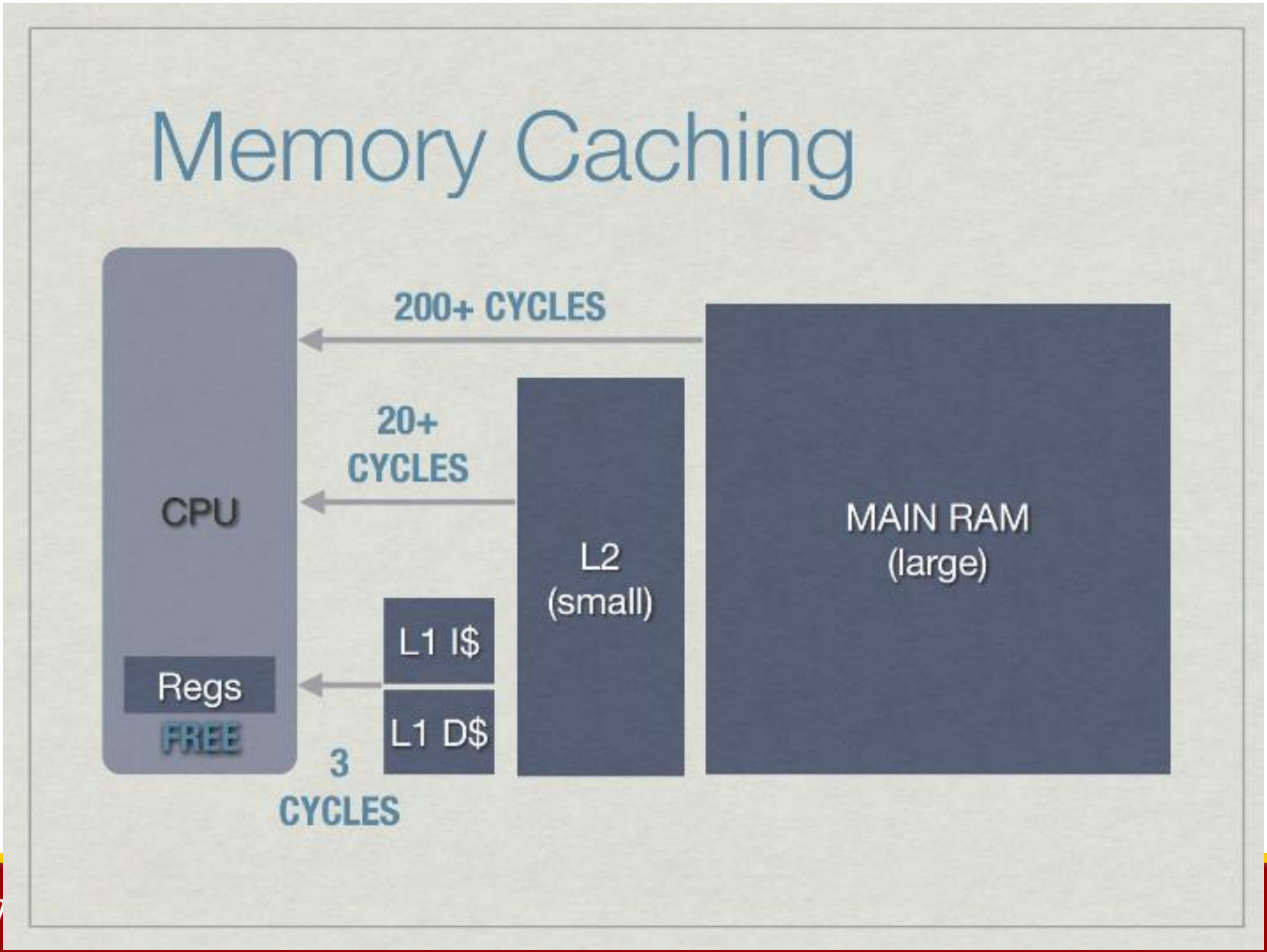
A Quick Explanation of the Cache

- If the CPU requests a value not in the cache, it's a ***cache miss***

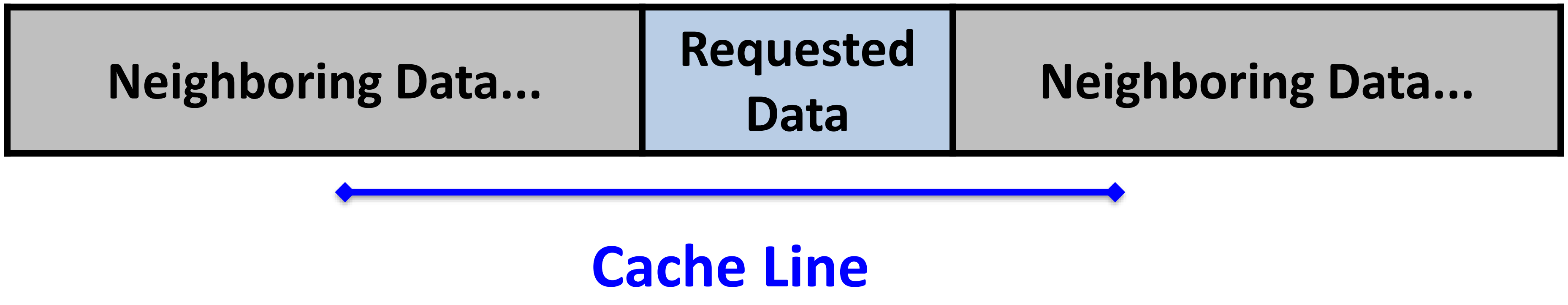


A Quick Explanation of the Cache

- If the CPU requests a value is in the cache, it's a ***cache hit***

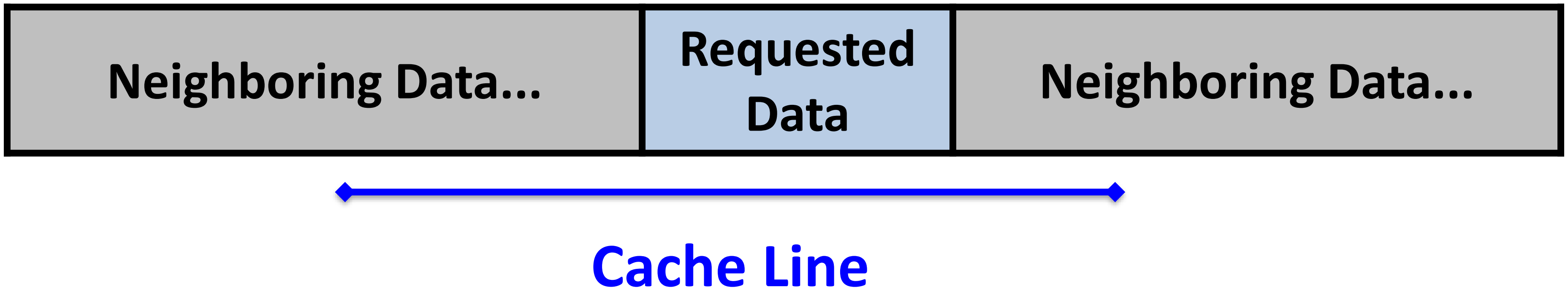


A Quick Explanation of the Cache



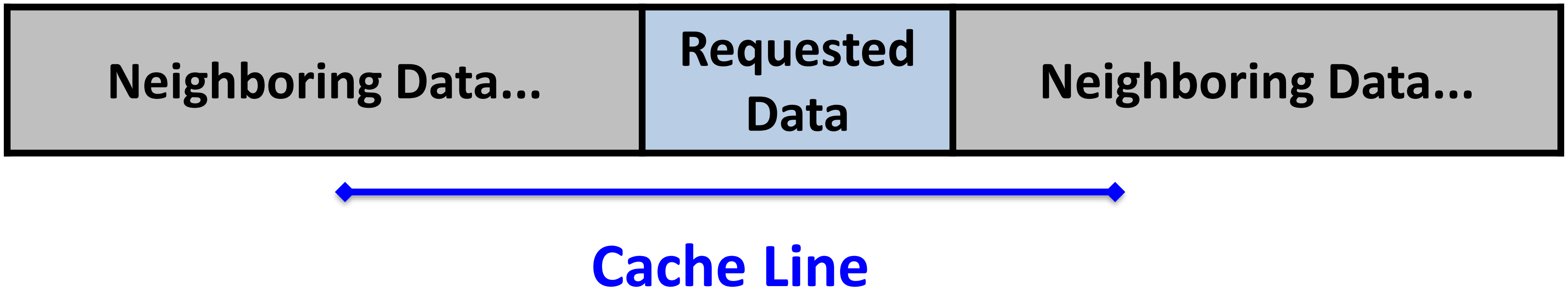
- When there's a cache miss, the cache loads in the requested data as well as the data immediately next to it in memory – this area is called a ***cache line***

A Quick Explanation of the Cache



- This means that there is a high desirability to have data next to the data you will be accessing soon!

A Quick Explanation of the Cache



- Most modern CPUs have a surprisingly small cache line – 64 bytes are loaded at a time

- (This is mostly because the L1 cache is quite small)

Vector Memory Layout

- In a vector, everything is right next to each other in memory

Index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Value	…	…	…	…	…	…	…	…	…

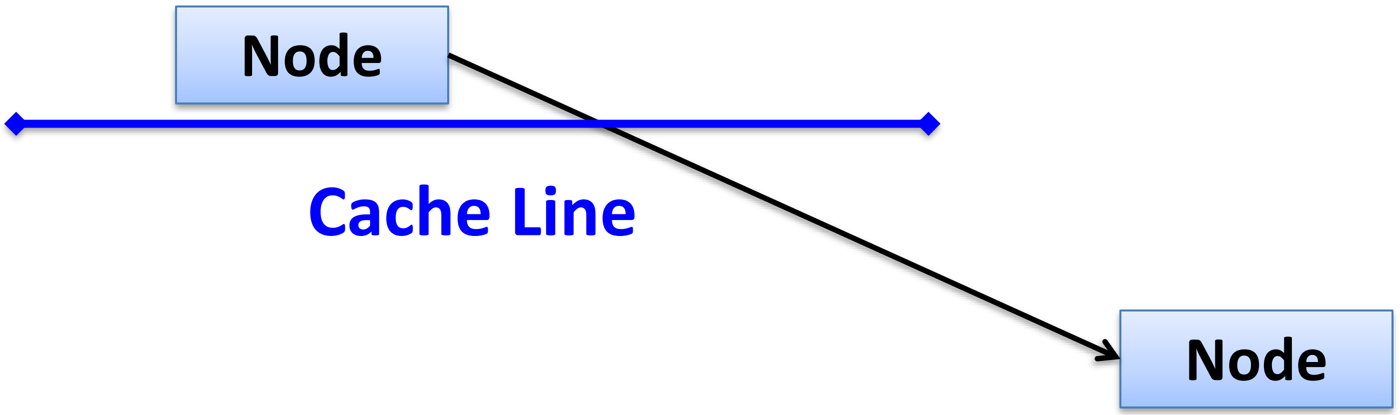
Cache Line

- Assuming ints...when index 0 is loaded into the cache, you get indices 1-15 for free

- If each cache miss is 200 cycles, you're spreading out that cost over 16 elements
- So with a vector 200/16 = **12.5 cycles/element**

Linked List Memory Layout

- By its nature, a linked list is all over the place in memory



- Thus one cache miss (200 cycles) will usually yield only 1 element ☹

Here’s a Quote

“Modern CPUs employ pipelining as well as techniques like hardware threading, out-of-order execution and instruction-level parallelism to utilize resources as effectively as possible. In spite of this, some types of software patterns and algorithms still result in inefficiencies. **For example, linked data structures are commonly used in software, but cause indirect addressing that can defeat hardware prefetchers. In many cases, this behavior can create bubbles of idleness in the pipeline while data is retrieved and there are no other instructions to execute**” – Intel on “How to Tune Applications”

Disclaimer

- This ***does not mean*** that Big-O is useless
- Understanding algorithmic complexity is important, and for example an ***O(n³)*** algorithm is almost always going to perform worse than an ***O(n)*** one

In-class Week 1 Lecture 2

- Big-O review

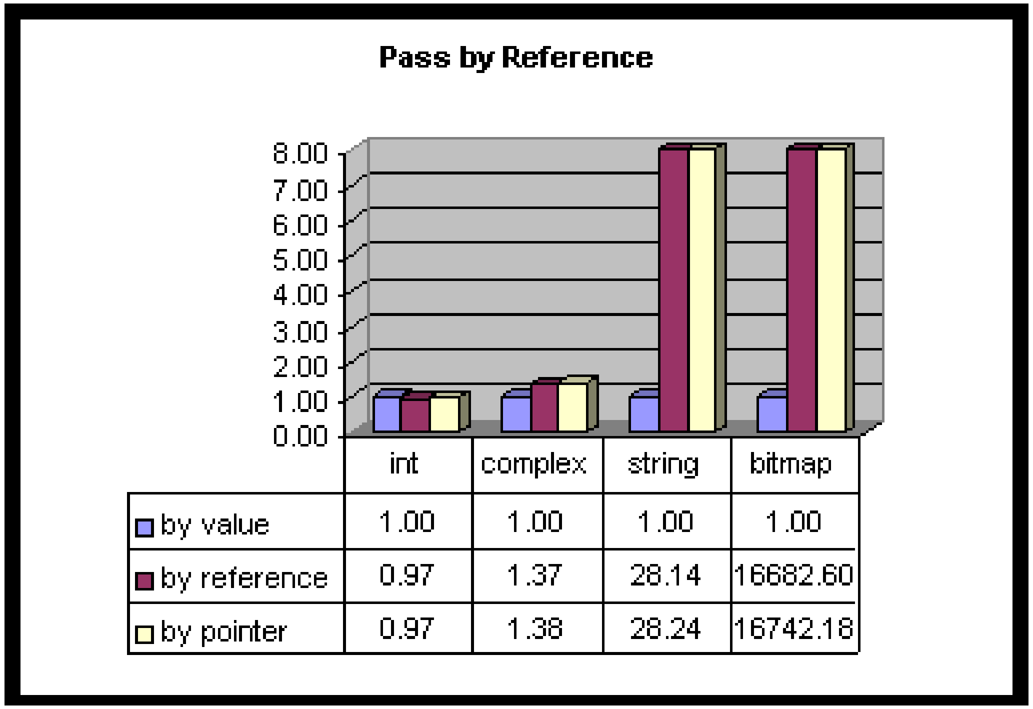
Making your code faster!!

Some Ideas

- Don’t do heavy optimization before your code works
- But there’s simple ideas you can follow that will help make your “unoptimized” code faster – aka “optimize as you go”

Good Rules to Follow

- Non-basic types should be passed by reference when possible:



	int	complex	string	bitmap
🟦 by value	1.00	1.00	1.00	1.00
🟥 by reference	0.97	1.37	28.14	16682.60
🟨 by pointer	0.97	1.38	28.24	16742.18

- ***Why?*** Because copying data is slow

Good Rules to Follow (cont'd)

- Postpone variable declarations when possible:

```
// Declare Outside (b is true half the time)
T x;
if (b) {
    x = t;
    // Do stuff...
}
// Declare Inside (b is true half the time)
if (b) {
    T x = t;
    // Do stuff...
}
```

- ***Why?*** Can avoid potentially expensive constructor operations

Good Rules to Follow (cont'd)

- Postpone variable declarations when possible:
- ***CAVEAT!***
- For non-basic types, avoid constructing on each iteration of a loop:

```
// This constructs a string on every iteration
for (...) {
    std::string myString;
}

// This constructs one string total
std::string myString;
for (...) {
}
```

Good Rules to Follow (cont'd)

- Prefer operator+= to operator+ (and like operators)

```
struct Vector2
{
     float X;
     float Y;

     Vector2& operator+=(const Vector2& rhs)
     {
          X += rhs.X;
          Y += rhs.Y;

          return *this;
     }

     Vector2 operator+(const Vector2& rhs)
     {
          Vector2 temp(*this);
          temp += rhs;
          return temp;
     }
};
```

- ***Why?*** Can avoid making an extra temporary object

Good Rules to Follow (cont'd)

- Use prefix instead of postfix for non-basic types:

```
const T T::operator++ (int) // postfix
{
    T orig(*this); // Have to make copy in postfix
    ++(*this); // call prefix operator
    return (orig);
}
```

- ***Why?*** Can avoid making an extra temporary object

Prefer switches to chains of if-else statements

- Instead of:

```
void GetAndProcessResult() {
   const int result = GetResult();
   if (result == DOWNLOADED)
      return ProcessDownloadedFile();
   else if (result == NEEDS_DOWNLOAD)
      return DownloadFile();
   else if (result == NOT_AVAILABLE)
      return ReportNotAvailable();
   else if (result == ERROR)
      return ReportError();
}
```

Prefer switches to chains of if-else statements

- Do:

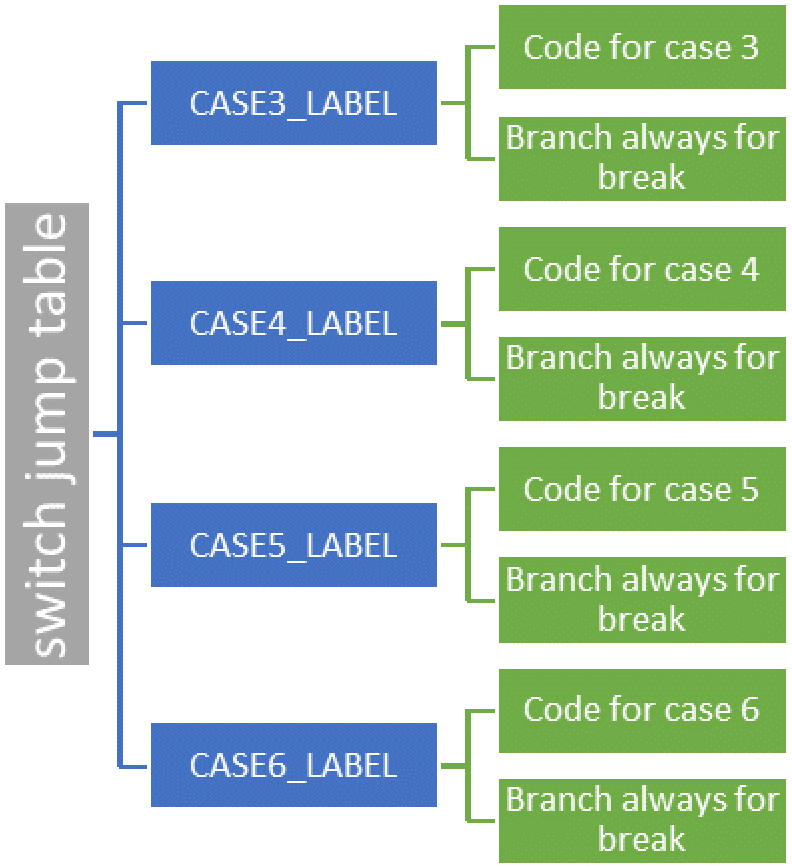
```
void GetAndProcessResult() {
    switch (GetResult()) {
        case DOWNLOADED:
            return ProcessDownloadedFile();
        case NEEDS_DOWNLOAD:
            return DownloadFile();
        case NOT_AVAILABLE:
            return ReportNotAvailable();
        case ERROR:
            return ReportError();
    }
}
```

Prefer switches to chains of if-else statements

- ***Why?***
- Switches can be better optimized such that they can figure out where to go in either *O(1)* or *O(log n)*
- if-else chain requires *O(n)* conditionals to figure out where to go

How can a switch be O(1)

- If we have high density of values in a range (like case 0, 1, 2, etc..) the compiler can use a ***jump table*** to jump to the correct case



Be careful with strings!

- std::string ***is not a basic type***
- Constructing a string might incur a dynamic allocation (depending on implementation)
- Appending to a string dynamically is expensive if it grows (it reallocates like vector!)

Growing Strings (Benchmark for big run)

Method	Time Spent (ms)
`std::string::append`	1399
`std::stringstream`	5102
`std::string::append` but first reserving with `std::string::reserve`	851

- (From ACCU 2015 talk by Arjan Ven Leeuwen)

Converting int-to-string (Benchmark for big run)

Method	Time Spent (ms)
`std::stringstream`	2959
`std::to_string`	1012
`boost::spirit::karma`	332

- (From ACCU 2015 talk by Arjan Ven Leeuwen)

Converting string-to-int (Benchmark for big run)

Method	Time Spent (ms)
`std::stoi`	3920
`boost::spirit::qi`	1276

- (From ACCU 2015 talk by Arjan Ven Leeuwen)

Prefer `'\n'` to `std::endl`

- Especially when writing to files, `'\n'` is significantly faster
- `std::endl` does two things:
  - Write a new line
  - Flush the buffer

- Flushing to a file is ***really slow***, because it writes to disk

- If you instead use `'\n'`, it will flush in these cases:
  - You manually call `.flush()` on the stream
  - The internal buffer reaches a certain size and it decides to flush
  - You close the file

Tips for PA2

- Instead of:

```
for i …
   for j …
      for k …
         phrase = words[i] + words[j] + words[k]
```

- Do:

```
for i …
   for j …
      ijword = words[i] + words[j];
      for k …
         phrase = ijword + words[k]
```

- Can make the above even better w/ +=
- Use \n!

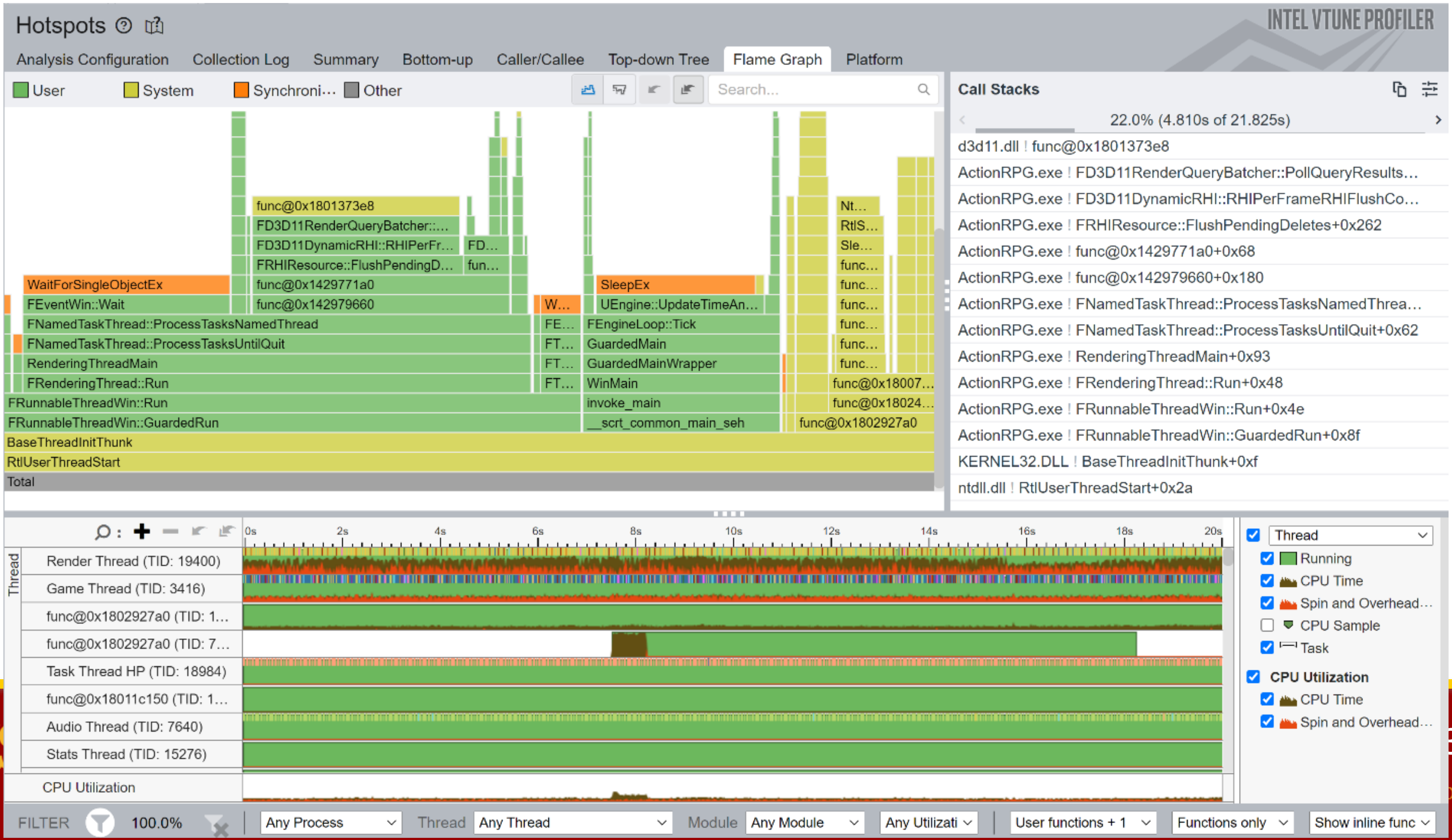
80/20 Rule

- “80-20” rule: 80% of your execution time is spent in 20% of your code

80% of the **results**
only come from **20%** of all **stuff**

Profiler

- Profiling programs can allow us to see which functions take up the most time…
- From Intel VTune:


Hotspots

INTEL VTUNE PROFILER

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Flame Graph | Platform

🟩 User 🟨 System 🟧 Synchroni… ⬛ Other

Search…

Flame Graph:
```
func@0x1801373e8
FD3D11RenderQueryBatcher::…
FD3D11DynamicRHI::RHIPerFr…   FD…
FRHIResource::FlushPendingD…  fun…
WaitForSingleObjectEx   func@0x1429771a0                 SleepEx
FEventWin::Wait         func@0x142979660          W…     UEngine::UpdateTimeAn…
FNamedTaskThread::ProcessTasksNamedThread         FE…   FEngineLoop::Tick
FNamedTaskThread::ProcessTasksUntilQuit           FT…   GuardedMain
RenderingThreadMain                               FT…   GuardedMainWrapper
FRenderingThread::Run                             FT…   WinMain                  func@0x18007…
FRunnableThreadWin::Run                                 invoke_main              func@0x18024…
FRunnableThreadWin::GuardedRun                          __scrt_common_main_seh  func@0x1802927a0
BaseThreadInitThunk
RtlUserThreadStart
Total
```
Nt… RtlS… Sle… func… func… func… func… func… func…

Call Stacks

22.0% (4.810s of 21.825s)

```
d3d11.dll ! func@0x1801373e8
ActionRPG.exe ! FD3D11RenderQueryBatcher::PollQueryResults…
ActionRPG.exe ! FD3D11DynamicRHI::RHIPerFrameRHIFlushCo…
ActionRPG.exe ! FRHIResource::FlushPendingDeletes+0x262
ActionRPG.exe ! func@0x1429771a0+0x68
ActionRPG.exe ! func@0x142979660+0x180
ActionRPG.exe ! FNamedTaskThread::ProcessTasksNamedThrea…
ActionRPG.exe ! FNamedTaskThread::ProcessTasksUntilQuit+0x62
ActionRPG.exe ! RenderingThreadMain+0x93
ActionRPG.exe ! FRenderingThread::Run+0x48
ActionRPG.exe ! FRunnableThreadWin::Run+0x4e
ActionRPG.exe ! FRunnableThreadWin::GuardedRun+0x8f
KERNEL32.DLL ! BaseThreadInitThunk+0xf
ntdll.dll ! RtlUserThreadStart+0x2a
```

Timeline: 0s 2s 4s 6s 8s 10s 12s 14s 16s 18s 20s

Thread
- Render Thread (TID: 19400)
- Game Thread (TID: 3416)
- func@0x1802927a0 (TID: 1…
- func@0x1802927a0 (TID: 7…
- Task Thread HP (TID: 18984)
- func@0x18011c150 (TID: 1…
- Audio Thread (TID: 7640)
- Stats Thread (TID: 15276)

CPU Utilization

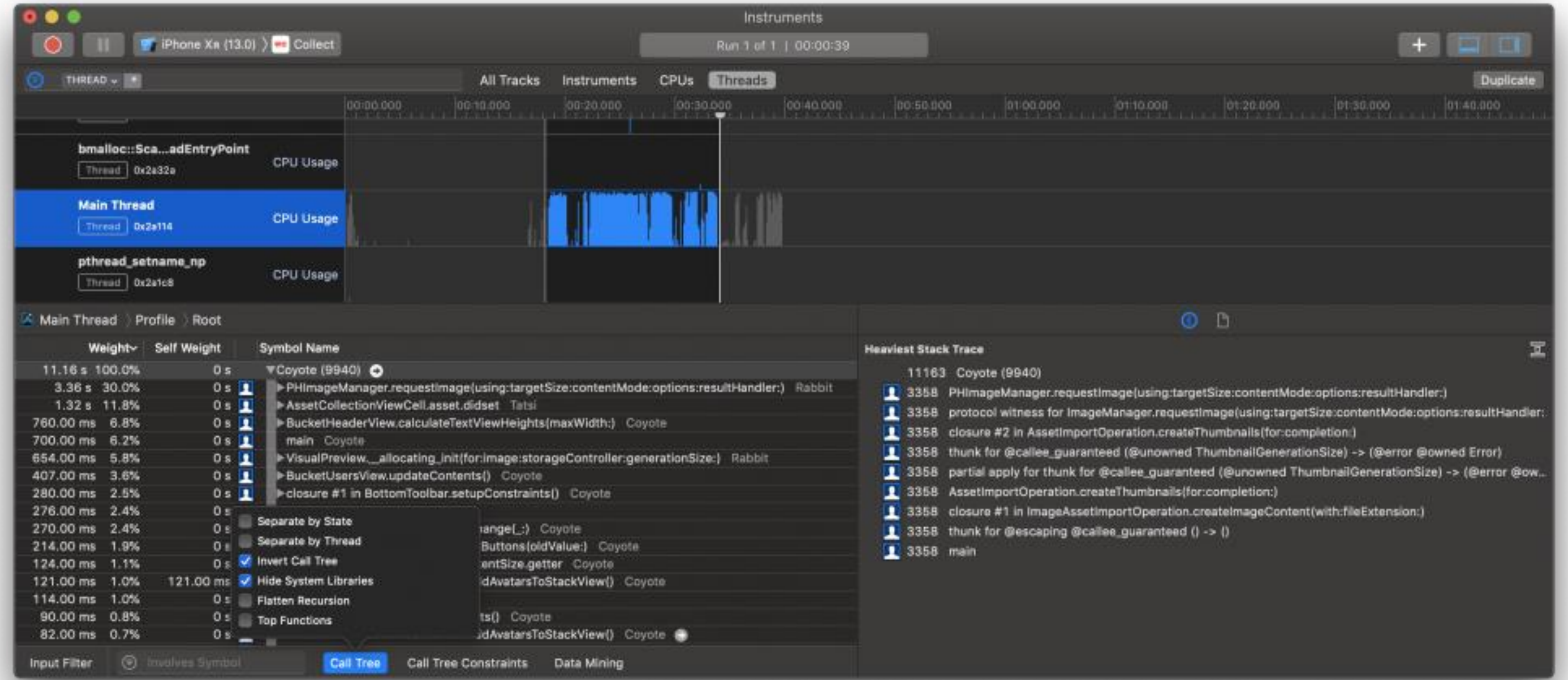
✓ Thread
- ✓ Running
- ✓ CPU Time
- ✓ Spin and Overhead…
- ☐ CPU Sample
- ✓ Task

✓ CPU Utilization
- ✓ CPU Time
- ✓ Spin and Overhead…

FILTER 100.0% | Any Process | Thread: Any Thread | Module: Any Module | Any Utilizati | User functions + 1 | Functions only | Show inline func

Xcode Profiler

- Product>Profile, select "Time Profiler"

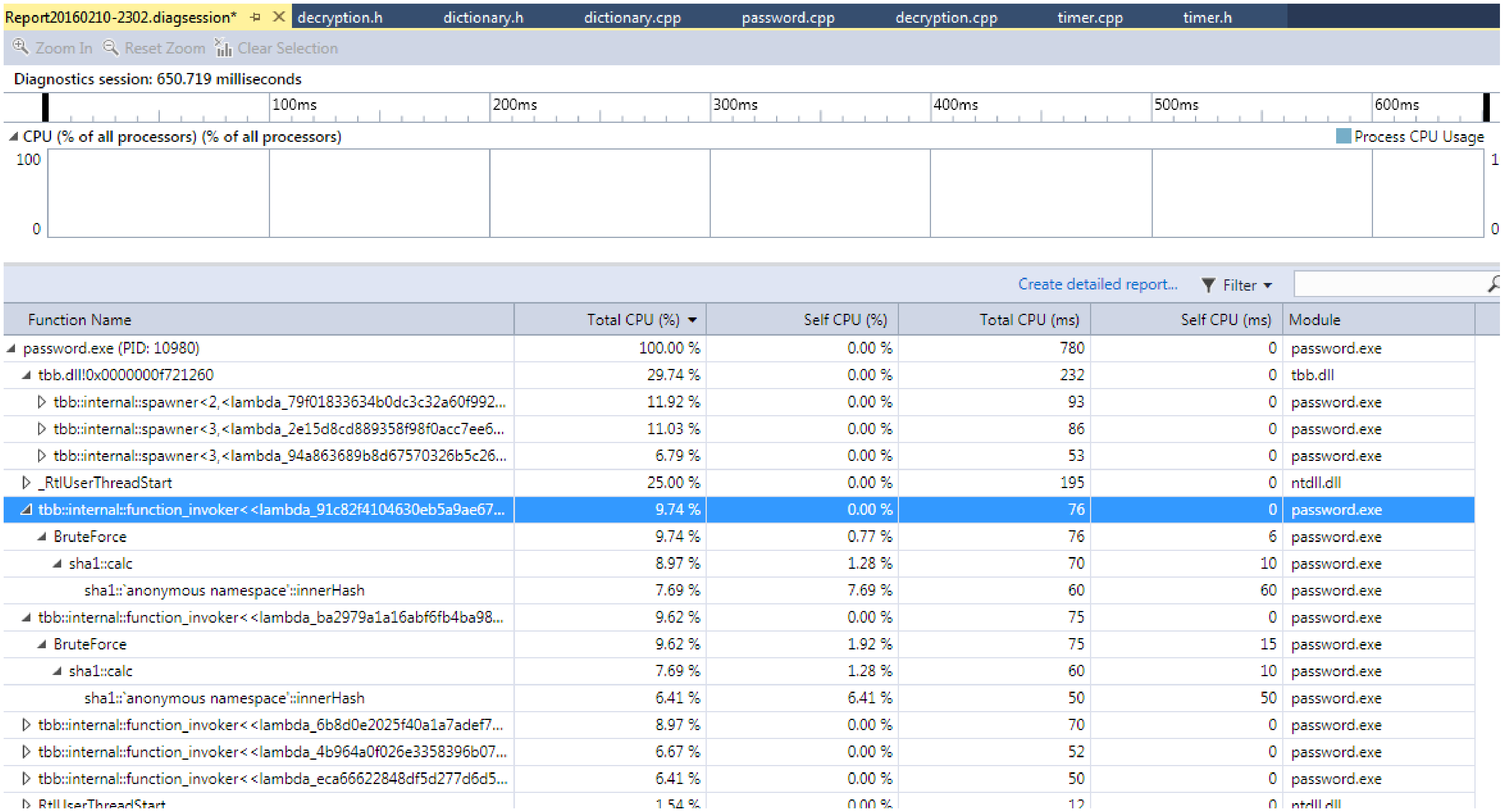


```
   Weight    Self Weight   Symbol Name
  11.16 s 100.0%     0 s   ▼Coyote (9940)
   3.36 s  30.0%     0 s    ▶PHImageManager.requestImage(using:targetSize:contentMode:options:resultHandler:)  Rabbit
   1.32 s  11.8%     0 s    ▶AssetCollectionViewCell.asset.didset  Tatsi
760.00 ms   6.8%     0 s    ▶BucketHeaderView.calculateTextViewHeights(maxWidth:)  Coyote
700.00 ms   6.2%     0 s     main  Coyote
654.00 ms   5.8%     0 s    ▶VisualPreview.__allocating_init(for:image:storageController:generationSize:)  Rabbit
407.00 ms   3.6%     0 s    ▶BucketUsersView.updateContents()  Coyote
280.00 ms   2.5%     0 s    ▶closure #1 in BottomToolbar.setupConstraints()  Coyote
276.00 ms   2.4%     0 s
270.00 ms   2.4%     0 s                    ange(_:)  Coyote
214.00 ms   1.9%     0 s                    Buttons(oldValue:)  Coyote
124.00 ms   1.1%     0 s                    entSize.getter  Coyote
121.00 ms   1.0%  121.00 ms                 dAvatarsToStackView()  Coyote
114.00 ms   1.0%     0 s
 90.00 ms   0.8%     0 s                    ts()  Coyote
 82.00 ms   0.7%     0 s                    dAvatarsToStackView()  Coyote
```

Heaviest Stack Trace

```
11163  Coyote (9940)
 3358  PHImageManager.requestImage(using:targetSize:contentMode:options:resultHandler:)
 3358  protocol witness for ImageManager.requestImage(using:targetSize:contentMode:options:resultHandler:
 3358  closure #2 in AssetImportOperation.createThumbnails(for:completion:)
 3358  thunk for @callee_guaranteed (@unowned ThumbnailGenerationSize) -> (@error @owned Error)
 3358  partial apply for thunk for @callee_guaranteed (@unowned ThumbnailGenerationSize) -> (@error @ow...
 3358  AssetImportOperation.createThumbnails(for:completion:)
 3358  closure #1 in ImageAssetImportOperation.createImageContent(with:fileExtension:)
 3358  thunk for @escaping @callee_guaranteed () -> ()
 3358  main
```

Visual Studio Profiler



Report20160210-2302.diagsession* | decryption.h | dictionary.h | dictionary.cpp | password.cpp | decryption.cpp | timer.cpp | timer.h

Zoom In | Reset Zoom | Clear Selection

Diagnostics session: 650.719 milliseconds

100ms 200ms 300ms 400ms 500ms 600ms

◢ CPU (% of all processors) (% of all processors) — Process CPU Usage

Create detailed report... Filter

Function Name	Total CPU (%)	Self CPU (%)	Total CPU (ms)	Self CPU (ms)	Module
◢ password.exe (PID: 10980)	100.00 %	0.00 %	780	0	password.exe
◢ tbb.dll!0x0000000f721260	29.74 %	0.00 %	232	0	tbb.dll
▷ tbb::internal::spawner<2,<lambda_79f01833634b0dc3c32a60f992...	11.92 %	0.00 %	93	0	password.exe
▷ tbb::internal::spawner<3,<lambda_2e15d8cd889358f98f0acc7ee6...	11.03 %	0.00 %	86	0	password.exe
▷ tbb::internal::spawner<3,<lambda_94a863689b8d67570326b5c26...	6.79 %	0.00 %	53	0	password.exe
▷ _RtlUserThreadStart	25.00 %	0.00 %	195	0	ntdll.dll
◢ tbb::internal::function_invoker<<lambda_91c82f4104630eb5a9ae67...	9.74 %	0.00 %	76	0	password.exe
◢ BruteForce	9.74 %	0.77 %	76	6	password.exe
◢ sha1::calc	8.97 %	1.28 %	70	10	password.exe
sha1::`anonymous namespace'::innerHash	7.69 %	7.69 %	60	60	password.exe
◢ tbb::internal::function_invoker<<lambda_ba2979a1a16abf6fb4ba98...	9.62 %	0.00 %	75	0	password.exe
◢ BruteForce	9.62 %	1.92 %	75	15	password.exe
◢ sha1::calc	7.69 %	1.28 %	60	10	password.exe
sha1::`anonymous namespace'::innerHash	6.41 %	6.41 %	50	50	password.exe
▷ tbb::internal::function_invoker<<lambda_6b8d0e2025f40a1a7adef7...	8.97 %	0.00 %	70	0	password.exe
▷ tbb::internal::function_invoker<<lambda_4b964a0f026e3358396b07...	6.67 %	0.00 %	52	0	password.exe
▷ tbb::internal::function_invoker<<lambda_eca66622848df5d277d6d5...	6.41 %	0.00 %	50	0	password.exe
▷ RtlUserThreadStart	1.54 %	0.00 %	12	0	ntdll.dll

Profiling Example

- Stack Overflow says this is a “great” string split function:

```
std::vector<std::string> SplitStackOverflow(std::string input)
{
   std::stringstream ss(input);
   std::vector<std::string> result;
   while (ss.good())
   {
      std::string substr;
      std::getline(ss, substr, ',’);
      result.push_back(substr);
   }
   return result;
}
```

Profiling Example (cont'd)

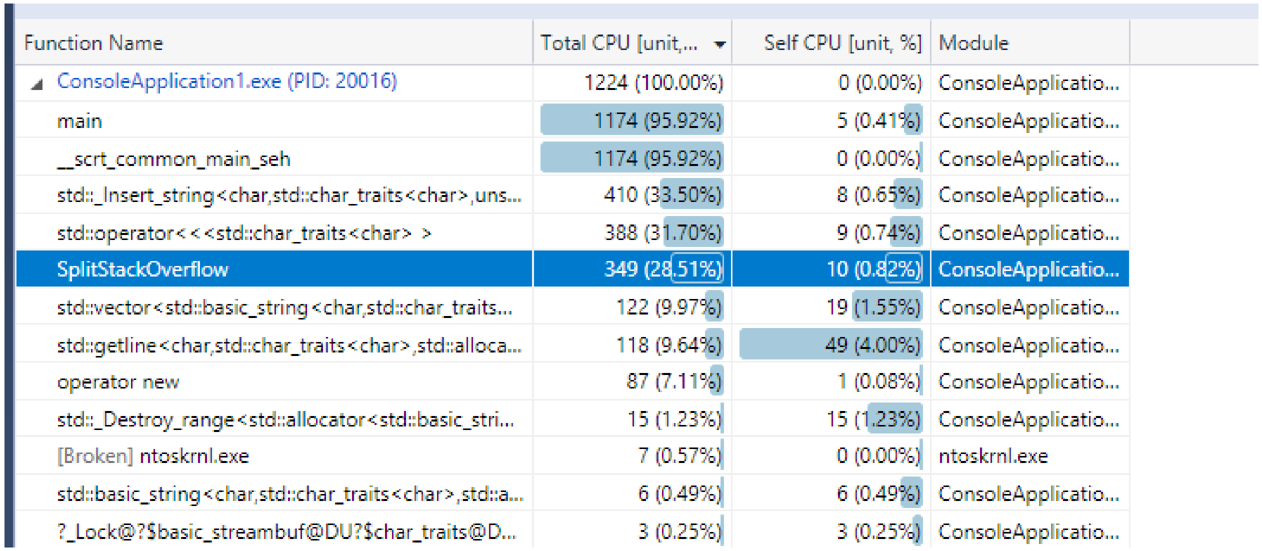
- Use this code to profile it (in optimized/release build)

```
// Top 20 US census names 1900
std::string
str("James,Mary,John,Patricia,Robert,Linda,Michael,Barbara,"
"William,Elizabeth,David,Jennifer,Richard,Maria,Charles,Susan,"
"Joseph,Margaret,Thomas,Dorothy");

std::vector<std::string> result;
for (int i = 0; i < 100000; i++)
{
   result = SplitStackOverflow(str);
   std::cout << result[0] << '\n';
}
```

Profiling Example (cont'd)

- Here's the overall profile from Visual Studio



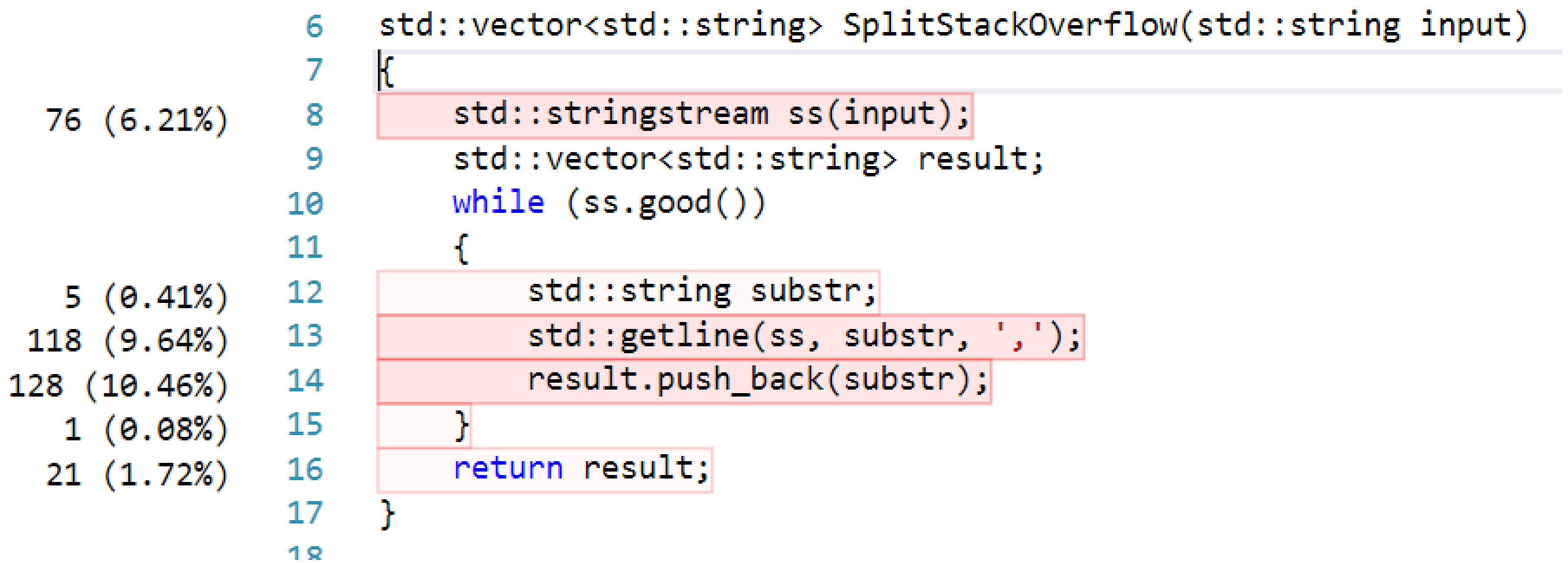
Function Name	Total CPU [unit,...	Self CPU [unit, %]	Module
◢ ConsoleApplication1.exe (PID: 20016)	1224 (100.00%)	0 (0.00%)	ConsoleApplicatio...
main	1174 (95.92%)	5 (0.41%)	ConsoleApplicatio...
__scrt_common_main_seh	1174 (95.92%)	0 (0.00%)	ConsoleApplicatio...
std::_Insert_string<char,std::char_traits<char>,uns...	410 (33.50%)	8 (0.65%)	ConsoleApplicatio...
std::operator<<<std::char_traits<char> >	388 (31.70%)	9 (0.74%)	ConsoleApplicatio...
SplitStackOverflow	349 (28.51%)	10 (0.82%)	ConsoleApplicatio...
std::vector<std::basic_string<char,std::char_traits...	122 (9.97%)	19 (1.55%)	ConsoleApplicatio...
std::getline<char,std::char_traits<char>,std::alloca...	118 (9.64%)	49 (4.00%)	ConsoleApplicatio...
operator new	87 (7.11%)	1 (0.08%)	ConsoleApplicatio...
std::_Destroy_range<std::allocator<std::basic_stri...	15 (1.23%)	15 (1.23%)	ConsoleApplicatio...
[Broken] ntoskrnl.exe	7 (0.57%)	0 (0.00%)	ntoskrnl.exe
std::basic_string<char,std::char_traits<char>,std::a...	6 (0.49%)	6 (0.49%)	ConsoleApplicatio...
?_Lock@?$basic_streambuf@DU?$char_traits@D...	3 (0.25%)	3 (0.25%)	ConsoleApplicatio...

- The console output is most expensive, but ignore that (we need the output or the compiler optimizes out everything since the return values aren't used otherwise)

- We spent ~349 ms in SplitStackOverflow

Profiling Example (cont'd)

- Clicking on the function highlights the most expensive lines in the function:



```
                std::vector<std::string> SplitStackOverflow(std::string input)
                {
 76 (6.21%)         std::stringstream ss(input);
                    std::vector<std::string> result;
                    while (ss.good())
                    {
  5 (0.41%)             std::string substr;
118 (9.64%)             std::getline(ss, substr, ',');
128 (10.46%)            result.push_back(substr);
  1 (0.08%)         }
 21 (1.72%)         return result;
                }
```

Profiling Example (cont'd)

- Let's try my version of the function:

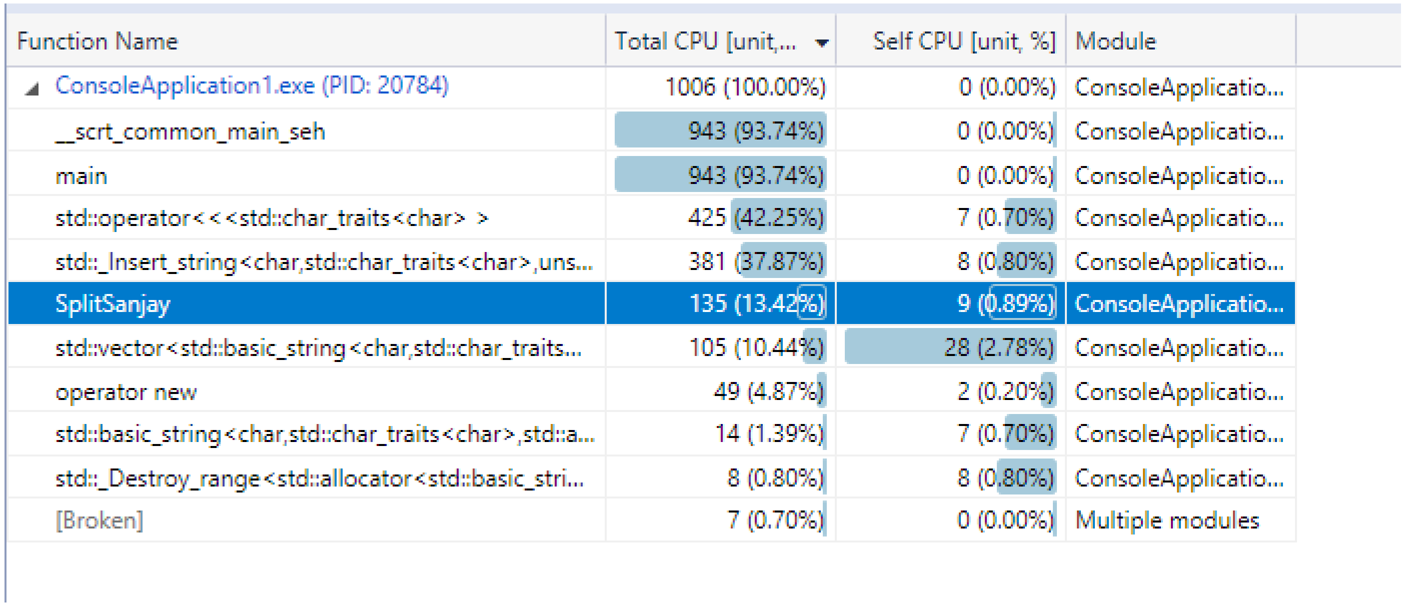
```
std::vector<std::string> SplitSanjay(const std::string& str) {
   const char delim = ',';
   std::vector<std::string> retVal;

   size_t start = 0;
   size_t delimLoc = str.find_first_of(delim, start);
   while (delimLoc != std::string::npos) {
      retVal.emplace_back(str.substr(start, delimLoc - start));
      start = delimLoc + 1;
      delimLoc = str.find_first_of(delim, start);
   }

   retVal.emplace_back(str.substr(start));
   return retVal;
}
```

Profiling Example (cont'd)

- Visual Studio says:

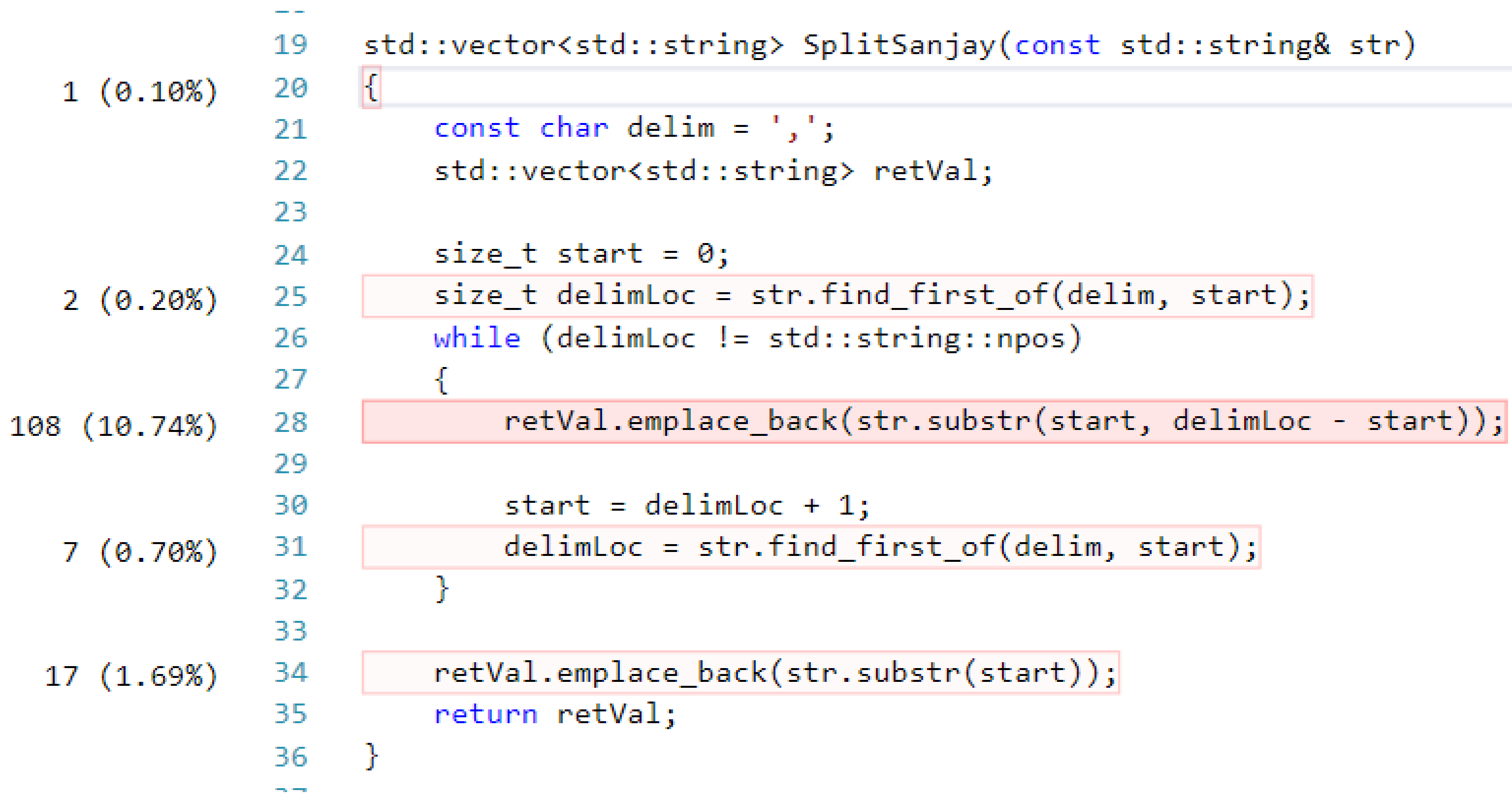


Function Name	Total CPU [unit,...	Self CPU [unit, %]	Module
◢ ConsoleApplication1.exe (PID: 20784)	1006 (100.00%)	0 (0.00%)	ConsoleApplicatio...
__scrt_common_main_seh	943 (93.74%)	0 (0.00%)	ConsoleApplicatio...
main	943 (93.74%)	0 (0.00%)	ConsoleApplicatio...
std::operator<<<std::char_traits<char> >	425 (42.25%)	7 (0.70%)	ConsoleApplicatio...
std::_Insert_string<char,std::char_traits<char>,uns...	381 (37.87%)	8 (0.80%)	ConsoleApplicatio...
SplitSanjay	135 (13.42%)	9 (0.89%)	ConsoleApplicatio...
std::vector<std::basic_string<char,std::char_traits...	105 (10.44%)	28 (2.78%)	ConsoleApplicatio...
operator new	49 (4.87%)	2 (0.20%)	ConsoleApplicatio...
std::basic_string<char,std::char_traits<char>,std::a...	14 (1.39%)	7 (0.70%)	ConsoleApplicatio...
std::_Destroy_range<std::allocator<std::basic_stri...	8 (0.80%)	8 (0.80%)	ConsoleApplicatio...
[Broken]	7 (0.70%)	0 (0.00%)	Multiple modules

- We only spent 135ms in SplitSanjay (compared to 349ms for SplitStackOverflow)
- Overall execution time went from 1.2s to 1s

Profiling Example (cont'd)

- The only hotspot in SplitSanjay is adding the string to the return value vector, which is unavoidable



```
                std::vector<std::string> SplitSanjay(const std::string& str)
1 (0.10%)       {
                    const char delim = ',';
                    std::vector<std::string> retVal;

                    size_t start = 0;
2 (0.20%)           size_t delimLoc = str.find_first_of(delim, start);
                    while (delimLoc != std::string::npos)
                    {
108 (10.74%)            retVal.emplace_back(str.substr(start, delimLoc - start));

                        start = delimLoc + 1;
7 (0.70%)               delimLoc = str.find_first_of(delim, start);
                    }

17 (1.69%)          retVal.emplace_back(str.substr(start));
                    return retVal;
                }
```

Why's SplitSanjay faster?

- By not using stringstream and getline, we avoid two hotspots from SplitStackOverflow

- We take advantage of move semantics and perfect forwarding (passing the substr result directly into emplace_back; we'll talk about perfect forwarding later in the semester)

Moral?

NEVER TRUST STACK OVERFLOW

Profiling your PA Code

- Follow the instructions here to profile your code: https://itp435-20243.github.io/Profiling.html
- (Unfortunately, it does require several steps to get setup)

How to optimize even further?

- Ok, you’ve identified what’s slow. Now how to make it faster?
- Often just algorithmic: If we’ re using an *O(n*^{2}*)* solution, maybe there’ s an *O(n log n)* one?
- If cache misses are a problem, is there more cache-friendly solution?
- Sometimes you also need to think about other lower-level optimizations!

What happens when you call a function?

1. The code has to “prepare” for the call by saving data on the stack
2. Jump to a different code location (where the function you’re calling is) – this might result in an instruction cache miss
3. The function needs to construct/initialize any local variables
4. The function executes its code
5. A return requires cleaning up the stack and jumping back to the caller’s location

What happens when you call a function?

1. The code has to "prepare" for the call by saving data on the stack
2. Jump to a different code location (where the function you're calling is) – this might result in an instruction cache miss
3. The function needs to construct/initialize any local variables
4. The function executes its code
5. A return requires cleaning up the stack and jumping back to the caller's location

For a short function, steps 1, 2, and 5 may be way more expensive than steps 3 and 4!

Inline Functions

- Tell compiler to copy/paste function code at every location it’s called, instead of incurring function call costs

- Method 1: Implementing a function within class declaration suggests you want it inlined:

```
class MyVector3
{
public:
   float DotProduct(const MyVector3& rhs)
   {
      return (x * rhs.x + y * rhs.y + z * rhs.z);
   }
};
```

Inline Functions, cont'd

- Method 2: For standalone functions, use the "inline" for a suggestion:

```
inline void MyInlineFunction()
{
    // Do something
}
```

Force Inlining

- To “force” inlining, which will inline in almost all cases…
- VS:
```
__forceinline void MyInlineFunction() { /* ... */}
```
- clang/gcc:
```
__attribute__((always_inline))
void MyInlineFunction() { /* ... */}
```

- However, there are still some cases where it won’t be inlined:
  - Debug build
  - Recursive function beyond a certain depth
  - Virtual function called virtually
  - Function pointer call
  - And a couple of other rare cases

What happens when you run a loop?

```
// Sum array of 1000 elements
float sum = 0.0f;
for (int i = 0; i < 1000; i++)
{
   sum += myArray[i];
}
```

At the end of every iteration we:

1. Increment
2. Do a conditional check
3. If true, jump back to the top of the loop body

Loop Unrolling, cont'd

- With ***loop unrolling***, we're trying to reduce the number of iterations
- Here's a partially "unrolled" version of the same loop (we've cut the number of iterations to a fourth):

```
// Sum array of 1000 elements
float sum = 0.0f;
for (int i = 0; i < 1000; i+=4)
{
   sum += myArray[i] + myArray[i + 1] +
          myArray[i + 2] + myArray[i + 3];
}
```

Why loop unroll?

- Every time we reach the end of an iteration, we have to increment, check if the condition is true, and jump back to the start of the loop body if so
- If the code inside the loop is smaller than this overhead, it might be worth to unroll
- ***Bonus***: When doing numerical computations as the previous example, the unrolled loop is more likely to be identified as a candidate for vectorization (SIMD)

Return Value Optimization

- When returning a class by value, on some compilers it may be more efficient to return an *unnamed* class as opposed to a named one

```
template <class T> T Original(const T& tValue)
{
    T tResult; // named object; optimization potential low
    tResult = tValue;
    return (tResult);
}

template <class T> T Optimized(const T& tValue)
{
    return T(tValue); // unnamed; optimization potential high
}
```

Small String Optimization

- The ***small string optimization*** assumes that most strings are small:
  - For these small strings, we can avoid the dynamic allocation by using the 24 bytes differently
  - This does mean all string operations are slightly more expensive, but given how dominant dynamic memory management is to performance, it's better!

Unions

- We have 24 bytes. We want to pick from one of these two memory layouts, depending on whether the string is big or small:

Normal (Large) String	**Small String**

`char* mData;`
`size_t mSize;`
`size_t mCapacity;`
	`char mData[22];`
	`char mSize;`
	`bool mIsSmallString;`

- *(So the "small" string can have up to 22 bytes of data in the char buffer)*
- *(I had to reorder large because of endianness)*

Union

- A ***union*** says "I want to refer to this memory in multiple ways"
- In our case, we want to have a struct for Large and a struct for Small, and be able to pick between the two (the syntax is a little gnarly)

Union

```
union {
   struct {
      char* mData;
      size_t mSize;
      size_t mCapacity;
   } Large;
   struct {
      char mData[22];
      char mSize;
      bool mIsSmallString;
   } Small;
} mStr;
```

Union

```
union {
   struct {
      char* mData;
      size_t mSize;
      size_t mCapacity;
   } Large;
   struct {
      char mData[22];
      char mSize;
      bool mIsSmallString;
   } Small;
} mStr;
```

This struct is if I decide I want a "Large" string

Union

```
union {
   struct {
      char* mData;
      size_t mSize;
      size_t mCapacity;
   } Large;
   struct {
      char mData[22];
      char mSize;
      bool mIsSmallString;
   } Small;
} mStr;
```

This struct is if I decide I want a "Small" string

Union

```
union {
   struct {
      char* mData;
      size_t mSize;
      size_t mCapacity;
   } Large;
   struct {
      char mData[22];
      char mSize;
      bool mIsSmallString;
   } Small;
} mStr;
```

This is the name of the member data in String.

Union

```
union {
   struct {
      char* mData;
      size_t mSize;
      size_t mCapacity;
   } Large;
   struct {
      char mData[22];
      char mSize;
      bool mIsSmallString;
   } Small;
} mStr;
```

Eg. If I want to access mSize in "Large" I would say:

```
mStr.Large.mSize
```

Implementing the Constructor

- I now have to decide which I want based on strlen

```
String(const char* str) {
   size_t len = std::strlen(str);
   // If the str is short enough, we can use the "small" string
   if (len <= 21) {
      mStr.Small.mIsSmallString = true;
      mStr.Small.mSize = len;
      std::memcpy(mStr.Small.mData, str, len + 1);
   } else {
      // We have to use "large" string :(
      mStr.Large.mSize = len;
      mStr.Large.mCapacity = len + 1;
      mStr.Large.mData = new char[mStr.Large.mCapacity];
      std::memcpy(mStr.Large.mData, str, len + 1);
   }
}
```

Implementing the rest…

- Now, basically every function needs to check if it’s a small string:

```
size_t Capacity() const {
   if (mStr.Small.mIsSmallString) {
      return 21;
   } else {
      return mStr.Large.mCapacity;
   }
}
```

- (So, the tradeoff is that we add an additional condition to every function)

The user can be oblivious to our optimization!

```
String small("Hello");
String large("This string is so long omgggggggg");
```

Gotcha

- Because of the byte layout of the union, the most significant byte of mCapacity conflicts with mIsSmallString
- This means our actual maximum capacity is now **2^{56}** (oh noes)

A Note on Small String

- With some bitwise manipulation, we can get 1 more byte in the char array, but it starts to get a bit tricky

- Some STL implementations do use small string by default!!