Scribe Notes for November 13
By:
Roy Zhang
Karl Krehbiel
Rabeea Ahmed

Systems Optimization

Overview of today:
-Understanding compiler optimization blockers
-Code motion
-Profiling tools
-Library perf debugging

How does a French person say “sup?”
“Soup”

How to generate a huge file (for debugging purposes):

Generate a huge file filled with Ys:
```
yes y | head - c $((1 << 30)) > f
```

double the file:
cat f f > g (double file f)
cat g g > h (double file g)

“yes” is a dumb command from ancient days.
“head” prints the beginning lines in a file
“-c” argument it prints the first *characters*
“((1 << 30))” is number of characters
“cat” takes input files and writes them to stdio

Making programs faster:

1. Improve locality
2. Reduce expensive operations (such as system calls)
3. Faster Algorithm(s)
4. Better Hardware
5. Avoid optimization blocker

Consequence of applying these strategies:
Program breaks because you make a mistake.

“Best speedup = working program” - Hurgmon Dinkszheimer
“Premature optimization is the root of all evil” -Knuth

Optimization is premature when:
1. You don't know how fast your code is running
2. Your code doesn't work.

strtol program:

```
void strtolower(char *s) {
    for (size_t i = 0; i < strlen(s); ++i)
     if (s[i] >= 'A' && s[i] <= 'Z')
         s[i] += 'a' - 'A';
}
```

-read a dictionary into a single string
-make the whole string lower case

Our first attempt takes 1.69 seconds
But we need a baseline!
Our baseline will be .13s for 100,000,000
Now, how long will our first attempt take for those 100,000,000 characters?
Why is our first attempt so slow?

-`for (size_t i = 0; i < strlen(s); ++i)`

-Calculates the string length strlen times. So the complexity is O(n^2)... very slow

Let's pull out that pesky strlen command to get:

```
void strtolower(char *s) {
    size_t len = strlen(s);
    for (size_t i = 0; i < len; ++i)
     if (s[i] >= 'A' && s[i] <= 'Z')
         s[i] += 'a' - 'A';
}
```

This runs at 0.26s for 100,000,000 characters!
We have removed an **optimization blocker**! This specific blocker is **aliasing**

Aliasing:

```
Options:
     1. for (size_t i = 0; i < strlen(s); ++i)
             /* stuff modifying s*/
     2. for (size_t i = 0; i < strlen(s); ++i)
             /* stuff not modifying s*/
```

"As-If" Rule: Optimizer can transform code as long as it behaves AS IF original code ran.

-For option 2, the optimizer can tell that the code in the loop will not modify s, so it can pull the loop condition out of the loop.

- It needs to KNOW strlen to be able to do this...
  - and it can sometimes know your code, but not always.

Why is it called aliasing?

Values in the loop might point to same data as one in loop condition

New Program: *Service Translate*

What it does:

-Take input file and connect it to port number

port number in networking used for differ

Sample Inputs and Ports:

Input	**Output**
www	www, 80
https	https, 443
blah	blah, 0
foo	foo, 0
ftp	ftp, 21

serviceTranslate:

	Size	time taken
Attempt 0	Small	1.02s
Attempt 0	Large	Too long.
Attempt 1- w/ nlines	Small	0.40s
Attempt 2 - w/ cache	Small	.02s
Attempt 2 - w/cache	Large	2.84s
Attempt 3 - w/better protocol	Large	2.40

DeSlowing it:

-Opportunity for code motion?

-countlines function is really not fast so let's pull it out of the loop

-now attempt 1

-Use a different API?

-We're gonna need a better tool...

Performance Measurement Tools:

1. *gprof* =Sampling Profiler

   takes timed random samples of the program's state and reports

2. *perf* = Whole-system profiler

   App + kernel

3. *gdb* = Human Sampling

using gprof:
without nlines, we have that all of the calls were in countlines which says that it is slow
with nlines, apparently only next_line is called
important caveat: gprof *doesn't include the performance of library functions or system calls in its results"*

using perf:

can't run on command line. Run it as root.

Run in two phases:

1. RECORD

   sudo perf record[command line]

2. REPORT

   sudo perf report

lists the most expensive things in reverse-order

using stupid debugging (GDB):

hit control c see where we stop. then backtrace

So why is our program so slow?

Because we keep opening... reading through... and closing the etsy services file... that's unnecessary in getsserverbyname. it makes complexity greater by a constant proportional to size of etsy services file. We will fix this by reading in the services file once... **Caching!**

Servicetranslate05.c has this caching and it's.... way faster!!!

Use perf again

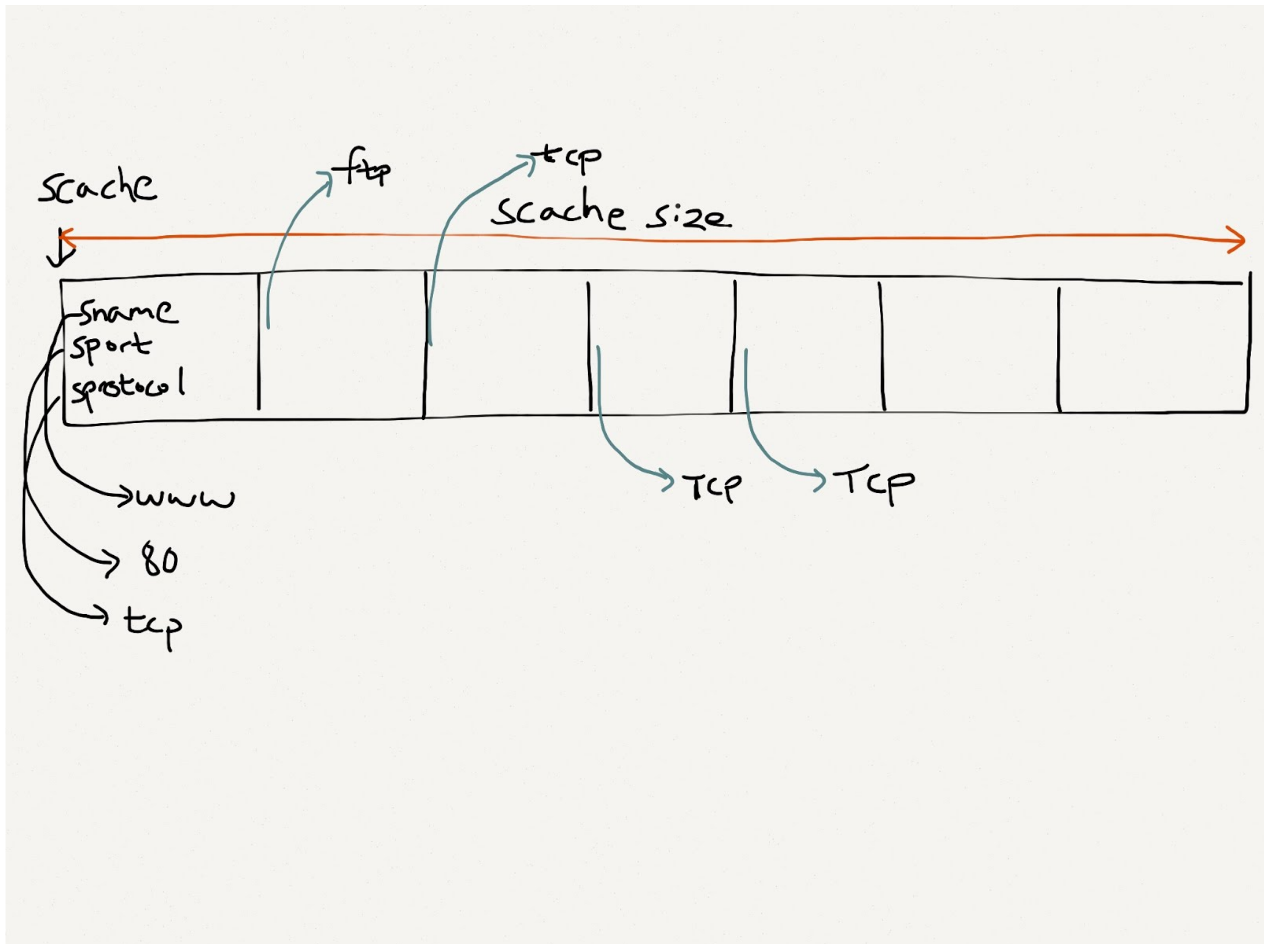
see that the strcmp is keeping us slow.

how many times in the array do we store the separate string "tcp"?

-this is a problem because you have to look it up from somewhere else

-takes up cache space

Avoid this by having each have a pointer to protocol.
Now we can pointer compare instead of strcmp...



And it’s even faster!