# Lecture 24—Massive Scalability; Course Summary

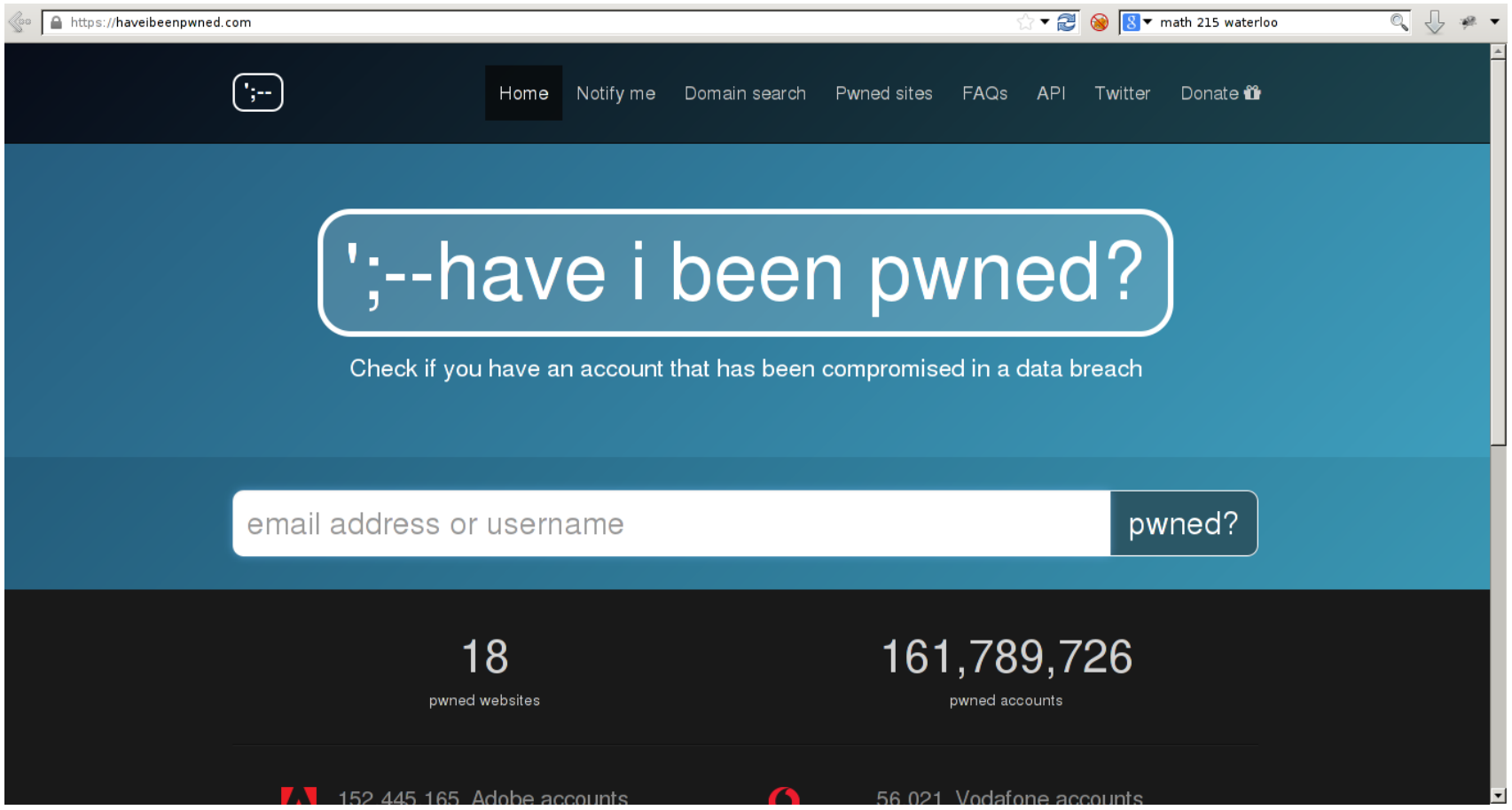
## ECE 459: Programming for Performance

April 3, 2014

# Part I

## Optimizing a Website

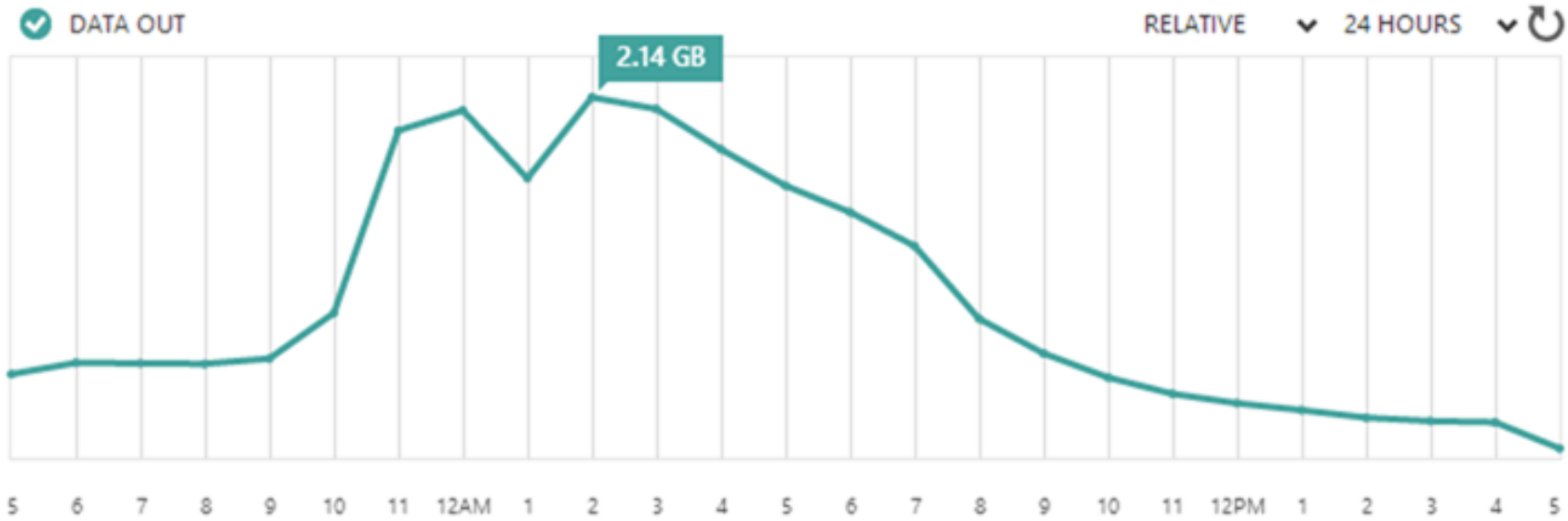
# December 2013: Slashdotted...




Discussion:

`http://www.troyhunt.com/2013/12/micro-optimising-web-content-for.html`

# The Issue is Bandwidth




23GB in 24 hours.

This is a problem that you could solve with money.
We can do better.

# Standard Website Tricks

HTML (and JavaScript) + CSS.

Specifically: Bootstrap, jQuery, Font Awesome.
Bundled and minified.
(2 requests: one for CSS—early, one for JS—late).

Cache expiration = 1 year; gzip everything.

5 images: well-optimized SVGs = 12.1kB.

Disabled all unnecessary response headers.

Ran through YSlow and Web Page Test (A).

# Conflicting Objectives

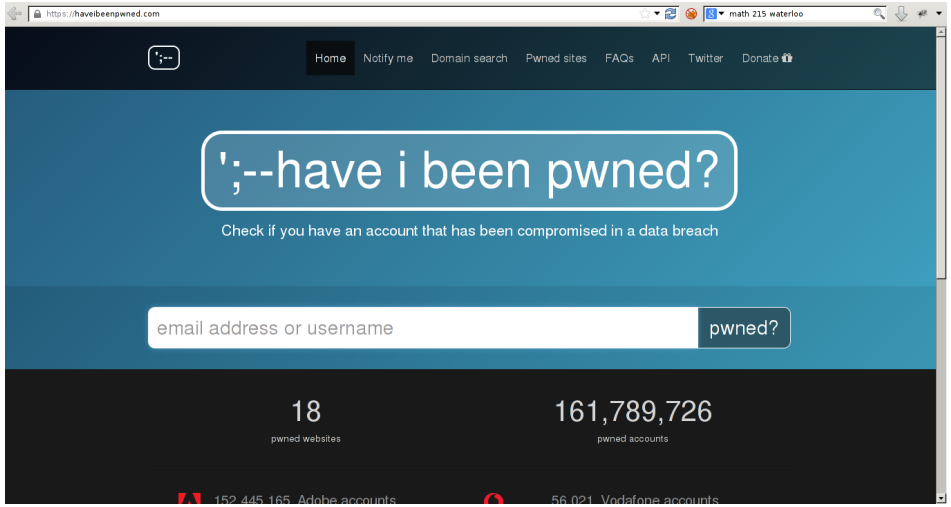
- Speed: load page as fast as possible;
- Data volume: minimize outbound data.

Example of conflict:

load jQuery from Google's Content Distribution Network, at the cost of more latency (separate requests).

# About Speed

Want site to be “usable” as soon as possible.




1. Must load the HTML first (compress it!)
2. Then you need the CSS.

That covers everything except for the logos.

At this point, the page is “usable” (but can’t process requests).

# Adding intelligence

To actually start processing, need JavaScript.

Don't actually need it until user enters an address
(well, also when clicking on a company;)
(but that's not going to happen instantly either.)

Graceful degradation:
without JS, use form submission and hyperlink.

Prefetching:
load 17 breaches directly in the HTML.

# Stages of the Load Lifecycle

1. Readable
2. Visually Complete
3. Functionally Complete

# Profiling Page Load

Use e.g. Chrome developer tools.

- Size: actual, compressed.
- Time: including latency.
- Sequencing/staggering of requests: parallelism!

31 requests, 174kB transferred.
Page ready after 200ms, complete after 800ms.

# Content Distribution Networks

Initially: bundle then minify JavaScript—1 request, 88kB.

Serve jQuery from Google:

- 2 more requests;
- increase bytes transferred by 2kB.

But you've outsourced JS serving;
now only serve 10kB of JS yourself.

Plus, it's faster: jQuery loaded in parallel, closer to the user, and may be cached.

Also works for other libraries.

# Other Tweaks

Make SVG files smaller.

Convert PNG to SVG.

Serve SVG from CDNs.

Do work on client-side (email address validation).

# Part II

# Massive Scalability

# People worth following

- Fran Allen (IBM)
- Jeff Dean (Google)
- Keith Packard (Intel)
- Herb Sutter (Microsoft)

# Some Thoughts from Jeff Dean

URL: `research.google.com/pubs/jeff.html`

Selections from:
"Software Engineering Advice for Building Large-Scale Distributed Systems".

On scaling:

- design for $\sim$ 10x, rewrite before 100x

Key concept for scaling:

- sharding, also known as partitioning.

# Why Distribute?

Let's say that we want a copy of the Web.

20+ billion web pages $\times$ 20KB $=$ 400+ TB.

$\sim$ 3 months to read the web.

$\sim$ 1000 HDs (in 2010) to store the web.

And that's without even processing the data!

# Magic solution: distribute the problem!

1000 machines $\Rightarrow$ $< 3$ hours.

No Free Lunch.
Problems: need to deal with ...

- communication & coordination;
- recovering from machine failure;
- status reporting;
- debugging;
- optimization;
- locality

...and that, from scratch, for each problem!

# Designing Systems as Services/Platforms

Steve Yegge on Google and Platforms:

`https://plus.google.com/112678702228711889851/posts/eVeouesvaVX`

[Bezos's] Big Mandate went something along these lines:
1) All teams will henceforth expose their data and functionality through service interfaces.
2) Teams must communicate with each other through these interfaces.
3) There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
4) It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols – doesn't matter. Bezos doesn't care.
5) All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
6) Anyone who doesn't do this will be fired.
7) Thank you; have a nice day! [j/k]

# Why Services?

Decouple different parts of a system.

“Fewer dependencies, clearly specified”.

“Easy to test new versions.”

“Small teams can work independently.”

# How to Design Stuff

Talk to people!

Write down a [some] rough sketch[es],
chat at a whiteboard.

Design good interfaces. (This is hard.)

# Using Back-of-the-Envelope Calculations I

"How long to generate image results page (30 thumbnails)?"

1. read serially, thumbnail 256K images on-the-fly:
   30 seeks $\times$ 10 ms/seek + 30 $\times$ 256K / 30MB/s = 560ms
2. issue reads in parallel:
   10 ms/seek + 256KB read / 30 MB/s = 18ms
   (plus variance: 30-60ms)

# Using Back-of-the-Envelope Calculations II

"How long to quicksort 1GB of 2-byte numbers?"

Comparisons: lots of branch mispredicts.
log($2^{28}$) passes over $2^{28}$ numbers = $\sim 2^{33}$ compares
$\sim 50\%$ mispredicts = $2^{32}$ mispredicts $\times$ 5 ns/mispredict = 21s.

Memory bandwidth: mostly sequential streaming.
$2^{30}$ bytes $\times$ 28 passes = 28GB;
memory bandwidth $\sim 4GB/s$, so $\sim 7$ seconds.

Total: about 30 seconds to sort 1GB on 1 CPU.

Also, write microbenchmarks. Understand your building blocks.

# Numbers Everyone Should Know

`http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html`

# Part III

# Course Summary

# Key Concepts I: goals

- Bandwidth versus Latency.
- Concurrency versus Parallelism.
- More bandwidth through parallelism.
- Amdahl's Law and Gustafson's Law.

# Key Concepts II: leveraging parallelism

- Features of modern hardware.
- Parallelism implementations: pthreads.
  - definition of a thread;
  - spawning threads.
- Problems with parallelism: race conditions.
  - manual solutions: mutexes, spinlocks, RW locks, semaphores, barriers.
  - lock granuarlity.
- Parallelization patterns; also SIMD.

# Key Concepts III: inherently-sequential problems

- Barriers to parallelization: dependencies.
  - loop-carried, memory-carried;
  - RAW/WAR/WAW/RAR.
- Breaking dependencies with speculation.

# Key Concepts IV: higher-level parallelization

- Automatic parallelization; when does it work?
- Language/library support through OpenMP.

# Key Concepts V: hardware considerations

- Unwelcome surprises: memory models & reordering.
  - fences and barriers; atomic instructions.
  - cache coherency implementations.

# Key Concepts VI: help from the compiler

- Three-address code.
- Compiler constructs: volatile, restrict.
- Inlining and other static optimizations.
- Profile-guided optimizations.

# Key Concepts VII: profiling

- Profiling tools and techniques.
- Call graphs, performance counters from profilers.
- When your profiler lies to you!
- Query-based DTrace approach.

# Key Concepts VIII: assorted topics

- Reduced-resource computing.
- Software transactions.

# Key Concepts IX: beyond single-core CPU programming

- Languages for high-performance computing.
- GPU Programming (e.g. with OpenCL).
- Clusters: MapReduce, MPI.
- Clouds.
- Big Data.

# Final Words

Good luck on the final & see you at convocation!