Chromium Resource Scheduling

by William Chan (陳智昌)
willchan@{chromium.org,google.com}

Overview

- Resource scheduling refers to the decision logic for how / when to request resources
- The primary goal of browser resource scheduling is to optimize the “page load experience” (make it fast)
- Difficulties
  - Unclear what load ordering improves the user experience
  - Need to discover resources first before they can be loaded
  - Fetching multiple resources may introduce contention
  - Many more!

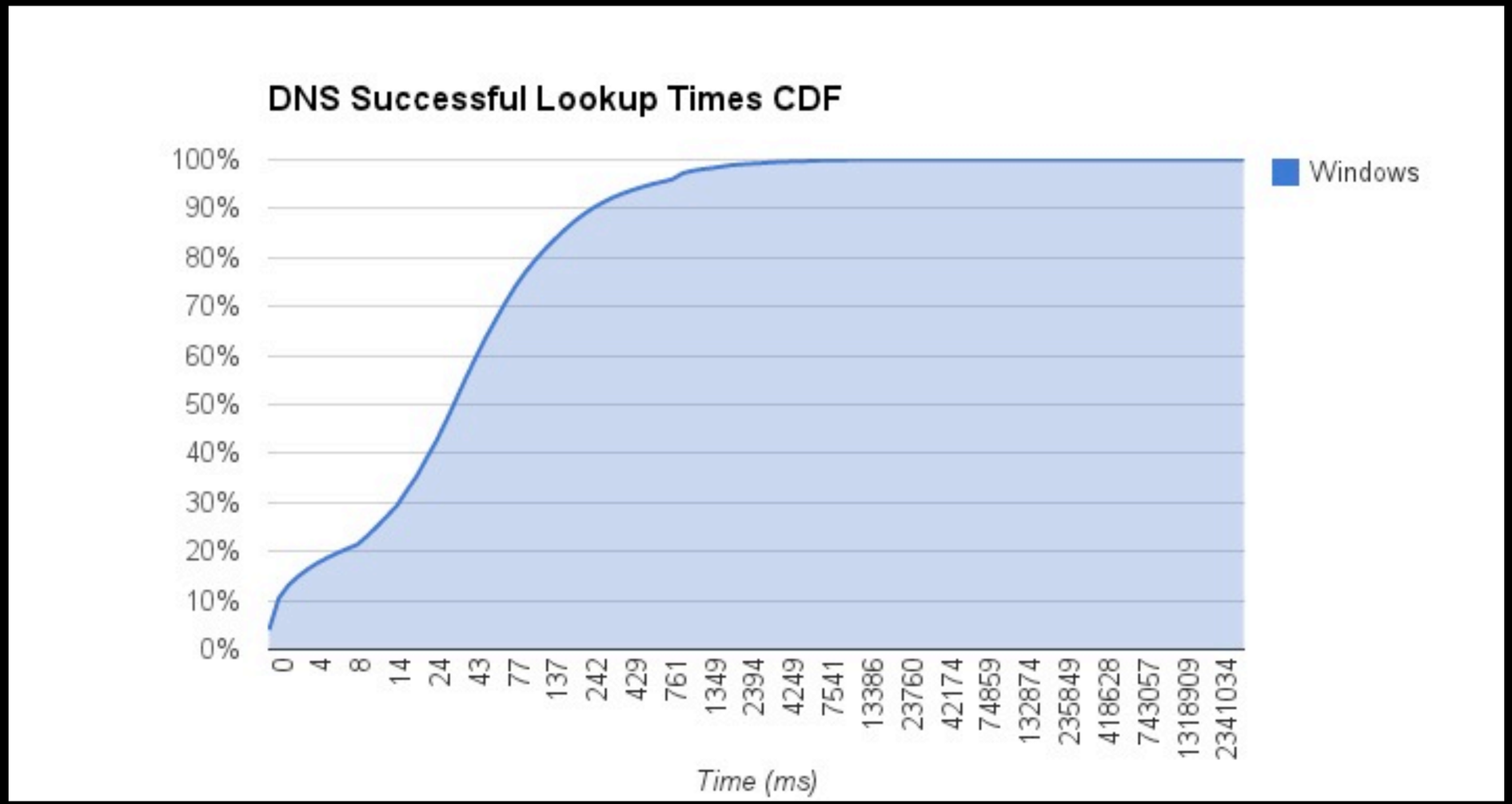
An aside: What does “fast” mean?

- Time to first paint?
  - Browsers will wait for stylesheets in <head> to download before first paint, in order to prevent FOUC
- Time until most of the above the fold content is visible?
- Time until all page assets are loaded?
- Time until the page becomes interactive/usable?
  - Many scripts will wait until the DOMContentLoaded event or load event before running script / installing event handlers / etc

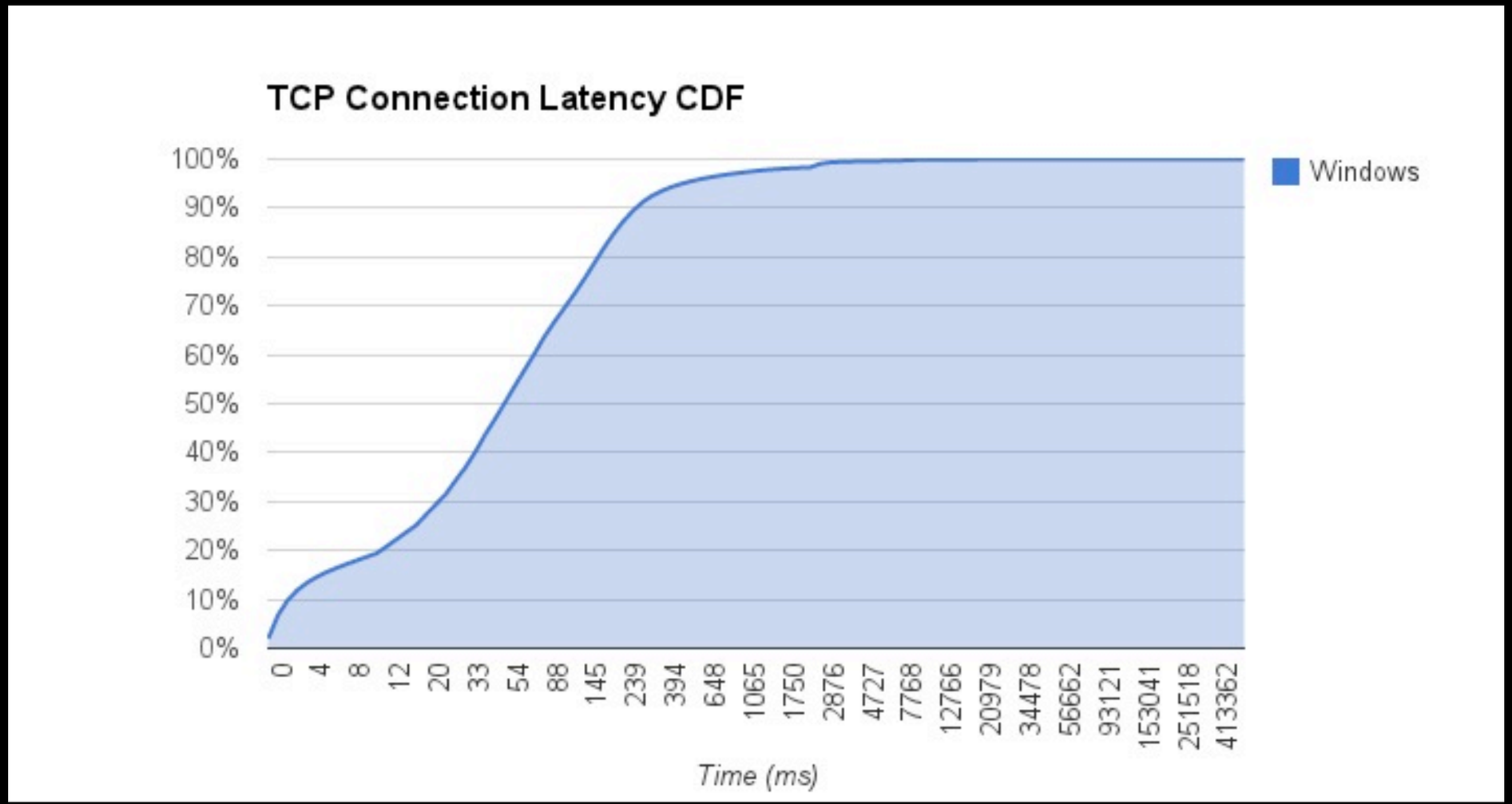
Problem: Network is slow

- Network roundtrip latency is high
  - DNS lookups are slow
  - TCP connection latencies are slow
  - HTTP roundtrips are slow
  - Speed of light is not getting faster
- Page download time is slow
  - Pages/resources are getting bigger
  - Bandwidth is improving, but many users are still on slow connections

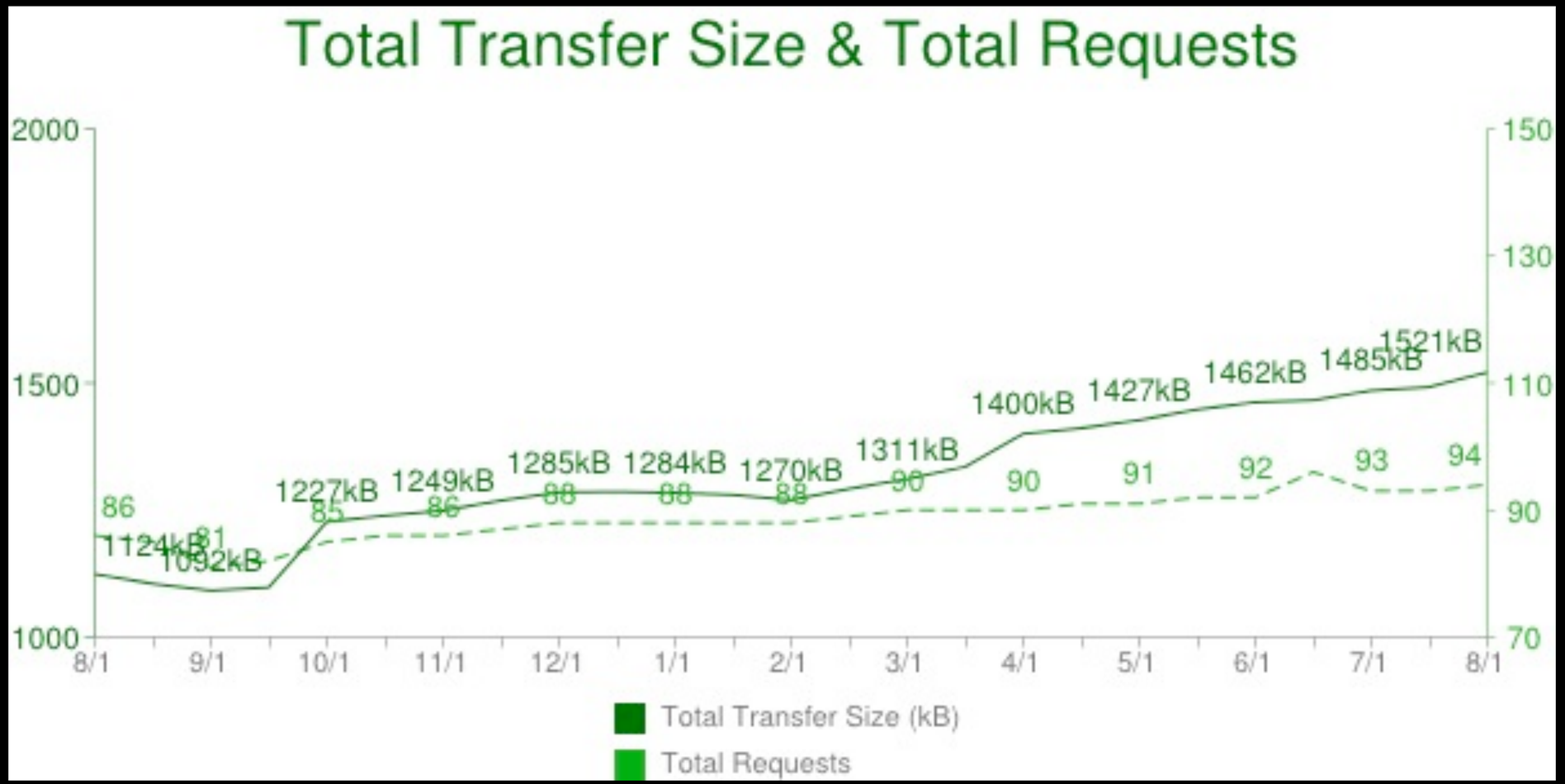
DNS latency stats from Chrome 28



TCP latency stats from Chrome 28



Total transfer time (HTTP Archive)



Solution: Fetch resources ASAP

- If the network's slow, the solution is to discover all the resources and begin fetching them as soon as possible.
- Browser implementors to the rescue: speculative parsing!
  - While the HTML parser is blocked waiting for a script or stylesheet to download, the speculative parser looks ahead in the HTML for resources to download
  - In Tony Gentilcore's test, it sped up the Alexa top 75 websites by 20%

Problem: Contention

- Major issue is bandwidth contention
  - Each new TCP connection will contend with existing TCP connections for available bandwidth
  - TCP tries to be “fair”
  - If there are X TCP connections, TCP tries to give each connection 1/X of available bandwidth.
- Fairness is a good thing though, right?

Resources are not created equal!

- Some resources let you discover other resources to fetch (e.g. scripts using XHR, iframes referencing other resources)
- Some resources block parsing (e.g. script, stylesheets)
- Stylesheets will block first paint (prevent FOUC)
- Some resources are more visually important than others (e.g. above the fold images)
- Some resources must be processed in entirety (e.g. JS), whereas some can be incrementally processed (e.g. HTML)
- Some parts of resources are more important than others (e.g. image headers, progressive images)
- etc, etc

Using fewer connections

- TCP will try to give each of X connections 1/X of the available bandwidth
  - Therefore, the obvious solution is reduce X
  - This is one reason browsers limit connections per host to 6
    - Resource fetches will sit in priority queues waiting for available connections
  - Browsers will also only fetch high priority resources before first paint, in order to reduce contention and paint sooner

Solution: Resource Prioritization

- Basic approach: prioritize by resource type (e.g. document/ script/stylesheet/etc) and then by discovery order (roughly parse order)
  - HTML > CSS > JS > Images
    - Just a naive heuristic, it's generally good, but sometimes bad
- But since TCP tries to be fair, how do we actually implement prioritization?

Does contention matter?

Case study: gap.com:

Problem: Underutilization

If we use too few connections, then we might not fully utilize the available bandwidth. May slow down overall page load.

SPDY & HTTP/2 to the rescue!

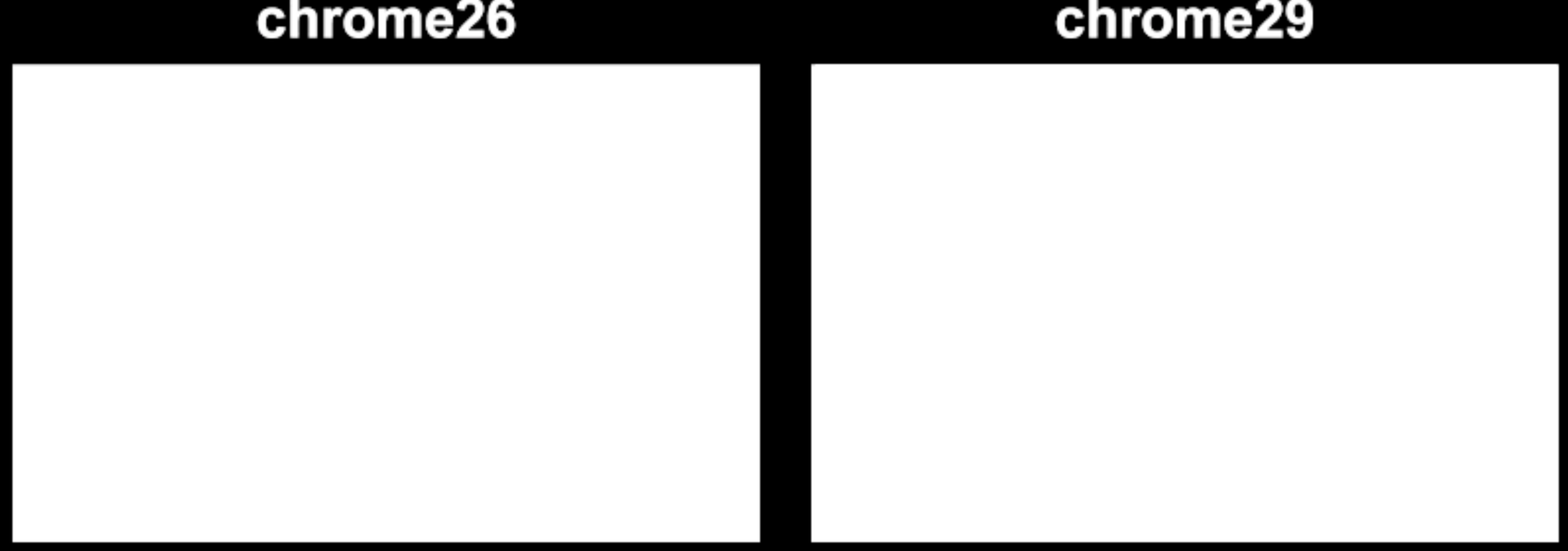
- SPDY introduces prioritized multiplexing within a single connection.
  - Each request is tagged with an advisory priority
  - The server maintains a priority queue for ordering its responses
  - Now the browser doesn't have to issue fewer resource fetches, it can fetch all the resources simultaneously and the server will send them back in priority order.

SPDY Prioritization Example

- Chrome 26 vs Chrome 29 (Chrome 29 disables the resource scheduler logic for SPDY)

SPDY Prioritization Example

- Chrome 26 vs Chrome 29 (Chrome 29 disables the resource scheduler logic for SPDY)



Problem: Contention again!

- Chromium's current prioritization by resource type is too coarse-grained
- Examples
  - Certain images are more important than other images
  - <script> should be loaded in parse order.
  - Image headers are more important than the image bodies, since they affect layout
  - ...

Better image prioritization

- It's impossible to know what is "above the fold" until layout happens, but that's too late.
- Sometimes images aren't fetched in parse order, since image loads may be initiated by stylesheets, which the speculative parser doesn't query.
- Rough heuristics
  - What about fixing image prioritization to match parse order?
  - What about assuming that the first images are more important than the later ones, so set a max concurrency limit for images?

Deprioritizing preloaded images

- When the speculative parser preloads an image, preload it at a lower priority than normal images. When the normal parser catches up, reprioritize the preloaded image back to normal.
- Example: bridepower.com

  - Navigation bars have background-images

Limiting concurrent image fetches

- The idea is that the first images are more important than later images.
- If Chromium limits the # of concurrent image fetches, it will prioritize earlier images.
- If the concurrency limit is too low, we get underutilization.
- If the concurrency limit is too high, we get contention again.
- Experiments show that a limit of 10 is a pretty good number in today's web.

Summarizing the information so far

- Resource discovery enables the browser to begin downloading resources sooner, which is important to achieve high bandwidth utilization
- Resource discovery may also lead to contention
- Chromium heuristically prioritizes resources based on type
  - With HTTP/1.X, browsers only have crude mechanisms for trying to prioritize resource fetches, often running the risk of underutilizing bandwidth
  - Browsers can more effectively implement prioritization in SPDY & HTTP/2, without the risk of underutilization

Advice for Web Developers

- Generally speaking, it's a good idea to enable the browser to discover your resources sooner
  - Fetching resources via declarative HTML markup enables the speculative parser to discover the resource even when parsing is blocked.
    - Protip: Chromium supports <link rel=subresource> which enables the parser to discover resources sooner. The problem with it is it lacks resource type info, so the browser doesn't have a good idea how to prioritize it.
  - Fetching resources via script and other mechanisms prevents the speculative parser from fetching them earlier.

Advice for Web Developers - 2

- Avoid hiding/munging the resource type from the browser
  - Becomes especially important with SPDY & HTTP/2
  - Examples of hiding/munging the resource type
    - Fetching via XHR and dynamically inserting in the DOM
    - Using an iframe to load inline <script>
  - Case study: Gmail (used JS in iframe and CSS via XHR)
    - Asked why CSS always finished slower than JS
  - Case study: Google+ (fetched CSS via XHR instead of <link>)
    - Chrome speedup: 4x speedup at the median, and 5x at 25th percentile
    - Firefox speedup: 5x speedup at median, and 8x at 25th percentile

Advice for Web Developers - 3

- Watch out for contention within the same resource type
  - SPDY & HTTP/2 doesn't fix it...yet? TBD
- <script src="core.js"></script>
  <script src="enhance.js"></script>
  <script src="enhance_more.js"></script>
  <script src="enhance_even_more.js"></script>
  - Ideally, these should be downloaded in parse order, but currently even with SPDY & HTTP/2 they'll contend

Future Work

- Analyze why the preconnect experiment showed no gains
  - We hold back lower priority requests at some points, but speculatively preconnecting *should* help hide latency
- Experiment with dynamically adjusting the image download concurrency limit
- Experiment with trying to leverage more information
  - Does the browser know RTT or available bandwidth? Tweak limits accordingly.