Characterizing and Improving Web Page Load Times

Xiao Wang

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2015

Reading Committee:

David J. Wetherall, Chair

Arvind Krishnamurthy, Chair

Henry M. Levy

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Characterizing and Improving Web Page Load Times

Xiao Wang

Co-Chairs of the Supervisory Committee:
Prof. David J. Wetherall
Computer Science and Engineering

Prof. Arvind Krishnamurthy
Computer Science and Engineering

Web page load time (PLT) is a key performance metric that many techniques aim to improve. PLT is much slower than lower-level latencies, but the reason was not well understood.

This dissertation first characterizes the Web page load time by abstracting a dependency model between network and computation activities. We have built a tool WProf based on this model, that identifies the bottlenecks of PLTs of hundreds of Web pages, and that provides basis for evaluating PLT-reducing techniques. Next, we evaluate SPDY's contributions to PLTs and find that SPDY's impact on PLTs is largely limited by the dependencies and browser computation. This suggests that the page load process should be restructured to remove the dependencies so as to improve PLTs. Thus, we propose SplitBrowser that preprocesses Web pages on a proxy server and migrate carefully crafted state to the client so as to simplify the client-side page load process. We have shown that SplitBrowser reduces PLTs by more than half under a variety of mobile settings that span less compute power and slower networks.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I am extremely fortunate to have worked with my amazing advisors, David Wetherall and Arvind Krishnamurthy, who have taught me how to do good research, inspired me with insightful thoughts, and supported me unconditionally throughout the past six years. David has always encouraged me to focus on practical problems that I feel passionate about and with long-term impact. His insightful comments and the ability to always point me to the right direction in the first place have helped me keep on track and make progress with projects. Arvind's optimism brings more joy to my life in graduate school. He provides me with the flexibility to explore things myself while almost always being the first to help me out when I get blocked. I feel incredibly fortunate to have a great combination of advisors who have made me a stronger person that I could not imagine before entering graduate school. I'm so in debt to them.

I thank my collaborators, Aruna Balasubramanian, David Choffnes, and Haichen Shen, my committee members, Henry M. Levy and Radha Poovendran, the undergraduate students that I have worked with, Tyler Jacob and Ruby Yan etc., and other students in the networks lab. Without their contributions or discussions I would not be able to always keep the high standard of the networks lab in my own work. I thank my friends, especially the UW/CSE Chinese community, for having fun with me during weekends when I make little process with research. I thank my other friends, local or remote, for encouraging and supporting me throughout the journey.

Last, I thank my family, especially my parents, for their unconditional support. I could not finish the dissertation that I feel proud of without their encouragement and support.

Chapter 1

Introduction

Web pages delivered by HTTP have become the de-facto standard for connecting billions of users to Internet applications. As a consequence, Web page load time (PLT) has become a key performance metric. Numerous studies and media articles have reported its importance for user experience [8, 7], and consequently to business revenues. For example, Amazon increased revenue 1% for every 0.1 second reduction in PLT, and Shopzilla experienced a 12% increase in revenue by reducing PLT from 6 seconds to 1.2 seconds [44].

Given its importance, significant efforts have been invested in reducing PLT. Many websites rely on single developers to follow the list of best practices [51, 73], which are ever-changing and thus hard to follow closely. Thus, it has been widely believed that the inefficiencies in PLT should be transparently corrected by proper techniques. A first kind of techniques focuses on automatic page rewritings such as mod_pagespeed [62] in order to enforce the best practices. A second kind focuses on improving the network transfer times. For example, techniques such as DNS pre-resolution [29], TCP pre-connect [22], and TCP fast open [45] reduce latencies, and the SPDY protocol improves network efficiency at the application layer [52]. A third kind lowers computation costs by either exploiting parallelism [33, 13] or adding architecture support [76, 15].

However, it is unclear how much the above techniques contribute to PLT because of the inherent complexity of the page load process. Web pages mix resources fetched by HTTP with JavaScript and CSS evaluation. These activities are inter-related such that the bottlenecks are difficult to identify. Web browsers complicate the situation with implementation strategies for parsing, loading

and rendering that significantly impact PLT. The result is that it is hard to explain why a change in the way a page is written or how it is loaded has an observed effect on PLT. As a consequence, it is difficult to know when proposed techniques will help or harm PLT.

This dissertation aims at improving PLT by using principled and provably effective techniques. To gauge the effectiveness of techniques, one has to understand how the activities that are involved in a page load interplay. This is because often techniques improve some parts of a page load that however contribute little to the overall PLT.

As a first step, we present WProf, a lightweight in-browser profiler that produces a detailed dependency graph of the activities that make up a page load. WProf is based on a model we abstracted to capture the dependencies and constraints between network load, page parsing, JavaScript/CSS evaluation, and rendering activity in popular browsers. Combined with real-world experiments, this model lets us pinpoint the bottleneck activities that limit PLT. This model also makes it possible to estimate the amount of overall impact on PLT given that parts of the activities within a page load are improved, providing a basis for measuring the effectiveness of PLT-reducing techniques.

Keeping this model in mind, we measure the contributions of SPDY (then HTTP/2) to PLT, a topical protocol that refactors the application-layer network protocol for the Web. To identify the factors that affect PLT, we proceed from simple, synthetic pages to complete page loads based on the top 200 Alexa sites. To make experiments reproducible, we develop a tool called `Epload` that controls the variability by recording and replaying the process of a page load at fine granularity, complete with browser dependencies and deterministic computational delays. We find that SPDY provides a significant improvement over HTTP/1.1 when dependencies in the page load process and the effects of browser computation are ignored. Unfortunately, the benefits can be easily overwhelmed by dependencies and computation. We also find that server push has good potential since it can break the dependencies that form the bottlenecks of PLT.

Our previous work suggests that dependencies and other complexities significantly slow down page loads. To this end, we design and implement SplitBrowser that splits the page load process such that the first phase loads the Web page fast without any dependencies and complexities and the second phase ensures correctness of this isolation. Evaluations show that SplitBrowser reduces

PLT by about 60% on a variety of mobile and desktop settings.

The following sections describe each piece of work in more detail.

1.1 Demystifying Page Load Times with WProf

Browsers were often treated as black boxes, and the relations between activities that occur in a page load remained a mystery. To provide a basis for advancing techniques that reduce PLT, this work aims at demystifying Web page load performance.

Previous studies in this space have measured Web performance in different settings, e.g., cellular versus wired [26], and correlated PLT with variables such as the number of resources and domains [11]. However, these factors are only coarse indicators of performance and lack explanatory power. The key information that can explain performance is the relations between activities in the page load process itself. Earlier work such as WebProphet [30] made clever use of inference techniques to identify some of these dependencies. But inference is necessarily time-consuming and imprecise because it treats the browser as a black-box. "Waterfall" tools such as Google's Pagespeed Insight [42] have proliferated to provide detailed and valuable timing information on the components of a page load. However, even these tools are limited to reporting what happened without explaining why the page load proceeded as it did.

In this work, we abstract the dependency policies in four browsers, i.e., IE, Firefox, Chrome, and Safari. We run experiments with systematically instrumented test pages and observe object timings using Developer Tools [18]. For cases when Developer Tools are insufficient, we deduce dependencies by inspecting the browser code when open source code is available. We find that some of these dependency policies are given by Web standards, e.g., JavaScript evaluation in script tags blocks HTML parsing. However, other dependencies are the result of browser implementation choices, e.g., a single thread of execution shared by parsing, JavaScript and CSS evaluation, and part of rendering. They have significant impacts on PLT and cannot be ignored.

Given the dependency policies, we develop a lightweight profiler, WProf, that runs in Webkit browsers (e.g., Chrome, Safari) while real pages are loaded. WProf generates a dependency graph and identifies a load bottleneck for any given Web page. Unlike existing tools that produce waterfall

or HAR reports [24], our profiler discovers and reports the dependencies between the browser activities that make up a page load. It is this information that pinpoints why, for example, page parsing took unexpectedly long and hence suggests what may be done to improve PLT.

To study page load performance, we run WProf while fetching pages from popular servers and apply critical path analysis, a well-known technique for analyzing the performance of parallel programs [47]. First, we identify page load bottlenecks by computing what fraction of the critical path the activity occupies. Surprisingly, we find that while most prior work focuses on network activity, computation (mostly HTML parsing and JavaScript execution) comprises 35% of the critical path. Interestingly, downloading HTML and synchronous JavaScript (which blocks parsing) makes up a large fraction of the critical path, while fetching CSS and asynchronous JavaScript makes up little of the critical path. Second, we study the effectiveness of different techniques for optimizing web page load. Caching reduces the volume of data substantially, but decreases PLT by a lesser amount because many of the downloads that benefit from it are not on the critical path. Disappointingly, SPDY makes little difference to PLT under its default settings and low RTTs because it trades TCP connection setup time for HTTP request sending time and does not otherwise change page structure. Mod_pagespeed also does little to reduce PLT because minifying and merging objects does not reduce network time on critical paths.

1.2 How Speedy is SPDY?

To make the Web faster, Google proposed and deployed a new transport for HTTP messages, called SPDY, starting in 2009. SPDY adds a framing layer for multiplexing concurrent application-level transfers over a single TCP connection, support for prioritization and unsolicited push of Web objects, and a number of other features. SPDY is fast becoming one of the most important protocols for the Web; it is already deployed by many popular websites such as Google, Facebook, and Twitter, and supported by browsers including Chrome, Firefox, and IE 11. Further, IETF is standardizing a HTTP/2.0 proposal that is heavily based on SPDY [25].

Given the central role that SPDY is likely to play in the Web, it is important to understand how SPDY performs relative to HTTP. Unfortunately, the performance of SPDY is not well understood.

There have been several studies, predominantly white papers, but the findings often conflict. Some studies show that SPDY improves performance [55, 41], while others show that it provides only a modest improvement [38, 54]. In our own study [66] of page load time (PLT) for the top 200 Web pages from Alexa [3], we found either SPDY or HTTP could provide better performance by a significant margin, with SPDY performing only slightly better than HTTP in the median case.

As we have looked more deeply into the performance of SPDY, we have come to appreciate why it is challenging to understand. Both SPDY and HTTP performance depend on many factors external to the protocols themselves, including network parameters, TCP settings, and Web page characteristics. Any of these factors can have a large impact on performance, and to understand their interplay it is necessary to sweep a large portion of the parameter space. A second challenge is that there is much variability in page load time (PLT). The variability comes not only from random events like network loss, but from browser computation (i.e., JavaScript evaluation and HTML parsing). A third challenge is that dependencies between network activities and browser computation can have a significant impact on PLT [66].

In this work, we present what we believe to be the most in-depth study of page load time under SPDY to date. To make it possible to reproduce experiments, we develop a tool called `Epload` that controls the variability by recording and replaying the process of a page load at fine granularity, complete with browser dependencies and deterministic computational delays; in addition we use a controlled network environment. The other key to our approach is to isolate the different factors that affect PLT with reproducible experiments that progress from simple but unrealistic transfers to full page loads. By looking at results across this progression, we can systematically isolate the impact of the contributing factors and identify when SPDY helps significantly and when it performs poorly compared to HTTP.

Our experiments progress as follows. We first compare SPDY and HTTP simply as a transport protocol (with no browser dependencies or computation) that transfers Web objects from both artificial and real pages (from the top 200 Alexa sites). We use a decision tree analysis to identify the situations in which SPDY outperforms HTTP and vice versa. We find that SPDY improves PLT significantly in a large number of scenarios that track the benefits of using a single TCP connection.

Specifically, SPDY helps for small object sizes and under low loss rates by: batching several small objects in a TCP segment; reducing congestion-induced retransmissions; and reducing the time when the TCP pipe is idle. Conversely, SPDY significantly hurts performance under high packet loss for large objects. This is because a set of TCP connections tends to perform better under high packet loss; it is necessary to tune TCP behavior to boost performance.

Next, we examine the complete Web page load process by incorporating dependencies and computational delays. With these factors, the benefits of SPDY are reduced, and can even be negated. This is because: i) there are fewer outstanding objects at a given time; ii) traffic is less bursty; and iii) the impact of the network is degraded by computation. Overall, we find SPDY benefits to be larger when there is less bandwidth and longer RTTs. For these cases SPDY reduces the PLT for 70–80% of Web pages, and for shorter, faster links it has little effect, but it can also increase PLT: the worst 20% of pages see an increase of at least 6% for long RTT networks.

In search of greater benefits, we explore SPDY mechanisms for prioritization and server push. Prioritization helps little because it is limited by load dependencies, but server push has the potential for significant improvements. How to obtain this benefit depends on the server push policy, which is a non-trivial issue because of caching. This leads us to develop a policy based on dependency levels that performs comparably to mod_spdy's policy [35] while pushing 80% less data.

1.3 SplitBrowser

The previous work suggests that the complexities of page loads significantly slow down PLT. We further identify three types of inefficiencies associated with Web pages and the page load process. The first inefficiency comes from the content size of Web pages. Many Web pages use JavaScript libraries such as jQuery [28] or include large customized JavaScript code in order to support a high degree of user interactivity. The result is that a large portion of the code conveyed to a browser is never used on a page, or is only used when a user triggers an action. The second inefficiency stems from how the different stages of the page load process is scheduled in order to ensure semantic correctness in the presence of concurrent access to shared resources. This results in limited overlap between computation and network transfer, thus increasing PLT. The third and related inefficiency

is that many resources included in a Web page are often loaded sequentially due to the complex dependencies in the page load process, and this results in sub-optimal use of the network and increased PLTs.

Reducing PLT is hard given the nature of these inefficiencies. Human inspection is not ideal since there is no guarantee that Web developers follow closely to the ever-changing best practices. Thus, it has been widely believed that the inefficiencies should be transparently corrected by proper techniques. Many techniques focus on improving the network transfer times. Other techniques lower computation costs by either exploiting parallelism [33, 13] or adding architecture support [76, 15]. While being effective in speeding up the individual activities corresponding to a page load, these techniques have limited impact in reducing overall PLT, because they still communicate redundant code, stall in the presence of conflicting operations, and are constrained by the limited parallelism in the page load process.

In this work, we advocate the SplitBrowser approach that splits the page load process and reduces the PLT by directly tackling the three inefficiencies listed above. A proxy server is set up to preload a Web page up to a time, e.g., when the *load* event is fired; the preload is expected to be fast since it exploits greater compute power at the proxy server and since all of the resources that would normally result in blocking transfers are locally available. When migrating state to the client, the proxy server prioritizes state needed for the initial page load over state that will be used later, so as to convey critical information as fast as possible. After all the state is fully migrated, the user can interact with the page normally as if the page were loaded directly without using a proxy server. Note that Opera mini [40] and Amazon Silk [4] also embrace a proxy approach but differ intrinsically. Their client-side browsers only handle display, and thus JavaScript evaluation is handled by the proxy server. This process depends on the network which is both slow and unreliable [49], and requires the proxy server to be placed near users. SplitBrowser has a fully functioning client-side browser and requires the proxy server to be placed near Web servers for the most performance gains.

SplitBrowser also ensures that the Web page functionality in terms of user interactivity is preserved and that the delivery process is compatible with latency-reduction techniques such as

caching and CDNs and security features such as the enforcement of same-origin policies. The resulting system is thus both efficient and practical. Our evaluations on the top 100 Alexa Web pages show that SplitBrowser reduces PLT by 60% on both a 1GHz Android phone and a variety of Linux desktop settings. The amount of resources loaded is decreased while the total amount of traffic is increased moderately by 1%.

1.4 Thesis and Contributions

The dissertation supports my thesis that *Web page load times can be radically improved by understanding the inefficiencies, pinpointing the bottlenecks, and developing techniques that remove the bottlenecks*. We make the following contributions in this dissertation:

- **A measurement tool, WProf.** To understand page load performance, we abstract a model of dependency policies that describes how page load activities interplay. Based on the policies, we develop WProf, a lightweight in-browser profiler that runs in Webkit browsers while real pages are being loaded. WProf generates a dependency graph and identifies a page load bottleneck for any given Web page. We run WProf from popular servers, apply critical path analysis to analyze the bottlenecks, and find the following results. 1) While most prior work focuses on network activities, browser computation (mostly HTML parsing and JavaScript evaluation) comprises 35% of the critical path, 2) Downloading HTML and synchronous JavaScript makes up a large fraction of the critical path while fetching CSS and asynchronous JavaScript makes up little of the critical path, 3) Caching reduces the volume of data substantially, but decreases PLT by a lesser amount because many of the downloads that benefit from it are not on the critical path, 4) Mod_pagespeed does little to reduce PLT because minifying and merging objects does not reduce network time on the critical path.
- **A systematic measurement study of the SPDY protocol.** Taking into account of the challenges, we conduct an in-depth study of SPDY as a result of controlled and reproducible experiments that isolate the different factors that affect PLT. We proceed from simple, synthetic pages to complete page loads based on the top 200 Alexa sites. Findings include: 1) SPDY provides a significant improvement over HTTP/1.1 when dependencies and browser

computation are ignored; 2) Unfortunately, the benefits can be easily overwhelmed by dependencies and computation; 3) Most SPDY benefits stem from the use of a single TCP connection, but the same feature is also detrimental under high packet loss; 4) Further benefits in PLT will require changes to restructure the page load process such as the server push feature of SPDY.

- **A prototype of SplitBrowser.** Informed by results from my previous work, we design and implement a prototype, SplitBrowser, that significantly reduces PLT at the client. SplitBrowser uses a proxy server to preload a Web page, quickly communicates an initial representation of the page's DOM to the client, and loads secondary resources in the background. SplitBrowser also ensures that the Web page functionality such as user interactivity is preserved and that the delivery process is compatible with latency-reduction techniques such as caching and CDNs and security features such as the enforcement of same-origin policies. The resulting system is thus both efficient and practical. Evaluations on the top 100 Alexa Web pages show that SplitBrowser reduces PLT by 60% on both a 1GHz Android phone and a variety of Linux desktop settings.

1.5 Organization

In the rest of this dissertation, we review background and related work in Chapter 2. The contributions are described in Chapter 3, Chapter 4, and Chapter 5 respectively. We conclude and discuss future work in Chapter 6.

Chapter 2

Background and Related Work

This chapter reviews the background of the Web architecture and browser workflows, provides a summary of the existing measurement metrics, techniques and studies related to PLT, and discusses several kinds of techniques that reduce PLT.

2.1 Background

This section describes the Web architecture and browser workflows, the key background for understanding where the time is gone when a page is being loaded.

2.1.1 Web Architecture

Figure 2.1 describes the architecture of the modern Web as a result of twenty years of evolution. A Web request usually starts from browsers, goes through an edge network (e.g., WiFi hotspots, cellular networks), then the core network, and arrives at the very front end of the target website. For large-scale websites, the very front end would be reverse proxies which both improve locality and balance the loads. The reverse proxies would then route the request to a front-end server that runs server-side code that produces the Web page. In response to the request, the front-end server sometimes contacts back-end servers and/or fetches personalized data. For static Web pages, the front-end server, or even the reverse proxy, responses without involving the back ends.

A Web request often results in contacting third parties for various reasons. Domain name system (DNS) translates a domain name to an IP address. Content distribution networks (CDNs) such

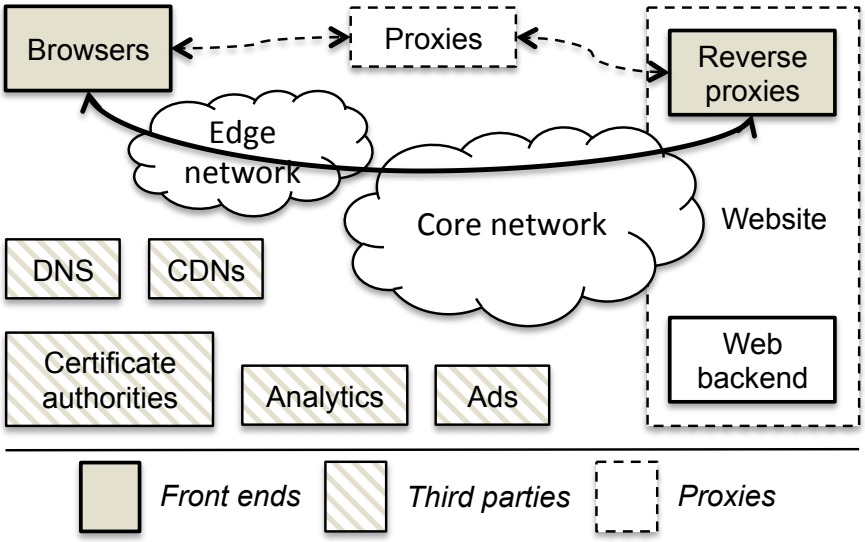


Figure 2.1: The architecture of the modern Web.

as Akamai [2] provide services to websites by hosting large (usually media) contents at multiple geographically distributed locations, thereby improving locality so as to speed up page loads. When necessary, certificate authorities (CA) certify the ownership of public keys as an initial step to setup end-to-end secure connections via HTTPS. Besides providing users with better performance and security, websites often use third-party advertisement networks (Ads) and Analytics for profit.

An increasingly popular component in the Web architecture is a proxy that sits between a mobile browser and a server aiming at faster page loads. The idea is to optimize both Web pages themselves and the way they will be loaded at a proxy for accommodating mobile devices with constrained computing power, batteries, and cellular connections. There are multiple choices as to where to place the proxies. Amazon Silk browser [4] and Android Chrome [72] place the proxies in their clouds, while Telefonica uses a Web accelerator at their edge networks [5].

This dissertation focuses on performance measurements and optimizations at the front ends (i.e., browsers, proxies, and front-end servers). As front ends work at the highest level of the page load process, they determine the inputs to other components and therefore their performance. Thus, we believe that working at front ends is primary in advancing Web performance, which has also

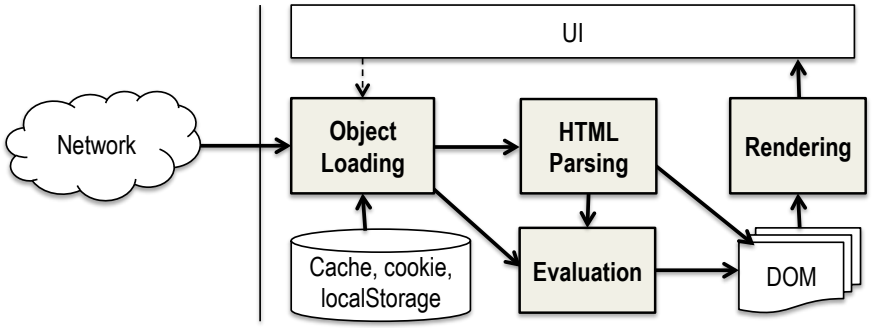


Figure 2.2: The workflow of a page load. It involves four processes (shown in gray).

been validated by other articles [51, 16].

2.1.2 Browser Workflows

While being primary in determining Web performance, browsers are arguably the least understood component. Here, we provide background on how browsers load Web pages. Figure 2.2 shows the workflow for loading a page. The page load starts with a user-initiated request that triggers the *Object Loader* to download the corresponding root HTML page. Upon receiving the first chunk of the root page, the *HTML Parser* starts to iteratively parse the page and download embedded objects within the page, until the page is fully parsed. The embedded objects are *Evaluated* when needed. To visualize the page, the *Rendering Engine* progressively renders the page on the browser. While the HTML Parse, Evaluator, and Rendering Engine are computation processes, the Object Loader is a network process.

HTML Parser: The Parser is key to the page load process, and it transforms the raw HTML page to a document object model (DOM) tree. A DOM tree is an intermediate representation of a Web page; the nodes in the DOM tree represent HTML tags, and each node is associated with a set of attributes. The DOM tree representation provides a common interface for programs to manipulate the page.

Object Loader: The Loader fetches objects requested by the user or those embedded in the HTML page. The objects are fetched over the Internet using HTTP or SPDY [52] which will then be

HTTP/2, unless the objects are already present in the browser cache. The embedded objects fall under different mine types: HTML (e.g., IFrame), JavaScript, CSS, Image, and Media. Embedded HTMLs are processed separately and use a different DOM tree. Inlined JavaScript and inlined CSS do not need to be loaded.

Evaluator: Two of the five embedded object types, namely, JavaScript and CSS, require additional evaluations after being fetched. JavaScript is a piece of software that adds dynamic contents to Web pages. Evaluating JavaScript involves manipulating the DOM, e.g., adding new nodes, modifying existing nodes, or changing nodes' styles.

Cascading style sheets (CSS) are used for specifying the presentational attributes (e.g., colors and fonts) of the HTML contents and is expressed as a set of rules. Evaluating a CSS rule involves changing styles of DOM nodes, often known as (CSSOM). For example, if a CSS rule specifies that certain nodes are to be displayed in blue color, then evaluating the CSS involves identifying the matching nodes in the DOM (known as CSS selector matching) and adding the style to each matched node. JavaScript and CSS are commonly embedded in Web pages today [11].

Rendering Engine: Browsers render Web pages progressively as the HTML Parser builds the DOM tree. Rendering involves two processes: Layout and Painting. Layout converts the DOM tree to the layout tree that encodes the size and position of each visible DOM node. Painting converts this layout tree to pixels on the screen.

2.2 Measuring Web Performance

There has been a large body of tools and studies on measuring Web performance using various metrics (§2.2.1). Some work considers network-specific measurements (§2.2.2), some others focus on computation at end hosts (§2.2.3), and a few span both network and computation.

2.2.1 Metrics

Before reviewing measurement tools and studies, we describe Web performance metrics regarding their definitions, accuracies, and measurement constraints. Table 2.1 summarizes the metrics.

The most popular Web performance metrics are time to fire W3C DOM event `DOMContentLoaded`

Metric	Category	User perceived	Measure
Time to load last object	Network	N	By network traces
Time to first byte			
DOMContentLoaded	DOM events	N	In browsers
load			
Render start	Browser	Y	
Above-the-fold-time	Webpagetest	Y	On webpagetest
Speed index			

Table 2.1: Web performance metrics.

and `load` [65]. `DOMContentLoaded` fires when the HTML parser finishes parsing the root HTML; and `load` fires when all embedded objects in the root HTML have been fetched. For example, when the HTML embeds an image, `load` waits until the image is loaded and `DOMContentLoaded` does not. Both DOM events can be measured with JavaScript on any browsers that follow W3C, even on headless browsers such as phantom.js [43]. Recently, the two metrics have been increasingly criticized for not being able to describe user-perceived page load times. For example, they overestimate on Web pages that embed many hidden image which can become visible by scroll downs; they underestimate on Web pages that use Ajax (e.g., XmlHttpRequest) to fetch content.

To convey user-perceived page load latencies, the industry designed metrics around painting progress. Start Render [46] measures the time to paint the first pixel on the browser screen and can be measured on a few browsers such as Chrome and Firefox. Above-the-fold time (AFT) [9] is defined as the moment when above-the-fold contents (those seen in the browser window) stops changing and reaches its final state. Start Render and AFT describe the initial and final state of the browser window respectively by ignoring intermediate state. To capture intermediate state, Speed Index (SI) [59] provides an averaged time at which visible parts of the page are displayed. Mathematically, speed index is defined as $SI(T) = XXX$ where $c(t)$ means the fraction of painted pixels at time t. In practice, speed index is used to gauge performance optimizations. While being the most related to user-perceived page load times, AFT and SI require video captures to be measured and are only available in limited lab settings [71].

Two other metrics based on assembling network traces are widely used in the network research community. One metric is time to first byte (TTFB) that is defined as the time from the start of the navigation to the reception of the first byte of a Web page. The other metric is time from the start of the navigation to completion of loading the last Web object. This metric varies in practice since the definition of last objects is vague on pages that constantly fetch objects. While trace-based metrics can be measured at any location that is not limited to browsers, they cannot reflect user-perceived delays.

2.2.2 Network-Specific Measurements

Measuring the network aspects of Web performance mostly falls under the following two categories. One uses application-level information that is captured inside browsers, usually by Developer Tools [18], and formatted in the HAR format [24]. This information tells, for example, how much time the browser uses to resolve DNS and sets up the TCP connection of a Web object, and how long it takes to load a Web object. The other category uses network-level traces that is captured by `tcpdump`, and later assembled to provide timing information similar to HAR. Network traces can be captured anywhere along the path but are limited by the obfuscation in front of HTTPS and by the lack of accuracy in reflecting user-perceived delays. Below reviews the measurement studies using the two kinds of information respectively.

Application level: The only measurement study in this space is performance by Butkiewicz et al. [11]. Butkiewicz et al. [11] conducted a macro-level study, measuring the number of domains, number of objects, JavaScript, CSS and so forth on top Web pages. While the study is valuable in characterizing the state of the art of top Web pages, it does not inform the bottlenecks in a Web page load.

Network level: Most of the measurements in the research community use network-level traces. Sundaresan et al. [61] measured page load performance at thousands of edge networks as part of the BisMark project. Ihm and Pai [27] presented a longitudinal study on Web traffic, Ager et al. [1] studied the Web performance with respect to CDNs, and Huang et al. [26] studied the page load performance under a slower cellular network. While providing measurements under diverse

settings, they do not let us pinpoint bottlenecks on the critical path.

WebProphet [30] identified dependencies in page loads by systematically delaying network loads. Because it instrumented a browser, it can use either application-level or network-level information and uses network-level information in the end. The focus of WebProphet is to uncover dependencies only related to object loading. As a result, WebProphet does not uncover dependencies with regard to parsing, rendering, and evaluation.

In addition to the measurements that use either application-level or network-level information. Chen et al. [14] provided a provider-side measurements of the Bing search response time. Because the authors have the knowledge of the Bing Web page and have access to timings in CDNs, they timestamped the arrival of specific Web contents that they needed.

2.2.3 Computation-Specific Measurements

There have been many profiling tools, benchmarks, and studies that are specific to browser computation with focuses on JavaScript and CSS evaluations, rendering, and breakdowns of overall performance. Here, we only describe tools that capture the overall performance.

Google provided developers with a number of tools that break down the performance of browser computation in several ways. Speed tracer [60] is a Chrome extension that measures and visualizes elapsed time spent on various computational tasks in browsers. The tasks can be JavaScript parsing, evaluation, garbage collection, CSS selector matching, style recalculation, DOM event handling, timer fires, network events handling, layout, painting, and so forth. The Profile tab in Chrome developer tools [18] provides CPU time that is idle, spent on JavaScript functions, and garbage collectors. It also collects heap usage by JavaScript objects. `chrome://profiler/` profiles elapsed times and counts of function calls in Chrome C++ source code.

2.2.4 Measurements that Span both Network and Computation

Google Pagespeed Insight [42] included a (non-open-sourced) critical path explorer once in a while. The explorer presented the critical path for the specific page load instance, but did not extract all dependencies innerent in a page. For example, the explorer did not include dependencies due to resource constraints and eager/late bindings. Also the explorer would likely miss

dependencies if objects were cached, if the network speed improved, if a new CDN is introduced, if the DNS cache changed, and many other instances. Therefore, it is difficult to conduct what-if analysis or to explain the behavior of Web page optimizations using the explorer. Unfortunately, the critical path explorer has disappeared on its Web page ever since 2013.

The Mystery Machine et al. [16] provided an end-to-end analysis of the Facebook Web pages that considered both network and computation activities, both front and back ends. The analysis inferred how activities are inter-related, which let them identify critical paths. The study found that most of the page load time on the critical path was spent on the front ends, which validated our hypothesis that front ends are the primary in advancing Web performance.

2.3 Making the Web Faster

The industry has driven Web accelerations for years, mostly in an ad hoc manner. Google is the leading company in driving this trend, and has been aggressively pushing new techniques such as mod_pagespeed [62] and SPDY [52]. Table 2.2 categorizes and summarizes the techniques proposed from both industry and academia.

2.3.1 Web Page Conversions

The first category converts Web pages to meet a list of best practices. The cost of such techniques is often a few additional computation on the front-end or proxy servers. A notable example in this category is mod_pagespeed [62], an Apache https module that automatically converts Web pages. We exemplify popular conversions below with an analysis of their benefits and risks.

- *Minifying bytes*: Minification such as removing empty spaces or comments reduces bytes so as to save bandwidths. This approach can be safely applied without inducing risks.
- *Combining objects*: Web objects are suggested to be combined for transfers. Because objects can be sent in parallel, it is not apparent as to why this improves page load performance.
- *Inlining objects*: Small objects are recommended to be inlined in the HTML page. While saving additional round trips to fetch the objects, it breaks the ability of caching. Therefore, only small objects are recommended to be inlined.
- *Externalizing objects*: To the contrary of inlining is externalizing which enables cacheability

Category	Techniques	Deployment	Cost	Benefits	Risks
Web page conversions	Minifying bytes	FE servers	computation at FE servers	bandwidth	N/A
	Combining objects			unknown	N/A
	Inlining objects			fast loads	broken caching
	Externalizing objects			caching	slow loads
	Reordering HTML text			comprehensive	broken pages
Network protocols	SPDY, HTTP/2	FE servers, browsers	N/A	efficient transfers	N/A
	TCP modifications	FE servers			
Latency reductions	TCP fast open	FE servers, browsers	N/A	latency	security
	ASAP [75]	DNS, FE, browsers			
	DNS pre-resolution	browser			unnecessarily resolved/connected
	TCP pre-connect				
Computation speedups	Parallel CSS [33]	browser	N/A	computation speed	N/A
	Parallel HTML [32]				
	ZOOMM [13]				
	Fast memory [76]				
	Hybrid DOM [15]				
Passive middleware	CDNs	CDNs	disk space	locality	N/A
	Sudaresan et al. [61]	edge networks			
	Silo [34]	browsers			
Active middleware	Amazon Silk	AWS cloud	computing nodes	same as page conversions; more computing power	depends
	Android Chrome Beta	Google cloud			
	Telefonica	edge networks			

Table 2.2: Summary of techniques that improve Web performance. FE stands for front end; BE stands for back end.

but increases round trips. Mod_pagespeed [62] recommends to externalize large objects.

- *Reordering HTML text*: Suggestions such as moving CSS to the front and JavaScript to the end have the potential to help page loads a lot because JavaScript evaluation blocks parsing and CSS evaluation blocks rendering. However, reordering HTML text could break Web pages and can only be used at the developers' discretion.

Web page conversions have a direct impact on page load performance, and are thus considered most effective among all kinds of techniques. However, it is hard to effectively convert pages without breaking functionalities.

2.3.2 End-Host Network Protocols

The next category modifies end-host network protocols at the application layer (i.e., SPDY or HTTP/2) or the transport layer (e.g., TCP).

SPDY: SPDY provides four key features:

- *Single TCP connection*. SPDY opens a single TCP connection to a domain and multiplexes multiple HTTP requests and responses (a.k.a., SPDY streams) over the connection. The multiplexing here is similar to HTTP/1.1 pipelining but is finer-grained. A single connection also helps reduce SSL overhead. Besides client-side benefits, using a single connection helps reduce the number of TCP connections opened at servers.
- *Request prioritization*. Some Web objects such as JavaScript are more important than others and thus should be loaded earlier. SPDY allows the client to specify a priority level for each object, which is then used by the server in scheduling the transfer of the object.
- *Server push*. SPDY allows the server to push embedded objects before the client requests for them. This improves latency but could also increase transmitted data if the objects are already cached at the client.
- *Header compression*. SPDY supports HTTP header compression since experiments suggest that HTTP headers for a single session contain duplicate copies of the same information (e.g., `User-Agent`).

SPDY is implemented by adding a framing layer to the network stack between HTTP and the

transport layer. A TCP segment can carry multiple frames, making it possible to batch up small HTTP requests and responses.

Previous SPDY studies include the SPDY white paper [55] and measurements by Microsoft [41], Akamai [38], and Cable Labs [54]. The SPDT white paper shows a 27% to 60% speedup for SPDY compared to HTTP, but the other studies show that SPDT helps only marginally. While providing invaluable measurements, these studies look at a limited parameter space. Studies by Microsoft [41] and Cable Labs [54] only measured single Web pages and the other studies consider only a limited set of network conditions. Our study extensively swept the parameter space including network parameters, TCP settings, and Web page characteristics.

TCP: Google have proposed and deployed several TCP enhancements to make the Web faster. TCP fast open eliminates the TCP connection setup time by sending application data in the SYN packet [45]. Proportional rate reduction smoothly backs off congestion window to transmit more data under packet loss [19]. Tail loss probe and other measurement-driven enhancements described in [20] mitigated or eliminated loss recovery by retransmission timeout.

2.3.3 Latency Reductions

Another category focuses on reducing latencies since latencies as opposed to bandwidths are found to limit page load performance. TCP fast open reduces the TCP connection setup time [45] and ASAP combines DNS resolution, TCP connection setup, and data transfers in TCP using one RTT [75]. Recent browsers speculatively perform DNS resolutions and TCP connection setups preemptively. The risk is that they can unnecessarily resolve DNS or set up TCP. Latency reduction approaches in general do not impose additional costs.

2.3.4 Computation speedups

Much work has been done to improve page load computations, with a focus on exploiting parallelism. Meyerovich et al. proposed a parallel architecture for computing Web page layout by parallelizing CSS evaluations [33]. The Adrenaline browser exploits parallelism by splitting up a Web page into many pieces and processing each piece in parallel [32]. The ZOOMM browser further parallelizes the browser engine by preloading and preprocessing objects and by speeding

up computation in sub-activities [13]. Due to the dependencies that are intrinsic in browsers, the level of parallelism is largely limited. Besides increasing parallelism, other efforts focus on adding architectural support. Zhu et al. [76] specialized the processors for fast DOM tree and CSS access. Choi et al. [15] proposed a hybrid-DOM to efficiently access the DOM nodes.

2.3.5 Passive Middleware

Passive middleware (e.g., CDNs and caching) are used to improve locality. Content distribution networks (CDNs) are widely used to host images and videos that are close to the clients. Caching improves locality on the path of transfers. Sudaresan et al. [61] found that caching at the edge networks can help much of page load performance, given the presence of browser caching. Silo [34] modified the structure of Web pages to improve browser caching. Zhang et al. [74] caches computation such as style formatting and layout calculation. Wang et al. [69] speculatively loaded and cached embedded Web objects. Using CDNs or caching imposes no risks but requires additional disk space.

2.3.6 Active Middleware

In contrast to passive middleware, active middleware are usually proxies between a browser (often mobile) and a server. The idea is to optimize Web pages and the way they will be loaded at a proxy so as to accommodate mobile devices with constrained computing power, batteries, and cellular connections. There are multiple choices to place the proxy. Amazon Silk browser [4] and Android Chrome Beta [72] place proxies in their clouds while Telefonica uses a Web accelerator at their edge networks [5]. This proxy can apply many techniques described above, for example, Web page conversions and using SPDY to deliver transfers to clients. Therefore, the benefits and risks depend on the applied techniques, and the additional cost is computing power (or disk space) at the proxy.

Chapter 3

Demystifying Page Load Times with WProf

This chapter describes WProf, a lightweight in-browser profiler that pinpoints the bottleneck path while a Web page is being loaded. This bottleneck information not only identifies the inefficiencies in a page load, but also enables effective evaluations of techniques on how much they contribute to the overall PLT.

Central to WProf is the relations between activities that make up a page load, which are yet to be thoroughly studied. To this end, we first abstract the dependency policies that describe the relations in four browsers, i.e., IE, Firefox, Chrome, and Safari. We run experiments with systematically crafted test pages and observe object timings using Developer Tools [18]. For cases when Developer Tools are insufficient, we deduce dependencies by inspecting the browser code when open source code is available. We find that dependency policies are often the result of Web standards (e.g., JavaScript evaluation in script tags blocks HTML parsing) and browser implementation choices (e.g., a single thread of execution shared by parsing, JavaScript and CSS evaluation, and rendering).

Given the dependency policies, we develop a lightweight profiler, WProf, that runs in Webkit browsers (e.g., Chrome, Safari) while real pages are being loaded. WProf generates a dependency graph and identifies a load bottleneck for any given Web page. Unlike existing tools that produce waterfall or HAR reports [24], our profiler discovers and reports the dependencies between the browser activities that make up a page load. It is this information that pinpoints why, for example, page parsing took unexpectedly long and hence suggests what may be done to improve PLT.

Process	Activity
HTML parsing	Parsing a single tag
Object loading	Loading a single object
Evaluation	Loading a single JavaScript or CSS
Rendering	Layout/painting for the current DOM

Table 3.1: Discretizing a process into a set of activities

To study the composition of PLT, we run WProf while fetching pages from popular servers and apply critical path analysis. First, we identify page load bottlenecks by computing what fraction of the critical path the activity occupies. Surprisingly, we find that while most prior work focuses on network activity, computation (mostly HTML parsing and JavaScript execution) comprises 35% of the critical path. Second, we study the effectiveness of different techniques for optimizing web page load. Caching reduces the volume of data substantially, but decreases PLT by a lesser amount because many of the downloads that benefit from it are not on the critical path. Disappointingly, mod_pagespeed does little to reduce PLT because minifying and merging objects does not reduce network time on critical paths.

3.1 Dependency Policies

Web page dependencies are caused by various factors such as co-dependence between network and computation activities, manipulation of common objects, limited resources, etc. Browsers use various policies to enforce these dependencies. Our goal is to abstract the dependency policies.

3.1.1 Definitions

Activity: Ideally, the four processes involved in a page load (described in Figure 2.2) would be executed in parallel, so that the page load time is determined by the slowest process. However, the processes are inter-dependent and often block each other. To represent the dependency policies, we first discretize the steps involved in each process. The granularity of discretization should be fine enough to reflect dependencies and coarse enough to preserve semantics. We consider the most coarse-grained atomic unit of work, which we call an *activity*. In the case of HTML Parser, the

activity is parsing a single tag. The Parser repeatedly executes this activity until all tags are parsed. Similarly, the Object Loader activity is loading a single object, the Evaluator activity is evaluating a single JavaScript or CSS, and the activity of the Rendering process is rendering the current DOM. Table 3.1 defines the activity associated with each of the four processes.

Dependency: We say an activity a_i is *dependent* on a previously scheduled activity a_j, if a_i can be executed only after a_j has completed. There are two exceptions to this definition, where the activity is executed after only a *partial* completion of the previous activity. We discuss these exceptions in §3.1.3.

3.1.2 Methodology

We extract browser dependency policies by (i) inspecting browser documentation, (ii) inspecting browser code if open-source code is available, and (iii) systematically instrumenting test pages. Note that no single method provides a complete set of dependency policies, but they complement each other in the information they provide. Our methodology assumes that browsers tend to parse tags sequentially. However, one exception is *preloading*. Preloading means that the browser pre-emptively loads objects that are embedded later in the page, even before their corresponding tags are parsed. Preloading is often used to speed up page loads.

Below, we describe how we instrument and experiment with test pages. We conduct experiments in four browsers: Chrome, Firefox, Internet Explorer, and Safari. We host our test pages on controlled servers. We observe the load timings of each object in the page using Developer Tools [18] made available by the browsers. We are unable to infer certain dependencies such as rendering using Developer Tools. Instead, for open-source browsers, we inspect browser code to study the dependency policies associated with rendering.

Instrumenting test pages (network): We instrument test pages to exhaustively cover possible loading scenarios: (i) loading objects in different orders, and (ii) loading embedded objects. We list our instrumented test pages and results at `http://wprof.cs.washington.edu/tests`. Web pages can embed five kinds of objects as described in §2.1.2. We first create test pages that embed all combinations of object pairs, e.g., the test page may request an embedded JavaScript

followed by an image. Next, we create test pages that embed more than two objects in all possible combinations. Embedded objects may further embed other objects. We create test pages for each object type that in-turn embeds all combinations of objects. To infer dependency policies, we systematically inject delays to objects and observe load times of other objects to see whether they are delayed accordingly, similar to the technique used in WebProphet [30]. For example, in a test page that embeds a JavaScript followed by an image, we delay loading the Javascript and observe whether the image is delayed.

Instrumenting test pages (computation): Developer tools expose timings of network activities, but not timings of computational activities. We instrument test pages to circumvent this problem and study dependencies during two main computation activities: HTML parsing and JavaScript evaluation. HTML parsing is often blocked during page load. To study the blocking behavior across browsers, we create test pages for each object type. For each page, we also embed an iframe in the end. During page load, if the iframe begins loading only after the previous object finishes loading, we infer that HTML parsing is blocked during the object load. Iframes are ideal for this purpose because they are not preloaded by browsers. To identify dependencies related to JavaScript evaluation, we create test pages that contain scripts with increasing complexity; i.e., scripts that require more and more time for evaluation. We embed an iframe at the end. As before, if iframe does not load immediately after the script loads, we infer that HTML parsing is blocked for script evaluation.

3.1.3 Policies

Using our methodology, we uncover the dependency policies in browsers and categorize them as: Flow dependency, Output dependency, Lazy/Eager binding dependency, and dependencies imposed by resource constraints. Table 3.2 tabulates the dependency policies. While output dependencies are required for correctness, the dependencies imposed by lazy/eager binding and resource constraints are a result of browser implementation strategies.

Flow dependency is the simplest form of dependency. For example, loading an object depends on parsing a tag that references the object (F1). Similarly, evaluating a JavaScript depends on

Dependency	Name	Definition
Flow	F1	Loading an object → Parsing the tag that references the object
	F2	Evaluating an object → Loading the object
	F3	Parsing the HTML page → Loading the first block of the HTML page*
	F4	Rendering the DOM tree → Updating the DOM
	F5	Loading an object referenced by a JavaScript or CSS → Evaluating the JavaScript or CSS*
	F6	Downloading/Evaluating an object → Listener triggers or timers
Output	O1	Parsing the next tag → Completion of a previous JavaScript download and evaluation
	O2	JavaScript evaluation → Completion of a previous CSS evaluation
	O3	Parsing the next tag → Completion of a previous CSS download and evaluation
Lazy/Eager binding	B1	[Lazy] Loading an image appeared in a CSS → Parsing the tag decorated by the image
	B2	[Lazy] Loading an image appeared in a CSS → Evaluation of any CSS that appears in front of the tag decorated by the image
	B3	[Eager] Preloading embedded objects does not depend on the status of HTML parsing. (breaks F1)
Resource constraint	R1	Number of objects fetched from different servers → Number of TCP connections allowed per domain
	R2	Browsers may execute key computational activities on the same thread, creating dependencies among the activities.This dependency is determined by the scheduling policy.

* An activity depends on *partial* completion of another activity.

Table 3.2: Summary of dependency policies imposed by browsers. → represents "depends on" relationship.

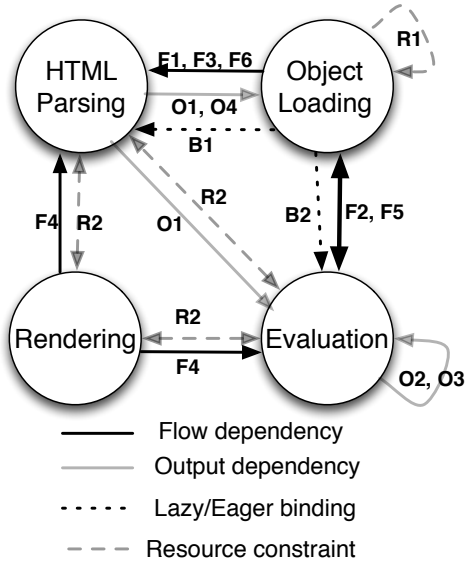


Figure 3.1: Dependencies between processes.

loading the JavaScript (F2). Often, browsers may load and evaluate a JavaScript based on triggers and timeouts, rather than the content of the page (F6). Table 3.2 provides the complete set of flow dependencies. Note that dependencies F3 and F5 are special cases, where the activity only depends on the partial completion of the previous activity. In case of F3, the browser starts to parse the page when the first chunk of the page is loaded, not waiting for the entire load to be completed. In case of F5, an object requested by a JavaScript/CSS is loaded immediately after evaluation starts, not waiting for the evaluation to be completed.

Output dependency ensures the correctness of execution when multiple processes modify a shared resource and execution order matters. In browsers, the shared resource is the DOM. Since both JavaScript evaluation and HTML parsing may write to the DOM, HTML parsing is blocked until JavaScript is both loaded and evaluated (O1). This ensures that the DOM is modified in the order specified in the page. Since JavaScript can modify styles of DOM nodes, execution of JavaScript waits for the completion of CSS processing (O2). Note that *async* JavaScript is not bounded by output dependencies because the order of script execution does not matter.

Lazy/Eager binding: Several lazy/eager binding techniques are used by the browser to trade off between decreasing spurious downloads and improving latency. Preloading (B3) is an example of an eager binding technique where browsers preemptively load objects that are embedded later in the page. Dependency B1 is a result of a lazy binding technique. When a CSS object is downloaded and evaluated, it may include an embedded image, for example, to decorate the background or to make CSS sprites. The browser does not load this image as soon as CSS is evaluated, and instead waits until it parses a tag that is decorated by the image. This ensures that the image is downloaded only if it is used.

Resource constraints: Browsers constrain the use of two resources: compute power and network resource. With respect to network resources, browsers limit the number of TCP connections. For example, Firefox limits the number of open TCP connections per domain to six by default. If a page load process needs to load more than six embedded objects from the same domain simultaneously, the upcoming load is blocked until a previous load completes. Similarly, some browsers allocate a

Dependency	IE	Firefox	WebKit
Output	all	no O3	no O3
Late binding	all	all	all
Eager Binding	*Preloads img, JS, CSS	Preloads img, JS, CSS	Preloads JS, CSS
Resource (R1)	6 conn.	6 conn.	6 conn.

Table 3.3: Dependency policies across browsers.

single compute thread to execute certain computational activities. For example, WebKit executes parts of rendering in the parsing thread. This results in blocking parsing until rendering is complete (R2). We were able to observe this only for the open-source WebKit browser because we directly instrumented the WebKit engine to log precise timing information of computational activities.

Figure 3.1 summarizes how these dependencies affect the four processes. Note that more than one dependency relationship can exist between two activities. For example, consider a page containing a CSS object followed by a JavaScript object. Evaluating the JavaScript depends on both loading the JavaScript (F2) and evaluating the previously appearing CSS (O3). The timing of these two dependencies will determine which of the two dependencies occur in this instance.

3.1.4 Dependency policies across browsers

Table 3.3 show the dependency policies across browsers. Only IE enforces dependency O3 that blocks HTML parsing to download and evaluate CSS. The preloading policies (i.e., when and what objects to preload) also differ among browsers, while we note that no browser preloads embedded iframes. The limit of parallel TCP connections per domain (R1) is consistent across browsers, but is often configurable. Note that flow dependency is implicitly imposed by all major browsers. We were unable to compare the compute dependency (R2) across browsers because it requires modification to the browser code.

3.1.5 Dependency graph of an example page

Figure 3.2 shows a dependency graph of an example page. The `DOMContentLoaded` refers to the event that HTML finishes parsing, `load` refers to the event that all embedded objects are

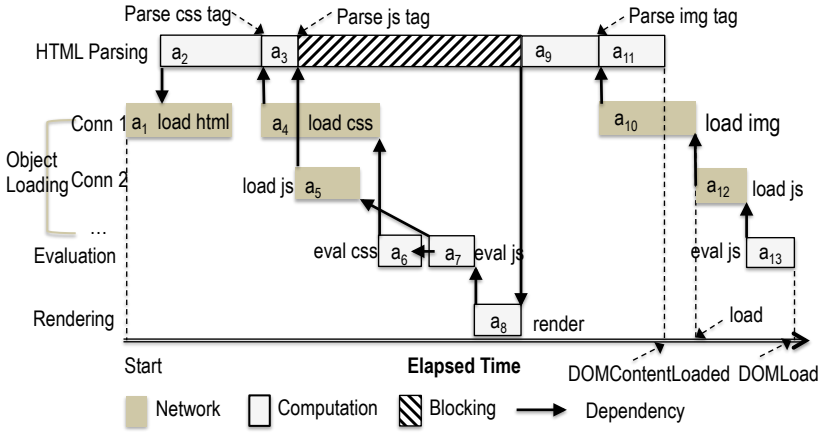


Figure 3.2: Dependency graph. Arrows denote"depends on" relation and vertices represent activities in a page load.

```
<html>
  <head>
    <link href='a.css'
      rel='stylesheet'>
    <script src='b.js' />
  </head>
  <!-- req a JS -->
  <body onload='...'>
    <img src='c.png' />
  </body>
</html>
```

Figure 3.3: Corresponding example code.

loaded, and `DOMLoad` refers to the event that DOM is fully loaded. Our example Web page (Figure 3.3) contains an embedded CSS, a JavaScript, an image, and a JavaScript triggered on the `load` event. Many Web pages (e.g., `facebook.com`) may load additional objects on the `load` event fires.

The page load starts with loading the root HTML page (a_1), following which parsing begins (a_2). When the Parser encounters the CSS tag, it loads the CSS (a_4) but does not block parsing. However, when the Parser encounters the JavaScript tag, it loads the JavaScript (a_5) and blocks parsing. Note that loading the CSS and the JavaScript subjects to a resource constraint; i.e., both CSS and JavaScript can be loaded simultaneously only if multiple TCP connections can be opened per domain. CSS is evaluated (a_6) after being loaded. Even though JavaScript finishes loading, it needs to wait until the CSS finishes evaluating, and then the JavaScript is evaluated (a_7). The rendering engine renders the current DOM (a_8) after which HTML parsing resumes (a_9). The Parser then encounters and loads an image (a_{10}) after which HTML parsing is completed and fires a `load` event. In our example, the triggered JavaScript is loaded (a_{12}) and evaluated (a_{13}). When all embedded objects are evaluated and the DOM is updated, the `DOMLoad` event is fired. Even our simple Web page exhibits over 10 dependencies of different types. For example, $a_2 \rightarrow a_1$ is a flow dependency; $a_7 \rightarrow a_6$ is an output dependency; and $a_9 \rightarrow a_8$ is a resource dependency.

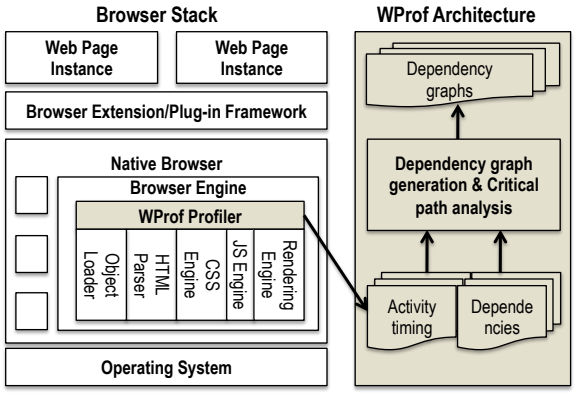


Figure 3.4: The WProf Architecture. WProf operates just above the browser engine, allowing it to collect precise timing and dependency information.

The figure also illustrates the importance of using dependency graphs for fine-grained accounting of page load time. For example, both activity a_4 and a_5 are network loads. However, only activity a_4 contributes to page load time; i.e., decreasing the load time of a_5 does not decrease total page load time. Tools such as HTTP Archival Records [24] provide network time for each activity, but this cannot be used to isolate the bottleneck activities. The next section demonstrates how to isolate the bottleneck activities using dependency graphs.

3.2 WProf

We present WProf, a tool that captures the dependency graph for any given Web page and identifies the delay bottlenecks. Figure 3.4 shows WProf architecture. The primary component is an in-browser profiler that instruments the browser engine to obtain timing and dependency information at runtime. The profiler is a shim layer inside the browser engine and requires minimal changes to the rest of the browser. Note that we do not work at the browser extension or plugin level because they provide limited visibility into the browser internals. WProf then analyzes the profiler logs offline to generate the dependency graphs and identify the bottleneck path. The profiler is lightweight and has negligible effect on PLTs (§3.2.4).

3.2.1 WProf Profiler

The key role of the WProf profiler is to record timing and dependency information for a page load. While the dependency information represents the structure of a page, the timing information captures how dependencies are exhibited for a specific page load; both are crucial to pinpoint the bottleneck path.

Logging Timing: WProf records the timestamps at the beginning and the end of each activity that is executed during the page load process. WProf also records network timing information, including DNS lookup, TCP connection setup, and HTTP transfer time. To keep track of the number of TCP connections being used/re-used, WProf records the IDs of all TCP connections.

Logging dependencies: WProf assumes the dependency policies described in §3.1.3 and uses different ways to log different kinds of dependencies. Flow dependencies are logged by attaching the URL to an activity. For example in Figure 3.2, WProf learns that the activity that evaluates b.js depends on the activity that loads b.js by recording b.js. WProf logs resource constraints by IDs of the constrained resources, e.g., thread IDs and TCP IDs.

To record output dependencies and lazy/eager bindings, WProf tracks a DOM-specific ordered sequence of processed HTML tags as the browser loads a page. We maintain a separate ordered list for each DOM tree associated with the page (e.g., the DOM for the root page and the various IFrames). HTML tags are recorded when they are first encountered; they are then attached to the activities that occur when the tags are being parsed. For example, when objects are preloaded, the tags under parsing are attached to the preloading activity, not the tags that reference the objects. For example in Figure 3.3, the Parser first processes the tag that references a.css, and then processes the tag that references b.js. This ordering, combined with the knowledge of output dependencies, result in the dependency $a_7 \rightarrow a_6$ in Figure 3.2. Note that the HTML page itself provides an implicit ordering of the page activities; however, this ordering is static. For example, if a JavaScript in a page modifies the rest of the page, statically analyzing the Web page provides an incorrect order sequence. Instead, WProf records the Web page processing order directly from the browser runtime.

Validating the completeness of our dependency policies would require either reading all the browser code or visiting all the Web pages, neither of which is practical. As browsers evolve constantly, changes in Web standard or browser implementations can change the dependency policies. Thus, WProf needs to be modified accordingly. Since WProf works as a shim layer in browsers that does not require knowledge about underlying components such as CSS evaluation, the effort to record an additional dependency policy would be minimal. The dependency policies still persist two years after the publication of WProf, meaning that the demand for changes in WProf is minimal given the slow evolutions of the Web standard and browser implementations that affect WProf.

3.2.2 Critical path analysis

To identify page load bottlenecks, we apply critical path analysis to dependency graphs (shown in Algorithm 3.2.1). Let the dependency graph $G = (V, E)$, where V is the set of activities required to render the page and E is the set of dependencies between the activities. Each activity $a \in V$ is associated with the attribute that represents the duration of the activity. The critical path P consists of a set of activities that form the longest path in the dependency graph such that, reducing the duration of any activity *not* on the critical path ($a \notin P$), will not change the critical path; i.e., optimizing an activity not on the critical path will not reduce page load performance. For example, the critical path for the dependency graph shown in Figure 3.2 is (a_1, a_2, a_4, a_6, a_7, a_8, a_9, a_{11}, a_{12}, a_{13}).

We estimate the critical path P of a dependency graph using the algorithm below. We treat preloading as a special case because it breaks the flow dependency. When preloading occurs, we always add it to the critical path.

Algorithm 3.2.1: CRITICALPATHANALYSIS(A)

$P \leftarrow \emptyset$

$a \leftarrow$ activity that completes last from A

$P \leftarrow P \cup a$

while a is not the first activity

do
- $A \leftarrow$ set of activities that a is dependent on
- **if** a is preloaded from an activity a' in A
  - **then** $a \leftarrow a'$
  - **else** $a \leftarrow$ activity in A that completes last from A
- $P \leftarrow P \cup a$

return (P)

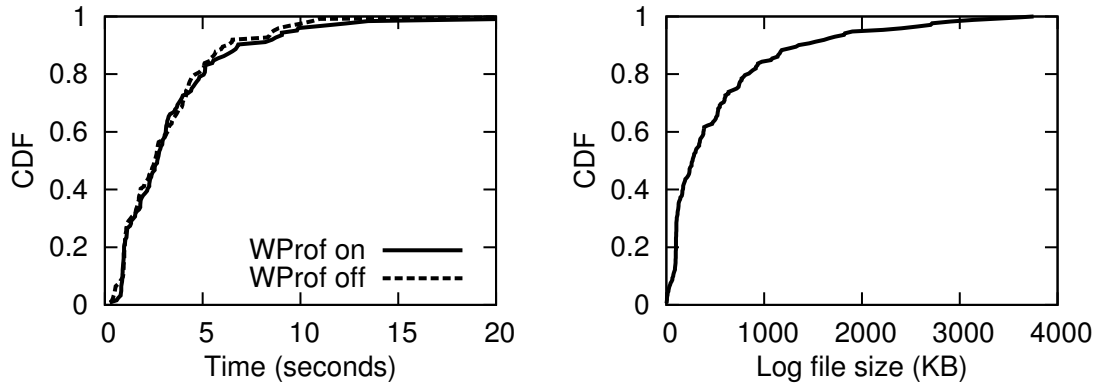
3.2.3 Implementation

We implement WProf on Webkit [70], an open source Web browser engine that is used by Chrome, Safari, Android, and iOS. The challenge is to ensure that the WProf shim layer is lightweight. Creating a unique identifier for each activity (for logging dependency relationship) is memory intensive as typical pages require several thousands of unique identifiers (e.g., for HTML tags). Instead, we use pointer addresses as unique identifiers. The WProf profiler keeps all logs in memory and transfers them to disk after the DOM is loaded, to minimize the impact on page load.

We extended Chrome and Safari by modifying 2K lines of C++ code. Our implementation can be easily extended to other browsers that build on top of Webkit. Our dependency graph generation and critical path analysis is written in Perl. The WProf source code is available at `http://wprof.cs.washington.edu`.

3.2.4 System evaluation

In this section, we present WProf micro-benchmarks. We perform the experiments on Chrome. We use a 2GHz CPU dual core and 4GB memory machine running MacOS. We load 200 pages



(a) CDF of page load time with and without WProf.

(b) CDF of log file sizes.

Figure 3.5: WProf evaluation.

WProf	CPU %	Memory %
on	58.5	65.5
off	53.5	54.9

Table 3.4: Maximum sampled CPU and memory usage.

with a five second interval between pages. The computer is connected via Ethernet, and to provide a stable network environment, we limit the bandwidth to 10Mbps using DummyNet [12].

Figure 3.5(a) shows the CDF of page load times with and without WProf. We define page load time as the time from when the page is requested to when the `DOMLoad` (Figure 3.2) event is fired. We discuss our rationale for the page load time definition in §3.3.1. The figure shows that WProf's in-browser profiler only has a negligible effect on the page load time. Similarly, Figure 3.5(b) shows the CDF of size of the logs generated by the WProf profiler. The median log file size is 268KB even without compression, and is unlikely to be a burden on the storage system. We sample the CPU and memory usage at a rate of 0.1 second when loading the top 200 web pages with and without WProf. Table 3.4 shows that even in the maximum case, WProf only increases the CPU usage by 9.3% and memory usage by 19.3%.

3.3 Studies with WProf

The goal of our studies is to use WProf's dependency graph and critical path analysis to (i) identify the bottleneck activities during page load (§3.3.2), (ii) quantify page load performance under caching (§3.3.3), and (iii) quantify page load performance under two proposed optimization techniques, SPDY and mod_pagespeed (§3.3.4).

3.3.1 Methodology

Experimental setup: We conduct our experiments on default Chrome and WProf-instrumented Chrome. We automate our experiments using the Selenium Webdriver [48] that emulates browser clicks. By default, we present experiments conducted on an iMac with a 3GHz quad core CPU and 8GB memory. The computer is connected to campus Ethernet at UW Seattle. We use DummyNet [12] to provide a stable network connection of 10Mbps bandwidth; 10Mbps represents the average broadband bandwidth seen by urban users [37]. By default, we assume page loads are *cold*, i.e., the local cache is cleared. This is the common case, as 40%–60% of page loads are known to be cold loads [34]. We report the minimum page load time from a total of 5 runs.

Web pages: We experiment with the top 200 most visited websites from Alexa [3]. Because some websites get stuck due to reported hang bugs in Selenium Webdriver, we present results from the 150 websites that provide complete runs.

Page load time metric: We define Page Load Time (PLT) as the time between when the page is requested and when the `DOMLoad` event is fired. Recall that the `DOMLoad` event is fired when all embedded objects are fetched and added to the DOM (see Figure 3.2). We obtain all times directly from the browser engine. There are a few alternative definitions to PLT. The *above-the-fold* time [9] metric is a user-driven metric that measures the time until the page is shown on the screen. However, this metric requires recording and manually analyzing page load videos, a cumbersome and non-scalable process. Other researchers use `load` [11] or `DOMContentLoaded` [34] events logged in the HTTP Archival Record (HAR) [24] to indicate the end of the page load process. Since we can tap directly into the browser, we do not need to rely on the external HAR.

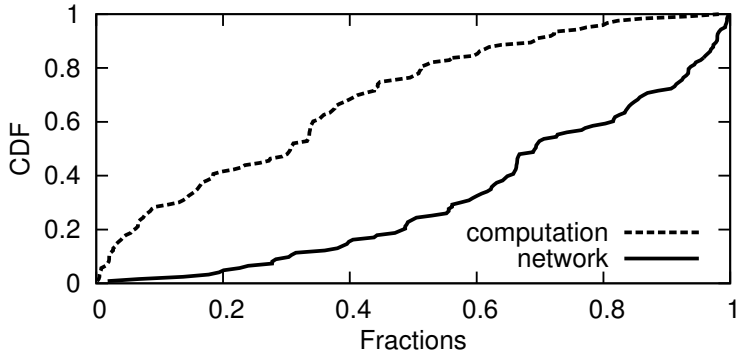


Figure 3.6: Fractions of time on computation v.s. network on the critical path.

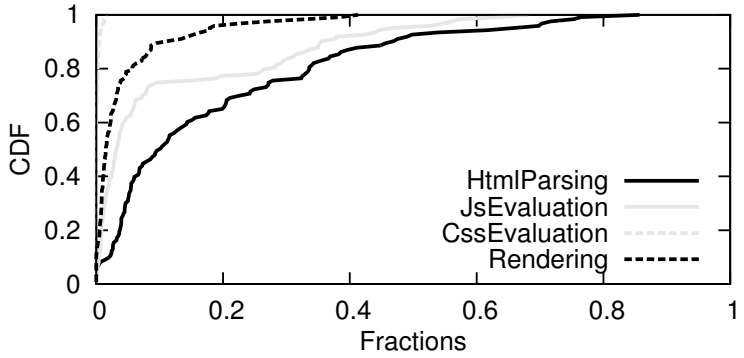


Figure 3.7: A breakdown of computational time to total page load time on the critical path.

We perform additional experiments with varying location, compute power, and Internet speeds. To exclude bias towards popular pages, we also experiment on 200 random home pages that we choose from the top 1 million Alexa websites. We summarize our results of these experiments in §3.3.5.

3.3.2 Identifying load bottlenecks (no caching)

The goal of this section is to characterize bottleneck activities that contribute to the page load time. Note that all of these results focus on delays on critical paths. In aggregate, a typical page we analyzed contained 32 objects and 6 activities on the critical path (all median values).

Computation is significant: Figure 3.6 shows that 35% of page load time in the critical path is spent

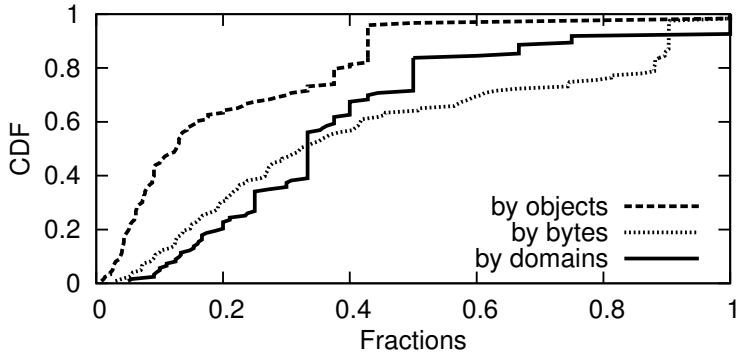
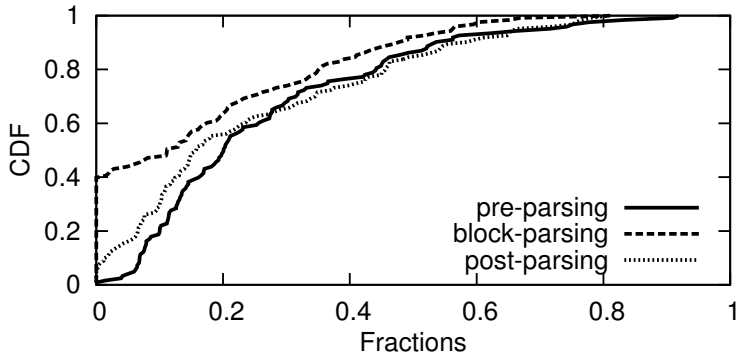


Figure 3.8: Fractions of domains, objects, and bytes on critical paths.

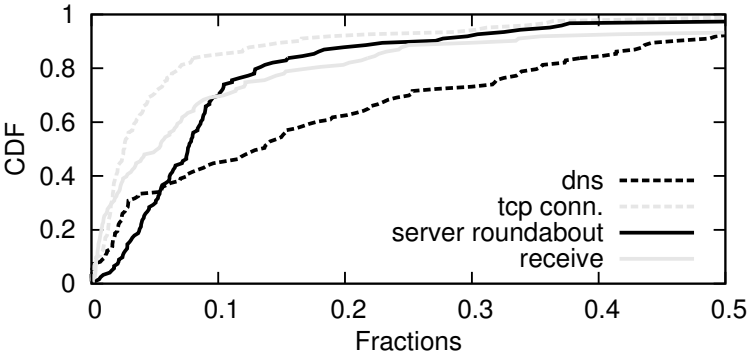
on computation; therefore computation is a critical factor in modeling or simulating page loads. Related measurements [27] do not estimate this computational component because they treat the browser engine as a black box.

We further break down computation into HTML parsing, JavaScript and CSS evaluation, and rendering in Figure 3.7. The fractions are with respect to the total page load time. Little time on the critical path is spent on firing timers or listeners, and so we do not show them. Interestingly, we find that HTML parsing costs the most in computation, at 10%. This is likely because: (i) many pages contain a large number of HTML tags, requiring a long time to convert to DOM; (ii) there is a significant amount of overhead when interacting with other components. For example, we find an interval of two milliseconds between reception of the first block of an HTML page and parsing the first HTML tag. JavaScript evaluation is also significant as Web pages embed more and more scripts. In contrast, CSS evaluation and rendering only cost a small fraction of page load time. This suggests that optimizing CSS is unlikely to be effective at reducing page load time.

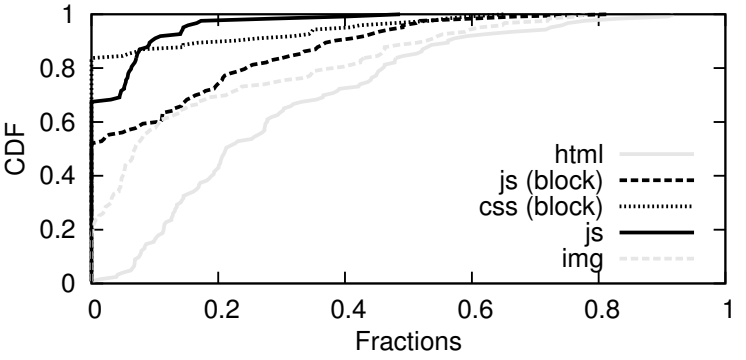
Network activities often block parsing. First, we break down network time by how it interacts with HTML parsing in Figure 3.9(a): pre-parsing, block-parsing, and post-parsing. As before, the fractions are with respect to the total page load time. The pre-parsing phase consists of fetching the first chunk of the page during which no content can be rendered. 15% of page load time is spent in this phase. The post-parsing phase refers to loading objects after HTML parsing (e.g., loading a_{10}



(a) By phase.

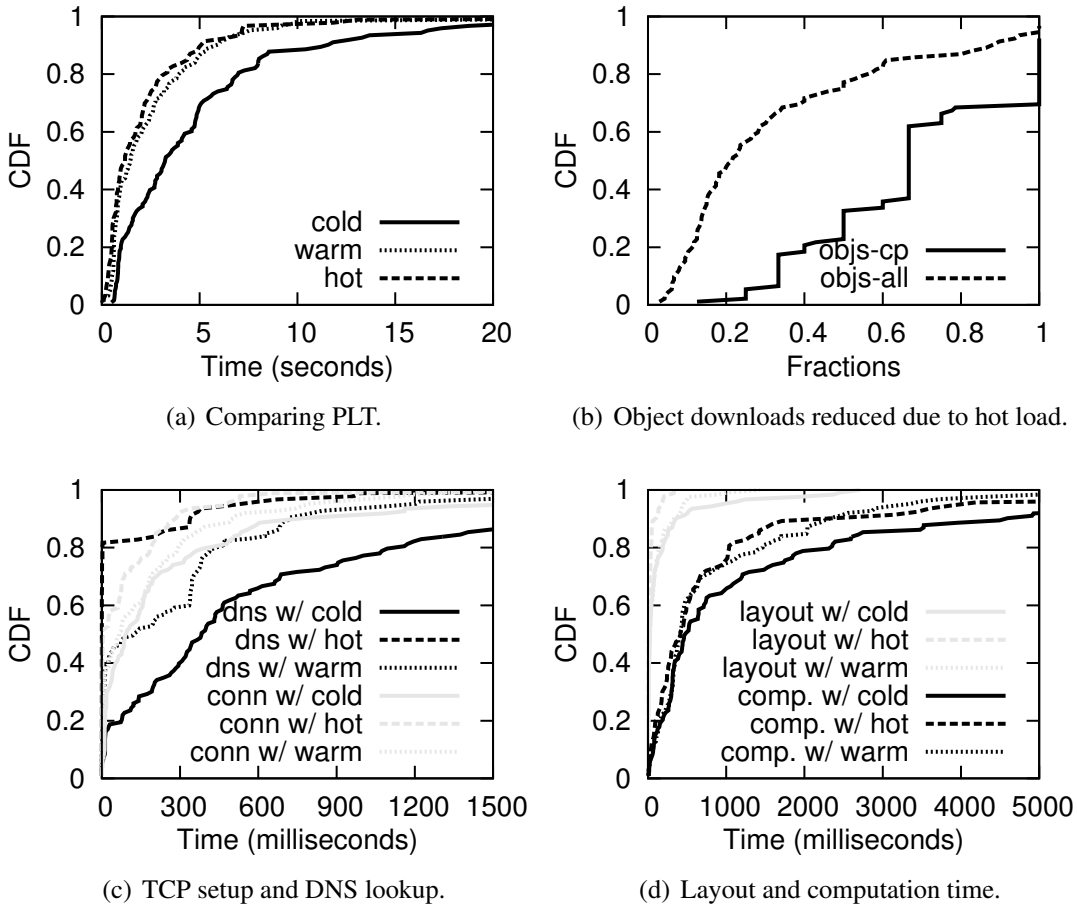


(b) By functionality.



(c) By mime type.

Figure 3.9: A breakdown of fractions of network time on the critical path.



(a) Comparing PLT.

(b) Object downloads reduced due to hot load.

(c) TCP setup and DNS lookup.

(d) Layout and computation time.

Figure 3.10: Warm and hot loads results. All results are a fraction of total page load time.

and a_{12} in Figure 3.2). Because rendering can be done before or during post parsing, post parsing is less important, though significant. Surprisingly, much of the network delay in the critical path blocks parsing. Recall that network loads can be done in parallel with parsing unless there are dependencies that block parsing, e.g., JavaScript downloads. Parsing-blocking downloads often occur at an early stage of HTML parsing that blocks loading subsequent embedded objects. The result suggests that a large portion of the critical path delay can be reduced if the pages are created carefully to avoid blocked parsing.

Second, we break down network time by functionality in Figure 3.9(b): DNS lookup, TCP

connection setup, server roundabouts, and receive time. DNS lookup and TCP connection setup are summed up for all the objects that are loaded, if they are on the critical path. Server roundabout refers to the time taken by the server to process the request plus a round trip time; again for every object loaded on the critical path. Finally, the receive time is the total time to receive each object on the critical path. DNS lookup incurs almost 13% of the critical path delay. To exclude bias towards one DNS server, we repeated our experiments with an OpenDNS [39] server, and found that the lookup time was still large. Our results suggest that reducing DNS lookup time alone can reduce page load time significantly. The server roundabout time is 8% of the critical path.

Third, we break down network time by MIME type in Figure 3.9(c). We find that loading HTML is the largest fraction (20%) and mostly occurs in the pre-parsing phase. Loading images is also a large fraction on critical paths. Parsing-blocking JavaScript is significant on critical paths while asynchronous JavaScript only contributes a small fraction. Interestingly, we find a small fraction of CSS that blocks JavaScript evaluation and thus blocks HTML parsing. This blocking can be reduced simply by moving the CSS tag after the JavaScript tag. There is almost no non-blocking CSS on critical paths, and therefore we omit it in Figure 3.9(c).

Last, we look at the page load time for external objects corresponding to Web Analytics and Ads but not CDNs. We find that one fourth of Web pages have downloads of external objects on their critical path. Of those pages, over half spends 15% or more page load time to fetch external objects and one even spends up to 60%.

Most object downloads are non-critical: Figure 3.8 compares all object downloads and the object downloads only on the critical path. Interestingly, only 30% bytes of content is on critical paths. This suggests that minifying and caching content may not improve page load time, unless they reduce content downloaded along the critical path. We analyze this further in the next section. Note that object downloads off the critical paths are not completely unimportant. Degrading the delay of some activities that are not on the critical path may cause them to become critical.

3.3.3 Identifying load bottlenecks (with caching)

We analyze the effects of caching under two conditions: hot load and warm load. Hot loads occur when a page is loaded immediately after a cold load. Warm loads occur when a page is loaded a small amount of time after a cold load when the immediate cache would have expired. We set a time interval of 12 minutes for warm loads. Both cases are common scenarios, since users often reload pages immediately as well as after a short period of time.

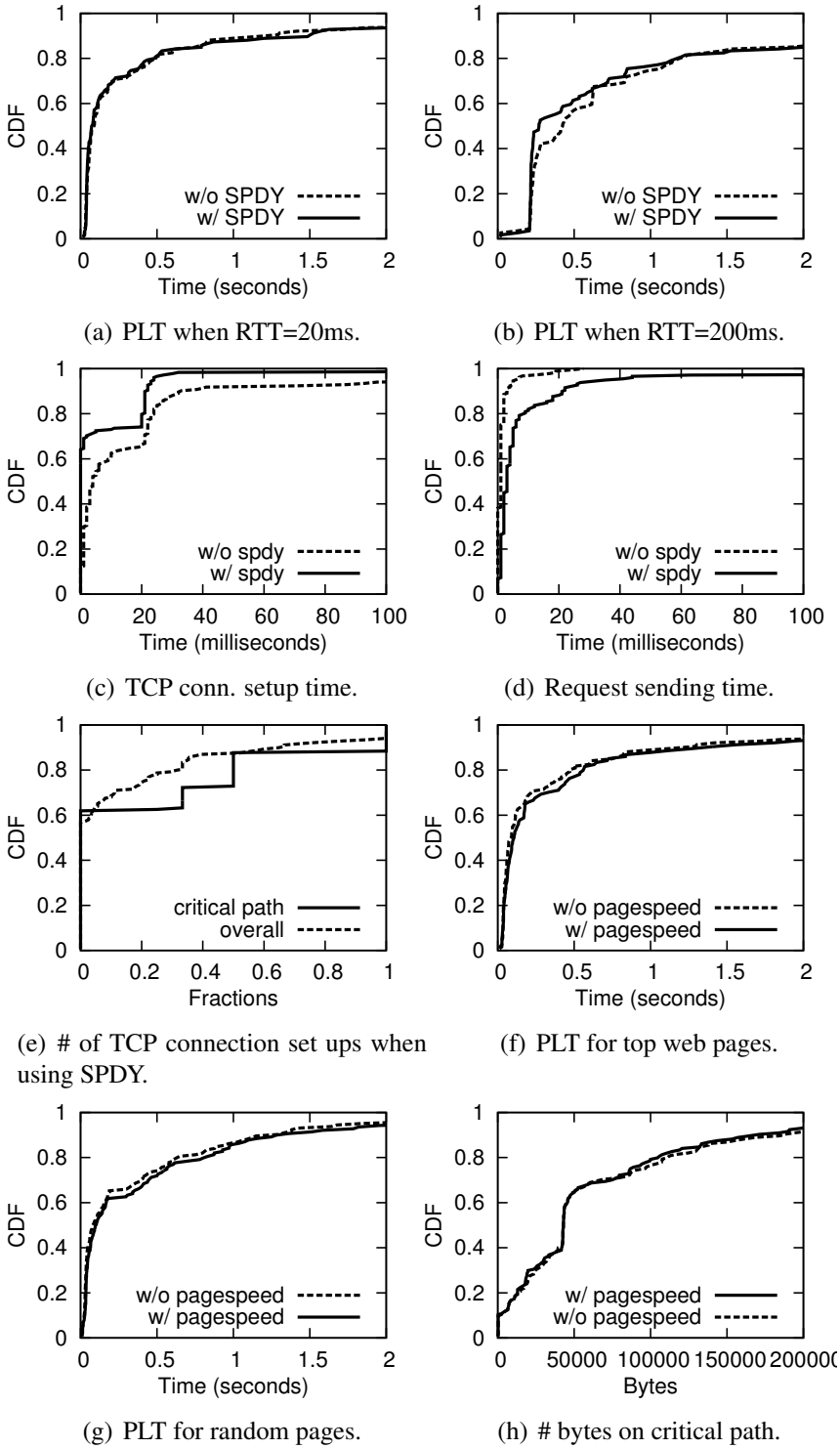
Caching gains are not proportional to saved bytes. Figure 3.10(a) shows the distributions of page load times under cold, warm, and hot loads. For 50% of the pages, caching decreases page load time by over 40%. However, further analysis shows that 90% of the objects were cached during the experiments. In other words, the decrease in page load time is not proportional to the cached bytes. To understand this further, we analyze the fraction of cached objects that are on and off the critical path, during hot loads. Figure 3.10(b) shows that while caching reduces 65% of total object loads (marked *objs-all*), it only reduces 20% of object loads on the critical path (marked *objs-cp*). Caching objects that are not on the critical path leads to the disproportional savings.

Caching also reduces computation time. Figures 3.10(c) and 3.10(d) compare the network times and computation times of hot, warm, and cold loads, on the critical path. As expected, hot and warm loads reduce DNS lookup time and TCP connection setup. Especially during hot loads, DNS lookup time is an insignificant fraction of the page load time in contrast to cold loads. Interestingly, caching not only improves network time, but also computation time. Figure 3.10(d) shows the time taken by compute and layout activities during page load time. The figure suggests that modern browsers cache intermediate computation steps, further reducing the page load time during hot and warm loads.

3.3.4 Evaluating proposed techniques

This section evaluates two Web optimization techniques, SPDY [52] and mod_pagespeed [62].

SPDY: SPDY is an application-level protocol in place of HTTP. The key ideas in SPDY are: (i) Multiplexing HTTP transactions into a single TCP connection to reduce TCP connection setup time and to avoid slow start, (ii) prioritizing some object loads over others (e.g., JavaScript over



(a) PLT when RTT=20ms.

(b) PLT when RTT=200ms.

(c) TCP conn. setup time.

(d) Request sending time.

(e) # of TCP connection set ups when using SPDY.

(f) PLT for top web pages.

(g) PLT for random pages.

(h) # bytes on critical path.

Figure 3.11: SPDY and mod_pagespeed results.

images), and (iii) reducing HTTP header sizes. SPDY also includes two optional techniques—Server Push and Server Hint. Exploiting these additional options require extensive knowledge of Web pages and manual configurations. Therefore, we do not include them here.

We evaluate SPDY with controlled experiments on our own server with 2.4GHz 16 core CPU 16GB memory and Linux kernel 2.6.39. By default, we use a link with a controlled 10Mbps bandwidth, and set the TCP initial window size to 10 (increased from the default 3, as per SPDY recommendation). We use the same set of pages as the real-world experiments and download all embedded objects to our server[1] to avoid domain sharding [38]. We run SPDY version 2 over SSL.

SPDY only improves performance at high RTTs. Figure 3.11(a) and Figure 3.11(b) compares the page load time of SPDY versus non-SPDY (i.e., default HTTP) under 20ms and 200ms RTTs, respectively. SPDY provides few benefits to page load time under low RTTs. However, under 200ms RTT, SPDY improves page load time by 5%–40% for 30% of the pages. We conduct additional experiments by varying the TCP initial window size and packet loss, but find that the results are similar to the default setting.

SPDY reduces TCP connection setup time but increases request sending time. To understand SPDY performance, we compare SPDY and non-SPDY network activities on the critical path and break down the network activities by functionality; note that SPDY does not affect computation activities. For the 20ms RTT case, Figure 3.11(c) shows that SPDY significantly reduces TCP connection setup time. However, since SPDY delays sending requests to create a single TCP connection, Figure 3.11(d) shows that SPDY increases the time taken to send requests. Other network delays such as server roundabout time and total receive time remained similar.

Further, Figure 3.11(e) shows that although SPDY reduces TCP setup times, the number of TCP connection setups in the critical path is small. Coupled with the increase in request sending time, the total improvement due to SPDY cancels out, resulting in no net improvement in page load time.

The goal of our evaluation is to explain the page load behavior of SPDY using critical path

[1]Because we are unable to download objects that are dynamically generated, we respond to these requests with a HTTP 404 error.

analysis. Improving SPDY's performance and leveraging SPDY's optional techniques are part of future work.

mod_pagespeed: mod_pagespeed [62] is an Apache module that enforces a number of best practices to improve page load performance, by minifying object sizes, merging multiple objects into a single object, and externalizing and/or inlining JavaScripts. We evaluate mod_pagespeed using the setup described in §3.3.4 with a 20ms RTT.

Figure 3.11(f) compares the page load times with and without mod_pagespeed on top 200 Alexa pages. mod_pagespeed provides little benefits to page load time. Since the top 200 pages may be optimized, we load 200 random pages from the top one million Alexa Web pages. Figure 3.11(g) shows that mod_pagespeed helps little even for random pages.

To understand the mod_pagespeed performance, we analyze minification and merging multiple objects. The analysis is based on loading the top 200 Web pages. Figure 3.11(h) shows that the total number of bytes downloaded on the critical path remains unchanged with and without mod_pagespeed. In other words, minifying object does not reduces the size of the objects loads on the critical path, and therefore does not provide page load benefits. Similarly, our experiments show that merging objects does not reduce the network delay on the critical path, because the object loads are not the bottleneck (not shown here). These results are consistent with our earlier results (Figure 3.8) that shows that only 30% of the object loads are on the critical path. We were unable to determine how mod_pagespeed decides to inline or externalize JavaScripts, and therefore are unable to conduct experiments on how inlining/externalizing affects page load performance.

3.3.5 Summarizing results from alternate settings

In addition to our default experiments, we conducted additional experiments: (i) with 2 machines, one with 2GHz dual core 4GB memory, and another with 2.4GHz dual core 2GB memory, (ii) in 2 different locations, one at UMass Amherst with campus Ethernet connectivity, and another in a residential area in Seattle with broadband connectivity, and (iii) using 200 random Web pages chosen from the top 1 million Alexa Web pages. All other parameters remained the same as default.

Below, we summarize the results of our experiments:

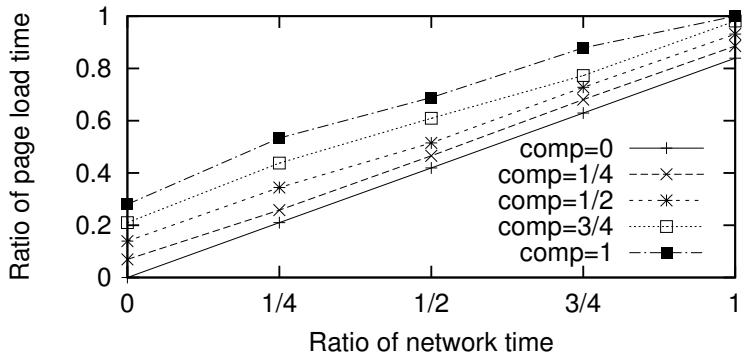


Figure 3.12: Median reduction in page load times if computation and network speeds are improved.

- The fraction of computation and network times on the critical path were quantitatively similar in different locations.
- The computation time as a fraction of the total page load time increased when using slower machines. For example, computation time was 40% of the critical path for the 2GHz machine, compared to 35% for the 3GHz machine.
- The 200 random Web pages experienced 2x page load time compared to the top Web pages. Of all the network components, the server roundabout time of random pages (see Figure 3.9(b)) was 2.3x of that of top pages. However, the total computation time on the critical path was 12% lower compared to the popular pages, because random pages embed less JavaScript.

3.4 Discussion

In this work, we have demonstrated that WProf can help identify page load bottlenecks and that it can help evaluate evolving techniques that optimize page loads. WProf can be potentially used in several other applications. Here, we briefly discuss two applications: (i) Conducting what-if analysis, and (ii) identifying potential Web optimizations.

What-if analysis: As CPUs and networks evolve, the question we ask is—how can page load times benefit from improving the network speed or CPU speed? We use the detailed dependency

graphs generated by WProf to conduct this what-if analysis. Figure 3.12 plots the reduction in page load time for a range of <network speedup, CPU speedup> combinations. When computational time is zeroed but the network time is unchanged, the page load time is reduced by 20%. If the network time is reduced to one fourth, but the computational time is unchanged, 45% of the page load time is reduced. Our results suggest that speeding up both network and computation together is more beneficial than just improving one of them. Note that our what-if analysis is limited as it does not capture all lower-level dependencies and constraints, e.g., TCP window sizes. We view our results here as estimates and leave a more extensive analysis for future work.

Potential optimizations: Our own experience and studies with WProf suggest a few optimization opportunities. We find that synchronous JavaScript significantly affects page load time, not only because of loading and evaluation, but also because of block-parsing (Figure 3.9(a)). Asynchronous JavaScript or in-lining the scripts can help reduce page load times. To validate this opportunity, we manually transform synchronous JavaScript to `async` on top five pages and find that this helps page load time. However, because asynchronous JavaScript may alter Web pages, we need further research to understand how and when JavaScript transformation affects Web pages. Another opportunity is with respect to prioritizing object loading according to the dependency graph. Prioritization can be either implemented at the application level such as SPDY or reflected using latency-reducing techniques [63]. For example, prioritizing JavaScript loads can decrease the time that HTML parsing is blocked. Recently, SPDY considers applying the dependency graph to prioritize object loading in version 4 [57]. Other opportunities include parallelizing page load activities and more aggressive preloading strategies, both of which require future exploration.

3.5 Summary

In this chapter, we abstract a model of browser dependencies, and design WProf, a lightweight, in-browser profiler that extracts dependency graphs of any given page. The goal of WProf is to identify bottleneck activities that contribute to the page load time. By extensively loading hundreds of pages and performing critical path analysis on their dependency graphs, we find that computation is a significant factor and makes up as much as 35% of the page load time on the critical path. We

also find that synchronous JavaScript evaluation plays a significant role in page load time because it blocks parsing. While caching reduces the size of downloads significantly, it is less effective in reducing page load time because loading does not always affect the critical path. We conducted experiments over SPDY and mod_pagespeed. While the effects of SPDY and mod_pagespeed vary over pages, surprisingly, we find that they help very little on average. In the future, we plan to extend our dependency graphs with more lower-level dependencies (e.g., in servers) to understand how page loads would be affected by these dependencies.

Chapter 4

How Speedy is SPDY?

This chapter focuses on analyzing the performance of SPDY (standardized as HTTP/2), an application-layer protocol designed for the Web. Given the central role that SPDY is likely to play in the Web, it is surprising that the performance of SPDY is not well understood. There have been several studies, predominantly white papers. But the findings often conflict [55, 41, 38, 54] because the performance of SPDY depends on many factors, even those that are external to SPDY.

We conduct what we believe to be the most in-depth study of PLT under SPDY to date. To augment the study with explanatory power, we isolate the different factors that affect PLT with experiments that progress from simple but unrealistic transfers to full page loads. Results across this progression let us systematically isolate the impact of the contributing factors and identify when SPDY helps significantly or performs poorly compared to HTTP. To make experiments reproducible, we develop a tool called `Epload` that controls the variability by recording and replaying the process of a page load at fine granularity, complete with browser dependencies and deterministic computational delays.

Our experiments progress as follows. We first compare SPDY and HTTP simply as a transport protocol (with no browser dependencies or computation) that transfers Web objects from both artificial and real pages (from the top 200 Alexa sites). We use a decision tree analysis to identify the situations in which SPDY outperforms HTTP and vice versa. We find that SPDY improves PLT significantly in a large number of scenarios that track the benefits of using a single TCP connection. Conversely, SPDY significantly hurts under high packet loss for large objects, because

a set of TCP connections tends to perform better under high packet loss;. Thus it is necessary to tune TCP behavior to boost PLT.

Next, we examine the complete Web page load process by incorporating dependencies and computational delays. With these factors, the benefits of SPDY are reduced, and can even be negated. This is because: i) there are fewer outstanding objects at a given time; ii) traffic is less bursty; and iii) the impact of the network is degraded by computation. Overall, we find SPDY benefits to be larger when there is less bandwidth and longer RTTs.

In search of greater benefits, we explore SPDY mechanisms for prioritization and server push. Prioritization helps little because it is limited by load dependencies, but server push has the potential for significant improvements. How to obtain this benefit depends on the server push policy, which is a non-trivial issue because of caching. This leads us to develop a policy based on dependency levels that performs comparably to mod_spdy's policy [35] while pushing 80% less data.

4.1 Background

In this section, we review issues with HTTP performance and describe how the new SPDY protocol addresses them.

4.1.1 Limitations of HTTP/1.1

When HTTP/1.1, or simply HTTP, was designed in the late 1990's, Web applications were fairly simple and rudimentary. Since then, Web pages have become more complex and dynamic, making it difficult for HTTP to meet the increasingly demanding user experience. Below, we identify some of the limitations of HTTP:

i) Browsers open too many TCP connections to load a page. HTTP improves performance by using parallel TCP connections. But if the number of connections is too large, the aggregate flow may cause network congestion, high packet loss, and reduced performance [23]. Further, services often deliver Web objects from multiple domains, which results in even more TCP connections and the possibility of high packet loss.

ii) Web transfers are strictly initiated from the client. Consider the loading of embedded objects. Theoretically, the server can send embedded objects along with the parent object when it receives

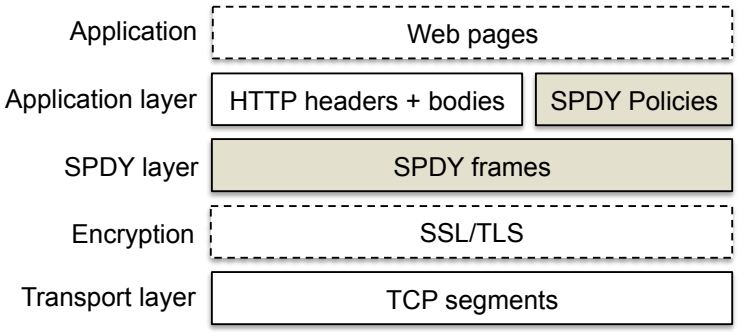


Figure 4.1: The network stack with SPDY. Boxes with dashed borders are optional.

a request for the parent object. In HTTP, because an object can be sent only in response to a client request, the server has to wait for an explicit request which is sent only after the client has received and processed the parent page.

iii) A TCP segment cannot carry more than one HTTP request or response. HTTP, TCP and other headers could account for a significant portion of a packet when HTTP requests or responses are small. So if there are a large number of small embedded objects in a page, the overhead associated with these headers is substantial.

4.1.2 SPDY

SPDY addresses several of the issues described above. We now review the key ideas in SPDY's design and implementation and its deployment status.

Design: There are four key SPDY features.

i) Single TCP connection. SPDY opens a single TCP connection to a domain and multiplexes multiple HTTP requests and responses (a.k.a., SPDY streams) over the connection. The multiplexing here is similar to HTTP/1.1 pipelining but is finer-grained. A single connection also helps reduce SSL overhead. Besides client-side benefits, using a single connection helps reduce the number of TCP connections opened at servers.

ii) Request prioritization. Some Web objects, such as JavaScript code modules, are more important than others and thus should be loaded earlier. SPDY allows the client to specify a priority level for each object, which is then used by the server in scheduling the transfer of the object.

iii) Server push. SPDY allows the server to push embedded objects before the client requests for them. This improves latency but could also increase transmitted data if the objects are already cached at the client.

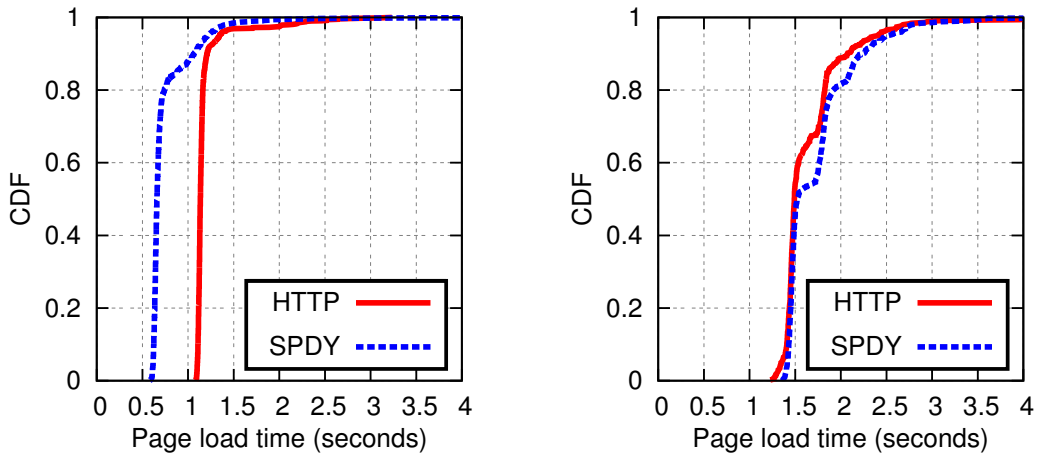
iv) Header compression. SPDY supports HTTP header compression since experiments suggest that HTTP headers for a single session contain duplicate copies of the same information (e.g., `User-Agent`).

Implementation: SPDY is implemented by adding a framing layer to the network stack between HTTP and the transport layer. Unlike HTTP, SPDY splits HTTP headers and data payloads into two kinds of frames. `SYN_STREAM` frames carry request headers and `SYN_REPLY` frames carry response headers. When a header exceeds the frame size, one or more `HEADERS` frames will follow. HTTP data payloads are sliced into `DATA` frames. There is no standardized value for the frame size, and we find that mod_spdy caps frame size to 4KB [35]. Because frame size is the granularity of multiplexing, too large a frame decreases the ability to multiplex while too small a frame increases overhead. SPDY frames are encapsulated in one or more consecutive TCP segments. A TCP segment can carry multiple SPDY frames, making it possible to batch up small HTTP requests and responses.

Deployment: SPDY is deployed over SSL and TCP. On the client side, SPDY is enabled in Chrome, Firefox, and IE 11. On the server side, popular websites such as Google, Facebook, and Twitter have deployed SPDY. Another popular use of SPDY is between a proxy and a client, such as the Amazon Silk browser [4] and Android Chrome Beta [72]. SPDY version 3 is the most recent specification and is widely deployed [56].

4.2 Pinning SPDY down

We would like to experimentally evaluate how SPDY performs relative to HTTP because SPDY is likely to play a key role in the Web. But, understanding SPDY performance is hard. Below, we identify three challenges in studying the performance of SPDY and then provide an overview of our approach.



(a) The flag page

(b) Twitter home page

Figure 4.2: Distributions of PLTs of SPDY and HTTP. Performed a thousand runs for each curve without caching.

4.2.1 Challenges

We identify the challenges on the basis of previous studies and our own initial experimentation. As a first step, we extensively load two Web pages for a thousand times using a measurement node at the University of Washington. One page displays fifty world flags [36], which is advertised by mod_spdy [35] to demonstrate the performance benefits of SPDY, and the other is the Twitter home page. The results are depicted in Figure 4.2.

First, we observe that SPDY helps the flag page but not the Twitter page, and it is not immediately apparent as to why that is the case. Further experimentation in emulated settings also revealed that both the magnitude and the direction of the performance differences vary significantly with network conditions. Taken together, this indicates that SPDY's performance depends on many factors such as Web page characteristics, network parameters, and TCP settings, and that measurement studies will likely yield different, even conflicting, results, if they use different experimental settings. Therefore, a comprehensive sweep of the parameter space is necessary to evaluate under what conditions SPDY helps, what kinds of Web pages benefit most from SPDY, and what

parameters best support SPDY.

Second, we observed in our experiments that the measured page load times have high variances, and this often overwhelms the differences between SPDY and HTTP. For example, in Figure 4.2(b), the variance of the PLT for the Twitter page is 0.5 second but the PLT difference between HTTP and SPDY is only 0.02 second. We observe high variance even when we load the two pages in a fully controlled network. This indicates that the variability likely stems from browser computation (i.e., JavaScript evaluation and HTML parsing). Controlling this variability is key to reproducing experiments so as to obtain meaningful comparisons.

Third, prior work has shown that the dependencies between network operations and computation has a significant impact on PLT [66]. Interestingly, page dependencies also influence the scheduling of network traffic and affects how much SPDY helps or hurts performance (§4.3 and §4.4). Thus, on one hand, ignoring browser computations can reduce PLT variability, but on the other hand, dependencies need to be preserved in order to obtain accurate measurements under realistic offered loads.

4.2.2 Approach

Our approach is to separate the various factors that affect SPDY and study them in isolation. This allows us to control and identify the extent to which these factors affect SPDY.

First, we extensively sweep the parameter space of all the factors that affect SPDY including RTT, bandwidth, loss rate, TCP initial window, number of objects on a page, and object sizes. We initially *ignore* page load dependencies and computation in order to simplify our analysis. This systematic study allows us to identify when SPDY helps or hurts and characterize the importance of the contributing factors. Based on further analysis of why SPDY sometimes hurts, we propose some simple modifications to TCP.

Second, before we perform experiments *with* page load dependencies, we address the variability caused by computation. We develop a tool called `Epload` that emulates the process of a page load. Instead of performing real browser computation, `Epload` records the process of a sample page load, identifies when computations happen, and replays the page load by introducing

the appropriate delays associated with the recorded computations. After emulating a computation activity, `Epload` performs real network requests to dependent Web objects. This allows us to control the variability of computation while also modeling page load dependencies. In contrast to the methodology that statistically reduces variability by obtaining a large amount of data (usually from production), our methodology mitigates the root cause of variability and thus largely reduces the amount of required experiments.

Third, we study the effects of dependencies and computation by performing page loads with `Epload`. We are then able to identify how much dependencies and computation affect SPDY, and to identify the relative importance of other contributing factors. To mitigate the negative impact of dependencies and computation, we explore the use of prioritization and server push that enable the client and the server to coordinate the transfers. Here, we are able to evaluate the extent to which these mechanisms can improve performance when used appropriately.

4.3 TCP and SPDY

In this section, we extensively study the performance of SPDY as a transfer protocol on both synthetic and real pages by ignoring page load dependencies and computation. This allows us to measure SPDY performance without other confounding factors such as browser computation and page load dependencies. Here, SPDY is only different from HTTP in the use of a single TCP connection, header compression, and a framing layer.

4.3.1 Experimental setup

We conduct the experiments by setting up a client and a server that can communicate over both HTTP and SPDY. Both the server and the client are connected to the campus LAN at the University of Washington. We use Dummynet [12] to vary network parameters. Below details the experimental setup.

Server: Our server is a 64-bit machine with 2.4GHz 16 core CPU and 16GB memory. It runs Ubuntu 12.04 with Linux kernel 3.7.5 using the default TCP variant Cubic. We use a TCP initial window size of ten as the default setting, as suggested by SPDY best practices [53]. HTTP and

SPDY are enabled on Apache 2.2.2 with the SPDY module, mod_spdy 0.9.3.3-386, installed. We use SPDY 3 without SSL which allows us to decode the SPDY frames in TCP payloads. To control the exact size of Web objects, we turn off gzip encoding.

Client: Because we issue requests at the granularity of Web objects and not pages, we do not work with browsers, and instead develop our own SPDY client by following the SPDY/3 specification [56]. Unlike other wget-like SPDY clients such as `spdylay` [58] that open a TCP connection per request, our SPDY client allows us to reuse TCP connections. Similarly, we also develop an HTTP client for comparison. We set the maximum number of parallel TCP connections for HTTP to six, as used by all major browsers. As the receive window is auto-tuned, it is not a bottleneck in our experiments.

Web pages: To experiment with synthetic pages, we create objects with pre-specified sizes and numbers. To experiment with real pages, we download the home pages of the Alexa top 200 websites to our own server. To avoid the negative impact of domain sharding on SPDY [53], we serve all embedded objects from the same server including those that are dynamically generated by JavaScript.

We run the experiments presented in the entire paper from June to September, 2013. We repeat our experiments five times and present the median to exclude the effects of random loss. We collect network traces at both the client and the server. We define page load time (PLT) as the elapsed time between when the first object is requested and when the last object is received. Because we do not experiment within a browser, we do not use the W3C `load` event [65].

4.3.2 Experimenting with synthetic pages

In experimenting with synthetic pages, we consider a broad range of parameter settings for the various factors that affect performance. Table 4.1 summarizes the parameter space used in our experiments. The RTT values include 20ms (intra-coast), 100ms (inter-coast), and 200ms (3G link or cross-continent). The bandwidths emulate a broadband link with 10Mbps [37] and a 3G link with 1Mbps [6]. We inject random packet loss rates from zero to 2% since studies suggest that Google servers experience a loss rate between 1% and 2% [19]. At the server, we vary TCP initial

Categ	Factor	Range	High
Net	rtt	20ms, 100ms, 200ms	$\geq$100ms
	bw	1Mbps, 10Mbps	$\geq$10Mbps
	pkt loss	0, 0.005, 0.01, 0.02	$\geq$ 0.01
TCP	iw	3, 10, 21, 32	$\geq$ 21
Page	obj size	100B, 1K, 10K, 100K, 1M	$\geq$ 1K
	# of obj	2, 8, 16, 32, 64, 128, 512	$\geq$ 64

Table 4.1: Contributing factors to SPDY performance. We define a threshold for each factor, so that we can classify a setting as being high or low in our analysis.

window size from 3 (used by earlier Linux kernel versions) to 32 (used by Google servers). We also consider a wide range of Web object sizes (100B to 1M) and object numbers (2 to 512). For simplicity, we choose one value for each factor which means that there is no cross traffic.

When we sweep this large parameter space, we find that SPDY improves performance under certain conditions, but degrades performance under other conditions.

(i) When does SPDY help or hurt

There have been many hypotheses as to whether SPDY helps or hurts based on analytical inference about parallel versus single TCP connections. For example, one hypothesis is that SPDY hurts because a single TCP connection increases congestion window slower than multiple connections; another hypothesis is that SPDY helps stragglers because HTTP has to balance its communications across parallel TCP. However, it is unclear how much hypotheses contribute to SPDY performance. Here, we sort out the most important findings, meaning that hypotheses that are shown here contribute more to SPDY performance than those that are not shown.

Methodology: To understand the conditions under which SPDY helps or hurts, we build a predictive model based on decision tree analysis. In the analysis, each configuration is a combination of values for all factors listed in Table 4.1. For each configuration, we add an additional variable s, which is the PLT of SPDY divided by that of HTTP. We run the decision tree to predict the configurations under which SPDY outperforms HTTP ($s < 0.9$) and under which HTTP outperforms SPDY ($s > 1.1$). The decision tree analysis generates the likelihood that a configuration works

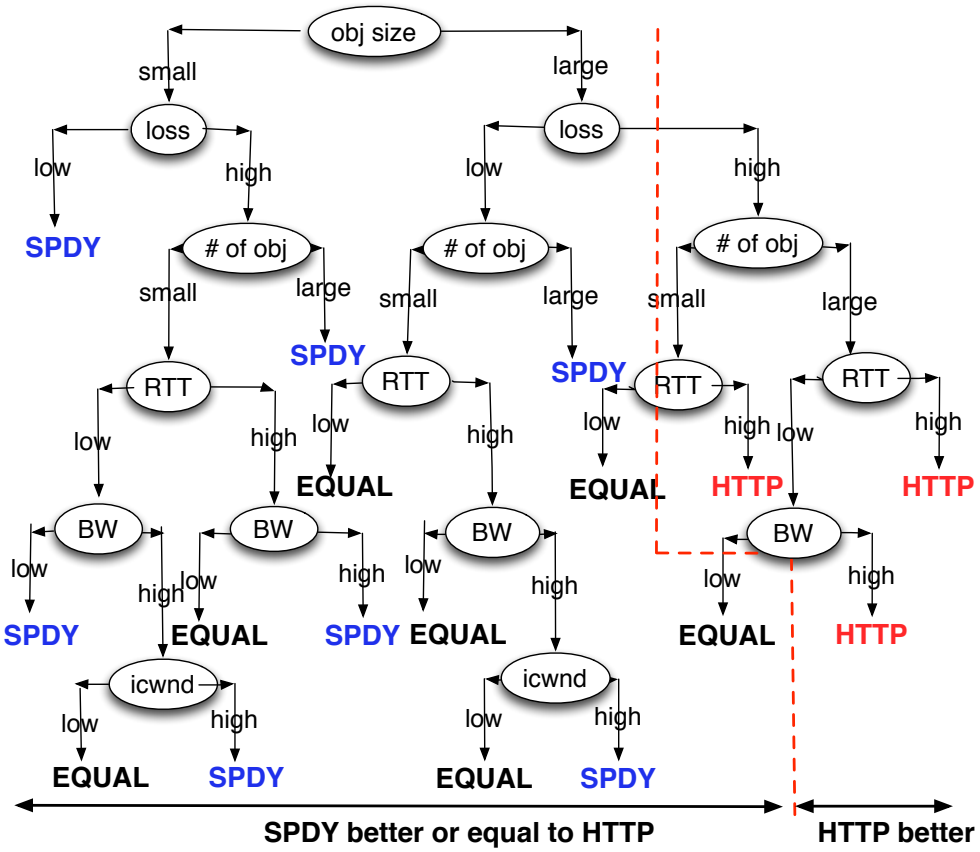


Figure 4.3: The decision tree that tells when SPDY or HTTP helps. A leaf pointing to SPDY (HTTP) means SPDY (HTTP) helps; a leaf pointing to EQUAL means SPDY and HTTP are comparable. Table 4.1 shows how we define a factor being high or low.

better under SPDY (or HTTP). If this likelihood is over 0.75, we mark the branch as SPDY (or HTTP); otherwise, we say that SPDY and HTTP perform equally.

We obtain the decision tree in Figure 4.3 as follows. First, we produce a decision tree based on all the factors. To populate the branches, we also generate supplemental decision trees based on subsets of factors. Each supplemental decision tree has a prediction accuracy of 84% or higher. Last, we merge the branches from supplemental decision trees into the original decision tree.

Results: The decision tree shows that SPDY hurts when packet loss is high. However, SPDY helps under a number of conditions, for example, when there are:

- Many small objects, or small objects under low loss.
- Many large objects under low loss.
- Few objects under good network conditions and a large TCP initial window.

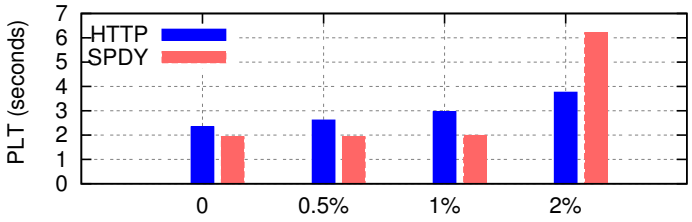
The decision tree also depicts the relative importance of contributing factors. Intuitively, factors close to the root of the decision tree affect SPDY performance more than those near the leaves. This is because the decision tree places the important factors near the root to reduce the number of branches. We find that object size and loss rate are the most important factors in predicting SPDY performance. However, RTT, bandwidth, and TCP initial window play a less important role.

How much SPDY helps or hurts: We present three trending graphs in Figure 4.4. Figure 4.4(a) shows that HTTP outperforms SPDY by half when loss rate increases to 2%, Figure 4.4(b) shows the trend that SPDY performs better as the number of objects increases, and Figure 4.4(c) shows the trend that SPDY performs worse as the object size increases. We publish the results, trends, and network traces at `http://wprof.cs.washington.edu/spdy/`.

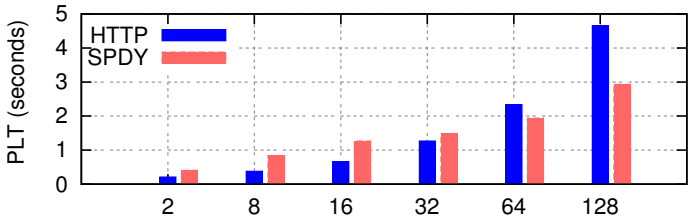
(ii) Why does SPDY help or hurt

While the decision tree informs the conditions under which SPDY helps or hurts, it does not explain why. To this end, we analyze the network traces we collected to explain SPDY performance. We discuss below our findings.

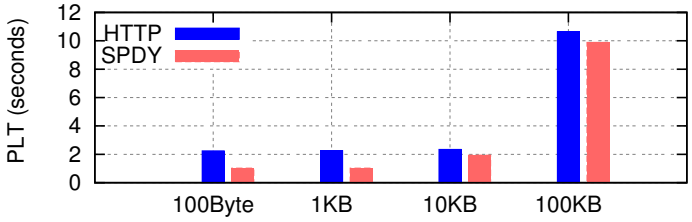
SPDY helps on small objects. Our traces suggest that TCP implements congestion control by counting outstanding packets not bytes. Thus, sending a few small objects with HTTP will



(a) Packet loss rate



(b) Object number



(c) Object size

Figure 4.4: Performance trends for three factors with a default setting: rtt=200ms, bw=10Mbps, loss=0, iw=10, obj_size=10K, obj_number=64.

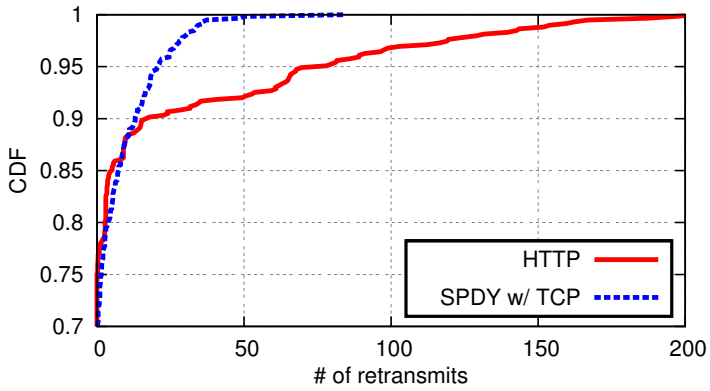
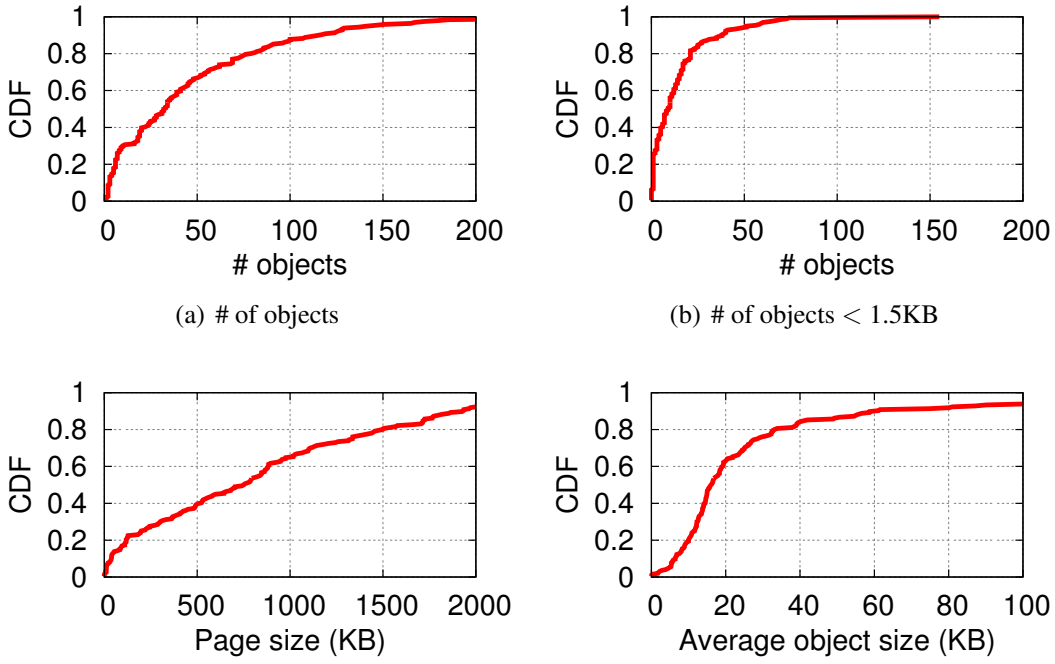


Figure 4.5: SPDY reduces the number of retransmissions.

promptly use up the congestion window, though outstanding bytes are far below the window limit. In contrast, SPDY batches small objects and thus eliminates this problem. This explains why the flag page [36], which mod_spdy advertised, benefits from SPDY.

SPDY benefits from having a single connection. We find several reasons as to why SPDY benefits from a single TCP connection. First, a single connection results in fewer retransmissions. Figure 4.5 shows the retransmissions in SPDY and HTTP across all configurations except those with zero injected loss. SPDY helps because packet loss occurs more often when concurrent TCP connections are competing with each other. There are additional explanations for why SPDY benefits from using a single connection. In our previous study [66], our experiments showed that SPDY significantly reduced the contribution of the TCP connection setup time to the *critical path* of a page download. Further, our experiments in §4.4 will show that a single pipe reduces the amount of time the pipe is idle due to delayed client requests.

SPDY degrades under high loss due to the use of a single pipe. We discussed above that a single TCP connection helps under several conditions. However, a single connection hurts under high packet loss because it aggressively reduces the congestion window compared to HTTP which reduces the congestion window on only one of its parallel connections.



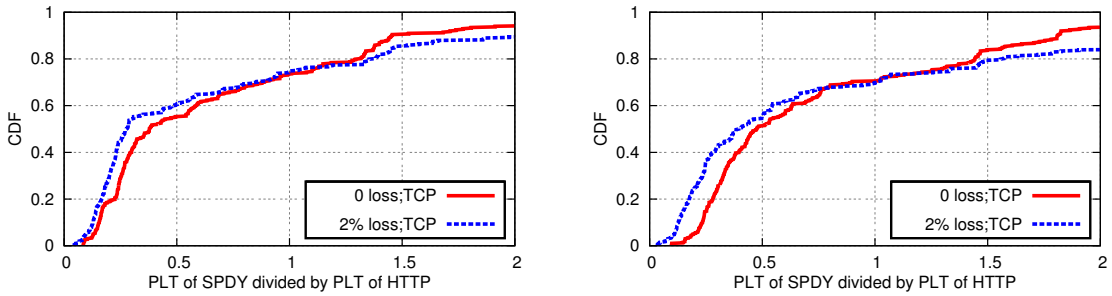
(a) # of objects

(b) # of objects $<$ 1.5KB

(c) Page size (KB)

(d) Mean object size (KB)

Figure 4.6: Characteristics of top 200 Alexa Web pages.



(a) rtt=20ms, bw=10Mbps (b) rtt=200ms, bw=10Mbps

Figure 4.7: SPDY performance across 200 pages with object sizes and numbers of objects drawn from real pages. SPDY helps more under a 1Mbps bandwidth.

4.3.3 Experimenting with real pages

In this section, we study the effects of varying object sizes and number of objects based on the distributions observed in real Web pages. We continue to vary other factors such as network conditions and TCP settings based on the parameter space described in Table 4.1. Due to space limit, we only show results under a 10Mbps bandwidth.

First, we examine the page characteristics of real pages because they can explain why SPDY helps or hurts when we relate them to the decision tree. Figure 4.6 shows the characteristics of the top 200 Alexa Web pages [3]. The median number of objects is 30 and the median page size is 750KB. We find high variability in the size of objects within a page. The standard deviation of the object size within a page is 31KB (median), even more than the average object size 17KB (median).

Figure 4.7 shows PLT of SPDY divided by that of HTTP across the 200 Web pages. It suggests that SPDY helps on 70% of the pages consistently across network conditions. Interestingly, SPDY shows a 2x speedup over half of the pages, likely due to the following reasons. First, SPDY almost eliminates retransmissions (as indicated in Figure 4.8). Compared to a similar analysis for artificial pages (see Figure 4.5), SPDY's retransmission rate is even lower. Second, we find in Figure 4.6(b) that 80% of the pages have small objects, and that half of the pages have more than

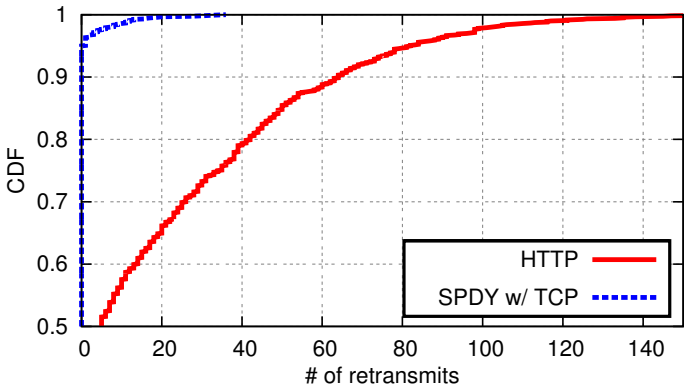


Figure 4.8: SPDY helps reduce retransmissions.

ten small objects. Since SPDY helps with small objects (based on the decision tree analysis), it is not surprising that SPDY has lower PLT for this set of experiments. In addition, we hypothesize that SPDY could help with stragglers since it multiplexes all objects on to a single connection and thus reduces the dynamics of congestion windows. To check this hypothesis, we ran a set of experiments with overall page size and the number of objects drawn from the real pages, but with equal object sizes embedded inside the pages. When we perform this experiment, HTTP's performance improves only marginally indicating that there is very little straggler effect.

4.3.4 TCP modifications

Previously, we found that SPDY hurts mainly under high packet loss because a single TCP connection reduces the congestion window more aggressively than HTTP's parallel connections. Here, we demonstrate that the negative impact can be mitigated by simple TCP modifications.

Our modification (a.k.a., TCP+) mimics behaviors of concurrent connections with a single connection. Let the number of parallel TCP connections be n. First, we propose to multiply the initial window by n to reduce the effect of slow start. Second, we suggest scaling the receive window by n to ensure that the SPDY connection has the same amount of receive buffer as HTTP's parallel connections. Third, when packet loss occurs, the congestion window (*cwnd*) backs off with a rate $\beta' = 1-(1-\beta)/n$ where β is the original backoff rate. In practice, the number of concurrent

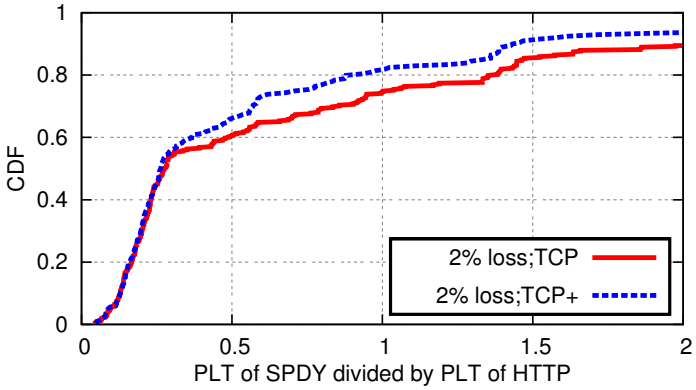


Figure 4.9: TCP+ helps SPDY across the 200 pages. RTT=20ms, BW=10Mbps. Results on other network settings are similar.

connections changes over time. Because we are unable to pass this value to the Linux kernel in real time, we assume that HTTP uses six connections and set $n = 6$. We use six here because it is found optimal and used by major browsers [50].

We perform the same set of SPDY experiments with both synthetic and real pages using TCP+. Figure 4.9 shows that SPDY performs better with TCP+, and the decision tree analysis for TCP+ suggests that loss rate is no longer a key factor that determines SPDY performance.

To evaluate the potential side effects of TCP+, we look at the number of retransmissions produced by TCP+. Figure 4.10 shows that SPDY still produces much fewer retransmissions with TCP+ than with HTTP, meaning that TCP+ does not abuse the congestion window under the conditions that we experimented with. Here, we aim to demonstrate that SPDY's negative impact under high random loss can be mitigated by tuning the congestion window. Because the loss patterns in real networks are likely more complex, a solution for real networks requires further consideration and extensive evaluations and is out of the scope of this paper.

4.4 Web pages and SPDY

This section examines how SPDY performs for real Web pages. Real page loads incur dependencies and computation that may affect SPDY's performance. To incorporate dependencies and

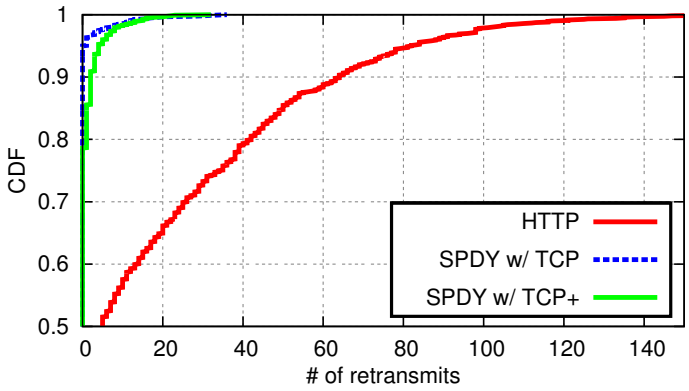


Figure 4.10: With TCP+, SPDY still produces few retransmissions.

computation while controlling variability, we develop a page load emulator `Epload` that hides the complexity and variations in browser computation while performing authentic network requests (§4.4.1). We use `Epload` to identify the effect of page load dependencies and computation on SPDY's performance (§4.4.2). We further study SPDY's potential by examining prioritization and server push (§4.4.3).

4.4.1 Epload: emulating page loads

Web objects in a page are usually not loaded at the same time, because loading an object can depend on loading or evaluating other objects. Therefore, not only network conditions, but also page load dependencies and browser computation, affect page load times. To study how much SPDY helps the overall page load time, we need to evaluate SPDY's performance by preserving dependencies and computation of real page loads.

Dependencies and computation are naturally preserved by loading pages in real browsers. However, this procedure incurs high variances in page load times that stem from both network conditions and browser computation. We have conducted controlled experiments to control the variability of network, and here introduce the `Epload` emulator to control the variability of computation.

Design: The key idea of `Epload` is to decouple network operations and computation in page loads. This allows `Epload` to simplify computation while scheduling network requests at the

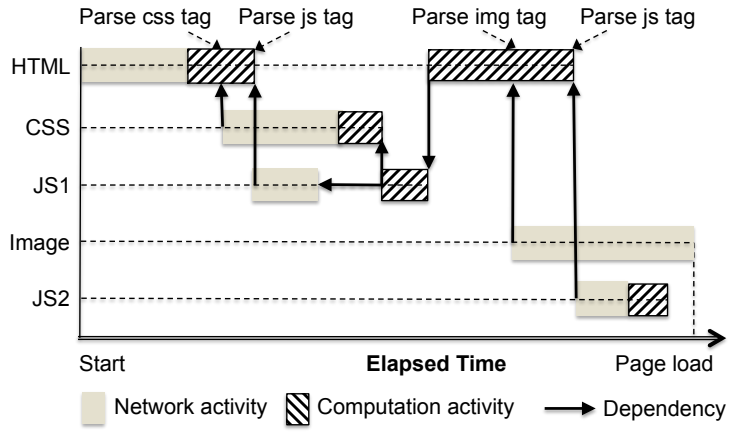


Figure 4.11: A dependency graph obtained from WProf.

appropriate points during the page load.

`Epload` records the process of a page load by capturing the dependency graph using our previous work, WProf [66]. WProf captures the dependency and timing information of a page load. Figure 4.11 shows an example of a dependency graph obtained from WProf where activities depend on each other. This Web page embeds a CSS, a JavaScript, an image, and another JavaScript. A bar represents an activity (i.e., loading objects, evaluating CSS and JavaScript, parsing HTML) while an arrow represents that one activity depends on another. For example, evaluating JS1 depends on both loading JS1 and evaluating CSS. Therefore, evaluating JS1 can only start after the other two activities complete. There are other dependencies such as layout and painting. Because they do not occur deterministically and significantly, we exclude them here.

Using the recorded dependency graph, `Epload` replays the page load process as follows. First, `Epload` starts the activity that loads the root HTML. When the activity is finished, `Epload` checks whether it should trigger a dependent activity based on whether all activities that the dependent activity depends on are finished. For example in Figure 4.11, the dependent activity is parsing the HTML, and it should be triggered. Next, it starts the activity that parses the HTML. Instead of performing HTML parsing, it waits for the same amount of time that parsing takes (based on the recorded information) and checks dependent activities upon completion. This proceeds until

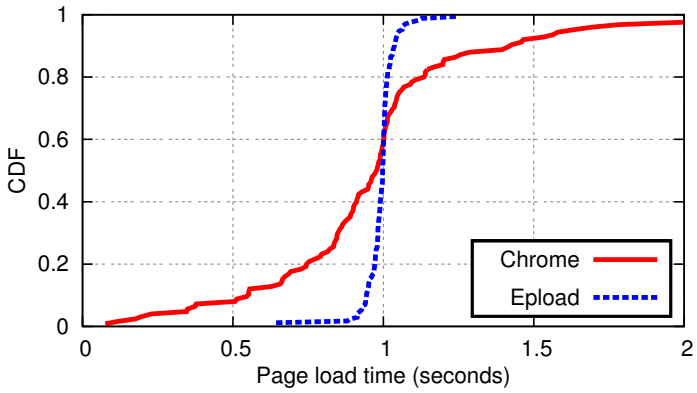


Figure 4.12: Page loads using Chrome v.s. Epload.

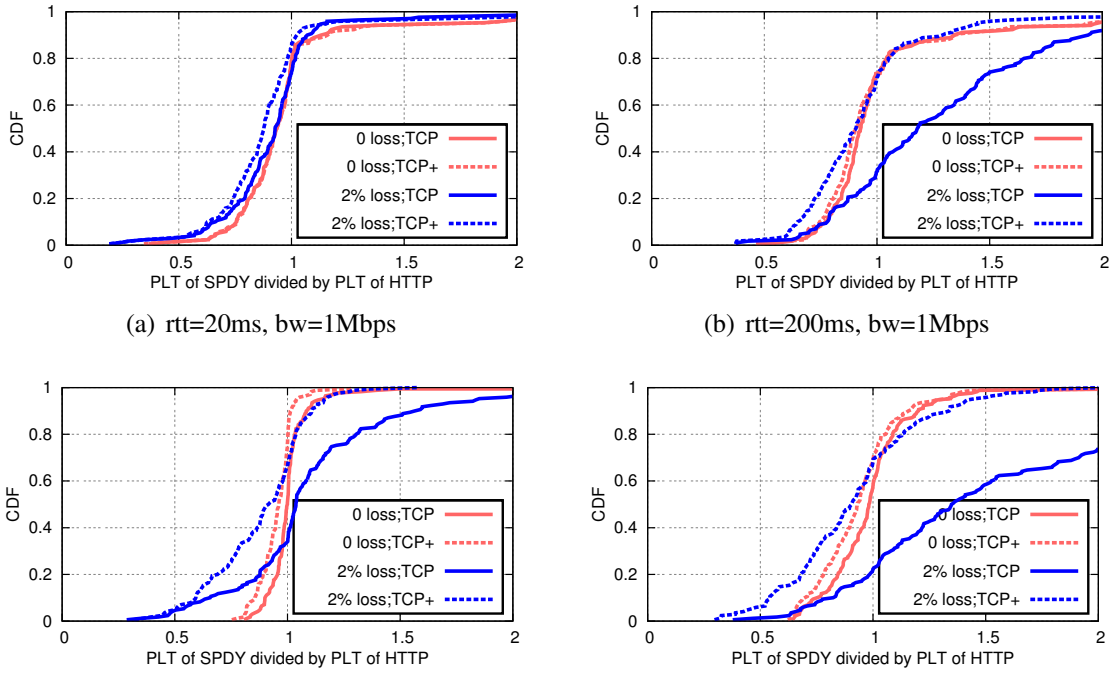
all activities are finished. The actual replay process is more complex because a dependent activity can start before an activity is fully completed. For example, parsing an HTML starts after the first chunk of the HTTP response is received; and loading the CSS starts after the first chunk of HTML is fully parsed. `Epload` models all of these aspects of a page load.

Implementation: `Epload` recorder is implemented based on WProf to generate a dependency graph that specifies activities and their dependencies. `Epload` records the computational delays while performing the page load in the browser, whereas the network delays are realized independently for each replay run. We implement `Epload` replayer using node.js. The output from `Epload` replayer is a series of throttled HTTP or SPDY requests to perform a page load. The `Epload` code is available at `http://wprof.cs.washington.edu/spdy/`.

Evaluation: We validate that `Epload` controls the variability of computation. We compare the differences of two runs across 200 pages loaded by `Epload` and by Chrome. The network is tuned to a 20ms RTT, a 10Mbps bandwidth, and zero loss. Figure 4.12 shows that `Epload` produces at most 5% differences for over 80% of pages which is a 90% reduction compared to Chrome.

4.4.2 Effects of dependencies and computation

We use `Epload` to measure the impact of dependencies and computation. We set up experiments as follows. The `Epload` recorder uses a WProf-instrumented Chrome to obtain the dependency



(a) rtt=20ms, bw=1Mbps

(b) rtt=200ms, bw=1Mbps

(c) rtt=20ms, bw=10Mbps

(d) rtt=200ms, bw=10Mbps

Figure 4.13: SPDY performance using emulated page loads. Compared to Figure 4.7, it suggests that dependencies and computation reduce the impact of SPDY and that RTT and bandwidth become more important.

graphs of the top 200 Alexa Web pages [3]. `Epload` runs on a Mac with 2GHz dual core CPU and 4GB memory. We vary other factors based on the parameter space described in Table 4.1. Here, we only show figures under a 10Mbps bandwidth.

Figure 4.13 shows the performance of SPDY versus HTTP after incorporating dependencies and computation. Compared to Figure 4.7, dependencies and computation largely reduce the amount that SPDY helps or hurts. We make the following observations along with supporting evidence. First, computation and dependencies increase PLTs of both HTTP and SPDY, reducing the network load. Second, SPDY reduces the amount of time a connection is idle, lowering the possibility of slow start (see Figure 4.14). Third, dependencies help HTTP by making traffic less bursty, resulting in fewer retransmissions (see Figure 4.15). Fourth, having fewer outstanding objects diminishes SPDY's gains, because SPDY helps more when there are a large number of outstanding objects (as suggested by the decision tree in Figure 4.3). Here, we see that dependencies and computation reduce and can easily nullify the benefits of SPDY, implying that speeding up computation or breaking dependencies might be necessary to improve the PLT using SPDY.

Interestingly, we find that RTT and bandwidth now play a more important role in the performance of SPDY. For example, Figure 4.13 shows that SPDY helps up to 80% of the pages under low bandwidths, but only 55% of the pages under high bandwidths. This is because RTT and bandwidth determine the amount of time page loads spend in network relative to computation, and further the amount of impact that computation has on SPDY. This explains why SPDY provides minimal improvements under good network conditions (see Figure 4.13(c)).

To identify the impact of computation, we scale the time spent in each computation activity by factors of 0, 0.5, and 2. Figure 4.16 shows the performance of SPDY versus HTTP, both with scaled computation and under high bandwidths, suggesting that speeding up computation increases the impact of SPDY. Surprisingly, speeding up computation to the extreme is sometimes no better than a x2 speedup. This is because computation delays the requesting of dependent objects which allows for previously requested objects to be loaded faster, and therefore possibly lowers the PLT.

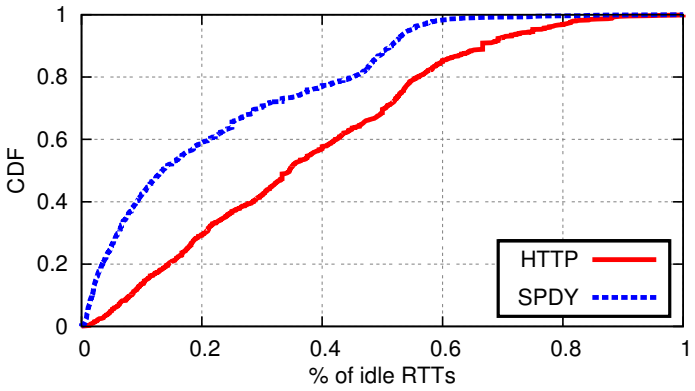


Figure 4.14: Fractions of RTTs when a TCP connection is idle. Experimented under 2% loss rate.

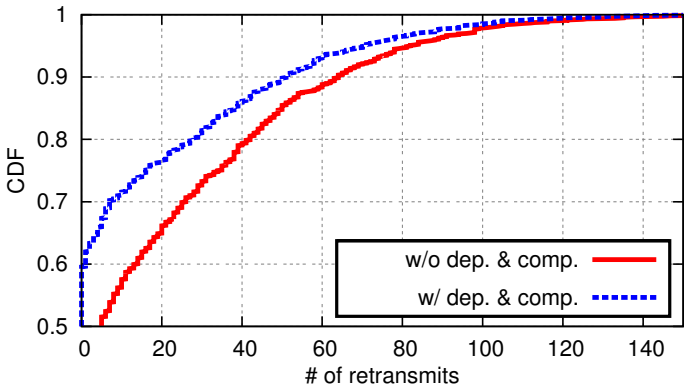


Figure 4.15: SPDY helps reduce retransmissions.

4.4.3 Advancing SPDY

SPDY provides two mechanisms, i) prioritization and ii) server push, to mitigate the negative effects of dependencies and computation of real page loads. However, little is known about how to better use the mechanisms. In this section, we explore advanced policies to speed up page loads using these mechanisms.

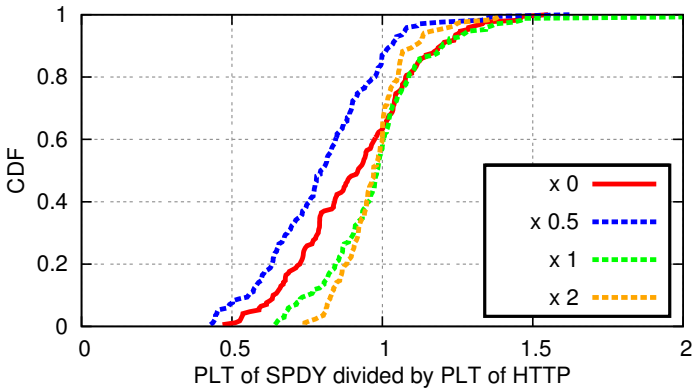


Figure 4.16: Results by varying computation when bw=10Mbps, rtt=200ms.

(i) Basis of advancing

To better schedule objects, both prioritization and server push provide mechanisms to specify the importance for each object. Thus, the key issue is to identify the importance of objects in an automatic manner. To highlight the benefits, we leverage the dependency information obtained from a previous load of the same page. This information gives us ground truth as to which objects are critical for reducing PLT. For example, in Figure 4.11, all the activities depend on loading the HTML, making HTML the most important object; but no activity depends on loading the image, suggesting that the image is not an important object.

To quantify the importance of an object, we first look at the time required to finish the page load starting from the load of this object. We denote this as time to finish (TTF). In Figure 4.11, TTF of the image is simply the time to load the image alone, while TTF of JS2 is the time to both load and evaluate it. Because TTF of the image is longer than TTF of JS2, this image is more important than JS2. Unfortunately in practice, it is not clear as to how long it would take to load an object, before we make the decision to prioritize or push it.

Therefore, we simplify the definition of importance. First, we convert the activity-based dependency graph to an object-based graph by eliminating computation while preserving dependencies (Figure 4.17). Second, we calculate the longest path from each object to the leaf objects; this pro-

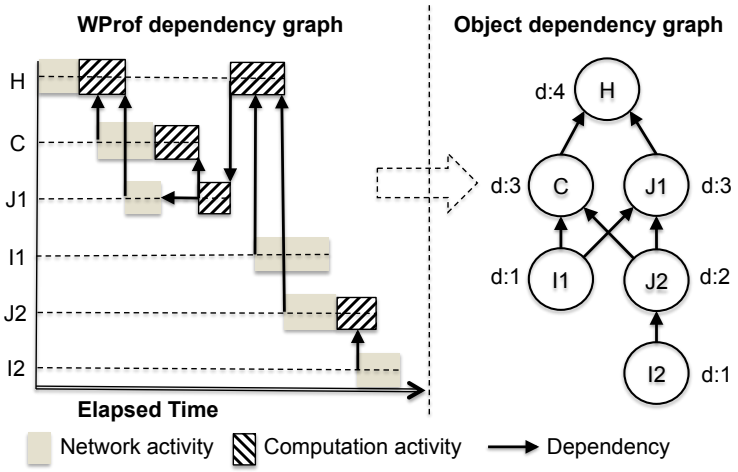


Figure 4.17: Converting WProf dependency graph to an object-based graph. Calculating a depth to each object in the object-based graph.

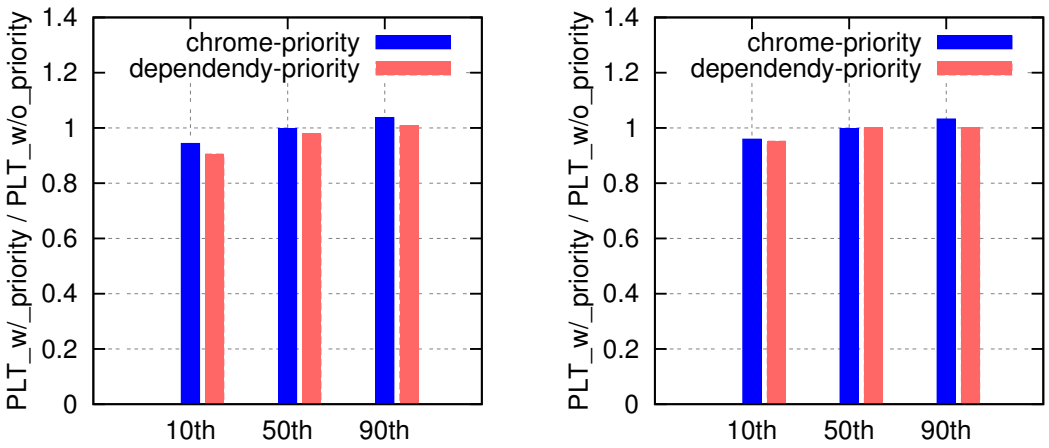
cess is equivalent to calculating node depths of a directed acyclic graph. Figure 4.17 (right) shows an example of assigned depths. Note that the depth here equals TTF if we ignore computation and suppose that the load of each object takes the same amount of time.

We use this depth information to prioritize and push objects. This implies that the browser or the server should know this beforehand. We provide a tool to let Web developers measure the depth information for objects transported by their pages.

(ii) Prioritization

SPDY/3 allows eight priority levels for clients to use when requesting objects. SPDY best practices website [53] recommends prioritizing HTML over CSS/JavaScript and CSS/JS over the rest (`chrome-priority`). Our priority levels are obtained by linearly mapping the depth information computed above (`dependency-priority`).

We compare the two prioritization policies to baseline SPDY in Figure 4.18. Interestingly, we find that there is almost no benefit by using `chrome-priority` while `dependency-policy` marginally helps under a 20ms RTT. The impact of *explicit* prioritization is minimal because the dependency graph has already *implicitly* prioritized objects. Implicit prioritization results from



(a) rtt=20ms, bw=10Mbps

(b) rtt=200ms, bw=10Mbps

Figure 4.18: Results of priority (zero packet loss) when bw=10Mbps. bw=1Mbps results are similar to (b).

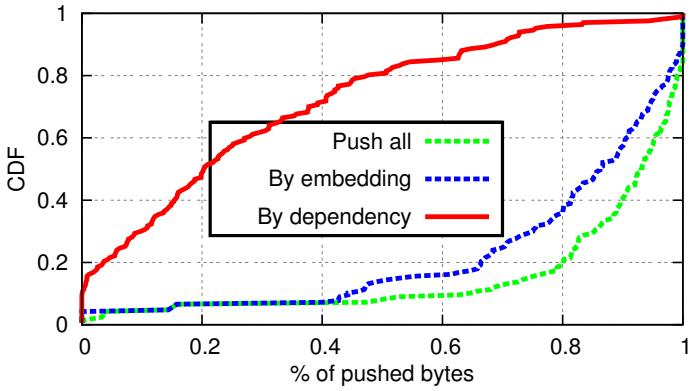
browser policies, independent of Web pages themselves. For example in Figure 4.11, all other objects cannot be loaded before HTML; Image and JS2 cannot be loaded before CSS and JS1. As dependencies limit the impact of SPDY, prioritization cannot break dependencies, and thus is unlikely to improve SPDY's PLT.

(iii) Server push

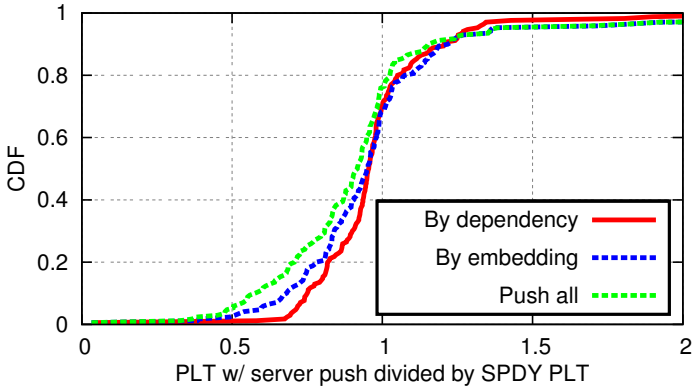
SPDY allows servers to push objects to save round trips. However, server push is non-trivial because there is a tension between making page loads faster and wasting bandwidth. Particularly, one should not overuse server push if pushed objects are already cached. Thus, the key goal is to speed up page loads while keeping the cost low.

We find no standard or best practices guidance from Google on how to do server push. Mod_spdy can be configured to push up to an *embedding level*, which is defined as follows: the root HTML page is at embedding level 0; objects at embedding level i are those whose URLs are embedded in objects at embedding level $i-1$. An alternative policy is to push based on the depth information.

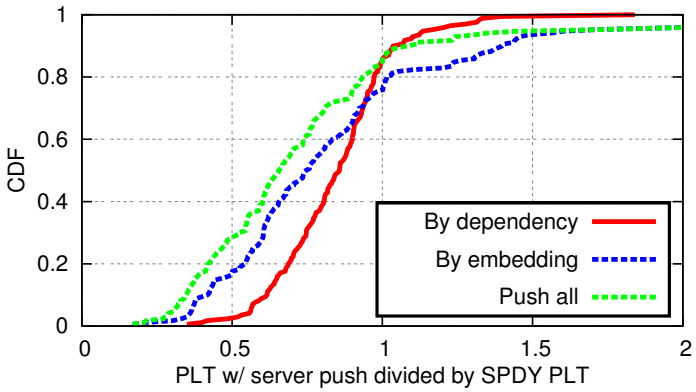
Figure 4.19 shows server push performance (i.e., push all objects, one embedding level, and



(a) Pushed bytes

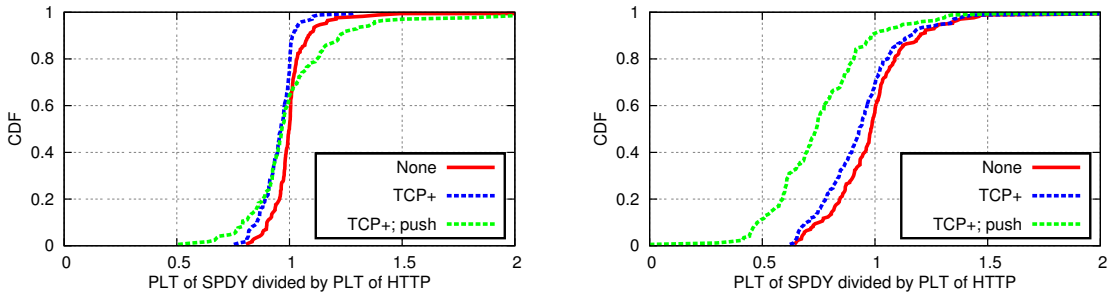


(b) rtt=20ms, bw=10Mbps



(c) rtt=200ms, bw=10Mbps

Figure 4.19: Results of server push when bw=10Mbps.



(a) rtt=20ms, bw=10Mbps (b) rtt=200ms, bw=10Mbps

Figure 4.20: Put all together when bw=10Mbps.

one dependency level) compared to baseline SPDY. We find that server push helps, especially under high RTT. We also find that pushing by dependency incurs comparable speedups to pushing by embedding, while benefiting from a 80% reduction in pushed bytes (Figure 4.19(a)). Note that server push does not always help because pushed objects share bandwidth with more important objects. In contrast to prioritization, server push can help because it breaks dependencies which limits the performance gains of SPDY.

4.4.4 Putting it all together

We now pool together the various enhancements (i.e., TCP+ and server push by one dependency level). Figure 4.20 shows that this improves SPDY by 30% under high RTTs. But this improvement largely diminishes under low RTTs where computation dominates page load times.

4.5 Discussions

SPDY in the wild: To evaluate SPDY in the wild, we place clients at Virginia (US-East), North California (US-West), and Ireland (Europe) using Amazon EC2 micro-instances. We add explanatory power by periodically probing network parameters between clients and the server, and find that RTTs are consistent: 22ms (US-East), 71ms (US-West), and 168ms (Europe). For all vantage points, bandwidths are high (10Mbps to 143Mbps) and loss rates are extremely low. These network parameters well explain our SPDY evaluations in the wild (not shown due to space limit) that

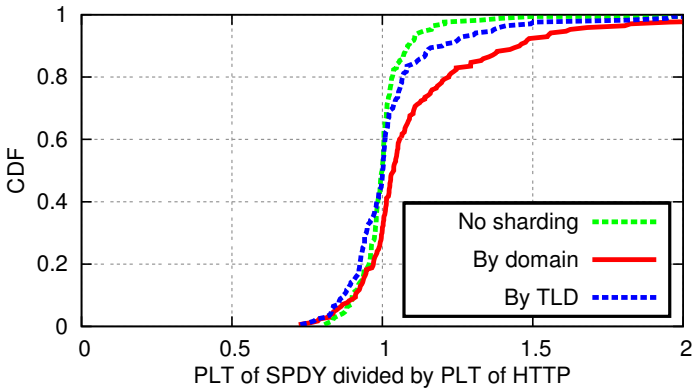


Figure 4.21: Results of domain shading when bw=10Mbps and rtt=20ms.

are similar to synthetic ones under high bandwidths and low loss rates. The evaluations here are preliminary and covering a complete set of scenarios would be future work.

Domain sharding: As suggested by SPDY best practices [53], we used a single connection to fetch all the objects of a page to eliminate the negative impact of domain sharing. In practice, migrating objects to one domain suffers from deployment issues given popular uses of third parties (e.g., CDNs, Ads, and Analytics). To this end, we evaluate situations when objects are distributed to multiple servers that cooperatively use SPDY. We distribute objects by full domain to represent the state-of-the-art of domain sharding. We also distribute objects by top-level domain (TLD). This demonstrates the situation when websites have eliminated domain sharding but still use third-party services. Figure 4.21 compares SPDY performance under these object distributions. We find that domain sharding hurts as expected but hosting objects by TLD is comparable to using one connection, suggesting that SPDY's performance does not degrade much when some portions of the page are provided by third-party services.

SSL: SSL adds overhead to page loads which can degrade the impact of SPDY, but it keeps the handshake overhead low by using a single connection. We conduct our experiments using SSL and find that the overhead of SSL is too small to affect SPDY's performance.

Mobile: We perform a small set of SPDY measurements under mobile environments. We assume

large RTTs, low bandwidths, high losses, and large computational delays, as suggested by related literature [6, 68]. Results with simulated slow networks suggest that SPDY helps more but also hurts more. It also shows that prioritization and server push by dependency help less (not shown due to space limit). However, large computational delays on mobile devices reduce the benefits provided by SPDY. This means that the benefits of SPDY under mobile scenarios depends on the relative changes in performance of the network and computation. Further studies on real mobile devices and networks would advance the understanding in this space.

Limitations: Our work does not consider a number of aspects. First, we did not evaluate the effects of header compression because it is not expected to provide significant benefits. Second, we did not evaluate dynamic pages which take more time in server processing. Similar to browser computation, server processing will likely reduce the impact of SPDY. Last, we are unable to evaluate SPDY under production servers where network is heavily used.

4.6 Summary

Our experiments and prior work show that SPDY can either help or sometimes hurt the load times of real Web pages by browsers compared to using HTTP. To learn which factors lead to performance improvements, we start with simple, synthetic page loads and progressively add key features of the real page load process. We find that most of the performance impact of SPDY comes from its use of a single TCP connection: when there is little network loss a single connection tends to perform well, but when there is high loss a set of connections tend to perform better. However, the benefits from a single TCP connection can be easily overwhelmed by dependencies in real Web pages and browser computation. We conclude that further benefits in PLT will require changes to restructure the page load process, such as the server push feature of SPDY, as well as careful configuration at the TCP level to ensure good network performance.

Chapter 5

SplitBrowser

Previous two chapters together suggest that the complexities of page loads significantly slow down PLT. The complexities not only stem from increasingly complex JavaScript libraries that require more time for evaluation, but also come from the inefficiencies as a result of inter-dependencies between page load activities. Reducing PLT is hard given the nature of these inefficiencies. It has been widely believed that the inefficiencies should be transparently corrected by proper techniques without human intervention. Many techniques focus on improving the network transfer times. Other techniques lower computation costs by either exploiting parallelism [33, 13] or adding architecture support [76, 15]. While being effective in speeding up the individual activities corresponding to a page load, these techniques have limited impact in reducing overall PLT, because they still communicate redundant code, stall in the presence of conflicting operations, and are constrained by the limited parallelism in the page load process.

In this chapter, we advocate the approach that pre-processes Web pages on a proxy server and migrate carefully crafted state to the client so as to simplify the client-side page load process. Our approach directly tackles the above inefficiencies. When migrating state to the client, the proxy server prioritizes state needed for the initial display over state that will be used later, so as to convey information for display as fast as possible. After all the state is fully migrated, the user can interact with the page normally as if the page were loaded directly without using a proxy server. The pre-processing is expected to be fast since it exploits greater compute power at the proxy server and since all of the resources that would normally result in blocking transfers are locally available.

Note that Opera mini [40] and Amazon Silk [4] also embrace a proxy approach but differ intrinsically. Their client-side browsers only handle display, and thus JavaScript evaluation is handled by the proxy server. This process depends on the network which is both slow and unreliable [49], and requires the proxy server to be placed near users. SplitBrowser has a fully functioning client-side browser and requires the proxy server to be placed near Web servers for the most performance gains.

SplitBrowser also ensures that the Web page functionality in terms of user interactivity is preserved and that the delivery process is compatible with latency-reduction techniques such as caching and CDNs and security features such as the enforcement of same-origin policies. The resulting system is thus both efficient and practical. Our evaluations on the top 100 Alexa Web pages show that SplitBrowser reduces PLT significantly by more than half on a variety of settings (e.g., mobile devices, varying RTT, bandwidth, CPU, and memory). The amount of resources is decreased while the total amount of traffic is increased moderately by 1%.

5.1 An analysis of page load inefficiencies

This section reviews the background on the Web page load process (§5.1.1), and identifies three inefficiencies in the process and quantifies them using a measurement study (§5.1.2).

5.1.1 Background: Web page loads

Web page compositions. A Web page comprises of several Web objects that can be HTML, JavaScript, CSS, images, and other media such as videos and Flash. HTML is the language (also the root object) that describes a Web page; it uses a markup to define a set of tree-structured elements such as headings, paragraphs, tables, and inputs. Cascading style sheets (CSS) are used for specifying the presentational attributes such as colors and fonts of the HTML elements and is expressed as a set of rules. Processing the CSS involves identifying the HTML elements that match the given rules (referred to as CSS selector matching) and adding the specified styles to matched elements. JavaScript is a piece of software that adds dynamic contents to Web pages; it can manipulate the HTML elements such as adding new elements, modifying existing elements, or

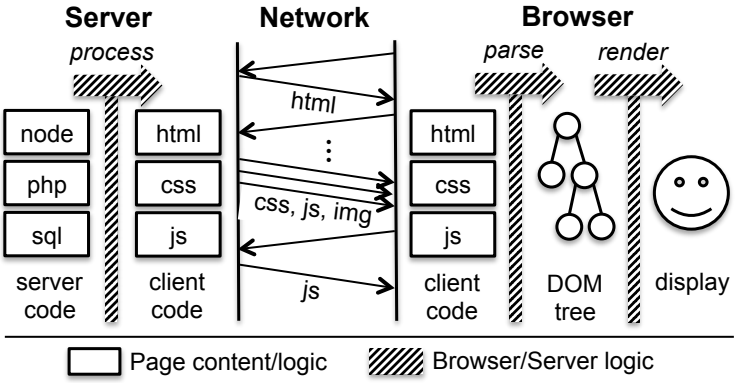


Figure 5.1: An overview of the page load process.

changing elements' styles, and can define and handle events. CSS and JavaScript are embedded in a Web page as HTML elements (i.e., `script`, `style`, and `link`) and can be either a standalone Web object or inline HTML.

Web page load process. Figure 5.1 shows an overview of the Web page load process. First, when a user inputs or clicks a URL, the browser initiates an HTTP request to that URL. Upon receiving the request, the server either responds with a static HTML file, or runs server-side code (e.g., Node.js or PHP) to generate the HTML content on the fly and sends it to the browser. The browser then starts to parse the HTML content; it downloads embedded files (e.g., CSS and JavaScript) until the page is fully parsed. The result of the parsing process is a document object model (DOM) tree, an in-memory representation of the Web page. The DOM tree provides a common interface for JavaScript to manipulate the page. The browser progressively renders the page during the load process; it converts the DOM tree to a layout tree and further to pixels on the screen.

The browser fires a `load` event when it finishes loading the DOM tree. The `load` event is commonly used for prioritizing Web page contents to improve user experience. For example, websites commonly use the `load` event to defer loading JavaScript that is not used in the initial page display. Such a design makes Web pages more responsive and provides better user-perceived page load times.

Dependencies in Web page loads. Ideally, the browser should fetch Web objects of a page fully

in parallel, but in practice the process is often blocked by dependencies among Web objects.

For example, one type of dependencies stems from accessing shared resources [66], which can be found in popular browsers including Chrome, Firefox, Safari, and IE. Particularly, when the parser encounters a `script` tag, it stops parsing, loads the corresponding JavaScript, evaluates the script (i.e., compilation and execution), and then resumes parsing. As both HTML and JavaScript can modify the DOM, this ensures that the DOM is modified in the order specified in the Web page. When a CSS appears ahead of this JavaScript, evaluating the JavaScript needs to wait until the CSS is loaded and evaluated (CSS evaluation includes parsing CSS rules, matching CSS selectors, and computing element styles). This is because both JavaScript and CSS can modify the elements' styles in the DOM. As a result, HTML parsing is often blocked to ensure the correctness of execution, thus significantly slowing down page loads.

These dependencies not only slow down page loads but also prevent optimizations from being more effective. For example, we find that the SPDY protocol would significantly improve page load times if all the objects in a page are fetched and processed in parallel; but this improvement is largely nullified by the page dependencies in real browser contexts. This is because the optimization technologies often just improve one aspect of page loads (e.g., network utilization) and they are further constrained by dependencies, so the marginal improvements are not significant.

Critical paths of Web page loads. Not all of the object loads on a Web page affect the page load times. The bottlenecks can be identified by performing a *critical path analysis* on the dependency graphs obtained when a page is being loaded. The notion of critical paths enables the comparison between times spent on network and on computation. For example, page loads on a 3GHz CPU machine use 65% of the time on network transfers and 35% on computation [66]. The time spent on the network comes not only from the time to load the first byte, but also from blocking loads of JavaScript or CSS, where the blocking is introduced to ensure semantic correctness. The notion of critical paths is also useful in evaluating optimizations, because techniques that improve activities off the critical path cannot reduce page load times. Often, optimization techniques focus on either computation (e.g., parallel CSS computations) or network transfers (e.g., SPDY). A common result

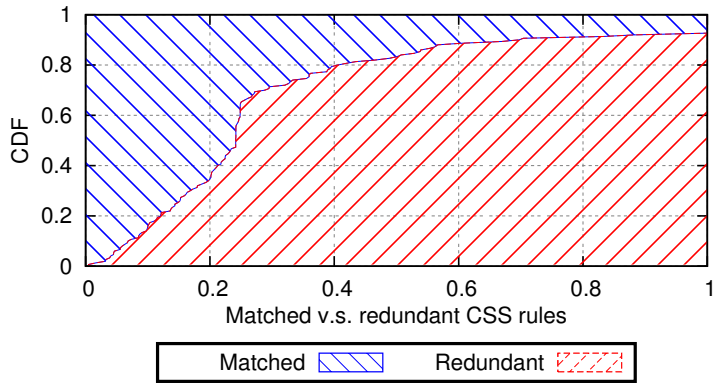


Figure 5.2: Matched CSS rules of top 100 pages in bytes.

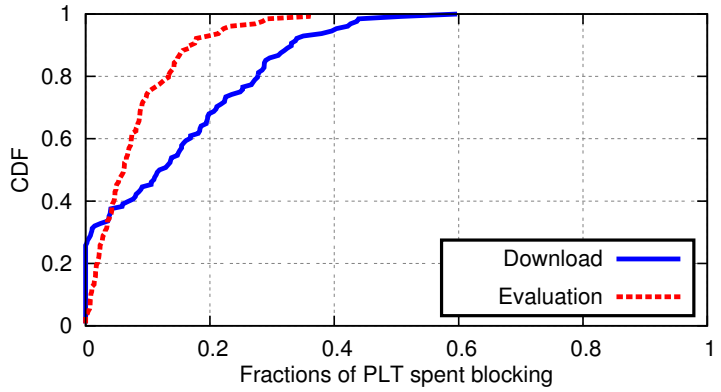


Figure 5.3: The fraction of parsing-blocking download/evaluation times versus the total page load time.

is that the reduction in page load time is not as much as the time reduced on the network (or computation), because reducing the network (computation) time would make computation (network) more on the critical path [66].

5.1.2 Page load inefficiencies

To understand inefficiencies in the Web page load process, we conduct a study on top 100 Alexa [3] pages by using Chrome, which is a highly optimized browser. To provide a controlled network environment, we download all pages to our own server, and use Dummynet [12] to provide a stable network connection of 20ms RTT and 10Mbps bandwidth. Our client is a machine with a 2GHz dual core CPU and 4GB memory. We clear the cache for every page load, and define page load

time (PLT) as the time between when the page is requested and when the `load` event is fired. We have observed the following factors that slow down the page load process.

Unused CSS in page loads. The first observation is that CSS files often contain rules that are either *never* used in a page or at least *not* used during the initial page load. Such CSS rules incur unnecessary network traffic and parsing efforts. We quantify the amount of used versus unused CSS rules in initial page loads (see Figure 5.2). In particular, 75% of CSS rules are unused in the median case. Surprisingly, 80% and 96% of CSS rules are unused for `google.com` and `facebook.com` respectively, while only 10% of CSS rules are unused for `wikipedia.org`. This suggests that CSS is likely to be redundant for interactive pages, because interactive pages tend to load lots of CSS rules for future interactions, at the cost of increased PLT.

Blocking JS/CSS. JavaScript and CSS often block parsing on the critical path. We extend the open source tool WProf [66] to measure the amount of additional round trips and parsing-blocking object downloads and evaluations. We further look at the amount of time used for downloading and evaluating JavaScript or CSS on the critical path. Figure 5.3 shows that most top pages contain parsing-blocking JavaScript or CSS. In particular, 15% of page load time is spent waiting for JavaScript or CSS to be loaded on the critical path; and 5% of page load time is used for evaluating JavaScript and CSS. Compared to the time to first byte which is difficult to reduce, there are significant gains to be had from optimizing the parsing-blocking object downloads and evaluations.

Additional round trips. Web objects are not loaded in a batch, but are often loaded sequentially due to the above reason. The result is that loading a page usually incurs many round trips, since loading an object often triggers a cascade of latencies such as redirections, DNS lookups, TCP connection setups, and SSL handshakes. We find that 80% of pages have sequentially loaded Web objects on the critical path.

5.2 Design

Our design aims to reduce PLTs by restructuring the page load process that removes the inefficiencies measured in §5.1.2. We pre-process Web pages on the proxy server, and migrate carefully

crafted state to the client so as to simplify the client-side page load process.

The key to our design is the state that we capture and migrate. On the one hand, page state needs to be captured at an appropriate processing stage so as to minimize the network and computational costs; on the other hand, the captured state should be comprehensive and ensure that the rendered page on the client displays and functions correctly. The challenges are detailed in §5.2.1. Next, we describe the *load-time state* (§5.2.2) that is captured for fast page loads, and the *post-load state* (§5.2.3) that is captured for interactivity and compatibility. In addition to the state that is migrated from the server to the client, we discuss the state that needs to be migrated from the client to the server (§5.2.4). We then discuss the backward compatibility of SplitBrowser (§5.2.5).

5.2.1 Challenges

We identify three challenges in designing SplitBrowser.

First, precisely identifying resources that are needed during a page load (load-time resources) is nontrivial since load-time resources and post-load resources are largely mingled. For example, some Web pages use a small portion of jQuery [28] to construct HTML elements while leaving a large portion of jQuery unused. Precisely identifying the load-time resources and migrating them to the client in the first place is key to reducing PLT.

Second, we need to ensure that the Web page rendered using SplitBrowser is functionally equivalent to one that is computed solely on the client. As load-time resources (such as JavaScript and CSS) have been evaluated on the server, evaluating them again on the client would be problematic. Similarly, not migrating these resources to the client would also be problematic, because they might be used later in user interactions. Further, the server needs proper client-side state to function properly. For example, some Web pages that adopt a responsive Web design provide layouts specific to browser size; and some other Web pages use information from cookies or HTML5 localStorage to assemble layouts. The server needs information regarding browser size, cookies or localStorage in order to function properly.

Third, completely recording and migrating the Web page state computed by the server is nontrivial. After the initial load process, the state computed by the server is largely dispersed across

various JavaScript code fragments that comprises the Web page. This state needs to be retrieved and then migrated to their equivalent locations on the client in order to ensure that the user has a seamless experience in interacting with the Web page.

In the rest of the section, we discuss how we address these challenges in designing Split-Browser.

5.2.2 Load-time state

The load-time state is designed primarily for display, being captured at a processing stage that minimizes the amount of work required for display. To this end, design decisions to the load-time state focus on how much we can eliminate JavaScrip/CSS evaluations while keeping the state small. As a result, the load-time state contains only HTML elements and their styles, but not JavaScript or post-load CSS. Below, we explain this in greater detail and also describe the state that is migrated to reflect JavaScript/CSS evaluations performed at the server.

(i) Load-time state in JavaScript

JavaScript itself does not directly reflect on display, but the result of JavaScript evaluation can. As JavaScript evaluation is slow and blocks rendering, a design decision is to avoid JavaScript in the load-time state and thus incurs no JavaScript evaluation in the initial page load. Instead the load-time state includes the result of JavaScript evaluation, which is already reflected in the HTML elements. For example, instead of transmitting a piece of D3 JavaScript [17] to construct an SVG graphic on the client, the JavaScript is evaluated at the server to generate the load-time state of HTML elements that represent the SVG. This design minimizes the computation time used in JavaScript evaluation for an initial page load, but at the cost of potentially increased size of migrated state.

(ii) Load-time state in CSS

CSS evaluation is also slow, blocks rendering, and thus should be avoided in the initial page load as much as possible. The result of CSS evaluation is a cumbersome but detailed list of styles for each HTML element. Including the detailed list of styles in the load-time state would fully eliminate

the CSS evaluation but incur a significant amount of time transferring the state.

Here, we seek for an intermediate state in CSS evaluation that incurs little amount of time finishing CSS evaluation while keeping the state small. CSS evaluation is discretized as follow. It goes through a sequential processes of CSS parsing, CSS selector matching, and style computation. The CSS selector matching process matches the selectors of *all* the CSS rules to *each* HTML element, requiring more than a linear amount of time. The style computation process applies matched CSS properties in a proper order to generate a list of styles for rendering.

Our design decision here is to leave style computations on the client by migrating all the inputs to style computations as load-time state. The required inputs are largely determined by the W3C algorithm that specifies the order according to which CSS properties are applied [64]. In addition to matched CSS selectors and properties for a given HTML element, the state also includes the importance (marked as important or not), the origin (from website, device, or user), and the specificity (calculated from CSS selectors) that determines this order. Here, the state is compact compared to the detailed list of styles, and eliminates CSS selector matching on the client.

(iii) Serialization and deserialization

The process to serialize the load-time state on the server is simple. When the page load event or any other defined event is fired, SplitBrowser recursively serializes each HTML element in the DOM (excluding CSS and JavaScript elements), its attributes, and references to matched CSS rules. Then, the details of the matched CSS rules are serialized. Each CSS rule includes a CSS selector, a list of CSS properties, the importance, and the origin. Note that we do not add CSS rules to each matched HTML element, but use references to link HTML elements to their matched CSS rules. This is because a CSS rule is likely to match with a large number of HTML elements. The HTML elements and matched CSS rules together provide complete information for a page to be displayed properly.

Deserializing the load-time state on the client determines the page load time, which is both simple and fast. SplitBrowser linearly scans the load-time state, uses HTML elements and attributes to construct the DOM. Instead of running a full CSS evaluation, SplitBrowser computes styles

from already matched CSS, requiring just a linear amount of time. SplitBrowser does not require any JavaScript evaluations because the state already contains the results of JavaScript evaluation. Compared to the page load process, the deserialization process does not block, does not incur additional network interactions, and avoids parsing of unused CSS or JavaScript, thereby significantly speeding up page loads on the client.

5.2.3 Post-load state

Following the load-time state, the post-load state needs to be processed transparently in the background in order to ensure that: (a) users can further interact with the page as if it wasn't delivered through a SplitBrowser (*interactivity*), and (b) latency-reduction techniques are still viable (*compatibility*). To ensure interactivity, the post-load state should include the portion of JavaScript that is not used in the load-time state, together with unused CSS, because they might be used later in user interactions. To ensure that latency-reduction techniques such as caching and CDNs can be used in SplitBrowser, we need to preserve the integrity of third-party objects. For example, we need to attach an unmodified version of third-party CSS and JavaScript in the post-load state.

The most compact approach would be migrating unmodified JavaScript/CSS snippets, which both ensures the integrity (compatibility) and includes all the information for post-load state (interactivity). Our design here focuses on examining the feasibility of migrating unmodified snippets, and processing unmodified snippets while excluding the effects of load-time state.

(i) Post-load state in CSS

Attaching unmodified CSS snippets (copies of inline CSS and links to external CSS) in the post-load state is both feasible and simple. We can just evaluate all the CSS rules here regardless of whether they had appeared in the load-time state. This is because CSS evaluation is idempotent—evaluating the same CSS rule any number of times would give the same results. In our design, the CSS rules in load-time state will be evaluated twice (one by SplitBrowser on the proxy server, and the other on the client) while post-load CSS is evaluated once.

This design is simple and satisfies the constraints, but at the cost of repeating the evaluation of load-time state. For example, if a snippet of external CSS is already being cached, our design

does not require loading any portion of this snippet from anywhere else. The price to pay is the additional energy consumption and latencies that result from the repeated evaluation of load-time state. But according to the usage of post-load state, latencies in processing post-load state are not a concern.

(ii) Post-load state in JavaScript

Attaching unmodified JavaScript snippets in the post-load state incurs a complex processing procedure, because not all JavaScript evaluation is idempotent. On the one hand, we need to ensure that JavaScript evaluation has equivalent results; on the other hand, we need to completely record all the state of JavaScript. Other approaches such as migrating the heap would incur significantly larger state (10x) and break the integrity of JavaScript objects, and are thus not an option here.

Ensuring that JavaScript evaluation has equivalent results. If we keep unmodified JavaScript in the post-load state, it is hard to ensure that JavaScript evaluation gives equivalent results as if no SplitBrowser were used. This is because the order in which JavaScript appears determines the results of JavaScript evaluation, but unfortunately is not preserved as a result of isolating load-time and post-load state. If we do not keep unmodified JavaScript in the post-load state, the compatibility would be compromised, so does the size of the state.

Our approach uses unmodified JavaScript, together with a small amount of the heap (described as partial heap in the rest of the chapter), to reconstruct the whole heap. The key is to extract as much as we can from the unmodified JavaScript so as to keep the partial heap small. The idea is to treat statements in the unmodified JavaScript differently according to whether evaluating them is idempotent. A JavaScript statement can be a function declaration, a function call, a variable declaration, and so forth. Evaluating function declarations is idempotent, but not necessarily the other statements. As all the statements have been evaluated once on the server, statements except for function declarations cannot be evaluated again on the client. Thus, at the client we only evaluate function declarations in post-load JavaScript, and directly apply the partial heap—the results of JavaScript evaluation that are migrated from the server.

What would be included in the partial heap? The answer to this question depends on the

function declarations extracted from the unmodified JavaScript. However, isolating function declarations from the other JavaScript is nontrivial because they are often largely mixed. Below, we discuss the situations under which function declarations are hard to isolate, with describing the partial heap when necessary.

(i) Recursively embedded instance variables. JavaScript does not distinguish between functions and objects, and thus a function declaration can recursively embed other function declarations and instance variables. Our approach recursively captures all the instance variables as the partial heap even if they are embedded in a function declaration, and then attach them to post-load state. When the client evaluates unmodified JavaScript, it just evaluates function declarations and ignores these instance variables. Then, the client applies the partial heap to restore the instance variables.

Note that our approach is able to distinguish between instance variables with the same name but within different scopes. This is because we work with native browsers and thus the partial heap does not have to comply with the JavaScript syntax.

(ii) Self-invoking functions. A self-invoking function combines a function declaration and a function call in a single statement. For example, `(function(n){alert(n);})(0)` is a self-invoking function that declares the function that pops up an alert and invokes the alert at the same time. Our approach is to split up the single statement into a function declaration and a function call.

(iii) `eval` *and* `document.write`*.* `eval` and `document.write` can convert strings to JavaScript code that embeds function declarations. The use of `eval` and `document.write` is considered as bad practices, not only because they slow down performance but also because they are vulnerable to cross-site scripting attacks. We disable the use of SplitBrowser for Web pages that have invoked `eval` and `document.write` before migrating state to the client. Note that the amount of such Web pages are less than that of all the Web pages that use the two functions.

Recording all the state of JavaScript. Recording all the state is challenging, because some state such as those in function closures and event callbacks are hard to capture.

(i) Instance variables in function closures. It is hard to record the state contained inside a func-

Events	Event state
DOM events	event name, callback and its arguments
`XmlHttpRequest`	internal fields of the object
`setTimeout`	time to fire, callback and its arguments
`setInterval`	time to fire, interval, callback and its arguments

Table 5.1: Summary of events and their states.

tion closure. A function closure is often used for isolating code execution environments (referred to as scopes). Similar to namespaces, function closures provide this isolation; but unlike namespaces, function closures can sometimes lack a reference to itself. To this end, we instrument the JavaScript engine with the ability to refer to function closures and serialize the instance variable for each closure respectively. Unlike other techniques that handle function closures by rewriting JavaScript [31], instrumenting the JavaScript engine allows us to handle function closures efficiently.

(ii) State in event callbacks. Besides function closures, event callbacks are also hard to capture. Here, we consider three kinds of events that can be added in an initial page load. This includes DOM events that can be added by invoking `addEventListener`, Ajax request `XmlHttpRequest`, and timers (i.e., `setTimeout` and `setInterval`). Serializing the event callbacks requires us to capture all the state in the event queue. We summarize the state for each type of event in Table 5.1. Because firing events can modify the DOM, we need to ensure that event callbacks do not modify the DOM while it is being serialized on the server; we use a lock-like mechanism to protect this from happening.

(iii) Serialization and deserialization

Serialization and deserialization of the post-load state happens in the background and is more complex than that of the load-time state.

On the server, SplitBrowser first serializes unmodified CSS or JavaScript snippets if they are inline (their links instead if they are external), ensuring compatibility. Next, SplitBrowser serializes the event callbacks and the partial heap excluding those that can be restored from function declarations in the unmodified JavaScript. The post-load state together with load-time state pro-

vides complete information for a Web page to function correctly. Note that the size of the load-time and post-load state together exceeds that of the original Web page. The extra portions include the matched CSS rules, the partial heap, and event state. Because they are computed from the original Web page and are thus repetitive, they can be compressed.

On the client, SplitBrowser first deserializes and parses unmodified CSS and JavaScript, fetching corresponding objects if they are external. Unlike in a Web page where fetching CSS and JavaScript has to comply with the dependency model [66], here JavaScript and CSS objects can be fetched completely in parallel. After all the objects are fetched, CSS is completely evaluated, and the function declarations in JavaScript are evaluated so as to avoid duplicate evaluations. Then, the partial heap is applied and events start to get fired. At this point, the Web page state on the client is restored as if the entire page load process happened on the client.

5.2.4 Client-side state

Website information stored in browsers. In addition to migrating state from the server to the client, some state stored in browsers needs to be first migrated from the client to the server. While constructing the DOM, the browser uses long-term storage including cookies, HTML5 localStorage, and Web database. Because the server does not keep a copy of this state, lacking client-side state might break the Web page. The simplest approach to handle client-side state is to migrate them from the client to the server. But this has the potential to increase the uplink transfers and thus slow down page loads. To this end, we conduct a measurement study on client-side state and have confirmed in §5.4.4 that the client-side state that needs to be migrated is small.

Other sources of unsynchronization. Besides browser storage, there can be differences in obtaining timestamps (`Date.now`), geolocation, and browser information from the client and the server [10]. The absolute timestamps should be the same on the client and the server as if they are both synchronized to the global clock. However, the time zone could be different and thus needs to be sent to the server. The geolocation can only be obtained by asking users for an explicit consent. Once this happens, we send the geolocation to the server. Browser information includes the window size and user agents. As user agents are always sent to the server, we do not need to

explicitly handle it. We send the window size to the server because it can be used to adjust the size of the layout (e.g., in a responsive design).

Privacy discussions. Note that our approach does *not* require access to additional user information, and thus fully respects privacy. This is because: (i) website information stored in browsers in the form of cookies or localStorage comes from the website itself; (ii) current browsers expose geolocation to websites once receiving consent from users; (iii) websites have already had access to the browser information (using JavaScript). To sum up, the client-side state either comes from the websites or is already exposed to the websites in the absence of SplitBrowser, and thus SplitBrowser respects privacy.

5.2.5 Compatibility

Latency-reduction techniques. Our design is compatible with existing latency-reduction techniques with notable examples of caching and CDNs. Both caching and CDNs use a URL as the key to store a Web object. To preserve the use of caching and CDNs, we need to preserve the integrity of both the Web object itself and its corresponding URL. We leave images and other media unmodified because they do not block HTML parsing, and we make the design decisions to migrate unmodified CSS and JavaScript in the post-load state. All the resources that are typically cached or served from CDNs are kept unmodified, meaning that all the caching and CDN abilities are preserved.

HTTPS. Our design is compatible with HTTPS if it is deployed on a reverse proxy, but breaks HTTPS when deployed as a globally distributed proxy. Similar to Amazon Silk and Opera mini, we would need to trust the proxy. The connection between the Web server and the proxy and the connection between the proxy and the client use two separate HTTPS connections. To handle SSL certificates we need to route the requests to the proxy so that the proxy can fetch Web pages on behalf of the client.

Security techniques backed by same-origin policies. Same-origin policy (SOP) is used to protect third-party scripts from accessing first-party assets such as cookies. Our design is compatible

with SOP when third-party scripts are embedded using an `iframe`, because frames and parent document are isolated from each other. When third-party scripts are embedded using a `script` tag, they are given full permissions to access the first-party assets in which case our design respects SOP.

5.3 Deployment and Implementation

5.3.1 Deployment

SplitBrowser can be deployed either in the reverse proxy (co-located with front-end servers) or as a separate globally distributed proxy service (similar to Opera mini [40] and Amazon Silk [4]).

Conventional wisdom suggests deployments near clients, so as to make use of edge caching and CDNs, and to offer low latencies when JavaScript offloading is needed. To the best of our knowledge, all page rewriting techniques (e.g., Opera Mini [40] and Amazon Silk [4]) are intended to deploy near clients. Unfortunately, such deployment slows down the preload process on the proxy server, because it adds additional round trips to the Web server, which is a key inefficiency especially for parsing-blocking object downloads in current Web pages (§5.1.2).

In our design of SplitBrowser, we find that exploiting caching/CDNs and reducing round trip delays to the origin server are not at odds and that a carefully designed system can achieve both. We only require the resources that are used as part of the initial page load to go through the proxy server, while the resources accessed after the initial load (e.g., images and videos) can still be cached or be fetched from CDNs. Therefore, we consider deployments wherein the proxy server is located near the Web content server and is ideally co-located with the reverse proxy of the Web service so as to reduce the preprocessing time in the proxy server.

5.3.2 Implementation

State format. We represent the migrated state in JSON format, because it is simple and compact. More importantly, the JSON format is suited for representing tree-structured elements such as the DOM. Note that other formats such as XML or HTML are also viable.

Server extension. We implement the server-side browser as a Web server extension. Our pro-

totype is based on Chrome's `content_shell` with most modifications to Blink and some to V8. `content_shell` is a lightweight browser that includes only page-specific features such as HTML5 and GPU acceleration, but not browser-specific features such as extensions, autofill, and spell checking [21]. We chose `content_shell` to implement the server extension since it is light weight.

Our instrumentation is primarily for state serialization, and is minimal before state serialization starts: we turn off downloads of images and other media because they are not part of the migrated state; we also block objects that are hosted on other domains because we mandate downloading all the required CSS and JavaScript to the Web server. While most of the state resides in Blink, some also resides in V8 (e.g., event callbacks and function closures). We created APIs so that the state inside V8 can be accessed from Blink. The server extension can be added to any Web server that allows process invocation.

Client-side browser. The client-side browser is also based on Chrome, and we modify it as little as possible. We implemented a JSON lexer to parse the migrated state, and this lexer is invoked instead of the HTML lexer. After obtaining the HTML elements from the JSON lexer, we perform DOM construction using unmodified Blink. Given that the migrated state contains matched CSS rules, we skip CSS selector matching and directly apply the CSS properties to compute the element styles. We modify V8 a little to selectively evaluate function statements in JavaScript and to apply server-side results of JavaScript evaluations. We modify Blink to create event listeners and timers from our serialized state instead of executing the load-time JavaScript. The client-side browser can opt in to using the SplitBrowser mechanisms using an HTTP header and thus can easily fallback to loading the original pages that SplitBrowser does not support.

Note that we chose to modify the browsers instead of implementing state migration using JavaScript because JavaScript evaluation is time consuming and because it does not provide the appropriate APIs necessary for all of the low-level manipulations. For example, JavaScript does not allow access to the matched CSS rules for an HTML element. By operating inside the browser code base, we have easy in-memory access to all of the desired information, and we also avoid

JavaScript execution at the client prior to the page load event.

5.4 Evaluation

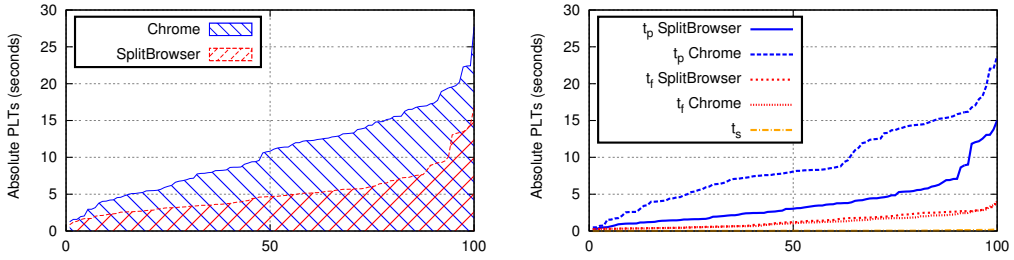
The evaluation aims to demonstrate that: (i) SplitBrowser significantly improves PLT under a variety of scenarios (§5.4.2), (ii) SplitBrowser does not significantly hurt data usage §5.4.3), and (iii) the amount of client-side state that needs to be transferred to the server is small (§5.4.4).

5.4.1 Experimental setup

We conduct the experiments by setting up a client that loads Web pages using our modified Chrome and a server that hosts pages using our server extension. Both the client and the server are connected to a LAN. We detail the experimental setup below.

Our server is a 64-bit machine with 2.4GHz 16 core CPU and 16GB memory, and has a Ubuntu 12.04 installation with the 3.8.0-29 kernel. We download the home pages of the Alexa top 100 websites to our server and use Apache to host them. We download all of the Web objects for a page, ensuring that the page loads by our server extension do not issue external network requests. In practice, only Web objects that are used in initial page loads need to be hosted on the server. For example, synchronous JavaScript needs to be placed on the server, but images and videos can be placed anywhere else. Our experimental setup emulates a deployment setting where SplitBrowser executes on a front-end server of a Web service.

We define page load time (PLT) as the time to display page contents that are rendered before the W3C `load` event [65] is fired. Note that our approach works with any metrics of page load times, though we use the W3C `load` metric to evaluate our prototype. Alternatives to PLT such as the above-the-fold time (AFT) [9] and speed index [59] represent user-perceived page load times, but they require cumbersome video recordings and analysis and are thus out of the scope of this paper. We clear browser cache between any two page loads, and do not consider client-side state that requires a login. We report the median page load times out of five runs for all the experiments.



(a) Overall page load times (b) Breakdowns of page load times

Figure 5.4: Page load times (seconds) on Nexus S with 1GHz Cortex A8 CPU, 512MB RAM, and Android 4.1.2, and with WiFi. SplitBrowser reduces page load times by 60% compared to Chrome in the median case.

5.4.2 Page load times

The page load process on SplitBrowser is simple: (i) the server pre-processes Web pages to obtain the state and migrate it to the client; (ii) the client linearly scans the load-time state and uses HTML elements and their attributes to construct the DOM; (iii) instead of running a full CSS evaluation, SplitBrowser just computes styles from matched CSS rules in load-time state, incurring a linear amount of time. This process continues until the page is loaded. Unlike other browsers, SplitBrowser neither blocks nor incurs additional network interactions, and it avoids evaluating unused CSS or JavaScript. Therefore, we expect SplitBrowser to significantly speed up page loads.

(i) PLT on mobile devices

We use a mobile phone, Nexus S with 1GHz Cortex-A8 CPU, 16GB internal memory (512MB RAM), and Android 4.1.2. The mobile phone is connected to the Internet via WiFi. We install our modified Android Chrome and automate experiments using `adb` shell. We load the Web pages with SplitBrowser and with unmodified Chrome on the mobile phone. Figure 5.4(a) shows that the PLTs with SplitBrowser are significantly reduced compared to those with unmodified Chrome. The reduction is as much as 60% in the median case.

Source of benefits: To identify the source of benefits, we further break down PLTs into time spent by the SplitBrowser server extension t_s, time to fetch the first chunk of the page t_f, and

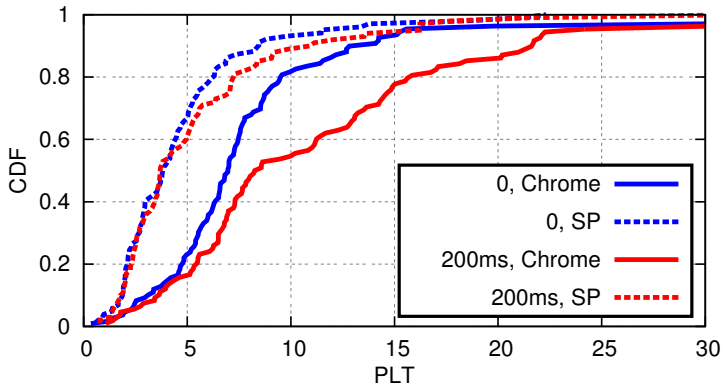


Figure 5.5: Varying RTT with fixed 1GHz CPU and 1GB memory.

time to parse the page (including parsing-blocking network fetch time) t_p. Figure 5.4(b) shows that SplitBrowser's server extension uses little time to pre-process pages (22ms in the median case, and 250 ms in the maximum case). Compared to client-side page loads that take a few seconds, server-side page loads have negligible overheads, due to the benefits accrued from more compute power (especially memory), a lightweight server browser, and mitigated network inefficiencies by deploying the cloud server near the Web server. The benefits together suggest that migrating page load computations to the server is effective. By comparing the client-side parsing times t_p of SplitBrowser and Chrome, we find that the benefits of SplitBrowser stem mainly from client-side parsing. This is because SplitBrowser requires no JavaScript evaluations, eliminates redundant CSS, and increases network utilization by eliminating blocking operations.

(ii) PLT on desktop VM

To demonstrate how much SplitBrowser helps PLTs on a variety of scenarios, we use a desktop VM with Ubuntu 12.04 kernel 3.8.0-29 installed and connected to the network using Ethernet. We use Dummynet [12] to emulate varying bandwidths and RTTs.

Varying RTT: We vary RTT from the minimal of the LAN to 200 milliseconds with fixed 1GHz CPU and 1GB memory, which are representative of current mobile devices. We do not cap the bandwidth. Figure 5.5 shows the cumulative distributions of PLTs of the 100 Web pages. The increased RTT affects much of PLT with Chrome but affects little of PLT with SplitBrowser, mean-

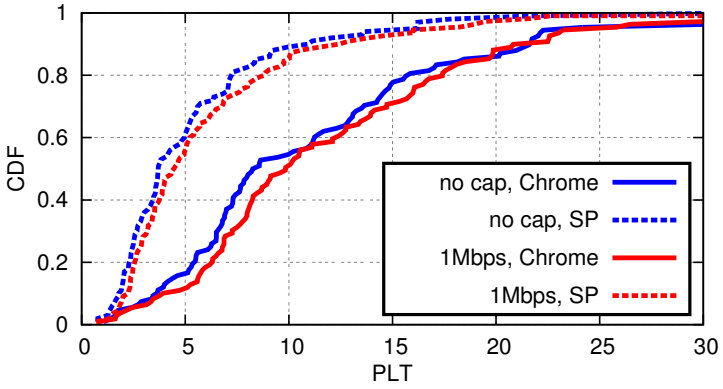


Figure 5.6: Varying bandwidth with fixed 1GHz CPU, 1GB memory, and 200ms RTT.

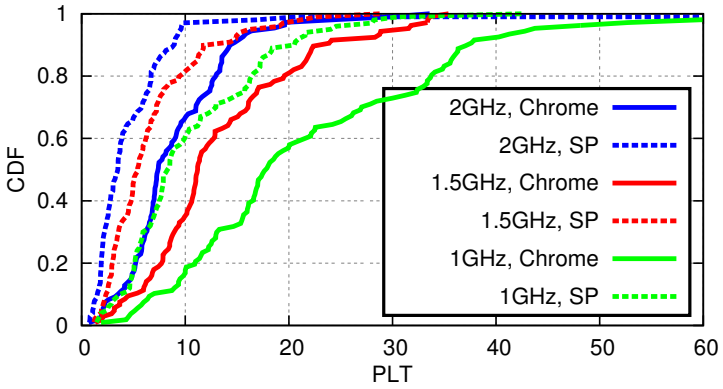


Figure 5.7: Varying CPU speed with fixed 1GB memory, no bandwidth cap, and no RTT insertion.

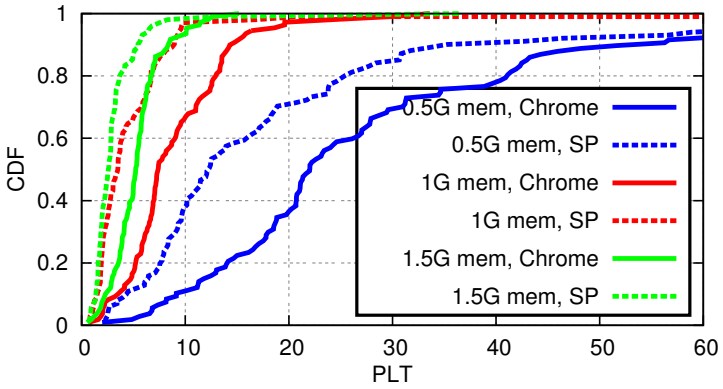


Figure 5.8: Varying memory size with fixed 2GHz CPU, no bandwidth cap, and no RTT insertion.

ing that SplitBrowser is insensitive to RTT. This is because among the breakdowns of PLT only t_f which is a small fraction of PLT is affected by RTT.

Varying bandwidth: We experiment with a 1Mbps bandwidth and with no bandwidth cap using fixed 1GHz CPU, 1GB memory, and 200ms RTT. Figure 5.6 shows that PLTs are affected little by bandwidths, which is consistent with previous findings [55, 51] that bandwidth is not a limiting factor of PLTs.

Varying CPU: We vary CPU speed from 1GHz to 2GHz while fixing memory size to 1GB. We do not tune RTT or bandwidth, meaning that PLT is dominated by computation. Figure 5.7 shows that the PLT improvement is linear to CPU increase. It also shows that CPU speed has the same amount of impact for both SplitBrowser and Chrome, because processing load-time state in SplitBrowser still incurs lots of CPU cycles. As PLT is dominated by computation, the results here approximate the situations when objects are inlined or cached. Clearly, SplitBrowser significantly improves PLTs than simply inlining objects since JavaScript evaluations and most of CSS evaluations are removed from the page load process.

Varying memory: We vary memory size from 0.5GB to 1.5GB with fixed 1GHz CPU and no network tuning. Figure 5.8 suggests that memory size has the same amount of impact for both SplitBrowser and Chrome, but a decrease in memory size has a more than linear negative impact on PLT.

In summary, SplitBrowser significantly improves PLT compared to Chrome under a variety of realistic mobile scenarios. This is rare since most techniques are specific to improve one of computation and network. But SplitBrowser improves both.

5.4.3 Size of transferred data

We evaluate the transferred data size as to (i) whether it hurts latencies and (ii) whether it hurts data usage.

To understand whether the size of transferred data helps or hurts latencies, we consider the size of the load-time state, because the post-load state and other objects do not affect the page load time. Figure 5.9 shows the size of the load-time state when a standard gzip compression is applied. The

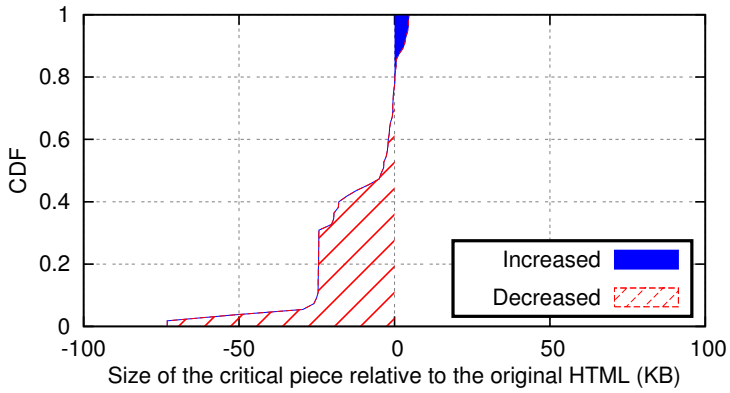


Figure 5.9: Size of the critical piece relative to the original HTML (KB).

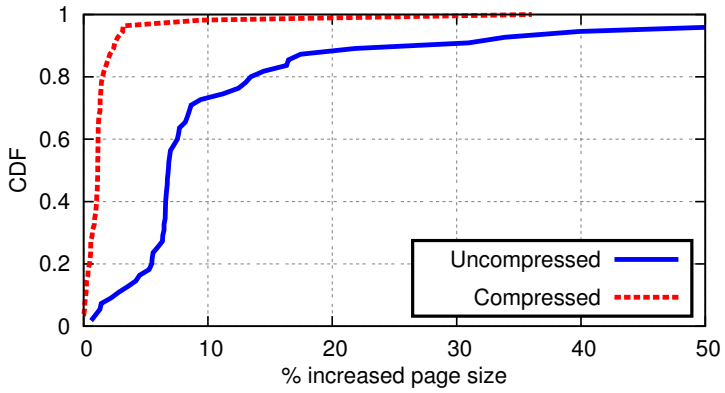


Figure 5.10: Percentage of increased page size (uncompressed v.s. compressed).

size of the load-time state relative to the original HTML decreases for most pages, and increases by a small amount only for less than 20% of the pages. This means that our migrated load-time state improves latencies in overall.

To evaluate whether the size of migrated state hurts data usage which is important for mobile browsers, we consider the total size of Web pages transferred to the client including all of the embedded objects. Figure 5.10 shows that the transferred data increases by 7% before compression, but it drops to 1% with standard gzip compression. This indicates that the overheads introduced by our approach are minimal.

5.4.4 Client-side state

We obtain client-side state from the browsers of a group of people, totaling 2,435 domains. The majority of the client-side state is HTML5 localStorage and cookies. We find that 90% of the websites use less than 460 bytes of localStorage, while 2% of the websites use more than 100KB of localStorage. We manually go through the 2% and find that large localStorage is almost always used for caching. For example, websites that use CloudFlare keep many caches in their localStorage; social networking websites such as Facebook store friends lists in the localStorage; and location-based services maintain the points of interest in the localStorage. Lack of such localStorage does not break Web pages because cache misses can be remedied by fetching from the server. Unlike localStorage, cookies are widely used but are always small, and Web databases are used in less than 1% of the websites and are small. The use of sessionStorage is also not much, likely because it only persists per-session state and cannot cache as long as localStorage. The study of client-side state suggests that migrating all of the necessary client-side state (e.g., cookies) to the server would not affect page load times by much.

5.5 Discussions

We believe that SplitBrowser is an important first step in mitigating dependencies that are the key bottleneck of latencies in page loads. Here, we discuss further optimizations that can be added to extend its efficiency and scope in the future.

Improving server-side concurrency. For dynamic pages where SplitBrowser's server extension needs to run in real time, reducing server concurrency is important to maintain low server cost and to improve end-to-end page load latencies. As a small set of object pieces are repeatedly processed for a large number of Web pages, an intuition of reducing latencies is to cache intermediaries of object pieces. The intermediaries can be cached in memory and shared across multiple instances, each of which processes a page load. Here, an object piece can be a CSS or JavaScript piece that is used by multiple pages, or static HTML pieces of a dynamic Web page. A piece of CSS or JavaScript is easy to identify; it is usually quoted by `link`, `style`, or `script` tags, or is a

stand alone Web object. However, a static HTML piece embedded in a dynamic Web page is hard to identify, because static HTML pieces and dynamic server-side code are largely mixed. Static HTML pieces can be identified by a diff-like analysis to a large set of dynamic pages. Intermediaries that correspond to the object pieces can be either parsed CSS rules for a CSS object, compiled JavaScript for a JavaScript object, or DOM nodes for a static HTML piece. When a CSS object is always attached to an HTML piece, DOM nodes with matched CSS styles can be further cached so as to eliminate CSS selector matching. Similarly, when a JavaScript object is always attached to an HTML piece, DOM nodes after JavaScript is applied can be further cached so as to eliminate JavaScript execution.

Using a cloud-based proxy for compression. SplitBrowser is orthogonal to existing cloud-based proxy approaches that do not restructure the page load process. This means that even if a proxy is already placed near the server for SplitBrowser, another proxy can be placed near the client for other purposes (e.g., Android Chrome Beta [72] for data compression, SPDY proxies for rewriting connections between the proxy and the device). However, approaches that restructure the page load process at clients (e.g., Opera mini [40] and Amazon Silk [4]) cannot be used together with SplitBrowser.

Extending the definition of PLT. Currently, SplitBrowser is designed for improving page load times defined by the W3C `load` event. But it would be trivial to extend SplitBrowser to improve any definition of page load times. The key is to capture the state of event listeners and the progress of HTML parsing for a given definition of page load time. The flexibility of PLT definitions is important because reports have shown that user-perceived page load times matter more than when the `load` event is fired [9].

5.6 Summary

This chapter presents SplitBrowser that improves PLT by simplifying the client-side page load process through a splitting page load architecture between the proxy server and the client. By preprocessing in the proxy server with more compute power, SplitBrowser largely reduces the inefficiencies of page loads on the client. SplitBrowser is fast for displaying Web pages, ensures

that users normally interact with the page, and is compatible with caching, CDNs, and security features that enforce same-origin policies. Our evaluations show that SplitBrowser reduces PLTs by more than half on a variety of mobile settings with varied RTT, bandwidth, CPU power, and memory size.

Chapter 6

Conclusions and Future Work

This chapter describes the contributions made by the dissertation (§6.1). We review future research directions (§6.2) and summarize the dissertation (§6.3).

6.1 Thesis and Contributions

The dissertation supports my thesis that *Web performance can be radically improved by understanding the inefficiencies, pinpointing the bottlenecks, and developing techniques that remove the bottlenecks*. We make the following contributions in this dissertation:

- **A measurement tool, WProf.** To understand page load performance, we abstract a model of dependency policies that describes how page load activities interplay. Based on the policies, we develop WProf, a lightweight in-browser profiler that runs in Webkit browsers while real pages are being loaded. WProf generates a dependency graph and identifies a page load bottleneck for any given Web page. We run WProf from popular servers, apply critical path analysis to analyze the bottlenecks, and find the following results. 1) While most prior work focuses on network activities, browser computation (mostly HTML parsing and JavaScript evaluation) comprises 35% of the critical path, 2) Downloading HTML and synchronous JavaScript makes up a large fraction of the critical path while fetching CSS and asynchronous JavaScript makes up little of the critical path, 3) Caching reduces the volume of data substantially, but decreases PLT by a lesser amount because many of the downloads that benefit from it are not on the critical path, 4) Mod_pagespeed does little to reduce PLT

because minifying and merging objects does not reduce network time on the critical path.

- **A systematic measurement study of the SPDY protocol.** Taking into account of the challenges, we conduct an in-depth study of SPDY as a result of controlled and reproducible experiments that isolate the different factors that affect PLT. We proceed from simple, synthetic pages to complete page loads based on the top 200 Alexa sites. Findings include: 1) SPDY provides a significant improvement over HTTP/1.1 when dependencies and browser computation are ignored; 2) Unfortunately, the benefits can be easily overwhelmed by dependencies and computation; 3) Most SPDY benefits stem from the use of a single TCP connection, but the same feature is also detrimental under high packet loss; 4) Further benefits in PLT will require changes to restructure the page load process such as the server push feature of SPDY.
- **A prototype of SplitBrowser.** Informed by results from my previous work, we design and implement a prototype, SplitBrowser, that significantly reduces PLT at the client. SplitBrowser uses a proxy server to preload a Web page, quickly communicates an initial representation of the page's DOM to the client, and loads secondary resources in the background. SplitBrowser also ensures that the Web page functionality such as user interactivity is preserved and that the delivery process is compatible with latency-reduction techniques such as caching and CDNs and security features such as the enforcement of same-origin policies. The resulting system is thus both efficient and practical. Evaluations on the top 100 Alexa Web pages show that SplitBrowser reduces PLT by 60% on both a 1GHz Android phone and a variety of Linux desktop settings.

6.2 Future Work

Besides setting out key steps in characterizing and improving PLTs, this dissertation also opens up several areas of future work.

6.2.1 Extending WProf

WProf is a first step to analyzing PLTs that are spent on complex browser artifacts and geographically distributed machines. There are many ways that WProf can be extended to obtain a more

complete characterization of PLTs.

Completing the dependency model at other levels. WProf's dependency model focuses on the Web page level at front ends. One direction is to extend the dependency model to back ends. For example, the Mystery Machine [16] has followed up to provide an end-to-end analysis of Facebook PLTs that involve both the front ends and the back ends. Another direction is to extend the dependency model to lower layers such as TCP. Such extensions would help conduct what-if analysis based on more realistic assumptions. For example, an assumption with WProf can be a 20% increase in loading an object, which does not relate to any physical events, but an assumption with an extended TCP model can be a packet loss at the third round trip, which is more realistic.

Characterizing mobile Web with WProf. We have only characterized PLTs in desktop settings. As the use and kinds of mobile devices are evolving, characterizing PLTs under mobile settings would be valuable. Mobile devices often use a slower network and have less compute power, which is likely to be the case given the limited battery power. The relative contributions to PLTs from the network and the computation would be interesting to learn.

Visualizing WProf dependency graphs. WProf is released as a tool that anyone can use, but it requires lots of installations that limit its usability. One direction is to provide a service that runs WProf on behalf of a user (e.g., Web developers) and that visualizes the WProf dependency graphs. The visualization can help users locate the bottleneck and provide what-if analysis as to how to improve a single Web page. We have an ongoing project in this direction.

6.2.2 Further Characterizing and Improving SPDY

Although we have provided a systematic study of SPDY, there are many ways that can extend the understanding and the performance of SPDY.

Characterizing SPDY in its practical usage. There are many practical uses of SPDY beyond the connection between a client and a server. SPDY is often used in a forward proxy that manages the connection between a client and the proxy, where resources are frequently multiplexed. SPDY is also used in a reverse proxy that manages the connection between a website's front ends and

back ends. These uses could incur differences from what we found. Another use scenario is when users change their locations and thus IP addresses frequently. How well SPDY performs under high mobility would be interesting to learn.

Exploring SPDY's advanced policies. We initiated studies of SPDY's advanced policies, which are still rudimentary. It is still unknown what an optimal server push strategy is like given lower-level assumptions and a Web page's dependency graph. Explorations in this direction would exploit SPDY's potential more thoroughly.

6.2.3 Restructuring the Page Load Process

There are many possible approaches to restructure the page load process.

Micro-caching. As Web pages are highly repetitive, websites can manage the browser cache at a micro-level in order to achieve the most performance benefits. We have evaluated the benefits if caching were performed at a finer granularity and at different levels (i.e., computed layout and compiled JavaScript) [67]. By analyzing Web pages gathered over two years, we find that both layout and code are highly cacheable. This suggests that PLTs could be radically improved if Web pages together with browsers' caching mechanisms were re-architected to support micro-caching.

Long-term impact. The page load process can be restructured differently given different assumptions. SplitBrowser assumed that both the client and the server can be modified. The approach would be different if only the client or the server can be modified, if browsers cannot be modified, or if Web standards can be modified. As the need towards a more performant Web is having a long-term impact on the stake holders of the Web (e.g., browsers, websites, and standards), we tend to appreciate approaches that are less restricted to the current Web.

6.3 Summary

Web page load time (PLT) is a key performance metric that is much slower than lower-level latencies, the reason of which was not well understood.

This dissertation first characterizes the Web page load time by abstracting a dependency model between network and computation activities. We have built a tool WProf based on this model, that

identifies the bottlenecks of PLTs of hundreds of Web pages, and that provides basis for evaluating PLT-reduction techniques. Next, we evaluate SPDY's contributions to PLTs and find that SPDY's impact on PLTs is largely limited by the dependencies and browser computation. This suggests that the page load process should be restructured to remove the dependencies so as to improve PLTs. Thus, we propose SplitBrowser that preprocesses Web pages on a proxy server and migrate carefully crafted state to the client so as to simplify the client-side page load process. We have shown that SplitBrowser reduces PLTs by more than half under a variety of mobile settings that span less compute power and slower networks.

Bibliography

[1] Bernhard Ager, Wolfgang Muhlbauer, Georgios Smaragdakis, and Steve Uhlig. Web Content Cartography. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2011*.

[2] Akamai. `http://www.akamai.com/`.

[3] Top sites in United States. `http://www.alexa.com/topsites/countries/US`.

[4] Amazon silk browser. `http://amazonsilk.wordpress.com/`.

[5] Awazza. `http://awazza.com/`.

[6] Aruna Balasubramanian, Ratul Mahajan, and Arun Venkataramani. Augmenting Mobile 3G Using WiFi. In *Proc. of the international conference on Mobile systems, applications, and services (Mobisys), 2010*.

[7] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into Web server design. In *Computer Networks Volume 33, Issue 1-6*, 2000.

[8] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. Quality is in the eye of the beholder: meeting users' requirements for Internet quality of service. In *Proc. of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI), 2000*.

[9] Jake Brutlag. Above the fold time: Measuring web page performance visually, March 2011. `http://en.oreilly.com/velocity-mar2011/public/schedule/detail/18692`.

[10] Brian Burg, Richard Bailey, Andrew J. Ko, and Michael D. Ernst. Interactive Record/Replay for Web Application Debugging. In *Proc. of the ACM UIST, 2013*.

[11] Michael Butkiewicz, Harsha V. Madhyastha, and Vyas Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2011*.

[12] Marta Carbone and Luigi Rizzo. Dummynet revisited. *ACM SIGCOMM Computer Communication Review*, 40(2):12–20, March 2010.

[13] Calin Cascaval, Seth Fowler, Pablo Montesinos Ortego, Wayne Piekarski, Mehrdad Reshadi, Behnam Robatmili, Michael Weber, and Vrajesh Bhavsar. ZOOMM: A Parallel Web Browser Engine for Multicore Mobile Devices. In *Proc. of the ACM PPoPP, 2013*.

[14] Yingying Chen, Ratul Mahajan, Baskar Sridharan, and Zhi-Li Zhang. A provider-side view of web search response time. In *Proc. of the ACM SIGCOMM*, 2013.

[15] Ryan H. Choi and Youngil Choi. Designing a high-performance mobile cloud web browser. In *Proc. of the International World Wide Web Conference (WWW)*, 2014.

[16] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proc. of the USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[17] D3.js. `http://d3js.org/`.

[18] Chrome Developer Tools. `https://developers.google.com/chrome-developer-tools/docs/overview`.

[19] Nandita Dukkipati, Matt Mathis, Yuchung Cheng, and Monia Ghobadi. Proportional rate reduction for TCP. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC)*, 2011.

[20] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing web latency: the virtue of gentle aggression. In *Proc. of the ACM SIGCOMM*, 2013.

[21] Google. Content module. `http://www.chromium.org/developers/content-module`.

[22] Ilya Grigorik. Chrome networking: DNS prefetch & TCP preconnect, June 2012. `http://www.igvita.com/2012/06/04/chrome-networking-dns-prefetch-and-tcp-preconnect/`.

[23] Thomas J. Hacker, Brian D. Noble, and Brian D. Athey. The Effects of Systemic Packet Loss on Aggregate TCP Flows . In *Proc. of IEEE Iternational Parallel and Distributed Processing Symposium (IPDPS), 2002*.

[24] HTTP Archive. `http://httparchive.org/`.

[25] HTTP/2.0 Draft Specifications. `https://github.com/http2/http2-spec`.

[26] Junxian Huang, Qiang Xu, Birjodh Tiwana, Z. Morley Mao, Ming Zhang, and Paramvir Bahl. Anatomizing application performance differences on smartphones. In *Proc. of the international conference on Mobile systems, applications, and services (Mobisys), 2010*.

[27] Sunghwan Ihm and Vivek S. Pai. Towards understanding modern web traffic. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2011*.

[28] jquery. `https://www.jquery.com/`.

[29] Eric Lawrence. Internet Explorer 9 network performance improvements, March 2011. `http://blogs.msdn.com/b/ie/archive/2011/03/17/internet-explorer-9-network-performance-improvements.aspx`.

[30] Zhichun Li, Ming Zhang, Zhaosheng Zhu, Yan Chen, Albert Greenberg, and Yi-Min Wang. WebProphet: automating performance prediction for web services. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI), 2010*.

[31] James Lo, Eric Wohlstadter, and Ali Mesbah. Imagen: Runtime Migration of Browser Sessions for JavaScript Web Applications. In *Proc. of the International World Wide Web Conference (WWW), 2013*.

[32] Haohui Mai, Shuo Tang, Samuel T. King, Calin Cascaval, and Pablo Montesinos. A Case for Parallelizing Web Pages. In *Proc. of HotPar, 2012*.

[33] Leo A. Meyerovich and Rastislav Bodik. Fast and parallel webpage layout. In *Proc. of the international conference on World Wide Web (WWW), 2010*.

[34] Jame Mickens. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads. In *Proc. of USENIX conference on Web Application Development (WebApps), 2010*.

[35] mod_spdy. `https://code.google.com/p/mod-spdy/`.

[36] World Flags mod_spdy Demo. `https://www.modspdy.com/world-flags/`.

[37] National broadband map. `http://www.broadbandmap.gov/`.

[38] Not as SPDY as you thought. `http://www.guypo.com/technical/not-as-spdy-as-you-thought/`.

[39] Open DNS. `http://www.opendns.com/`.

[40] Opera mini browser. `http://www.opera.com/mobile/`.

[41] Jitu Padhye and Henrik Frystyk Nielsen. A comparison of SPDY and HTTP performance. In *MSR-TR-2012-102*.

[42] Google Pagespeed Insights. `https://developers.google.com/speed/pagespeed/insights`.

[43] PhantomJS. http://phantomjs.org/.

[44] Shopzilla: faster page load time = 12% revenue increase. http://www.strangeloopnetworks.com/resources/infographics/web-performance-and-ecommerce/shopzilla-faster-pages-12-revenue-increase/.

[45] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. TCP Fast Open. In *Proc. of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2011.

[46] Render Start. https://sites.google.com/a/webpagetest.org/docs/usingwebpagetest/metrics.

[47] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989.

[48] Selenium web driver. http://seleniumhq.org/.

[49] Ashiwan Sivakumar, Vijay Gopalakrishnan, Seungjoon Lee, Sanjay Rao, Subhabrata Sen, and Oliver Spatscheck. Cloud is not a silver bullet: A Case Study of Cloud-based Mobile Browsing. In *Proc. of HotMobile, 2014*.

[50] Chapter 11. HTTP 1.X. http://chimera.labs.oreilly.com/books/1230000000545/ch11.html.

[51] Steve Souders. *High Performance Web Sites*. O'Reilly Media, 2007.

[52] Spdy. http://dev.chromium.org/spdy.

[53] SPDY best practices. http://dev.chromium.org/spdy/spdy-best-practices.

[54] Analysis of SPDY and TCP Initcwnd. http://tools.ietf.org/html/draft-white-httpbis-spdy-analysis-00.

[55] SPDY whitepaper. http://www.chromium.org/spdy/spdy-whitepaper.

[56] SPDY protocol–Draft 3. http://www.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3.

[57] Proposal for Stream Dependencies in SPDY. https://docs.google.com/document/d/1pNj2op5Y4r1AdnsG8bapS79b11iWDCStjCNHo3AWD0g/edit.

[58] Spdylay - SPDY C Library. https://github.com/tatsuhiro-t/spdylay.

[59] Speed index. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

[60] Speed Tracer. https://developers.google.com/web-toolkit/speedtracer/.

[61] Srikanth Sundaresan, Nick Feamster, Renata Teixeira, and Nazanin Magharei. Measuring and mitigating web performance bottlenecks in broadband access networks. In *Proc. of the ACM Internet Measurement Conference (IMC)*, 2013.

[62] Apache module for rewriting web pages to reduce latency and bandwidth. http://www.modpagespeed.com/.

[63] Ashish Vulimiri, Oliver Michel, P. Brighten Godfrey, and Scott Shenker. More is Less: Reducing Latency via Redundancy. In *Proc. of the ACM Workshop on Hot Topics in Networks (HotNets-XI), 2012*.

[64] Cascading Style Sheets level 2 revision 1 (CSS 2.1) specification, June 2011. http://www.w3.org/TR/CSS21/.

[65] Document Object Model (DOM) Level 3 Events specification, September 2014. http://www.w3.org/TR/DOM-Level-3-Events/.

[66] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying page load performance with WProf. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI), 2013*.

[67] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. How much can we micro-cache web pages? In *Proc. of the ACM Internet Measurement Conference (IMC)*, 2014.

[68] Xiao Sophia Wang, Haichen Shen, and David Wetherall. Accelerating the Mobile Web with Selective Offloading. In *Proc. of the ACM SIGCOMM Workshop on Mobile Cloud Computing (MCC), 2013*.

[69] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. How far can client-only solutions go for mobile browser speed? In *Proc. of the international conference on World Wide Web (WWW), 2012*.

[70] The Webkit Open Source Project. http://www.webkit.org/.

[71] WebPagetest. http://www.webpagetest.org/.

[72] Matt Welsh. Data compression in Chrome Beta for Android, March 2013. http://blog.chromium.org/2013/03/data-compression-in-chrome-beta-for.html.

[73] YSlow. http://yslow.org/.

[74] Kaimin Zhang, Lu Wang, Aimin Pan, and Bin Benjamin Zhu. Smart caching for web browsers. In *Proc. of the international conference on World Wide Web (WWW), 2010*.

[75] Wenxuan Zhou, Qingxi Li, Matthew Caesar, and Brighten Godfrey. ASAP: A Low-Latency Transport Layer. In *Proc. of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2011*.

[76] Yuhao Zhu and Vijay Janapa Reddi. WebCore: Architectural Support for Mobile Web Browsing. In *Proc. of the 41st International Symposium on Computer Architecture (ISCA)*, 2014.