RESEARCH PROFICIENCY EXAMINATION REPORT

Improving page load time on mobile devices

Javad Nejati

Committee:
Professor Aruna Balasubramanian (Advisor)
Professor Samir Das
Professor Anshul Gandhi

STONY BROOK UNIVERSITY
Department of Computer Science

Contents

Chapter 1

Introduction

1.1 Problem statement

Based on recent reports, number of smartphone subscriptions worldwide is expected to exceed 6.1 *billion* mark in 2020 [14]. In the US, the use of mobile Internet has already exceeded laptops and tablets [30]. For a considerable amount of users, the smartphone is the only compute device in their possession [31]. Not surprisingly, for many users, mobile pages are the primary portal for content, and mobile browsers continue to be one of the most popular applications on smartphones [9]. However, the page load performance on mobile devices does not match up to its importance: mobile page load times are an order of magnitude slower than desktop, often taking 10s of seconds to load just the landing page [8].

Unfortunately, it is not easy to improve mobile Web performance: several Web optimizations have been designed, but their effect on improving mobile page load times have been mixed.
FlyWheel [2], Google's data compression proxy, significantly reduces data usage on mobile devices, but its effect on page load time is mixed. HTTP 2.0 [45] is known to significantly improve performance over HTTP for desktop browsers, but studies show that the improvement over mobile browsers is not significant [15]. Other research projects target specific aspects of the mobile browsing such as the network latency [44] or user QoE [8], but have not been successful in improving the performance of the entire mobile page load time.

1.2 Challenges

There are three main challenges in improving mobile page load times: *dependency*, *complexity* and *variances*. In order to understand challenges in improving mobile page load times, we first need to understand how the browser loads a web page.

So, we start with some browser background.

1.2.1 Browser Background

Browser is a sophisticated software which comprises several interdependent activities. These activities can holistically be categorized as computation and network activities.

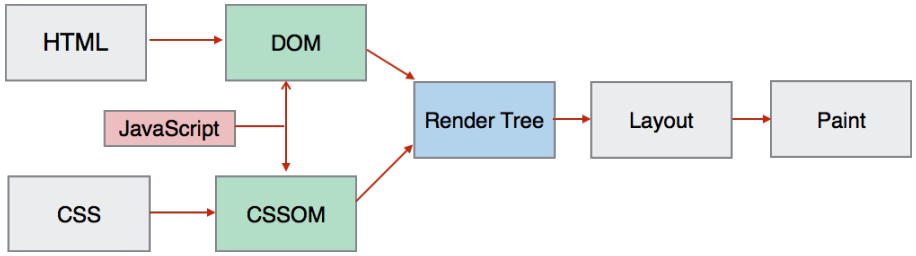


Figure 1.1: Page load process inside a browser

Browser starts off with fetching the html page (Figure 1.1), a network activity. After receiving the first chunk of HTML page, HTML parsing process starts in parallel to generate the Document Object Model, or DOM[48], a computation process.
DOM provides a structured representation of the document as a tree and it defines a way that the structure can be accessed from programs so that they can change the document structure, style and content.
For example, for the sample following HTML code, first, the browser identifies HTML *objects* such as *html*, *head*, *body* , ...
Then, since the HTML markup defines relationships between tags, objects are connected in a tree data structure to hold the parent-child relationship from the original markup. *html* object is parent of the *body* object, the *body* object is parent of the *paragraph* object and so on.

```
<html>
  <head>
    <link href="style.css" rel="stylesheet">
  </head>
  <body>
    <p>Hello DOM!</p>
    <div><img src="d.jpg"></div>
  </body>
</html>
```

The corrosponding DOM tree is depicted in figure 1.2.

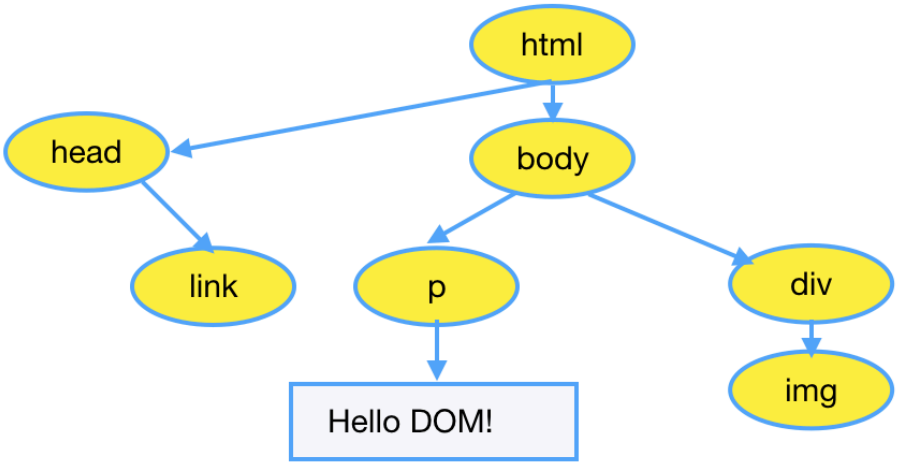


Figure 1.2: DOM construction

Rendering process also continuously renders the page, based on the intermediate DOM structure. In rendering process, layout computes the exact position and size of each object and paint process takes in the final render tree and renders the pixels to the screen.

As the parsing progresses, new objects need to be downloaded based on the page's HTML code. If the object happens to be a regular Javascript (instead of asynchronous Javascript) or a CSS file, DOM construction process is blocked until these scripts are downloaded and parsed, Imposing *dependency* between activities.

1.2.2 Dependency

We start with *dependency* as the first challenge in improving page load time. Each computation or networking elements running during the page load process is called an activity. As we have seen so far, activities lie in either *computation* or *networking* class.

From a different perspective, We can classify activities based on their dependency behavior.
There are activities that can be loaded/executed independent of other activities[1].

For example, two images in the following HTML code can be downloaded in parallel. Loading one of them does not depend on loading the other.

[1]Obviously, all activities are dependent on the loading and parsing of the main html file. When we talk about activities, we are often talking about objects inside the root html file.

```
<html>
  <body>
     <img src = "a.jpg"/>
     <img src = "b.jpg"/>
  </body>
</html>
```

In order to observe how activities are ordered in a page load process, We use WProf[49]. WProf is discussed in §3.
For example, for the above mentioned HTML code, figure1.3 illustrates the corresponding dependency graph.
The X axis shows time in milliseconds and Y axis which grows from top to down, only shows the order of objects inside an HTML file. The light blue circle is the very first download HTML activity which is followed by a HTML evaluation activity (dark blue). Two long purple bars refer to download of "a.jpg" and "b.jpg" ,respectively.

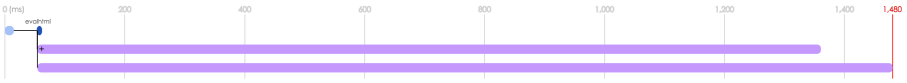


Figure 1.3: Dependency graph for a HTML page with two independent images

On the other hand, there are several activities inside a page load process that *affects/will be affected* by other activities. For instance, adding more activities to the simple HTML code, brings about two different possible dependency scenarios:

1. Adding Javascript after loading all images.

```
  <html>
  <body>
    <img src = "a.jpg"/>
    <img src = "b.jpg"/>
    <img src = "c.jpg"/>
    <img src = "d.jpg"/>
    <img src = "e.jpg"/>
    <img src = "f.jpg"/>
    <img src = "g.jpg"/>
    <img src = "h.jpg"/>
    <img src = "i.jpg"/>
    <script src = "b.js"> </script>
  </body>
</html>
```

In this case, as Javascript is loaded and executed after all images, image loading is not blocked by Javascript load, that is, loading of images does not *depend* on loading of Javascript code.

However, there still is an internal dependency between loading the Javascript and it's execution.

Generally speaking, for all activities which need computation, such as HTML, Javascript, CSS, ... there always exists a *Load dependency*. The dependency graph for this scenario is depicted in Figure1.4
The bottom bars are showing Javascript (*b.js*) download and execution. They start after all images has been started.

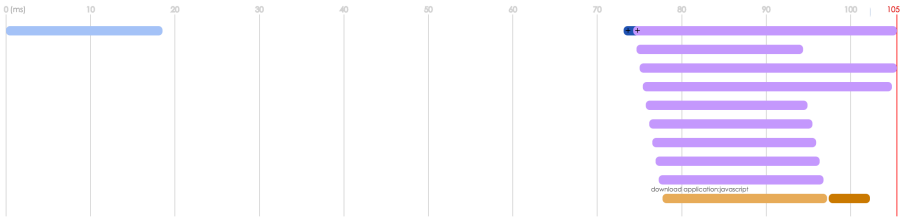


Figure 1.4: Dependency graph for a HTML page when Javascript is loaded after all images

2. Adding Javascript before image load.

   The interesting case is when Javascript code is located before images' ones. In this case, when the HTML parser encounters a dynamic activity that needs to be computed and is likely to modify the DOM tree. Browsers block subsequent activities until the dependency relations used in constructing the DOM tree are resolved. If the dynamic activity does affect subsequent activities, It has to be loaded and computed before upcoming ones.

   However, in cases where parser realizes that this dynamic activity has no effect on download of subsequent ones, subsequent activities can be safely downloaded shortly after the execution starts .

   The dependency graph for this scenario is depicted in figure1.5
   The top bars on the right are showing Javascript (*b.js*) download and execution which start before image loads.

```
<html>
<body>
  <script src = "b.js"> </script>
  <img src = "a.jpg"/>
  <img src = "b.jpg"/>
  <img src = "c.jpg"/>
```

```
    <img src = "d.jpg"/>
    <img src = "e.jpg"/>
    <img src = "f.jpg"/>
    <img src = "g.jpg"/>
    <img src = "h.jpg"/>
    <img src = "i.jpg"/>
  </body>
</html>
```

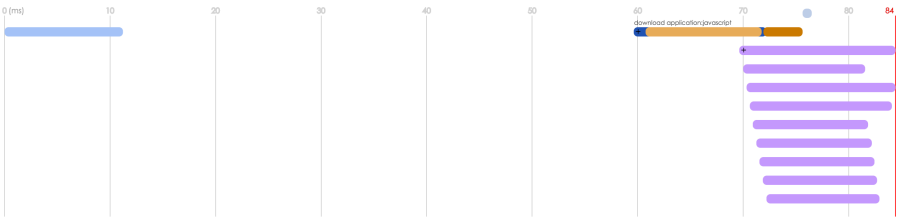


Figure 1.5: Dependency graph for a HTML page when Javascript is loaded before images

We can observe, if we reorder javascript and image locations in an HTML file, the resulting page and consequently the page load time is different.

In other words, the performance, depends directly on how the dependency graph between markup, stylesheets, and Javascript is resolved

1.2.3 Complexity

The second challenge in improving page load time that we will look into is complexity. Complexity of a website can be characterized as the number of objects inside it.
Adding more objects to a web page structure, not only increases the total size (which naturally means more download and computation time), but also has an effect on complexity of the dependency graph.
For instance, adding a Javascript to a page structure, spontaneously imposes one dependency between downloading and execution of that Javascript. Moreover, since Javascript modifies the DOM tree, all subsequent activities need to be blocked until the dependency issues are resolved.
In figure 1.6, each X axis ticks corresponds to a year (J-95 for January 1995) and the right Y axis shows average number of objects for all website analyzes in that particular year. We can observe that average number of objects in a webpage has doubled in last 7 years. The figure 1.6 also shows that size of websites has doubled in only 3 years and Although computation power and network bitrate are also increasing, but we are dealing with an ever increasing trend in page size and the

average number of objects in a web page.

Increased size and number of objects, in turn, actuates the dependencies inside the page structure. These all makes improving page load time even more complicated.

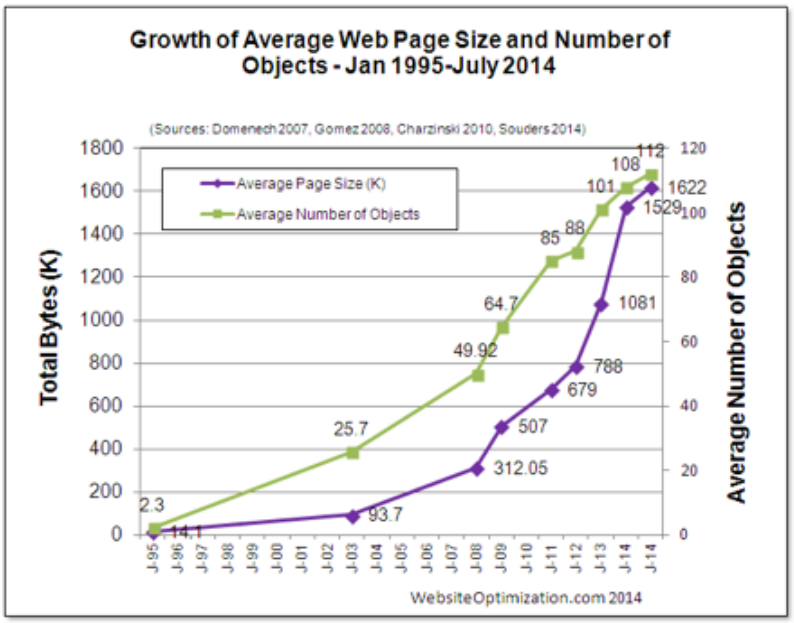


Figure 1.6: Websites' size and objects growth (source: websiteoptimization.com)

1.2.4 Variances

Another challenge that makes studying page load time complicated, is the variability of page load time over wireless/cellular networks. Page load time depends on several factors such as:

- Underlying network parameters including bandwidth, delay and packet loss.
- Connection Type can be Wi-Fi, 4G, 3G, . . . , each with different characteristics.
- Device computation and network resources such as CPU, RAM, wireless chipset, . . .
- Even time and location of access may have effect on quality of network service one receives.

Taking into account all these parameters is not a trivial task.

1.3 Overview

Our primary object is to improve Web page load times. Although there are several approaches to improve Web pages, we propose the following: We will design algorithms and techniques to determine the best optimization(s) for a given page. Our goal is to provide an optimization framework for Web developers that will quickly suggest the best optimizations for a page.
The remainder of this report is organized as follows.
In order to address dependency and variances challenges, we start off by building a testbed to run our experiments in a controlled environment. This testbed resolves the variances issue and lets us focus on each varying factor one at a time. (chapter 3)

Using the testbed, we first study the characteristics of the mobile page load process. From the experiments ran on this testbed, we observe that although networking time is the main bottleneck in desktop environment, but in mobile environment, main bottleneck in a page load process is the computation time. (chapter 4)
However, most mobile optimizations today including FlyWheel[2], Parcel [44], HTTP/2 [45], attempt to reduce the network latency during page load time.
Our dependency graph and critical path analysis approach using this testbed, enables us to study page load time behavior when a certain optimization is applied into a page structure (such as inlining) or when we change the way browsers and servers communicate (for example, enabling compression on server side).
Then, we will employ these findings to build a platform (PLTSpeed, our current project) to suggest users, what optimizations they need to apply and what is the effect of applied optimizations on their page's load time. (chapter 5)
In future, we propose to design and build a platform to predict page load time in a fast and scalable manner. Then, using this prediction mechanism, we suggest to users optimization (or an ordered list of optimizations) that most improve their Web site's page load time.

Chapter 2

Related works

In this chapter, we investigate previous efforts related to our work. We start by studying other similar testbeds. Then we explore proposed page load time improvement techniques, whether they are client side, server side or industry tools. We wrap up this chapter by looking into notable measurement studies related to our work.

2.1 Testbeds

Mahimahi [33] is a framework to record traffic from HTTP-based applications, and later replay it under emulated network conditions. Mahimahi highlights it's features in 3 main parts:

1. It takes into account the multi-server nature of a webpage (external linked objects). Mahimahi launches separate web server instances for each external server to better emulate how web objects are delivered to the browser. Their results show that lack of multi-server emulation yields significantly worse performance at higher link rates.

2. It employs linux network namespace feature to isolate TCP/IP interactions for each instance.

3. The design is modular in the sense that different shells can be put together to build different architectures (like Lego blocks). Mahimahi is structured as a set of UNIX shells:

    - RecordShell, allows a user to record all HTTP traffic for any process spawned within it.
    - ReplayShell, replays recorded content using local servers that emulate the application servers.

- To emulate network conditions, Mahimahi includes DelayShell, which emulates a fixed network propagation delay.
- LinkShell, which emulates both fixed-capacity and variable-capacity links.

While we share using Linux network name space with Mahimahi, there are some fundamental differences between Mahimahi and our approach.
In our testbed, we mainly focus on studying the critical path, and therefore record the page load time at the object level rather than at the HTTP level as Mahimahi does.

Also, we are using well-proven Linux's Traffic Control program instead of a new link emulation software.

In addition, site manipulation is easier in our testbed as we have the whole page structure locally. For example, we can easily minify scripts and replay the new modified version locally.

Moreover, our testbed supports real mobile access, while MahiMahi emulates mobile devices.

WebProphet [25] was one of the first works to discuss dependencies for desktop browsers. WebProphet extracts dependencies in desktop browsers, and treats all computational activities as a black box. WProf, improved over WebProphet by uncovering dependencies in both networking and computational aspects of page load. We are also using mobile version of WProf to study dependencies in a mobile environment.

Finally, there are measurement platforms such as WebPageTest[54] and HTTP Archives [21] which allow researchers to perform Web page measurements from several vantage points. They provide measurement data from a large number of networks and devices. However, we still lack tools to directly compare the performance of desktop and mobile browsers, or analyze the critical path.

2.2 Page load time improvements

There has been number of proposals to improve page load time in mobile environment. They either incline toward server or client side.

2.2.1 Client side improvements

Qian et al.[36], study mobile web cache implementations across a wide range of applications and browsers, using data collected both from a large cellular operator

(AT&T) as well as 20 real users over several months. They study the impact of redundant transfers and identify the key root cause being broken caching logic in many HTTP library implementations used by native apps. The result is that 18% to 20% of HTTP transfers are redundant and could be eliminated with correct caching implementations.
Their results show that there exists a big gap between the protocol specification and implementation on mobile devices, which leads to significant amounts of redundant *network* traffic.

Zhue et al. [61] take a "hardware-level" optimization approach by proposing the *"WebCore"* , a general-purpose core customized and specialized for the mobile Web browsing workload, to achieve both energy-efficiency and performance improvements in mobile web browsing.
They aim to improve the *computation* part of mobile web browsing. Their findings show that there are two main bottlenecks in current designs: Instruction delivery and data feeding. *WebCore* adds two new components to the current CPU architecture: Style Resolution Unit (SRU) and Software-Managed Browser Engine Cache.
Their results show that customizations alone on the existing general-purpose mobile processor design lead to 22.2% performance improvement and 18.6% energy saving.

Wang et al.[51] add speculative loading to the current caching and pre-fetching solutions. In speculative loading, instead of trying to predict user behavior (prefetching), server's behavior is predicted.
Also, speculative loading starts loading the resources only after the user requests a webpage's URL in contrast to prefetching that loads the resources beforehand.
In their experiments, they observe on average, 76% of the resources in one webpage are shared by at least one other webpage from the same website. Hence, they conclude that after a user visits a website for enough times and the resource graph is constructed, the browser can potentially predict the majority of the subresources (mostly CSS and Javascript) needed for a new webpage visit, and thus speculatively load them.

On their experiments they observed a maximum of 1.4 seconds improvement in page load time and consider this as a limit for any client-only solution.
However, they consider network RTT as the major bottleneck in mobile browser performance .

2.2.2 Server side improvements

Sivakumar et al.[43] compare CB (cloud based) with Direct (device based) mobile browsing and how they affect page load time and total energy. In a CB, all or

part of the computation or netwoorking tasks are offloaded to the cloud, while in Direct, every thing is done on the client device. By using CB, mobile device can offload heavy processing components such as JavaScript to the cloud in order to reduce CPU time and energy. Then the processed page is returned in a specific notation (Called CBML: Cloud Based Markup Language) to the mobile device.

Another benefit of using CB is data compaction performed in cloud which aims to decrease page load time and transferred bytes. Their evaluations indicate that neither Direct nor CB is better under all scenarios. For e.g. while CB decreases the download time compared to Direct for 38.87% of pages, it increases it by as much as 29.8s for other pages. Similarly CB increases energy usage by up to 21.31J compared to Direct for some pages. They have found root causes of these variances to be: Extent and duration of Javascript run in the page and compaction ratio.

PARCEL[44] moves the task of identifying and downloading objects needed to render a Webpage to a proxy. Main focus of PARCEL is to support cellular friendly data-transfers by reducing the number of HTTP request-response interactions. Unlike *cloud based browsing* [43], PARCEL keeps most other functionalities including Javascript execution in the client browser and does not try to offload the *computation* part. PARCEL uses it's own browser to communicate with the proxy part and therefore is more of a mixed (client and server) solution.

FlyWheel[2] is Google's data reduction proxy service that provides an average 58% byte size reduction of HTTP content. FlyWheel also focuses on *network* bytes transfer. Overhead of compression and decompression on CPU time and energy consumption has not been studied in FlyWheel.

KLOTSKI[8] states that "the increasing complexity of web page content and decreasing user tolerance will outpace the benefits from incremental performance enhancements such as compression, caching, cloud based browsers, . . . ". Therefore, they focus on improving user experience, instead of decreasing total page load time. In order to do that, they propose a platform which dynamically reprioritizes web content so that the resources on a page that are critical to the user experience are delivered sooner. To respond to a request, KLOTSKI first selects the subset of resources on the page that it should prioritize. Thereafter, as the client executes the page load, the front-end alters the sequence in which the page's content is delivered to the client, in order to prioritize the delivery of the selected subset of resources.

2.2.3 Industry tools

Google's PageSpeed Insights [20] is a tool that analyzes web pages and generates tailored suggestions to make the pages faster. PageSpeed Insights analysis does not use real devices, instead it changes the user-agent for desktop and mobile in webkit renderer accordingly.
PageSpeed Insights uses a set of predefined rules to evaluate a webpage. Each PageSpeed rule generates an impact number that indicates the importance or priority of implementing the rule-result suggestions for the rule, relative to other rules.

YSLOW [56] is Yahoo's performance analysis tool that grades a web page, based on one of three predefined ruleset or a user-defined ruleset. It also offers suggestions for improving the page's performance.

Neither PageSpeed Insights nor YSlow metrics, directly indicate page load time. Part of our current project is defining a new scoring system that is correlated with page load time and shows amount of improvement based on the page load time improvement.

2.3 Measurement studies

Qian et al. [35] provide a comprehensive measurement study to examine the resource usage of mobile browsers, but mainly focus on cellular bandwidth and energy usage rather than computation and page load times.

Bui et al. [5] propose techniques to optimize the energy consumption of web page loading on cellphones. Their network-aware resource processing and adaptive content painting, aim to address energy inefficiency issues of the current mobile web browsers in its content processing and graphic processing pipelines.
Application assisted scheduling's goal is to balance the trade-off between the energy saving and the QoS in ARM's big.LITTLE platforms.
However, their focus is more on lowering energy consumption, without affecting the original page load time.

Singh et al. [42] show that compression middleboxes are not always helpful. Compression middleboxes are middle servers that compress page content and are usually deployed as part of a content delivery network. commonly used today. They observe from extensive measurements that the compression middle-box should be used only when network conditions are bad and otherwise, should be directly

fetched from the original web server. Based on this observation, they build FlexiWeb, a framework that supports network-aware middle-box usage. In addition, FlexiWeb performs dynamic network-aware compression to provide further performance gain.

Butkiewicz et al. [7] characterize Web page complexity. They show a website's popularity is not a good indicator of its complexity, whereas its category does matter. For example, *www.google.com* is a popular website but not a complex one. They have also found news sites load more objects from more servers and origins than other categories. Their analysis show that number of objects and number of servers are the dominant indicators of page load time and variability in page load times, respectively.

Wang et al. [50] show that although SPDY has been designed to decrease transfer time because of it's use of a single TCP connection, but this feature is also detrimental under high packet loss.

Chapter 3

Design and Methodology

To overcome challenges addressed in §1, we design and build our own testbed. To address *dependency* challenges, we use WProf [49] which takes into account internal page dependencies and to address high *variances* in a page load process, we provide a controlled environment to run our experiments.

3.1 WProf

WProf is a tool built atop the open source Chromium browser and is able to infer dependency policies of the browser. WProf uses fine-grained timing of activities involved in a page load process to accurately extract the page *dependency graph* and the *critical path* inside such graph. We say an activity a_2 is dependent on a previously scheduled activity a_1 , if a_2 can be executed only after a_1 is being executed or completed.
Considering activities as nodes of a graph and inter-dependency relations as the edges of such graph, the resulting graph would be a DAG (Directed Acyclic Graph) which we call *dependency graph* hereafter.

Critical path is the sequence of activities which add up to the longest overall duration inside the dependency graph.

3.2 WProf-M

To perform critical path analysis on mobile devices, we use a version of WProf on Android Chromium Version 31.0.1626.0. As a first step, we repeat our browser instrumentation experiments on mobile browsers to infer the dependency policies on the Android Chromium browser. To infer dependency policies in networking loading and computational activities, we load carefully crafted test pages.

The set of Web pages we use to infer the dependency policies can be found at `wprof.cs.washington.edu/tests/`.

Next, we instrument the Android Chromium browser to record the timings of each page load activity and use the timing and the dependency policy to extract the dependency graph and compute the critical path.

3.3 Testbed Design

One of the challenges in studying effect of an optimization on page load time is that Web page loads have high variance [50]. The variance makes it non-trivial to isolate the performance of the mobile browser: if the page load times on the mobile browser is very different when an optimization is applied, the difference could be because of the optimization or because of page load variance.
We design an experimental testbed that minimizes variance when loading pages before and after optimizations. We make the following design choices:

- We serve pages from the local server. To this end, we download the entire page locally on the server and convert all the external links to local links. This minimizes variances caused by changes to the page.
- When comparing mobile and desktop browsers, we load the same page on the mobile and the desktop browser, rather than loading the mobile version of the page (or *mpages*) to perform more direct comparisons.
- We emulate different network conditions using a traffic controller to ensure that both the mobile and the desktop browser load pages under the same network conditions.

To ensure that the results we get from our controlled setting applies more broadly, we perform additional experiments where the three restrictions are removed; i.e., we load pages directly from the Web server, we load *mpages*, and we use real networks. Our additional experiments show that the conclusions we derive from our controlled experiments also apply more generally.

Figure 3.1 shows our experimental testbed. At the client side, we load Web pages on a phone running Android Chromium instrumented with WProf-M, and on a desktop running WProf instrumented Chromium. All Web page loads go through the experiment manager. The manager stores logs generated by WProf and WProf-M and configures the traffic controller and the Web server if needed. We use USB tethering to connect the mobile device to the experiment manager rather than connect using WiFi because we observed large variances in WiFi latencies.

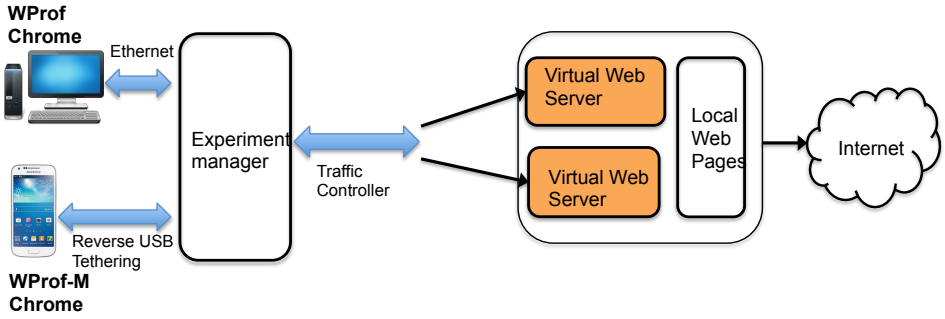


Figure 3.1: Testbed architecture

On the server side, we leverage virtual machines to run multiple web servers on the same platform. To isolate the different network stacks for the difference virtual servers, we use Linux network namespace [34], similar to [33]. To emulate different network conditions, we are using Linux Traffic Control (TC). Before each experiment, we run ping and iperf tools to test that the emulated network has the expected bandwidth and delay values.

3.4 Parameters

The parameters used in our testbed are as follow

Server and Client: All Linux instances are virtual machines running inside a VMware ESXi 6.0 Bare-Metal Hypervisor. On average, 4 cores at 2.6GHz and 2GB of RAM has been assigned to each virtual machine.

We use two Samsung Galaxy S4 phones running Android KitKat and Samsung Galaxy S6 phones running Android Lollipop. By default, we present experiments conducted on Samsung Galaxy S4 phones on the controlled testbed.

Network: We run the emulated network experiments on 6 different network profiles under the following bandwidth: 1Mbps, 5Mbps and 20Mbps. We experiment with two round trip delays: 50ms and 150 ms. We inject up to 2% packet loss rate based on real world studies [13].

Table 3.1 shows the different network profiles. Based on our lab experiments we categorize each pair of bandwidth and round trip delays as latencies seen in WiFi, 3G, and 4G networks.

Webpages: We experiment with 200 Web pages. We randomly choose 40% of the

	lab_WiFi	lab_3G	lab_4G
Average	b20-d50	b1-d50	b5-d50
Poor	b20-d150	b1-d150	b5-d150

Table 3.1: Different network profiles set up in testbed. b: bandwidth in each direction, d: round-trip delay time

Web pages from the top 200 websites in Alexa [4], 30% from the pages from the bottom of Alexa's 1 million web sites, and the remaining 30% from news websites on Alexa. We choose a mix of Web pages for the following reason: typically the popular top 200 Web pages on Alexa are smaller (for example, google.com) and are highly optimized. The performance of such Web pages may not be typical. Instead, we include unpopular pages in our mix because they are likely to not be optimized. We also choose news websites because they tend to be complex pages.

In the common case, we load the original page on both desktop and mobile. We perform addition experiments where we load mobile version of the page, that we call *mpage*. For example, `m.cnn.com` is an mpage, where the original page is `www.cnn.com`. Note that by default, mobile browsers always redirect to the *mpage*. We modify the user agent field to force the mobile browser to load the original page. We do this to directly compare the performance differences between mobile and desktop browsers.

Metrics:
We measure page load performance using the Page Load Time (PLT) metric. The Page Load Time metric is commonly defined as the time between when the page is requested and when the *DOMLoad* event is fired [49]. The DOMLoad event is fired when all objects are fetched, processed, and added to the DOM. There has been several alternate metrics to define page load performance such as the above-the-fold metric [1]. However, these alternate metrics are not easy to compute and are not yet widely used.

For each page load, we estimate the critical path. We divide the critical path into the following components:

- Computation: We sum the time taken by the following activities on the critical path: HTML Parsing, Javascript/CSS evaluation, and rendering. We call this the computational component of the critical path.
- Network: We sum the time taken to load each object on the critical path.

Chapter 4

Characterizing bottleneck in mobile page load

As discussed earlier in §1, lots of parameters are involved in a page load process. We also categorized activities involved in a page load, in two main classes: networking activities and computation activities.
In this chapter we start off by studying the importance of these two classes and observe how they effect the page load time. Then, we will look into the critical path and how that would be affected in different situations.
Characterizing mobile page load times would be our first step towards improving page load performance.

4.1 Computation is the bottleneck

In this section, we show the results of experiments we have run under different network profiles for our set of Web pages, as discussed in §3.4. We test both desktop and mobile browsers and show the main bottleneck in a page load process.

Our main observations are as follows:

- On mobile devices, when analyzing critical path, we observe that in almost all network profiles, ranging from *average lab_Wifi* to *poor lab_3G*, computation time, rather than networking time, is the dominant factor in the whole page load time . On contrary, on desktop browsers, networking time takes the majority of the page load time, independent of network profile.
- Computation still continues to be the bottleneck even when we switch to the new Samsung Galaxy S6 phones, with better computation resources.
- Computation is still bottleneck on mobile browsers when pages are loaded *in-the-wild*. i.e from the original Web server on real WiFi connections rather

than in our controlled environment.

4.1.1 Page load times

Figure 4.1 shows the page load times to load pages on desktop browsers, mobile browsers, and to load *mpages* on mobile browsers. These are the results of all Web pages and across all network profiles. We can observe that median of page load time on mobile browsers (both original pages and *mpages*) is two times greater than that of desktop's in the same network profile. The difference in tail is much higher. We see later (§4.1.5) that when loading pages on desktop browsers using Ethernet connection, the difference in page load time between mobile and desktop browsers is even higher.

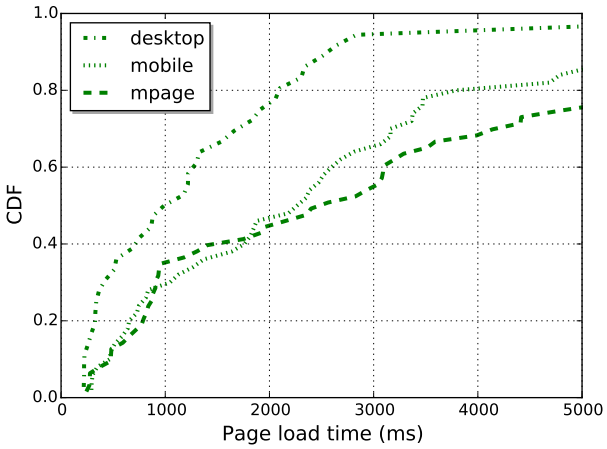
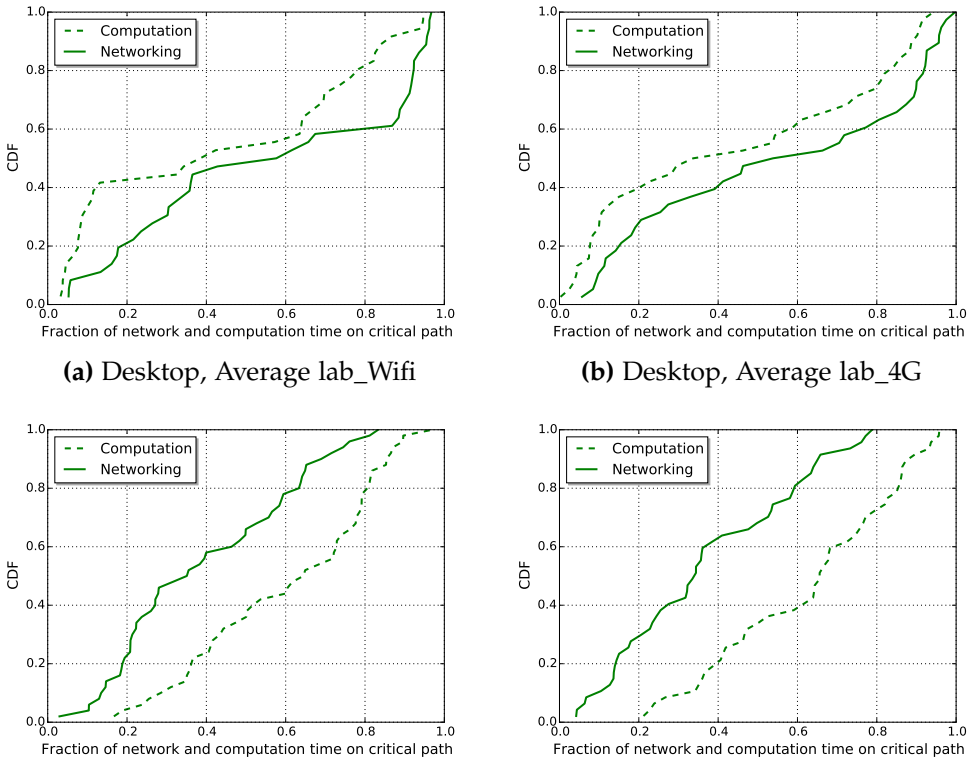


Figure 4.1: Comparing the page load times between loading original pages on mobile browser vs desktop browser, and loading *mpages* on the mobile browser.

Although *mpages* are smaller versions of the original pages, we do not see a significant difference between page load times when loading original pages and *mpages* on mobile. Reducing object sizes, would directly affect the download time for those objects but does not necessarily reduce the computation requirement.

4.1.2 Bottleneck in mobile vs desktop browsers

In order to pinpoint the bottleneck in a page load process, we load the same set of Web pages on desktop ad the mobile browsers, under the same network conditions. Figure 4.2 shows the fraction of computation activities on the critical path and the fraction of network activities on the critical path for desktop browsers. Figures 4.2a and 4.2b show that downloading objects takes 60% of the total delay on the critical path, while the computation activities is responsible for only 40% of the total delay on the critical path. This relation holds for different network profiles.



(a) Desktop, Average lab_Wifi

(b) Desktop, Average lab_4G

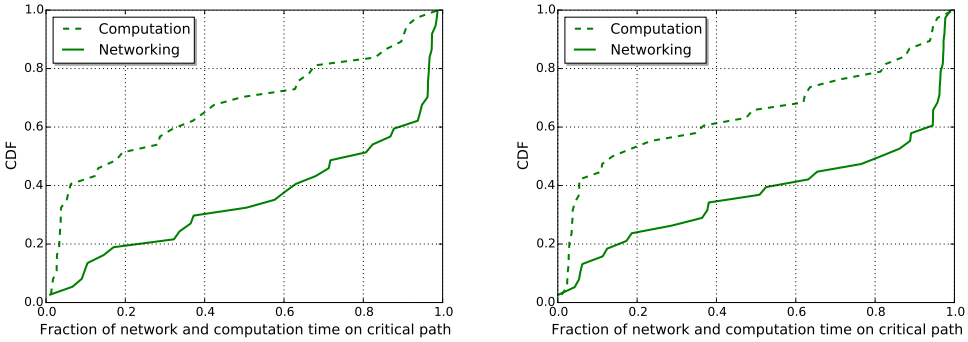
(c) Mobile, Average lab_Wifi

(d) Mobile, Average lab_4G

Figure 4.2: Average Wireless Connectivity: Fraction of network and computation time on critical path. Under average connectivity, computation is the main bottleneck for mobile browsers.

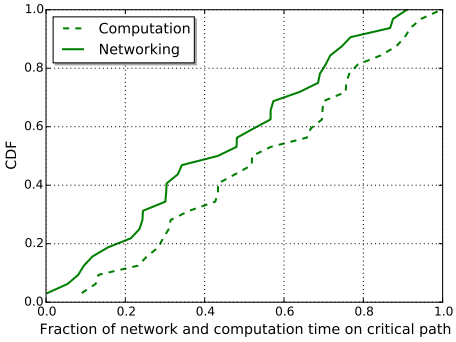
However, for mobile browsers computation time versus network activities time on the critical path is different. Figure 4.2c and 4.2d shows that on both the average lab_Wifi and the average lab_4G networks, the bottleneck is the computation activities when loading a page on mobile browsers. Computation accounts for over 60% of the page load time on the critical path in the median case when loading pages on mobile browsers. The results for the lab_3G network is quantitatively similar (not shown here).

Figure 4.3 shows the same bottleneck analysis but when the wireless network is of poor quality with longer round trip times of 150ms. Surprisingly, even when the network is poor, Figure 4.3c shows that computation remains a bottleneck for page load on poor lab_WiFi, accounting for 55% of the page load time in the median case. It is only in the poor lab_4G environment that the network becomes the bottleneck when loading pages on the mobile browser. On the other hand, on desktop browsers, poor wireless condition only makes the network bottleneck even more pronounced (Figures 4.3a and 4.3b).

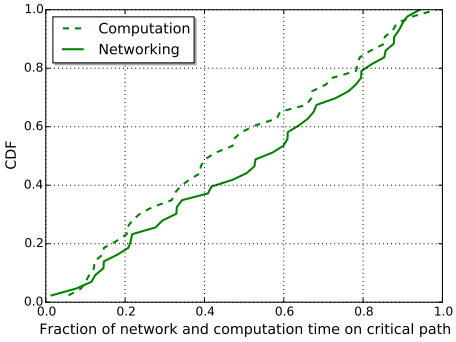


(a) Desktop browser, Poor lab_Wifi

(b) Desktop browser, Poor lab_4G



(c) Mobile browser, Poor lab_Wifi



(d) Mobile browser, Poor lab_4G

Figure 4.3: Poor Wireless Connectivity: Fraction of network and computation time on critical path. Even under poor network connectivity, computation is the bottleneck for mobile browsers. In contrast, under poor connectivity, network becomes even more of a bottleneck for desktop browsers.

4.1.3 Bottleneck when loading mpages

Figure 4.4 shows that when loading *mpages* on the mobile browsers, computation is once again the bottleneck. In the median case, more than 60% of the critical path is spent on computational activities in the median case.

4.1.4 Experiments on Samsung Galaxy S6

We ran the controlled page load experiments again on Samsung Galaxy S6, which has a 2100MHz CPU and octo-core, compared to the quad-core, 1700MHz Samsung S4 phones. Figure 4.5a shows that the page load time on S6 is not much different compared to the page load time on S4.
Interestingly, Figure 4.5b shows that the computation is still a bottleneck for the page load, even under a better CPU specification. This results suggests that browsers are not utilizing the additional CPU capacity in newer phones effectively.

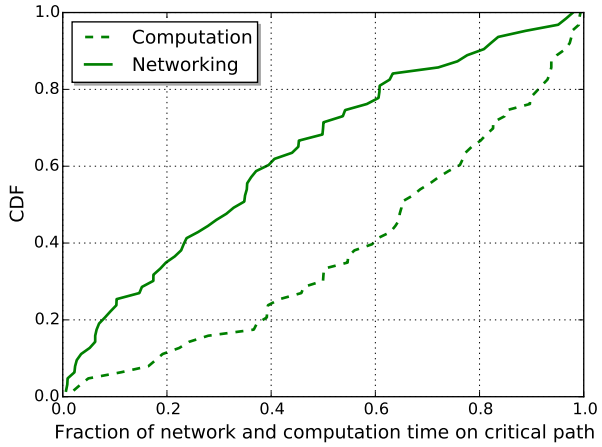
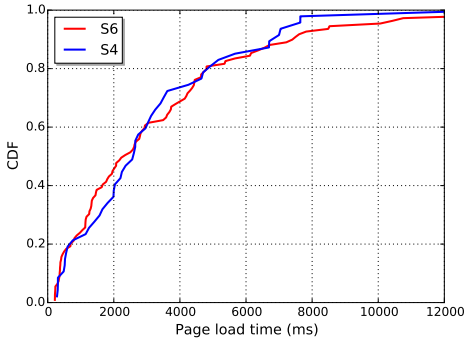
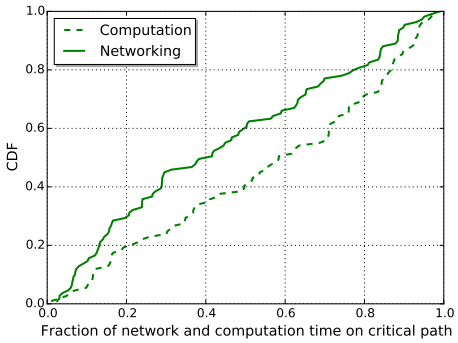


Figure 4.4: Loading *mpages* on lab_Wifi: The fraction of network and computation on critical path.

4.1.5 Experiments "in-the-wild"



(a) PLTs when loading web pages on Samsung Galaxy S4 and S6 phones. The pages were loaded in the average lab_4G network.



(b) Computation vs Networking on critical path when loading pages in Samsung Galaxy S6 phones under average lab_4G network conditions.

Figure 4.5: Results from experiments on Samsung Galaxy S6

Finally, we load web pages on desktop and mobile browsers outside of our experimental testbed. The web pages are fetched from the original Web server. The desktop browser uses the campus Ethernet connection (bandwidth 250Mbps), while the mobile browser uses the campus WiFi connection (bandwidth 30Mbps). The goal of this experiment is to study if the observations we make in the controlled setting also hold true in general. Note that as there are variances in networking parameters when running experiments *in-the-wild*, we cannot make generalizations in this case.

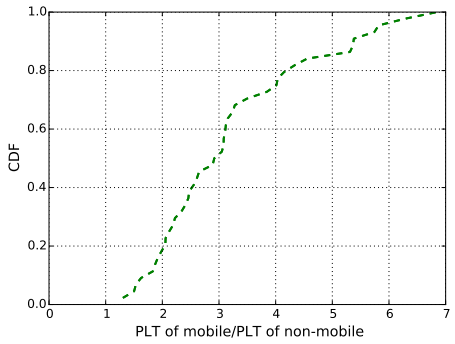


Figure 4.6: PLT difference between loading pages on mobile and desktop in-the-wild.

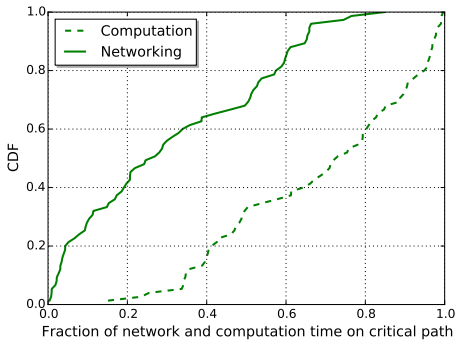


Figure 4.7: Mobile browser in the wild: computation vs networking on critical path.

Figure 4.6 shows the ratio of the time to load the page on mobile browsers and desktop browsers. The difference in page load time is even higher compared to the experiment in the controlled setting (Figure 4.1). Loading pages on the mobile browser is three times as slow as the desktop in the median case. This is largely because of the difference in network speeds between Ethernet and WiFi.
Surprisingly, Figure 4.7 shows that, similar to the controlled setting, the bottleneck in mobile browser remains computation.

4.2 Critical path analysis

Next, we study critical path in loading Web pages on mobile and desktop browsers. Summary of our observations are as follows:

- Although number of Javascripts, CSS, images, and HTML evaluations seems to be similar on critical path, when the same web pages are loaded on the same network profiles, but objects are not exactly the same. In other words, activity type distribution remains similar, while they are different activities.
- Each computation activity on the critical path, takes 4 times longer when is run on mobile browser compared to the desktop browser.

4.2.1 Object downloads

Figure 4.8 depicts the total number of bytes downloaded when loading the original page and loading the *mpage* on a mobile browser in all network profiles. Even though *mpages* are designed to be smaller, we find that for 60% of the pages, the total data downloaded remains the same. But for 30% of the pages, the difference in size is over 60%. The results suggests that *mpages* significantly reduce the size

of a small number of pages, but for a large fraction of pages, there is not much difference between *mpages* and the original page.

Figure 4.9 shows the bytes downloaded on the critical path for original page and *mpage*. Recall that only objects loaded in the critical path contribute to the page load time; other objects are loaded in parallel. Here we find that *mpages* reduce the number of objects loaded in the critical path for over 40% of the pages. In other words, *mpages* does reduce the network latency on the critical path.

However, the page load time does not reduce significantly when loading *mpages* compared to the original pages (see Figure 4.1). This is because the computation is the bottleneck when loading *mpages* on the mobile browser, as shown in Figure 4.4. In a separate experiment (not shown here), we find that loading *mpages* does not reduce the computation time on the critical path. In fact, the computation time is slightly worse.

4.2.2 Similarity metric

We load the same pages on mobile and desktop browsers, under the same network environments. This lets us compare the critical path of the page load on mobile and desktop browsers. The critical path consists of a series of activities such as loading an object, evaluating a Javascript, etc. In addition, each activity is associated with a unique URL corresponding to the activity. For example, the URL of the object to be loaded, or the URL of the Javascript to be evaluated.

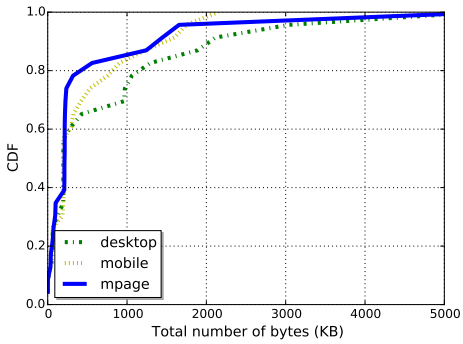


Figure 4.8: Number of total bytes downloaded for *mobile* and *mpage*

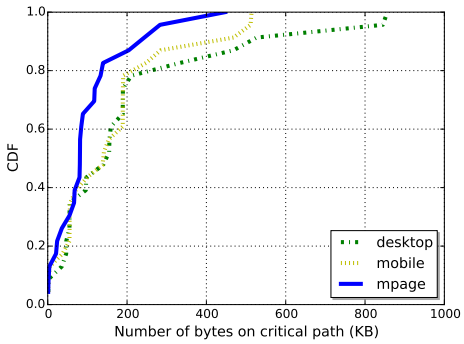


Figure 4.9: Number of bytes download on critical path for *mobile* and *mpage*

We define similarity metric as follows: the fraction of time the same <URL, activity> pair occurs both on the critical path of the mobile page load and the critical path of the desktop page load. If the number of elements on the critical path are not equal, we pad the smaller critical path with null activities.
Figure 4.10 shows the similarity metric across all network profiles. Even when the same page is being loaded under the same network profile, the critical path

is identical only for 20% of pages. For another 20% of the pages, only 50% of the critical path is similar. This result has big implications for optimization. It shows that optimizing a specific object, such as making a specific Javascript object smaller, may not have the same effect on mobile browsers as they would desktop browsers.

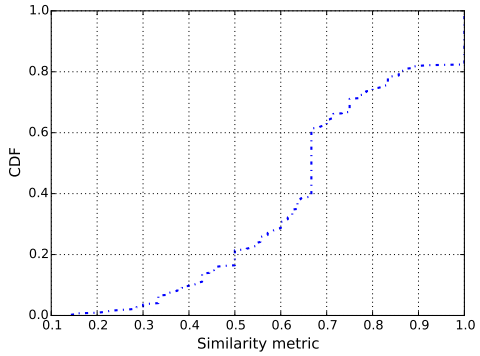


Figure 4.10: Similarity metric: the fraction of time the same <URL, activity> pair occur on the critical path of both mobile and desktop page load.

Next, we relax the definition of similarity and look at the percentage of time the same activity occur on the critical paths. For example, if there two Javascript activities on both critical paths, but the URLs corresponding to the Javascript are different, we still define the two activities as being similar. For this definition of similarity, we also include the experiments where we loaded the mobile version of the pages.
Figure 4.12 shows the percentage of each activity on the critical path across all pages and network profiles. In terms of the activities, we see that the critical path is very similar. In other words, for every page, the activities on the critical path in terms of Javascript evaluation, HTML parsing etc are similar on desktop and mobile browsers. But the critical paths differ in terms of, for example, which Javascript is being evaluated. One of the implications of this result is that, optimizations that target a class of activities, such as reducing the time to download all Javascript objects, are likely to provide benefits across mobile and desktop browsers.

4.2.3 Latency for each activity on the critical path

Finally, we measure the difference in latencies when performing the same activity on the mobile versus the desktop browser. To this end, we identify each activity on the critical path of either the desktop load or the mobile load. We then compare the latency of this activity when loading on the desktop versus loading on the mobile browser. Since we load the exact same page, an activity such as loading a specific Javascript will have to be performed both on the desktop load and the mobile load. Figure 4.12 shows the time take of each activity to be performed on the mobile

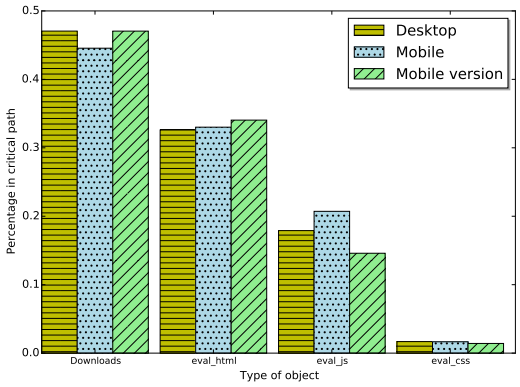


Figure 4.11: Fraction of different activities on the critical path.

versus the desktop browser. Each computation activity takes over 4 times longer to perform on the mobile browser versus the desktop browser. This is the core reason for the computational bottleneck on mobile browsers. For 50% of the web pages, it takes slightly longer to perform network activities on the mobile browser compared to desktop browser. This can possibly be because of a less optimized network stack on the mobile browser compared to desktop browser.

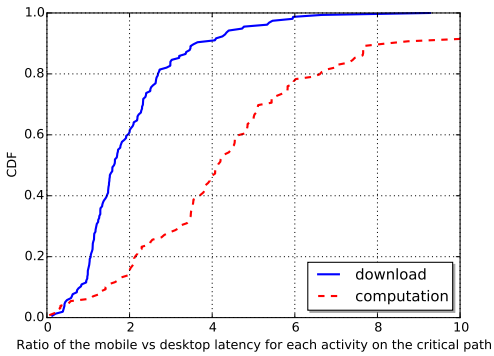


Figure 4.12: Time taken for each activity on the critical path to be performed on the mobile versus desktop browser. Results from loading pages on average lab_Wifi network.

Chapter 5

Page optimization

In pursuing our goal to improve page load time so far, we have characterized the main bottleneck in a page load process on mobile devices. Our next step is to study the effect of optimizations on the page load time. There has been several industry tools that also study the page and suggest optimizations. Google PageSpeed Insights [20] and Yahoo's YSlow [56] are two noteable current web optimization platform. Although they both are trying to suggest optimizations to improve the overall web site's performance, but they suffer from some major problems. Two main issues with Google PageSpeed Insights and Yahoo's YSlow solutions are as follows:

- They do not directly measure the benefit of an optimization on the page load time. Instead, they use a scoring mechanism that do not necessarily correlate well with the real page load improvements.
- They are using fixed set of rules and do not consider optimizing for each particular case.

  For example, PageSpeed Insights encourages users to inline (small) scripts in the HTML source file to avoid additional roundtrip times. On the contrary, YSlow recommends to make Javascript and CSS scripts external, to benefit from possible caching. This differences arise because of not considering the varying nature of servers, clients and network connections.
  We should also note that modification do not always help. For example, a popular improvement to reduce the download time is enabling compression in server side. Modern browsers support this feature automatically and decompress the received objects. Although this technique seems to be always beneficial, one should be careful about the gain achieved in reducing download time against increased computing time needed to decompress the whole session.

In this chapter, first, we study current web page optimization solutions, mainly Google's PageSpeed Insights and Yahoo's YSlow . Then we analyze these optimizations and study their effect on page load time using our testbed.
We then, illustrate building blocks of our future platform which scores each optimization based on it's contribution to the page load time. Having a platform that shows the effects of optimizations on page load time and considers the dynamic nature of servers, clients and network connections at the same time, would enable us to more precisely suggest customized optimizations for each particular case.

5.1 Current page optimization solutions

5.1.1 Google's PageSpeed Insights

PageSpeed Insights [20] is a tool that helps developers optimize their web pages by analyzing the pages and generating tailored suggestions to make the pages faster. PageSpeed Insights measures the performance of a page for mobile devices and desktop devices. It fetches the URL twice, once with a mobile user-agent, and once with a desktop-user agent. PageSpeed Insights analysis does not use real devices. It fetches a site with a webkit renderer (the same rendering engine that powers Chrome and Safari) that emulates both mobile device and desktop devices.

The PageSpeed Score ranges from 0 to 100 points. The higher the score, the better the performance. However, since the performance of a network connection varies considerably, PageSpeed Insights only considers the network-independent aspects of page performance: the server configuration, the HTML structure of a page, and its use of external resources such as images, JavaScript, and CSS. Implementing the suggestions should improve the relative performance of the page. However, the absolute performance of the page will still be dependent upon a user's network connection.

5.1.1.1 Architecture:

Developer makes a simple HTTP request to google PageSpeed Insights servers and gives a URL to fetch. PageSpeed Insights will fetch and render that URL, then runs the Page Speed library on it and gives back the result as depicted in figure 5.1

5.1.1.2 PageSpeed SDK

A webpage is fetched and rendered using *pagespeed_webkit* and then the defined rules in *lib/trunk/src/pagespeed/rules* are applied and the formatted results are pre-

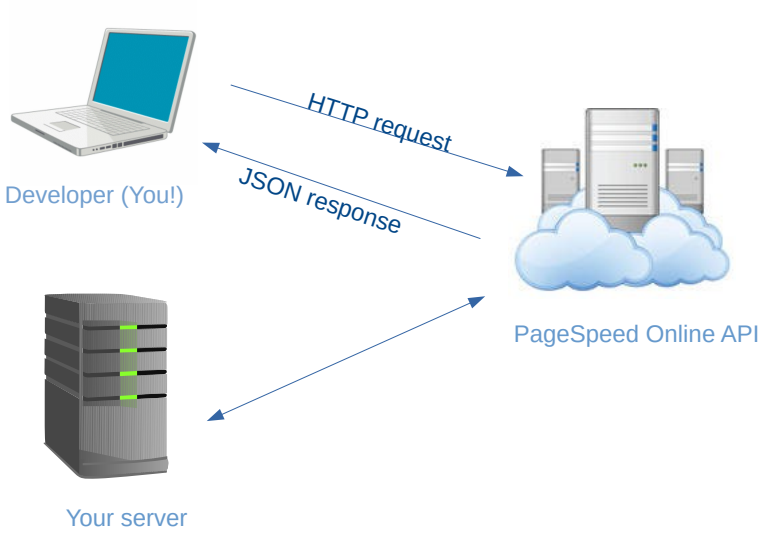


Figure 5.1: PageSpeed Insights online API

sented.
Results of all the rules are presented to the user. Each PageSpeed rule generates an impact number (an unbounded floating point value) that indicates the importance or priority of implementing the rule-result suggestions for the rule, relative to other rules.

For instance, if enabling compression would save 1MB, while optimizing images would save 500kB, the *"Enable Compression"* rule would have an impact value that is twice that of the *"Optimize Images"* rule. An impact of zero indicates that there is no suggestion for improvement for that rule. Following is a code snippet of a sample response of PageSpeed Insights API:

```
{
 "kind": "pagespeedonline#result",
 "id": "/speed/pagespeed",
 "responseCode": 200,
 "title": "PageSpeed Home",
 "score": 90,
 "pageStats": {
  "numberResources": 22,
```

```
  "numberHosts": 7,
  "totalRequestBytes": "2761",
  "numberStaticResources": 16,
  "htmlResponseBytes": "91981",
  "cssResponseBytes": "37728",
  "imageResponseBytes": "13909",
  "javascriptResponseBytes": "247214",
  "otherResponseBytes": "8804",
  "numberJsResources": 6,
  "numberCssResources": 2
 },
   "comment": "..."
   "MinifyJavaScript": {
    "localizedRuleName": "Minify JavaScript",
    "ruleImpact": 0.1417,
   "comment": "..."
 }
}
```


The improvements/fixes shown on PageSpeed Insights are sorted in decreasing order of their *"Rule Impact"* value. The one that has more Impact value will be shown first, which means that fixing it would have significant improvement in webpage performance.

5.1.1.3 Effect of PageSpeed Insights' rules on page load time

Goal of these experiments is to see how much effect the suggestions of PageSpeed Insights has on page load time. For this purpose, we have considered 5 websites for which PageSpeed suggests to Inline JS/CSS, Minify HTML/CSS/JS and Optimize images. For each website, entire website is downloaded locally on testbed and modified to implement three suggestions; Inlining, minification and optimizing images separately to see the impact of each change on page load times.
We performed these experiments with three different network profiles b1-d50, b5-d50 and b20-d50 where "b" stands for bandwidth and "d" stands for delay. Results for b20-d5- are only shown here.

1. **Inlining:**
   PageSpeed Insights suggests to inline only specific JS/CSS. Inlining should ideally help to reduce overall page load time, specifically total download time. Experiments were done with 4 websites of which inlining actually helped two and didn't help other two as shown in figure 5.2 . For first two sites page load time did not improve; in fact, worsened.

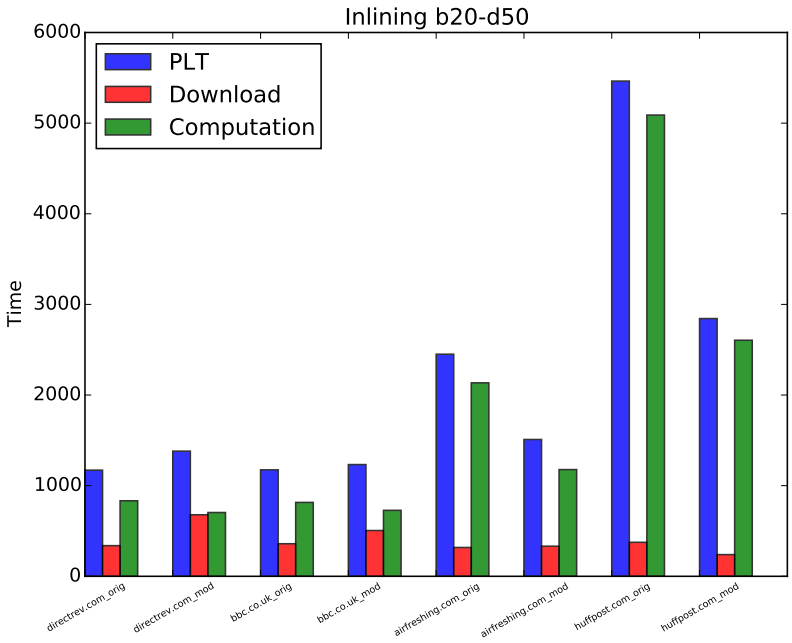


Figure 5.2: Page load time before/after inlining using PageSpeed insights

2. **Optimize Images:**
   PageSpeed Insights suggests to optimize few images so that their size is reduced which should eventually help reducing page load time. One disadvantage to this suggestion is size reduction is at the cost of high quality. Compressing images results in low quality which can affect user experience.

   As we observe from figure 5.3, optimizing images, helps to reduce page load time for *huffpost.com* and *airfreshing.com* while does not help in *youku.com* and *directrev.com* and even hurts in *bbc.co.uk*.

3. **Minification:**
   Page speed Insights suggests to minfiy JS/CSS/HTML to reduce the size of those files so that download time is reduced and page load time is improved. As shown in figure 5.3, minifcation alone is not always helping to reduce page load time. We can see again, minification has helped *directrev.com* , *airfreshing.com* and *huffpost.com* but, hurt *youku.com* and *bbc.co.uk*.

5.1.2 Yahoo's YSlow

Yahoo's YSlow [56] grades web page based on one of three predefined ruleset or a user-defined ruleset. It also offers suggestions for improving the page's perfor-

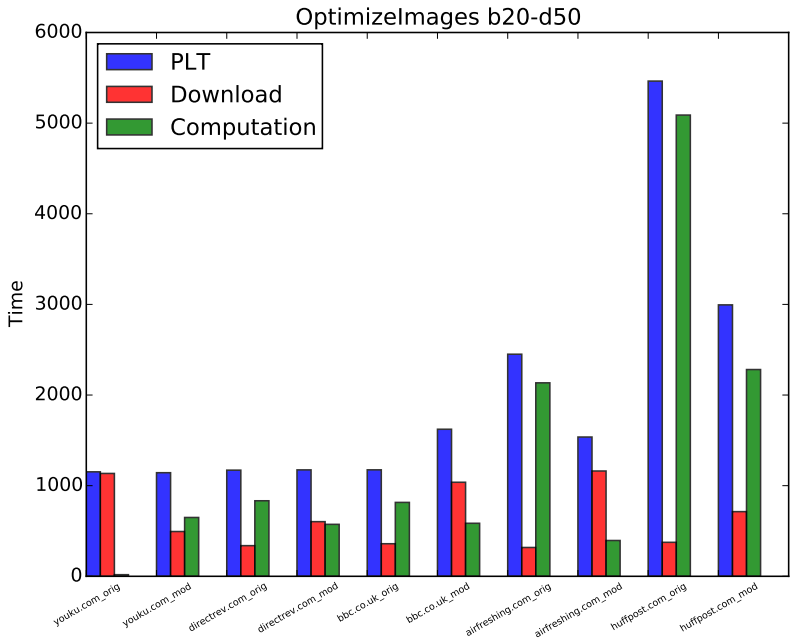


Figure 5.3: Page load time before/after optimizing images using PageSpeed insights

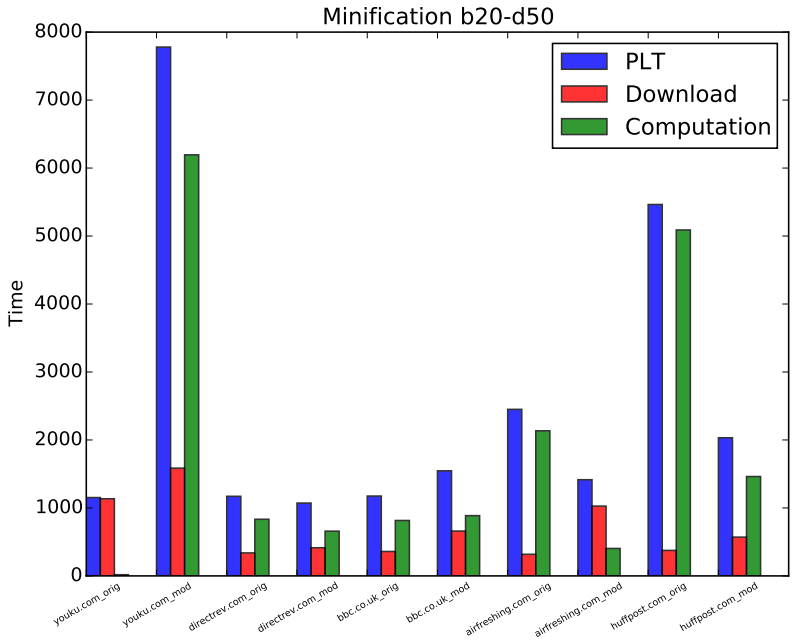


Figure 5.4: Page load time before/after Minification using PageSpeed insights

mance and provides tools for performance analysis.

YSlow works in three phases to generate its results.

1. YSlow crawls the DOM to find all the components (images, scripts, stylesheets, etc.) in the page. After crawling the DOM, YSlow loops through Firebug's [16] Net Panel components and adds those to the list of components already found in the DOM.

2. YSlow gets information about each component: size, whether it was gzipped, Expires header, etc. YSlow gets this information from Firebug's Net Panel if it's available. If the component's information is not available from Net Panel (for example, the component was read from cache or it had a 304 response), YSlow makes an XMLHttpRequest to fetch the component and track its headers and other necessary information.

3. YSlow takes all this data about the page and generates a grade for each rule, which produces the overall grade.

Figure 5.5 shows a sample analysis for *www.cs.stonybrook.edu* using YSlow.

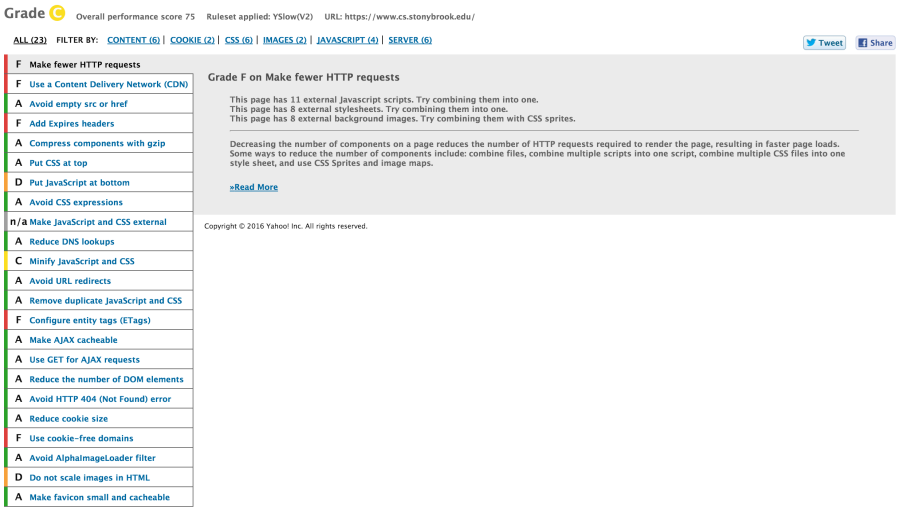


Figure 5.5: Sample YSLOW analysis for *www.cs.stonybrook.edu*

Since YSlow share the same problems with Google's PageSpeed Insight's(Fixed rule sets and unrealistic weights), the optimization results would be similar to the PageSpeed's Insights results.

5.2 Our Approach: PLTSpeed

In this section, we briefly discuss our approach in which we will score each optimization based on it's contribution to the page load time. We also take into account different network and client conditions when suggesting an optimization.

5.2.1 A framework to predict page load time

As we have discussed earlier (§3), WProf can capture the various activities performed during page load. We analyze these capture files to generate a JSON file for each page load process. Each JSON file contains the timing for all objects in a web page along with their dependencies. A sample snippet of such a JSON file is shown below.


```
{
  "n_download_no_trigger" : 1,
  "start_activity" : "download_0",
  "name" : "original.testbed.localhost_test4.html",
  "load_activity" : -1,
  "objs" : [
     {
        "when_comp_start" : 1,
        "url" : "http://original.testbed.localhost/test4.html",
        "id" : "r1",
        "comps" : [
           {
              "s_time" : 59.6280004829168,
              "time" : 12.5759998336435,
              "id" : "r1_c1",
              "type" : "evalhtml",
              "e_time" : 72.2040003165603
           },
           {
              "s_time" : 75.6310001015663,
              "time" : 0.867000780999703,
              "type" : "evalhtml",
              "id" : "r1_c2",
              "e_time" : 76.498000882566
           }
        ],
        "download" : {
           "receiveFirst" : 2,
```

```
            "len" : 295,
            "dns" : 0,
            "dnsEnd" : -1,
            "receiveHeadersEnd" : 9,
            "sslEnd" : -1,
            "connectEnd" : 6,
            "connectStart" : 0,
            "id" : "download_0",
            "receivedTime" : "11.2190004438162",
            "sslStart" : -1,
            "dnsStart" : -1,
            "receiveLast" : 2.21900044381618,
            "s_time" : 0,
            "sendEnd" : 7,
            "sendStart" : 6,
            "type" : "text/html"
        }
    }
```


These JSON files can be seen as structured representation of the whole page load. Now, if we apply an optimization to a Web page, the resulting JSON file will reflect the result of the applied optimization along with new timings and dependencies.

If we had access to the new timings beforehand, we could rewrite the original JSON file and produce the new JSON file. In other words, if we could predict the new computation and download time for each activity involved in a page load process, we were able to rewrite the original JSON file and predict the new page load time after optimizations has been applied. This process is illustrated in figure 5.6.
We process each of these new generated JSON files and calculate the critical rendering path. The length of the critical path is the page load time.

5.2.2 The prediction engine

To complete our framework from previous section, we now need an infrastructure to predict computation and download times after a certain parameter is changed. This parameter can be applying a new optimization or just a change in connection bandwidth.
For example, to predict new download time after a certain optimization is applied to a Web page, in our testbed, we can apply optimization to lots of locally served web pages. By actually running experiments before and after that optimization, we will be able to model download time for each object present in a page structure.

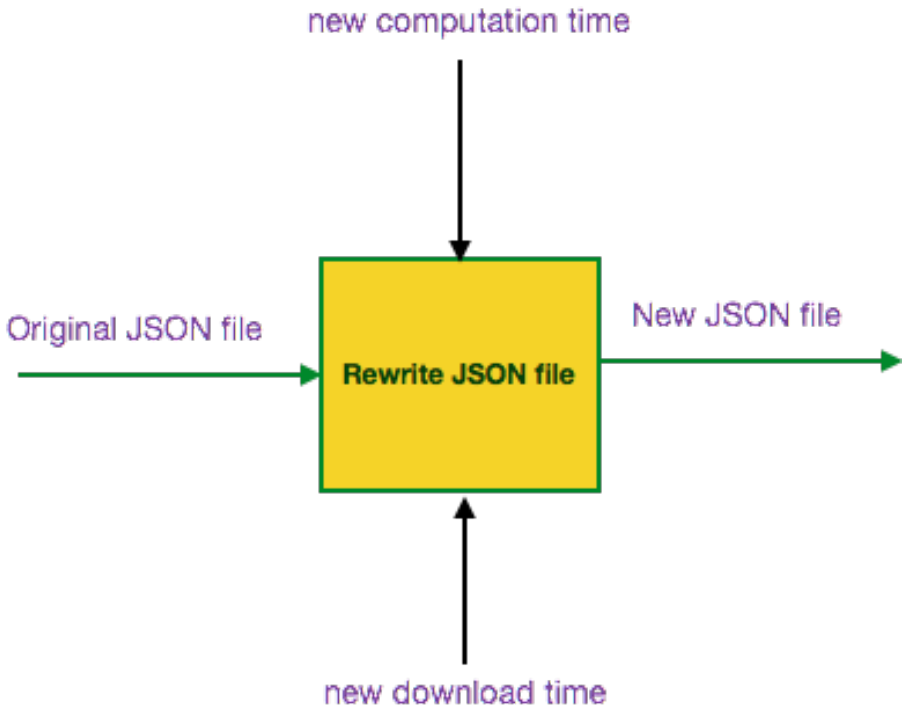


Figure 5.6: Rewriting the original JSON file to predict new page load time.

This model can predict the future network/computation time for each object. This feature (As discussed in previous section) enables us to manipulate the original JSON file and calculate the new critical rendering path time for a particular Web page without applying that particular optimization.
The block diagram for predicting new download times is depicted in figure 5.7

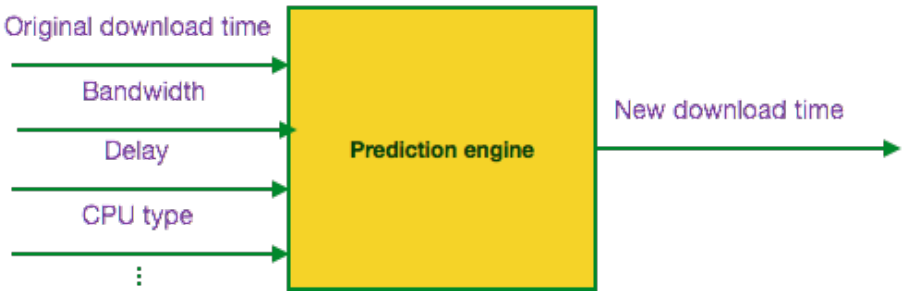


Figure 5.7: Prediction engine

We walk through an example to see how computation/networking time can be modeled for a certain type of modification.
We apply minification to an external CSS/Javascript and use our testbed to model

the impact of this modification. "Minification is the process of removing all unnecessary characters from source code without changing its functionality"[55].
The experiment is set up to run an emulated 4G connection. We use Alexa's top 200 web sites and in this case, experiments are run in a desktop environment. To ignore already minified scripts, we use a simple heuristic which looks for white spaces in the beginning of script source file. If there is no white space, we consider that script as "already minified".
The goal is to model computation and network time for minified scripts. We have the timing for all objects in the original web page. Using the testbed, we also have access to the timing for all objects in the new modified web page. This new modified web page has all of it's scripts minified. In order to model how minification affects the computation and networking time, we use linear regression modeling as our prediction algorithm.
In figure 5.8, the gray bars are length of the original *www.youtube.com* (before minification) and the color bars are for minified *www.youtube.com*. As we can observe, minification reduces length of objects on critical path in this case and leads to a decreased page load time.

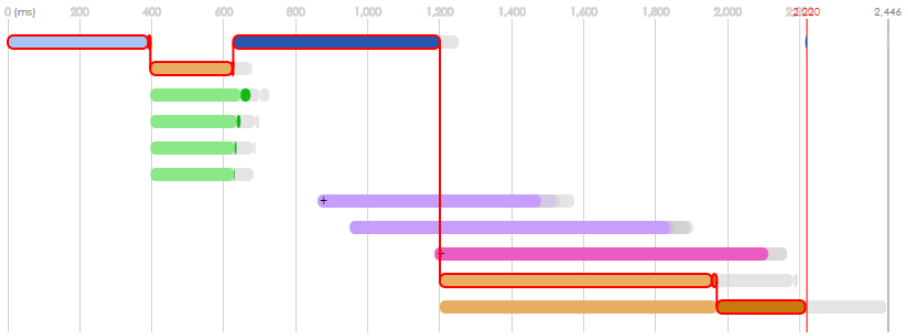


Figure 5.8: Page load time before/after minification on www.youtube.com

5.2.3 Future work

After completing our framework to predict page load time, in future, we will build a web based platform that suggests to users best optimizations applicable to their Web sites. This platform will consider optimization's contribution to page load time along with different networking conditions and user setups.
One should note that when combining optimizations, the whole gain/loss will not be equal to the sum of all optimizations. Hence, part of our future job will be to study the effect of combination of optimizations. For example, we may observe that minification alone, will improve page load time by 10% and inlining improves the page load time by 5% in a certain environment. Obviously we can not conclude

that when minification and inlining are both applied to a Web page, we will see overall 15% improvement in page load time.

References

[1] *Above the fold time*. http://www.webperformancetoday.com/.

[2] Victor Agababov et al. "Flywheel: Google's Data Compression Proxy for the Mobile Web". In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. USENIX Association, 2015.

[3] Bernhard Ager et al. "Web Content Cartography". In: *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2011*.

[4] *Alexa*. http://www.alexa.com/.

[5] Duc Hoang Bui et al. "Rethinking Energy-Performance Trade-Off in Mobile Web Page Loading". In: *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. ACM. 2015, pp. 14–26.

[6] 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*.

[7] Michael Butkiewicz, Harsha V Madhyastha, and Vyas Sekar. "Understanding website complexity: measurements, metrics, and implications". In: *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM. 2011, pp. 313–328.

[8] Michael Butkiewicz et al. "Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/butkiewicz.

[9] Xiaomeng Chen et al. "Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization". In: *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. MobiCom '15. ACM, 2015, pp. 40–52. ISBN: 978-1-4503-3619-2. DOI: 10.1145/2789168.2790107. URL: http://doi.acm.org/10.1145/2789168.2790107.

[10] *Chrome developer tools*. https://developer.chrome.com/devtools/.

[11] *d3.js: Data Driven Documents*. http://d3js.org/.

[12] *DNS pre-resolution*. http://blogs.msdn.com/b/ie/archive/2011/03/17/internet-explorer-9-network-performance-improvements.aspx.

[13] Nandita Dukkipati et al. "Proportional Rate Reduction for TCP". In: *Proceedings of the 11th ACM SIGCOMM Conference on Internet Measurement 2011, Berlin, Germany - November 2-4, 2011*. 2011. URL: http://conferences.sigcomm.org/imc/2011/program.htm.

[14] *Ericsson Mobility Report June 2015: http://www.ericsson.com/res/docs/2015/ericsson-mobility-report-june-2015.pdf*.

[15] Jeffrey Erman et al. "Towards a SPDY'Ier Mobile Web?" In: *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT '13. New York, NY, USA: ACM, 2013, pp. 303–314. ISBN: 978-1-4503-2101-3. DOI: 10.1145/2535372.2535399. URL: http://doi.acm.org/10.1145/2535372.2535399.

[16] *Firebug*. http://getfirebug.com/.

[17] *Firefox Developer Tools*. https://developer.mozilla.org/en-US/docs/Tools.

[18] *Google developers: Analyzing critical rendering path performance*. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/?hl=en.

[19] *Google I/O 2014: Making the Mobile Web Faster*. https://www.google.com/events/io/schedule.

[20] *Google Pagespeed Insights*. https://developers.google.com/speed/pagespeed/insights.

[21] *HTTP Archive*. http://httparchive.org/.

[22] Junxian Huang et al. "Anatomizing application performance differences on smartphones". In: *Proc. of the international conference on Mobile systems, applications, and services (Mobisys), 2010*.

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

[24] Rupa Krishnan et al. "Moving Beyond End-to-End Path Information to Optimize CDN Performance". In: *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2009*.

[25] Zhichun Li et al. "WebProphet: automating performance prediction for web services". In: *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI), 2010*.

[26] *Make sure your site's ready for mobile-friendly Google search results*. https://support.google.com/adsense/answer/6196932?hl=en.

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

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

[29] James Mickens et al. “Crom: Faster web browsing using speculative execution”. In: *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI), 2010*.

[30] *Mobile Internet Usage Skyrockets in Past 4 Years to Overtake Desktop as Most Used Digital Platform*. http://www.comscore.com/Insights/Blog/Mobile-Internet-Usage-Skyrockets-in-Past-4-Years-to-Overtake-Desktop-as-Most-Used-Digital-Platform.

[31] *Mobile-only users surpass desktop-only users*. http://marketingland.com/mobile-only-users-surpassed-pc-only-users-in-march-comscore-126952.

[32] *mod_pagespeed*. http://www.modpagespeed.com/.

[33] Ravi Netravali et al. “Mahimahi: Accurate Record-and-Replay for HTTP”. In: *USENIX Annual Technical Conference 2015*. Santa Clara, CA, 2015.

[34] *Network Namespace*. http://blog.scottlowe.org/2013/09/04/introducing-linux-network-namespaces.

[35] Feng Qian, Subhabrata Sen, and Oliver Spatscheck. “Characterizing Resource Usage for Mobile Web Browsing”. In: *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*. MobiSys ’14. Bretton Woods, New Hampshire, USA: ACM, 2014, pp. 218–231. ISBN: 978-1-4503-2793-0. DOI: 10.1145/2594368.2594372. URL: http://doi.acm.org/10.1145/2594368.2594372.

[36] Feng Qian et al. “Web Caching on Smartphones: Ideal vs. Reality”. In: *MobiSys ’12*. Low Wood Bay, Lake District, UK, 2012, pp. 127–140.

[37] *Quic*. https://www.chromium.org/quic.

[38] Sivasankar Radhakrishnan et al. “Crom: Faster web browsing using speculative execution”. In: *Proc. of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2011*.

[39] Haichen Shen et al. “Enhancing Mobile Apps to Use Sensor Hubs Without Programmer Effort”. In: *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. UbiComp ’15. Osaka, Japan: ACM, 2015, pp. 227–238. ISBN: 978-1-4503-3574-4. DOI: 10.1145/2750858.2804260. URL: http://doi.acm.org/10.1145/2750858.2804260.

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

[41] Shailendra Singh et al. "FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers". In: *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. MobiCom '15. Paris, France: ACM, 2015, pp. 604–616. ISBN: 978-1-4503-3619-2. DOI: 10.1145/2789168.2790128. URL: http://doi.acm.org/10.1145/2789168.2790128.

[42] Shailendra Singh et al. "FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers". In: *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. ACM. 2015, pp. 604–616.

[43] Ashiwan Sivakumar et al. "Cloud is not a silver bullet: a case study of cloud-based mobile browsing." In: *HotMobile*. California: ACM, 2014.

[44] Ashiwan Sivakumar et al. "PARCEL: Proxy Assisted BRowsing in Cellular Networks for Energy and Latency Reduction". In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT '14. New York, NY, USA: ACM, 2014.

[45] *SPDY*. https://www.chromium.org/spdy/spdy-whitepaper.

[46] *TCP pre-connect*. http://www.igvita.com/2012/06/04/chrome-networking-dns-prefetch-and-tcp-preconnect/.

[47] *The Trace Event Profiling Tool*. https://www.chromium.org/developers/how-tos/trace-event-profiling-tool.

[48] *W3C: Document Object Model*. http://www.w3.org/DOM/.

[49] Xiao Sophia Wang et al. "Demystify Page Load Performance with WProf". In: *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI)*. 2013.

[50] Xiao Sophia Wang et al. "How Speedy is SPDY?" In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 387–399. ISBN: 978-1-931971-09-6. URL: http://dl.acm.org/citation.cfm?id=2616448.2616484.

[51] Zhen Wang et al. "How far can client-only solutions go for mobile browser speed?" In: *Proc. of the international conference on World Wide Web (WWW), 2012*.

[52] Zhen Wang et al. "How Far Can Client-only Solutions Go for Mobile Browser Speed?" In: *Proceedings of the 21st International Conference on World Wide Web*. WWW '12. Lyon, France: ACM, 2012, pp. 31–40. ISBN: 978-1-4503-1229-5. DOI: 10.1145/2187836.2187842. URL: http://doi.acm.org/10.1145/2187836.2187842.

[53] Zhen Wang et al. “Why are web browsers slow on smartphones?” In: *Proc. of the Workshop on Mobile Computing Systems and Applications (HotMobile), 2011*.

[54] *WebPagetest*. http://www.webpagetest.org/.

[55] Wikipedia. *Minification (programming)*. [Online; accessed 3-July-2015]. 2015. URL: https://en.wikipedia.org/wiki/Minification_(programming).

[56] *YSlow*. http://yslow.org/.

[57] Yasir Zaki et al. “Dissecting Web Latency in Ghana”. In: *Proceedings of the Internet Measurement Conference (IMC). Vancouver, Canada, November 2014.* 2014.

[58] Kaimin Zhang et al. “Smart caching for web browsers”. In: *Proc. of the international conference on World Wide Web (WWW), 2010*.

[59] Wenxuan Zhou et al. “ASAP: A Low-Latency Transport Layer”. In: *Proc. of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2011*.

[60] Yuhao Zhu, Matthew Halpern, and Vijay Janapa Reddi. “The Role of the CPU in Energy-Efficient Mobile Web Browsing”. In: *In IEEE Micro – Special Issue on Mobile Systems*. (EEE Micro’2015. 2015.

[61] Yuhao Zhu and Vijay Janapa Reddi. “WebCore: Architectural Support for Mobileweb Browsing”. In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA ’14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 541–552. ISBN: 978-1-4799-4394-4. URL: http://dl.acm.org/citation.cfm?id=2665671.2665749.