Understanding and Improving the Performance of Web Page Loads

by

Vaspol Ruamviboonsuk

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2020

Vaspol Ruamviboonsuk

vaspol@umich.edu

ORCID iD: 0000-0003-0256-1613

To my family.

Acknowledgments

I am grateful to have worked with Professor Harsha V. Madhyastha, who guided me through my Ph.D journey with kindness. His endless valuable guidance puts me in the correct direction to make progress and his attention to detail improves me not only as a researcher, but also as a person when looking at issues surrounding me. In addition, Harsha always makes himself available to help with not only on research problems, but also important decisions such as providing me with suggestions on career advice. Deciding to work with him is one of the best decisions I made.

I am also thankful for my collaborators, Ravi Netravali, who provided me with an additional guidance and perspective on how to tackle research projects, as well as, Simon Pelchat, Tom Bergan, and Michael Buettner, who showed me what it is like to work on cutting-edge research projects in an industry setting. I thank my dissertation committee members Professor Morley Mao, Mosharaf Chowdhury, and Steve Oney for providing valuable feedback on my dissertation.

I would like to thank members of BBB 4929, Chris Baik, David Devecsary, Zhe Wu, Muhammed Uluyol, Joseph Lee, Jingyuan Li, Xianzheng Dou, and Andrew Quinn, for all the fond memory working in the office, and all the trips to the water cooler in the kitchen. I also thank my Thai friends who made home feel closer than it actually is.

Most importantly, I am indebted to my family, Paisan, Valya, and Varis, as well as my girlfriend, Puntaree, for their unconditional support. They provide me with a safe place whenever I needed emotional support and invaluable guidance for every decision that I have to make. Without them I wouldn't be able to successfully complete this journey.

TABLE OF CONTENTS

List of Figures

List of Tables

ABSTRACT

The web is vital to our daily lives, yet web pages are often slow to load. The inefficiency and complexity of loading web pages can be attributed to the dependencies between resources within a web page, which also leads to underutilization of the CPU and network on client devices.

My thesis research seeks solutions that enable better use of the client-side CPU and network during page loads. Such solutions can be categorized into three types of approaches: 1) leveraging a proxy to optimize web page loads, 2) modifying the end-to-end interaction between client browsers and web servers, and 3) rewriting web pages. Each approach offers various benefits and trade-offs.

This dissertation explores three specific solutions. First, CASPR is a proxy-based solution that enables clients to offload JavaScript computations to proxies. CASPR loads web pages on behalf of clients and transforms every page into a version that is simpler for clients to process, leading to a 1.7s median improvement in web page rendering for popular CASPR web pages. Second, Vroom rethinks how page loads work; in order to minimize dependencies between resources, it enables web servers to provide resource hints to clients and ensures that resources are loaded with proper prioritization. As a result, Vroom halves the median load times for popular news and sports websites. Finally, I conducted a longitudinal study to understand how web pages have changed over time and how these changes have affected performance.

CHAPTER 1

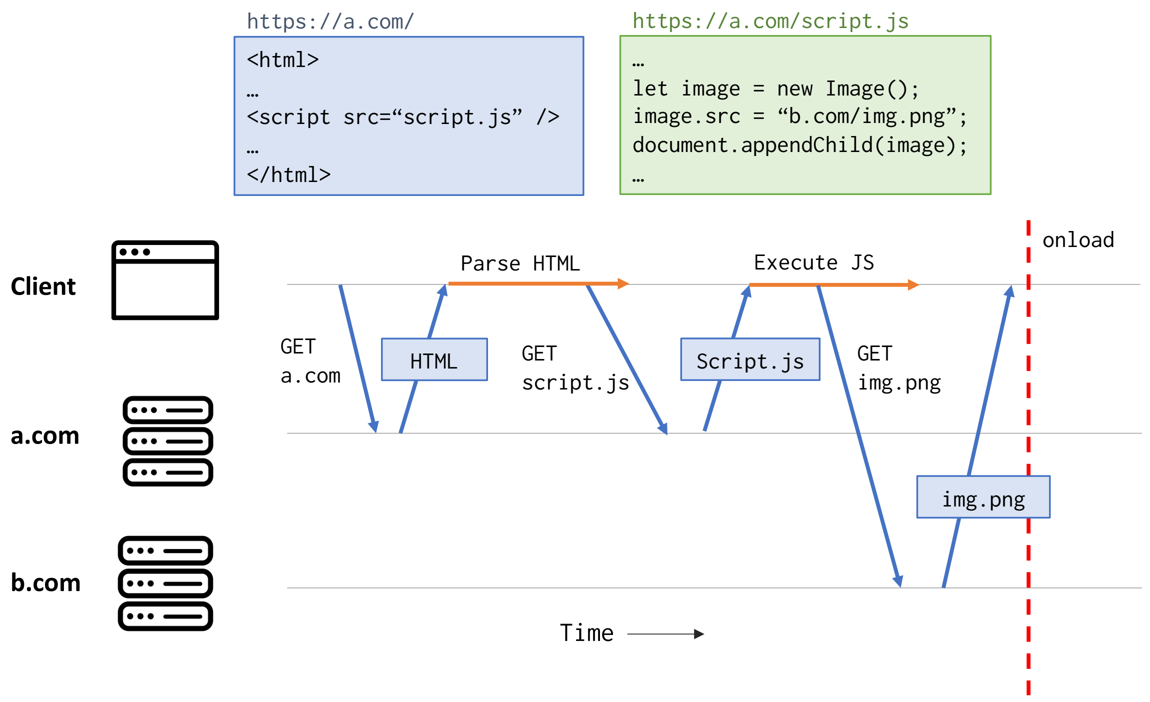
Introduction

The web is an integral of everybody's life as users use it to do various tasks such as access knowledge, keep up with the news, order a late-night meal, etc. While nobody can argue its importance, many websites are disappointingly slow, especially when loaded on mobile devices. In January 2018, Google observed that 70% of mobile landing pages took more than 5 seconds for all visual content to appear on the screen and more than 7 seconds to fully render the page [15].

Slow page loads is a critical issue as users are very sensitive to latency. Content providers with a slow web page are at risk of losing users and in turn revenue. A recent study has shown that the probability of users abandoning page loads almost double as the page load time grows from 1s to 5s [15]. In addition, many studies have shown that an increase in latency results in reduction in revenue. For example, Amazon.com saw a 1% reduction in sales with every 100ms increase in latency [5, 16].

1.1 Why are web pages slow to load?

Dependencies between resources lead to network and CPU underutilization. First, page load inefficiency can be attributed to the complexity of web pages. A web page comprises not just the page's HTML, but other resources such as images, CSS, and JavaScripts, etc. These resources can start a fetch of another resource, e.g., JavaScripts can be used to



```
<html>
…
<script src="script.js" />
…
</html>
```

```
…
let image = new Image();
image.src = "b.com/img.png";
document.appendChild(image);
…
```

Figure 1.1: An illustration of how a web page load works.

fetch images, creating a dependency between them. Many studies found that the inefficiency of page loads stems from dependencies between resources [83, 74, 81].

Figure 1.1 illustrates how a web browser loads a simple web page at `a.com`. First, it fetches the main HTML from the web server that owns the HTML, `a.com`. At this point, the browser has to wait on the network until the fetch is complete. Depending on the network conditions, the fetch can potentially require a long time to complete in a network constrained setting. Once the browser receives the HTML, it parses the HTML and then discovers and requests additional resources, e.g., `script.js`.

While processing the fetched resources, the browser can discover additional resources that can be embedded within other resources, e.g., a script can be used to fetch an image or an additional script. In Figure 1.1, after the browser fully fetches `script.js`, the browser executes it and starts the fetch of `image.png`. The browser can fetch `image.png` only after the browser executes the line of code that refers to it. The speed with which the browser can process a resource is dictated by the client's CPU—a client with a slower CPU will take

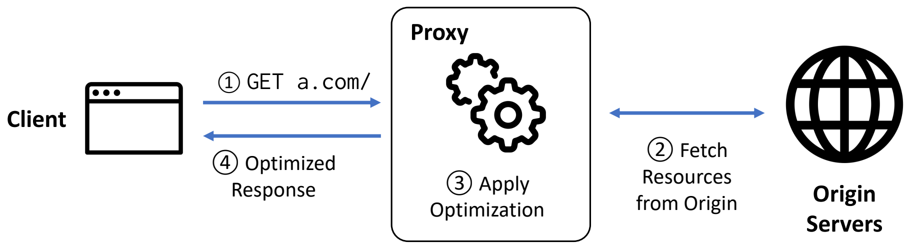


Figure 1.2: A typical architecture of a proxy-based solution.

more time processing a resource. The interleaving between resource fetches and resource processing creates a coupling between the use of network and the use of the CPU, i.e., computation can block resource discovery (and, vice versa).

Slow network and devices further slow down page loads. While the dependencies between resources is an issue regardless of the user's device, its impact is more pronounced in the mobile web setting because of the low compute power and poor network conditions. On one hand, less performant CPU slows down the processing of resources leading to an increase in the resource discovery delays at the client. Furthermore, additional CPU cores does not reduce the computation much for mobile browsers because web browsers execute most tasks in a single-threaded manner [63]. On the other hand, poor network conditions, i.e., high latency and low bandwidth, delay the receipt of resources, which causes the browser to wait longer before it can start processing resources.

1.2 Speeding Up Web Page Loads

Any solution that seeks to speed up page loads needs to address one or both reasons that cause web loads inefficiency. Generally speaking, there are three classes of approaches.

Proxy-based Solutions. In this class of solution, the client offloads computation to proxies leveraging the powerful CPU and well provisioned network of the proxy. Figure 1.2 depicts a typical architecture of a proxy-based solution. Instead of directly interacting with the origins, the client sends requests through a proxy server. In prior work, proxies use the information from page loads at the proxy in the following ways.

- **Reducing network delay penalties by applying network optimization at the proxy.** Solutions such as Google Chrome's Data Saver proxy [57] apply a myriad of resource-level optimizations to reduce the penalty due to network delays. For example, the Data Saver proxy serves as a resource cache; if a requested resource is present in the cache, the proxy can simply return the response. It can also minimize the size of each requested resource (e.g., when a JavaScript file is not compressed from the origin server, the proxy will compress it before sending it back to the client).
- **Minimizing dependencies between resources using remote dependency resolution.** Solutions using this approach such as Parcel [82], and Cumulus [78] use the proxy to send the dependent resources that it discovers during the page load to the client. The client stores the received resources upon receiving it from the proxy so that it can be immediately used when the client discovers the resource during the page load. Since the proxy can load the page at a much faster speed than the client, it is likely that when a client discovers a resource it will already be present in the client's local cache. Thus, the client can use the resource with minimal the network delays.
- **Reducing bandwidth consumption and client-side computation by snapshotting web pages.** Solutions employing this approach such as Shandian and Prophecy [85, 76] use the proxy to *snapshot* web pages by removing unnecessary state from the final page load. Snapshotting web pages speeds up the web especially for users with slow network and devices by reducing both computation and bandwidth requirements to load the page. The proxy then sends the snapshotted version of the page to the client, instead of the original page.

While proxy-based solutions enable significant improvements in page load performance, they are subject to two fundamental drawbacks. First, clients must trust that proxies preserve the integrity of HTTPS resources. Second, clients must share their cookies with proxies to enable the proxies to appropriately handle personalized content.

Modifying Client-Server Interactions. To address the limitations associated with the use of web proxies, solutions such as Polaris [75] and Vroom [81] reduce the dependencies between resources by altering the end-to-end interactions between clients and web servers. Unlike proxy-based solutions, clients directly contact origin servers to request for resources. In addition to the requested content, origin servers also include in their responses additional metadata for optimizing the page load. For example, a web server can include a high-level dependency structure of the page and the client can use that structure to prioritize fetches of resources that on the critical path. Since clients are directly contacting origin servers, this class of solutions preserve the end-to-end nature of the web, thus, ensuring that security and privacy guarantees still hold.

Such solutions require that content providers need to integrate the logic for generating the metadata for optimizing page loads into their serving infrastructure. This can be an involved effort, thus, leading to friction in adoption.

Rewriting Web Pages. The extent to which solutions compatible with legacy web pages can speed up the web is hindered by how pages are written. Therefore, a third class of solutions relies on modifications to web pages using off-the-shelf optimization primitives to both reduce the dependencies between resources and client resource usage. For instance, to reduce dependencies between resources, content providers can aid dependency discovery by including `<link rel="preload">` along with the main HTML to trigger resource fetches at the client without needing to process other resources. To reduce client resource usage, content providers can, for example, enable resource caching to reduce the bandwidth requirements to load the page as the client can simply reuse resources from its local cache, precluding the impact of the network. However, the friction of deploying this type of solution is high, as some optimizations may require removal of some page functionality. Thus, each website provider faces a trade-off between preserving functionalities on its web pages and ensuring fast page loads.

1.3 Thesis and Contributions

This dissertation supports the following thesis: *client-side CPU and network often go underutilized in web page loads; this underutilization can be reduced to speed up page loads or leveraged to add content to pages without any performance penalty.*

I explore solutions guided by the classes of the approaches described in Section 1.2. My dissertation work makes the following contributions.

Optimizations to CASPR **snapshot rendering.** We begin by exploring the use of HTTP proxies to snapshot web pages thereby reducing bandwidth usage and computation requirements for page loads. CASPR is a solution deployed at the Google Chrome Data Saver proxies [57] and works by moving the execution of JavaScripts to the cloud. When a user loads a web page through the Data Saver proxy, the proxy starts the load of the same web page, captures a "snapshot" of the page which contains the results of the page load at the proxy, and sends the snapshot to the client. While CASPR snapshots are lightweight, they still suffer from inefficient rendering. My primary contribution is on improving the rendering performance of CASPR snapshots. The optimizations lead to a 50% improvement in rendering performance for the median web page.

An end-to-end page load optimization framework with Vroom**.** While CASPR substantially improves rendering of page loads, it still suffers from the drawbacks of a proxy-based solution. Given the steady increase in HTTPS adoption [29], CASPR may be less applicable to users in the future.

With Vroom, we propose a rethink of the interaction between clients and servers with the goal of minimizing the coupling between resources while preserving the end-to-end nature of client-server interactions. Vroom alters page loads by having web servers preemptively *hint* resources that are needed during the page load to the client, which allows the client to discover resources without having to wait to process other resources. This helps decouple the client's use of the network and CPU. To discover the dependencies to hint, VROOM compliant web servers run a dependency resolution that only hint resources that will be part of the page load. To effectively use the hints, the client must be judicious in using the network (i.e., it cannot simply fetch all hinted resources at the same time). So, I implemented a scheduler that prioritizes critical resources (e.g., synchronous JavaScripts, CSS) over less critical resources (e.g., images). Our evaluation shows that VROOM speeds up the Alexa top news and sports websites by 50% for the median site.

A longitudinal study of web pages on the relationship between complexity and performance. In the final part of my thesis, I conduct a longitudinal study between 2016 and 2019. I study how the landing pages of popular websites have changed over the years and what has been the corresponding impact on performance? Between 2016 and 2019, web pages are roughly equally split based on the amount of content that has changed on the page. We observe that the changes in web page complexity and the changes in page load performance are not intuitively correlated. In particular, web pages that see an increase in page content do not see a proportionate degradation in page load performance. Our key finding is that fetches that occur serially (e.g., fetches of resources from JavaScripts and HTTP redirections), which are known to slow down page loads, enable web pages to include additional content without proportionately degrading performance. However, serialized fetches also prohibit web pages that observe a decrease in complexity from experiencing improvements in page load performance. Using the insights gathered from our study, we implement and evaluate simple optimizations that can potentially speed up pages with a decrease in complexity without altering page functionality.

1.4 Organization

The remainder of this dissertation is organized as follows. In chapter 2, we explore prior work in the area. Chapters 3, 4, and 5 describe the listed contributions in detail. Finally, chapter 6 summarizes the dissertation and discusses avenues for potential future work.

CHAPTER 2

Background and Related Work

2.1 Background

We begin by presenting a range of measurements that illustrate the poor web performance today on mobile devices, estimate the potential to reduce page load times, and show that existing solutions are insufficient.

Problem: Poor load times. We demonstrate the slowness of the mobile web using two sets of websites: the Alexa US top 100 websites and the top 50 sites each in the News and Sports categories [3]; these popular sites apply known best practices such as minifying JavaScript content and eliminating HTTP redirects [21]. We load the landing page for each site five times on a Nexus 6 smartphone that is connected to Verizon's LTE network with excellent signal strength; we report median page load times.[1]

Figure 2.1 shows that the median site among the top 100 takes roughly 5 seconds to load, which is higher than the 2–3 second period that a typical user is willing to wait [66]. When considering News and Sports sites, which are more complex than the average site [59], the median load time is even higher, exceeding 10 seconds. Since the need for faster loads is particularly acute on News and Sports sites, we focus on these sites in the rest of this section.

Cause: Poor CPU/network utilization. We now consider how low page load times

[1]We compute page load time as the time between when a page load begins and when the `onload` event fires.

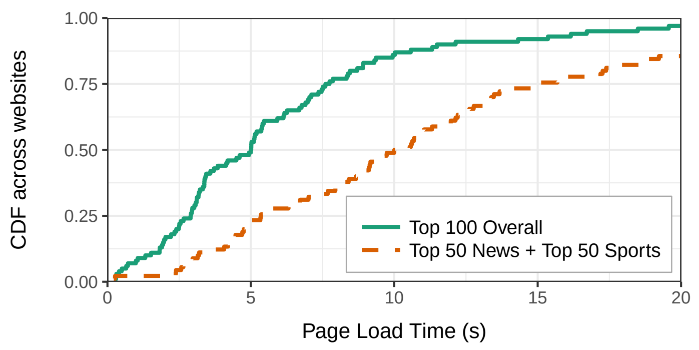


Figure 2.1: Page load times on today's mobile web.

can be reduced to without web pages being rewritten [2]. For a client to not have to trust proxies to execute page loads on its behalf, the client must fetch all resources on a page directly from origin web servers and locally process all of these resources. This places two constraints on web performance: the client's network connection and its CPU. To compute a lower bound on page load times, we estimate the potential gains from a redesign of the page load process that would fully utilize at least one of these two resources.

To mimic a setting where the network bandwidth is the bottleneck, we replay each page load after modifying the root HTML to list all resources required to load the page in a manner that instructs the browser to fetch these resources but not evaluate them. To emulate a setting where the client's CPU is the bottleneck, we load every page with the client phone connected via USB to a desktop which hosts all of the web servers. In both cases, we use Mahimahi [78] to record page content and replay page loads, and we use HTTP/2 between the client and all web servers in order to make efficient use of the network. We use the same mobile device and cellular network as above, and we further describe our replay setup in Section 4.5.

Figure 2.2 shows that, when exactly one of the network or the CPU is the bottleneck, rather than both limiting each other as is the case today, page load times on popular News and Sports websites are significantly lower than the status quo (the median load time drops from 10.5 seconds to 5 seconds). Our results also show that the CPU is typically the bottleneck in mobile page loads, corroborating the findings of recent studies [74].

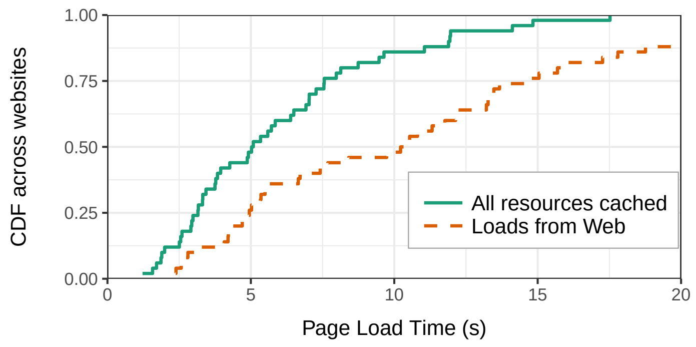


Figure 2.2: Potential for reducing page load times by making better use of the client's CPU and network.

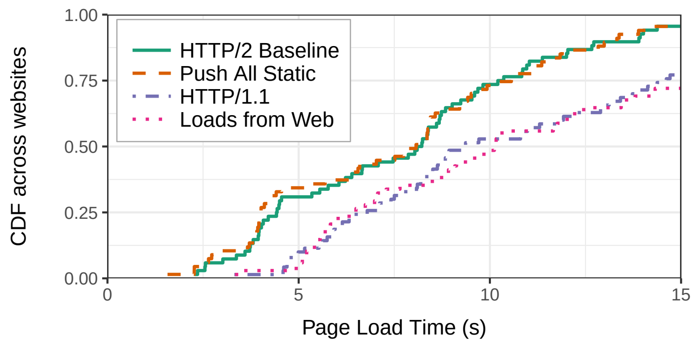


Figure 2.3: Estimation of page load time improvements that would be enabled once HTTP/2 is globally adopted. Given that the adoption of HTTP/2 is still in its nascency today, the fidelity of our replay setup is confirmed by the close match between load times measured when loading pages on the web and in our HTTP/1.1 replay environment.

Furthermore, page load times in the case where the CPU is the bottleneck remain largely the same even if we disable 1 of the 4 cores on the Nexus 6 smartphone, indicating that adding more cores will not help improve mobile web performance.

Existing solutions are insufficient. Since we seek a solution that improves mobile web performance while preserving the end-to-end nature of the web, we consider two existing solutions that satisfy this property.

First, we consider a setting where all domains on the web have adopted the latest version of the HTTP protocol, HTTP/2. HTTP/2 reduces inefficiency in the use of the network by enabling requests to be multiplexed on the same TCP connection. To estimate the potential impact of HTTP/2, we replay page loads in an environment where HTTP/2

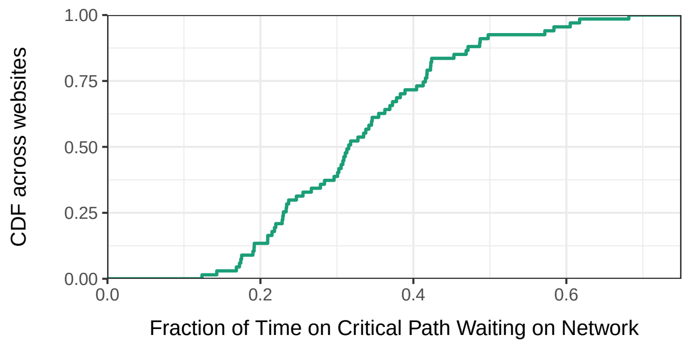


Figure 2.4: Fraction of the critical path spent waiting for the network, when the client uses HTTP/2 to communicate with all domains.

is universally used (“HTTP/2 Baseline"). The results in Figure 2.3 portend that, though the global adoption of HTTP/2 will reduce the median page load time across popular News and Sports websites to roughly 8 seconds, mobile web performance will remain significantly short of optimal; the lower bound we saw in Figure 2.2 was more than 2 seconds lower, a substantial gap given that web providers have found that even a few 100 milliseconds of additional delay significantly reduces their revenue [34]. Configuring the first party domain of every page to push (leveraging HTTP/2’s server push capability) all static resources that it hosts offers little additional benefit, for reasons discussed later.

Performance with HTTP/2 falls significantly short of the lower bound because the root cause for high page load times remains: since both CPU-bound and network-bound activities are typically on the critical path of a page load [83], neither the client’s CPU nor its access link is utilized to capacity. The client browser cannot parse an HTML/CSS object or execute a JavaScript file until it has incurred the latency to fetch that resource, which in turn it can begin to do only after discovering the need to fetch that resource by parsing/executing another resource. Indeed, Figure 2.4 shows that a significant fraction of time on the page load’s critical path—over 30% on the median page—is spent waiting to receive data over the network, leading to under-utilization of the CPU, the bottleneck resource (Figure 2.2).

Summary. Together, the measurements in this section lead to the following takeaways:

- Page loads on mobile devices are currently significantly slow even for popular websites, particularly for sites in categories that have more complex web pages than others.
- The reduction in load times that we can expect from the adoption of existing end-to-end solutions will not suffice.
- However, we could potentially halve the median load time if the page load process were redesigned to more efficiently use the client's CPU and network.

2.2 Related Work

This section provides an overview of the prior work done in the area of optimizing the performance of web page loads. In particular, it discusses prior studies on how to measure the complexity of web pages, key metrics for measuring the performance of page loads, and existing optimizations to page loads.

2.2.1 Measuring the Complexity of Web Pages

A number of prior efforts study the complexity of web pages to better understand the implications of a page's complexity on its performance. Butkiewicz *et al.* [59] studied the overall complexity of web pages from various perspectives: such as number of bytes on the page, number of requests, number of domains, etc. In addition, prior work [70, 71, 64] also measured web complexity to understand security and privacy implications for the web ecosystem. For instance, Kumar *et al.* [70] conducted a study to show various attack surfaces stemming from the high complexity of web pages.

2.2.2 Measuring the Performance of Web Page Loads

Given the importance of fast page loads, various efforts have designed metrics to measure page load performance. For example, Google recently announced the Web Vitals project [52], where it lists a combination of three critical performance metrics that websites providers can use to ensure a positive user experience on their site. Generally speaking, we can characterize each metrics into the following categories based on what aspect of the page load that the metric measures: 1) completion of page loads, 2) visual rendering progress, and 3) page interactivity.

Measuring the completion of page loads. To measure the end of the page load, page load time (PLT) is an often used metric as it measures the time when the browser finishes loading and processing resources required for the onload event to be emitted. However, the key drawback of PLT is that it can be affected by fetches of resources that do not contribute to the visual aspect of the page load. For example, PLT can be affected by invisible 1x1 pixel images that are included in many web pages for analytics purposes [64]. Thus, this means that PLT does not capture a user's perception of how quickly a page loads.

Measuring rendering performance. Given the limitations of PLT, other metrics attempt to better capture rendering performance. Many metrics leverage screenshots of the page load to measure how fast web pages render. For example, the above-the-fold time [67] metric uses the screenshots to find the time when the portion of the page visible to users before they begin scrolling stabilizes. Speed Index [44] uses the screenshots to measure the "average" time it takes to render the above-the-fold content. While these two metrics can measure the rendering performance of page loads, using these metrics on the live web at scale is challenging as collecting screenshots from users is not trivial.

To measure the rendering performance of web pages from live users, websites have to instead use rendering events exposed by web browsers. State-of-the-art browsers provides websites with important rendering events such as First Contentful Paint (FCP) [18], the time at which the first image or text element is rendered, or the Largest Contentful Paint (LCP) [33], the time at which the largest element is rendered. Measuring rendering performance from real users is critical for evaluating page load performance as users are equipped with a wide-range of devices and network configurations, which leads to variance in performance.

Measuring interactivity performance. While rendering plays an important role in experiencing web pages, the full web user experience also includes interacting with pages. There have been various efforts in measuring the interactivity for web pages from both industry and academia. For example, Google introduced the Time-to-Interactive (TTI) [48] and First CPU Idle [19] metrics to measure the point during the page load that the page is ready to respond to user interaction. Vesper [77], on the other hand, identify the JavaScript and DOM state used for interactivity, measures the load progress of such state during the page load, and measures the average time in which state is loaded.

Developer tools for measuring page load performance. To help developers in reasoning and improving the performance of page loads, modern web browsers include developer tools that expose key performance metrics as well as an in-depth tracing log of page loads. For example, Google Chrome includes Chrome DevTools [10] and, similarly, Firefox has Firefox Developer Tools [17]. In addition, Google introduces Lighthouse [35]: a web page auditing tool that summarizes the efficiency of page loads via various key performance metrics and provides optimization recommendations to developers.

2.2.3 Optimizing Web Page Load Performance

The final body of prior work relates to optimizing web page loads, i.e., making web pages load faster. At a high-level, we can group the work into three categories based on whether they optimize for the network side, the compute side, of both sides of page loads.

Optimizing the network usage of page loads. First, we focus on approaches tackles the issue at the network protocol level. For example, HTTP/2 (formerly SPDY) [30] was proposed to address the inefficiencies of HTTP/1.1 such as head-of-line blocking as well as introduce performance enhancing features such as HTTP/2 push [31]. In addition, Google presented QUIC (soon to be HTTP/3) [12], a transport protocol over UDP that optimizes the establishment of TLS connections.

Another line of prior work uses proxies to make network utilization more efficient. For example, when users enable data saver mode on Chrome, Google proxies all HTTP requests through its Data Saver proxies [57]. The Data Saver proxies then apply network optimizing techniques such as compression, applying image transcoding (i.e., convert images to a smaller format), etc. In addition, another group of prior work that leverages proxies, such as Parcel [82] and Cumulus [78], tackles the impact of high network latency on mobile page loads by leveraging the compute power and well-provisioned network of proxies. In this type of solution, the client sends the requests via a proxy, which completes the page loads significantly faster than on a mobile client. Then, the proxy can package the discovered the resources and send all of them to the client, thereby making efficient use of the network and minimizing the impact of network latency.

Another group of solutions on improves network efficiency of web page loads by using some knowledge of dependencies within page loads. For example, Klotski [60], Polaris [75], and Gaze [69] construct a dependency graph of the page. These solutions can then prioritize fetches of resources that are more "important" (e.g., both Klotski and Gaze use HTTP/2 push [31] to preemptively send resources critical to user experience).

Approaches focusing on reducing computation at the client. Given the effects of slow network and devices, many prior system reduce the computation necessary at the client to load web pages. There are two primary lines of work in this space.

On one hand, the client can employ a thin client where it offloads the majority of computation on a proxy or web server. When a user uses this type of client, a headless browser is typically run in parallel at a proxy. The cloud browser loads web pages on behalf of the client, sends necessary inputs for the client to render the page and construct the page state, as well as receives commands (e.g., interactions) from the client. Examples of such solutions include Amazon Silk [6] and Opera Mini [39].

Another type of solution reduces client-side computation by converting a web page into a highly optimized snapshot. For example, Shandian [85] and Prophecy [76] use proxies and web servers to extract state necessary for users to use the web page (e.g., final DOM state, and final JavaScript heap state) and package the extracted state into a snapshot. Given the minimal state included in the snapshot, the size of a typical snapshot is typically substantially smaller than the unmodified version of the page; thus, reducing bandwidth usage and computation at the client.

CHAPTER 3

CASPR: **Enabling Cloud-Assisted Web Page Loads at Scale**

3.1 Overview

The mobile web now dominates web traffic, with studies showing that 50% of global web traffic originates from mobile devices, up from 25% in 2015 [45]. Much of this increase stems from the growth of mobile web usage in resource constrained environments.

However, a large portion of the users in emerging markets are likely equipped with slow phones and poor network conditions [62]. For example, in 2018, the average download speed in Indonesia is 4.52Mbps [46]. Furthermore, while the availability of 4G is 70% on average, users using 4G still experience an average latency of 70ms, and users experience an average latency as high as 160ms using 3G connections.

This chapter describes CASPR (Cloud-Assisted Speedy and Progressive Rendering), a technology which uses cloud rendering to improve the web experience for users on slow networks and devices. CASPR optimizes web page loads by rendering any requested page at a proxy server and sending the outcome of the rendered page to the client in the form of a "snapshot" which enables the client to restore a fully functional page. CASPR was built with two important goals in mind:

- **Reduce computation at the client.** With less powerful phones, computation be-

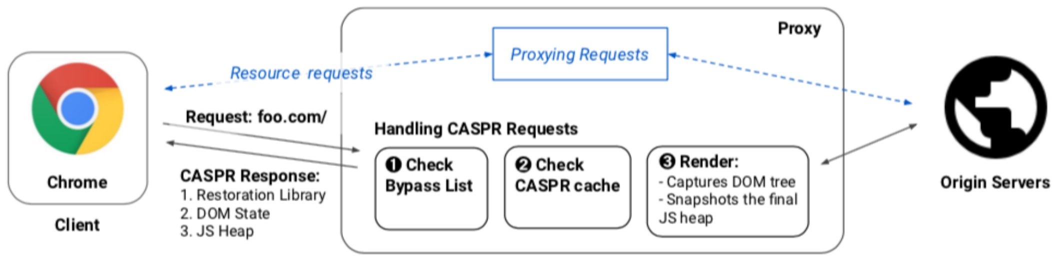


Figure 3.1: High-level architecture of CASPR

comes very constrained. Network optimizations alone are not sufficient.

- **Minimize shared state between the client and the server.** Strongly coupled state between the client and the server results in a system that is expensive to build and deploy, and where private information such as login credentials and other cookies are shared with the server. CASPR should not rely on shared state.

In this chapter, I describe my contribution to CASPR's design and implementation and its evaluation. I designed and implemented prototypes of two rendering optimization techiniques to CASPR, which were later incorporated into production at Google. The first technique accelerates CASPR rendering by incrementally sending rendering updates from the proxy to the client while using standard Web APIs. The second techique optimizes CSS delivery by removing 1) external fetches of CSS to reduce the impact of network latency, and 2) unused CSS rules to reduce bandwidth usage.

My evaluation of CASPR compares its performance impact in two extreme cache settings: an empty client cache and a pre-warmed cache from a previous load. The results show that the state of the client cache has a significant impact on CASPR's benefits; CASPR improves FCP by 1.8s when the cache is empty, but see no improvements when the cache is warm.

3.2 CASPR **Implementation**

CASPR accelerates page loads by moving JavaScript fetches and execution to a cloud browser. Users requesting a CASPR page make a request to a proxy server, which starts a headless cloud browser for rendering the page. The proxy server then starts rendering the page using the cloud browser and sends the state of the DOM, excluding JavaScripts, using a JSON-based protocol to the client browser. The client browser starts rendering the page immediately once it receives the updates from the proxy server. Once the page has been fully rendered at the proxy, the cloud browser generates a JavaScript file that restores the cloud browser's JavaScript state at the client. Figure 3.1 depicts the flow of a CASPR request.

This approach nicely satisfies our two goals. First, the client browser can render the page without executing any of the page's original JavaScript; since CASPR snapshots the JavaScript heap, it removes all original `<script>` elements from the final DOM tree. The heap restoration script is only fetched and executed once the DOM has been fully restored. Second, the snapshot is self-contained. The client can restore a fully functional page without the need to maintain shared state with the proxy. The client only needs to fetch resources, such as images, that are referenced in the DOM tree. Further, since CASPR represents the DOM tree with ordinary HTML and CSS and uses JavaScript to construct the page at the client, we can implement a CASPR service without adding custom rendering code to the browser.

While CASPR uses a variety of optimizations to accelerate page loads, this thesis focuses on two important contributions I made to CASPR. It does not cover details on capturing the final JavaScript state.

3.2.1 Streaming DOM Updates

One way to capture the DOM tree is to generate a DOM snapshot once the page load is visually complete. However, web pages can take a long time to reach visual completeness, especially when many external resources are needed. This can lead to an unpleasant experience as users will not see any progress until the DOM snapshot is finally generated and sent to the client. Prophecy's online mode [76] minimizes this problem by fetching all resources from local memory in the server, something we cannot do from a proxy. To enable users to see incremental progress as the page loads, unlike Prophecy and Shandian [85], CASPR sends back incremental DOM tree updates instead of a single monolithic snapshot.

One way to enable incremental rendering is by modifying the client browser so that the cloud-renderer can directly send rendering commands to the client. This approach is very challenging to deploy in practice because it introduces a strong coupling between the client and the proxy, which leads to prohibitive maintainance cost, e.g., modifying a feature can be very challenging because of client browser version fragmentation. Thus, we need a solution that does not introduce coupling between the client and the proxy.

Instead of enabling incremental rendering at the rendering layer inside the browser, CASPR relies on standard web technologies such as JavaScript and JSON to ensure that the solution is compatible across a large fraction of users. The incremental DOM updates are sent directly in the main HTML response. The proxy keeps the main response alive by not sending the closing `</html>` tag until the page is visually complete. In the cloud renderer, we use the `MutationObserver` API [37] to detect changes to the DOM tree, then we encode updates using JSON and append the JSON into the kept-alive response. JSON updates are interpreted by a small JavaScript runtime that is injected into the `<head>` of the HTML response.

3.2.2 Inlining CSS and Removing Unused CSS Rules

While enabling incremental rendering at the client allows users to have a more positive user experience, the overall rendering can still be improved. Recall that browsers block rendering when there is an outstanding fetch of CSS, so fetching external CSS can degrade rendering, especially on poor network conditions where latency is high and bandwidth is limited. We can divide the CSS delivery optimizations into two parts: 1) reducing the

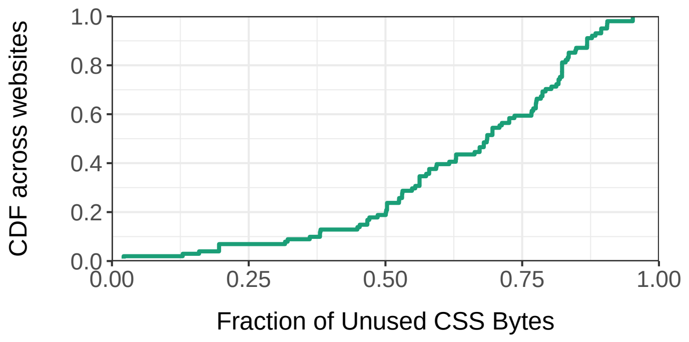


Figure 3.2: A large fraction of CSS bytes are not used.

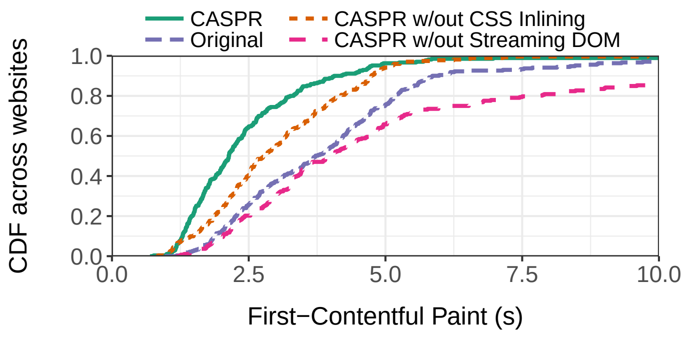
impact of network latency on fetching the CSS, and 2) reducing the impact of limited network bandwidth.

Minimizing impact of network latency by removing externally linked CSS. CSS files are often included in web pages as external links, using `<link rel="stylesheet">`, which requires the browser to fetch them over the network and fetching any resource without any existing connection requires at least two round-trips. In a very constrained network setting, the round-trip time can be as high as 150ms [46], which results in requiring at least 300ms before getting the first byte of the resource.

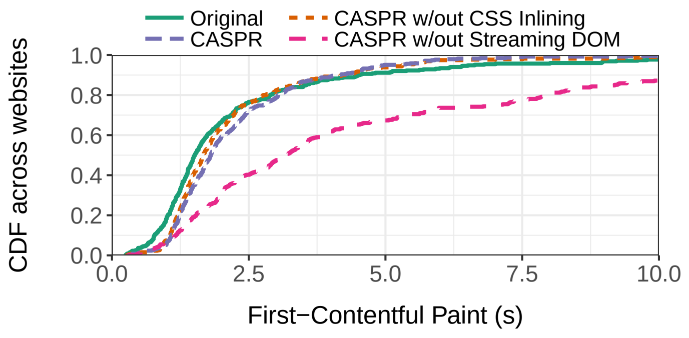
To minimize the effects of network latency, CASPR removes any references to externally linked CSS and inlines them with the main HTML using the `<style>` tag. By inlining the CSS, the client browser only needs to fetch the main HTML and does not need to incur additional round-trips to fetch CSS resources.

While inlining CSS can eliminate round-trips, CSS can no longer be separately cached from the HTML. A typical CSS file is often static and a recent study showed that the median CSS file can be cacheable for 1 year [9]. However, since CSS is now inlined with the main HTML, the caching policy of the CSS is now governed by the main HTML, which is typically not cacheable. Not being able to cache CSS can potentially lead to additional bandwidth consumption.

Minimizing impact of network bandwidth: removing unused CSS rules. To reduce bandwidth consumption, we leverage an observation that web pages often include a large



(a)



(b)

Figure 3.3: CASPR evaluation comparing (a) cold cache setting, and (b) warm cache setting on the First-Contentful Paint (FCP) metric.

number of unused rules—Figure 3.2 shows that 68% and 87% of the CSS bytes are left unused for the median and 90th percentile pages, which accounts for 76 KB and 558 KB of unnecessarily fetched bytes. In a setting with limited network bandwidth, reducing the amount of bytes required to load the page is beneficial to the client. Thus, CASPR reduces bandwidth consumption by removing unused CSS rules from the page and including only the used ones with the final snapshot.

3.3 CASPR **Evaluation**

Methodology. We evaluate CASPR in a lab setting using WebPageTest [53] with Moto G phones. We throttled the network to 400 Kbps bandwidth and 400 ms latency. Note that WebPageTest is not a replay setup; we load pages from the wild even in this controlled setup. Our corpus of pages includes one randomly selected page from each of the 500 most popular domains that are served CASPR pages in production. For each configuration, we load each page three times and randomly choose the result from one successful run. We consider only pages that have at least one successful run across all configurations, as WebPageTest can occasionally fail. Finally, we evaluate CASPR on two cache settings 1) *cold cache* (i.e., when the client cache is empty), and 2) *warm cache* (i.e., when the client cache is populated by loading the web page once right before evaluating CASPR in a subsequent back-to-back load).

Results Summary. Figure 3.3(a) compares fully-optimized CASPR with three other versions: CASPR with streaming DOM updates disabled, CASPR with CSS inlining disabled, and CASPR served from the proxy cache only. The most important optimization is streaming DOM updates—with the optimization, the median FCP improves from 4s to 2.1s, a 1.8s reduction. Without streaming DOM updates, CASPR loads are often slower than unoptimized pages.

Figure 3.3(b) shows that the overall benefits of CASPR are significantly less pronounced when the client cache is warm. There is little difference between an unoptimized page load and a CASPR page. When disabling streaming DOM updates, CASPR renders significantly slower than unmodified and Flywheel page loads with a 1.6s reduction in FCP for the median page. This further emphasizes that streaming DOM updates is a vital optimization.

Surprisingly, CASPR renders faster by 150ms when disabling inline CSS in a warm cache setting. We believe that this happens because most stylesheets are cacheable and are present in the browser cache. Whereas, with CSS inlining enabled, we give up the ability to cache stylesheets at the client and increase the size of the HTML. Since our network is throttled to 400 Kbps bandwidth, an extra 150 ms corresponds to about 7.5 KB of extra data.

We also evaluate CASPR on the Speed Index [44] metric as computed by WebPageTest, using the same methodology and corpus of pages. Unfortunately, due to a bug in Web-

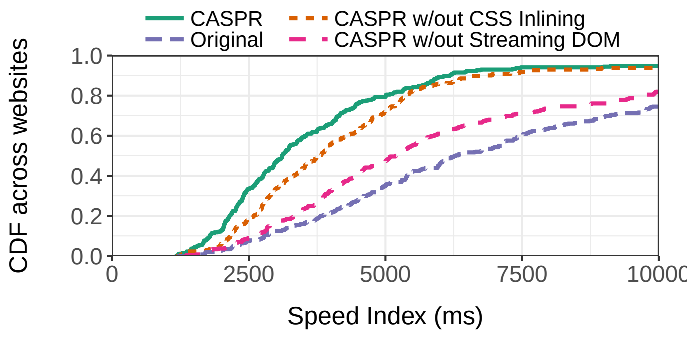


Figure 3.4: CASPR evaluation on the Speed Index metric.

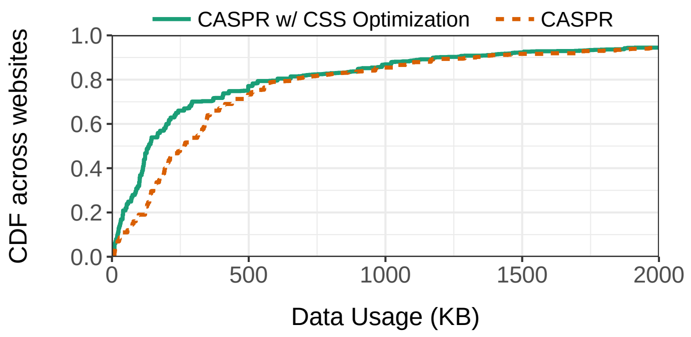


Figure 3.5: Distribution of network bytes fetched for CASPR with and without CSS Optimization.

PageTest, we could not measure Speed Index for warm cache page loads. We present cold cache results only.

Figure 3.4 shows that CASPR's effect on Speed Index is dramatically greater than its effect on FCP. On the median site, CASPR reduces Speed Index by 50%—from 6.3s to 3.1s. The dramatic speedup comes from the fact that CASPR pages can render without fetching or executing any JavaScript. CASPR has a higher reduction in Speed Index than FCP (50% vs. 40%) because, although it is common for pages to require JavaScript to reach FCP, it is even more common for pages to require JavaScript to reach a visually-complete status.

Figure 3.4 reaffirms that streaming DOM updates is the most important optimization. However, CASPR often has lower Speed Index than the original page–17% lower at the median–even with streaming DOM updates disabled. Once again, this illustrates the benefit of moving JavaScript fetch and execution to the cloud browser.

CASPR **Data Usage.** Figure 3.5 shows the impact of optimizing CSS for CASPR pages. While CASPR pages are already small, eliminating unused CSS further reduces the amount of bytes fetched. The median bytes fetched reduces from 265KB to 135KB, a 130KB reduction, while the 90th percentile sees a reduction of almost 150KB.

3.4 Summary

With a significant growth in users equipped with suboptimal smartphones and network, the Google Data Saver proxy team deployed a system that optimizes web pages into *snapshots*. In this chapter, I presented my contributions to improve the rendering of CASPR pages by enabling incremental DOM streaming and optimizing CSS delivery. My evaluation showed that these optimizations were able to significantly reduce FCP for the median web page by almost 45% when the client's cache is empty.

CHAPTER 4

Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution [1]

4.1 Overview

While CASPR is able to substantially improve the performance of page loads, it suffers from the drawbacks associated with a proxy-based solution; users must trust the integrity of HTTPS content served from proxies and must share the their cookies so that personalization is appropriately handled. With the recent increase in HTTPS usage [29], proxy-based solutions may not be as applicable to users as it was in the past. Thus, we explore optimizing page loads with the goal of preserving the end-to-end interaction between clients and web servers, enabling users to receive the benefits of faster page loads without the need to trust a third-party entity.

Recent studies [75, 83, 84] have found that dependencies between the resources on any web page are a key reason for slow page loads. Today, even mobile-optimized web pages include roughly one hundred resources [26] on average, and client browsers can discover each of these resources only after they have fetched, parsed, and executed other resources that appear earlier in the page. For instance, a browser may learn that it needs to fetch

[1]Appeared in: Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V. Madhyastha. 2017. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'17). Association for Computing Machinery, New York, NY, USA, 390-403. DOI:https://doi.org/10.1145/3098822.3098851

an image after executing a script which it discovers after downloading and parsing the page's HTML.

Prior work has taken one of two approaches to address the impact of these dependencies on web performance, and both approaches suffer from fundamental drawbacks.

- **Offloading to proxies.** In one class of solutions, when a client loads a page, discovery of resources on the page is offloaded to a proxy [78, 82, 85]. Solutions that take this approach attempt to reduce page load times by leveraging the faster CPUs and network connectivity of proxy servers. However, clients must trust that proxies preserve the integrity of HTTPS content; proxies that disregard HTTPS traffic are limited in the benefits they can provide given the increased usage of HTTPS [72]. Moreover, to preserve the ability of web providers to personalize content, a client must share with the proxy its cookies for all domains from which resources must be fetched to load the page.

- **Reprioritizing requests at client.** An alternative class of solutions [60, 75] lets the client itself discover all resources on a page. Instead of fetching resources in the order that they are discovered, these systems preferentially fetch certain resources (e.g., those that lead to longer dependency chains [75]) based on precomputed, high-level characterizations of the page's dependency structure.

  For example, with Polaris, the client receives a characterization of the page's dependency structure at the start of the page load and uses this knowledge to prioritize requests for more critical resources. However, such an approach can do little to reduce the network delays encountered on the critical path. The fundamental constraint with this approach is that the client must discover all resources on the page on its own (i.e., fetching a resource and then evaluating it to identify new resources to fetch). As a result, once the browser discovers a resource by parsing/executing other resources that appear earlier in the page load, the latency of fetching that resource must be incurred *at that time* before the browser can begin processing the resource. Since HTTP/2 eliminates head-of-line blocking and network bandwidth is not the bottleneck in mobile page loads as shown in section 2.1, reordering requests for discovered resources offers little benefit. We provide further evaluation and discussion of Polaris in Section 4.5.

The problem with these approaches is that, once the client browser discovers the need for a resource, the client must necessarily wait for that resource to be fetched over the network before it can start processing the resource, resulting in under-utilization of the CPU. Since the client CPU is the primary bottleneck when loading web pages on mobile devices ([74] and §2.1), this class of solutions has limited ability to speed up the mobile web.

These limitations of prior approaches motivate the need for a new solution that both preserves the end-to-end nature of the web and aids clients in discovering the resources on any page. Specifically, a client must receive every resource directly from the domain hosting that resource, thereby enabling the client to verify the integrity of HTTPS content and requiring the client to share its cookies for a domain only with servers in that domain. Yet, the new solution must also preserve the primary benefit of proxy-based dependency resolution, which is to decouple the client's downloading of resources on a page from the processing of those resources. Decoupling these functions maximizes resource utilization during page loads because fetching of resources is constrained by the network, whereas the CPU limits the parsing and execution of each fetched resource.

We argue that the way to realize this desired end-to-end solution is to redesign page loads such that web servers securely aid clients in resource discovery. In addition to returning a requested resource, a web server should inform the client of other dependent resources that the client will need in order to load the page. Though there are resource overheads associated with identifying these dependent resources, content providers have a strong incentive to incur this burden in order to decrease load times for their clients, thereby increasing revenue [73, 24]. We make three contributions in designing Vroom to realize this approach.

First, to aid clients in resource discovery, Vroom-compliant web servers not only *push* the content of dependent resources (leveraging the server push capability in HTTP/2 [58]) but also return *dependency hints* in the form of URLs for resources that the client should fetch. The use of HTTP/2 push alone is insufficient because content on modern web pages is often served by multiple domains [59], each of which can only securely push the content that it owns. In contrast, dependency hints enable a server to inform a client of the dependent resources that it should fetch from *other domains*, without providing the content of those resources. This additional input from servers ensures that a client's ability to discover and start downloading required resources is not constrained by the speed with which it can process fetched content. In fact, by the time the client discovers the need for a resource during its execution of a page load, that resource will likely already be in its cache.

Second, we develop the mechanisms that Vroom-compliant web servers must employ to identify the resources they should push and the dependency hints they should include with their responses. In contrast to prior efforts, which have relied exclusively on either online [78, 82] or offline [60, 75] dependency resolution, we show how to *combine* the two approaches to accurately identify the set of resources that a client will need to fetch within a specific page load. Critically, our design maximizes the number of dependent resources that the client is made aware of while avoiding sending dependency hints for intrinsically unpredictable resources—ones which vary even across back-to-back loads of the page—so that the client does not incur the overhead of fetching resources that are unnecessary for its page load.

Lastly, while the additional input from Vroom-compliant web servers reduces the latency for clients to discover all resources on a web page, fetching all resources as soon as they are discovered increases contention for the access link's bandwidth, delaying the receipt of some resources. To maximize CPU utilization, we leverage the property that resources that need to be parsed/executed (HTML, CSS, and JS objects) constitute only a quarter of the bytes on the average mobile web page [26]. We coordinate server-side pushes and client-side fetches such that resources that need to be parsed/executed are received earlier than other resources; the client can fetch the latter set of resources while processing the former set. In doing so, we ensure that resources arrive at the client in the order in which they will be processed.

Our implementation of Vroom enables the use of Google Chrome to load pages from the Mahimahi [78] page load replay environment. On a corpus of web pages from popular News and Sports sites, the median page load time reduces from the status quo of 10.5 seconds to 5.1 seconds with Vroom. These improvements stem from Vroom's ability to enable server-side identification of dependent resources with a median false negative rate of less than 5%, which in turn results in a 22% median decrease in client-side latency to discover all resources on a page.

4.2 Approach

Our results in Section 2.1 indicate that the key to optimizing mobile web performance is to maximize the utilization of the client's CPU. To do this, we need to ensure that the browser's processing of any resource is not delayed waiting to receive the resource over the network. To achieve this decoupling of the browser's use of the CPU and network, web page loads would ideally work as follows. When a client browser issues a request for a page, it would receive back *all* the resources needed to render the page, rather than just the HTML for the page. This would optimize page load performance for two reasons. On the one hand, in contrast to the status quo wherein the client incrementally fetches resources as it discovers them during the page load, receiving all of the objects on the page at once would maximize the utilization of the client's access link and eliminate the need for repeated latency-onerous interactions between the client and web servers. On the other hand, if the resources on the page are delivered in the order they need to be

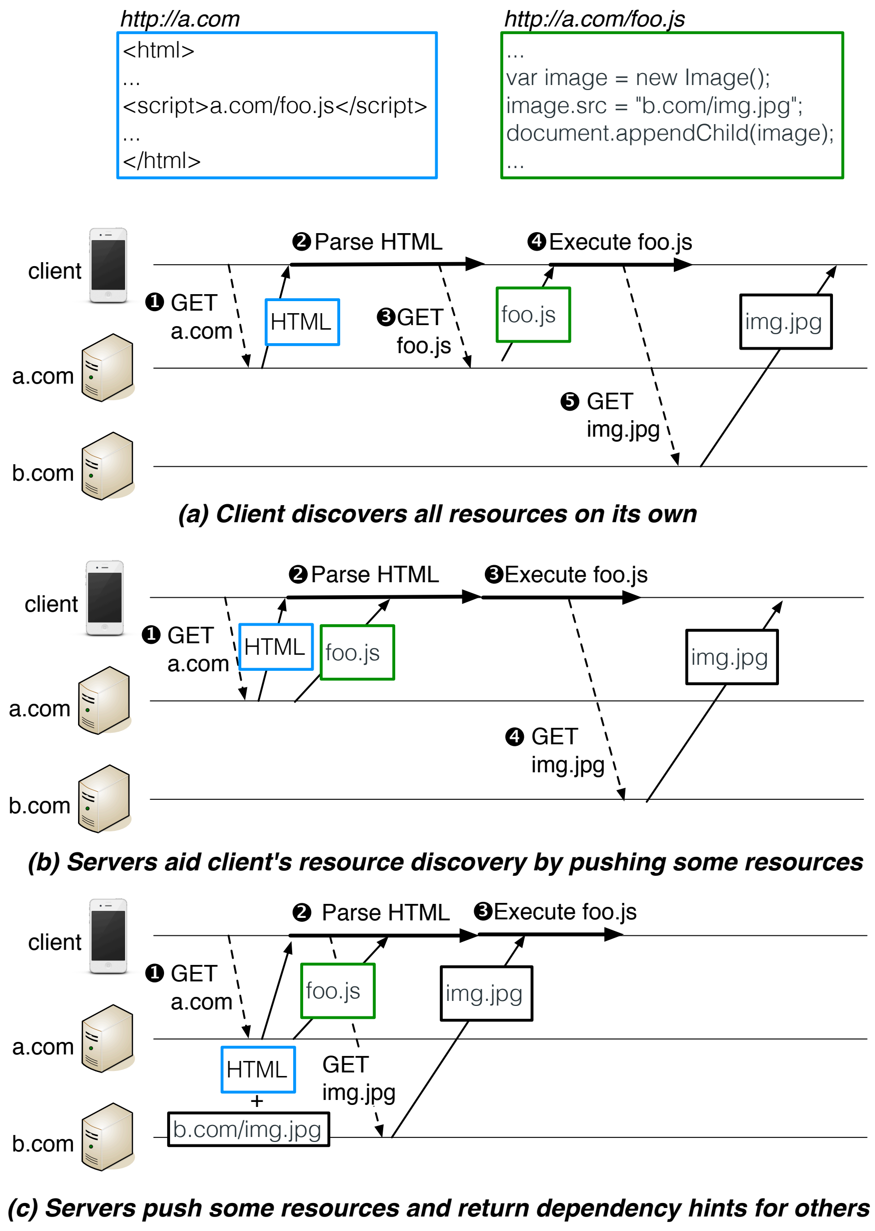


Figure 4.1: Comparison of critical path across different approaches for loading web pages, in all of which the client receives every resource from the domain from which it is served, so as to preserve personalization and the client's ability to verify the integrity of secure content: (a) 5 stages on critical path with CPU use blocking use of the network in steps 2 and 4 and vice-versa in steps 3 and 5, (b) 4 stages on critical path with CPU use blocking use of the network in steps 2 and 3 and vice-versa in step 4, (c) 3 stages on critical path with CPU and network utilized throughout.

processed, the client can make full use of its CPU, processing resources while fetching other resources in parallel.

4.2.1 Limitations of HTTP/2 PUSH

It may appear that such an ideal design of the page load process is feasible today because HTTP/2-compliant web servers can speculatively *push* resources to clients [58]. However, today, 1) the resources on a page are often spread across multiple domains [59], e.g., a web page from one provider often includes advertising, analytics, JavaScript libraries, and social networking widgets from other providers; 2) HTTPS adoption is rapidly growing [72, 29]; and 3) page content is increasingly personalized. These typical characteristics of modern web pages make the use of HTTP/2 PUSH inefficient for the following reasons.[2]

- When a domain receives a request for the HTML of a page that it hosts, it can only return resources that it hosts and not resources served by other domains. In the example in Figure 4.1, in response to the request for the HTML, `a.com`'s servers can only push the contents of `foo.js` which is served from the same domain, but not the third-party resource `img.jpg`. If web servers were to fetch resources from external domains and push them to clients [60, 85, 78], clients would be unable to verify the integrity of secure page content. Moreover, since any client's request to a web server will only include the client's cookie for that domain, resources fetched from other domains by that web server will not reflect any personalization of content by those domains.
- If web servers only push locally hosted resources, the client will discover resources that it needs to fetch from other domains only after processing previously fetched resources, e.g., in Figure 4.1(b), the client can discover the need to fetch `img.jpg` only after executing `foo.js`. This makes the CPU a potential bottleneck in the client's fetching of resources.
- During a page load, if every domain independently pushes its resources to the client, the client's receipt from one domain of a resource that must be processed (i.e., HTML,

[2]Another commonly cited limitation of PUSH is the potential for bandwidth wastage when a resource cached at the client is pushed. However, this problem could be addressed by having the client send a summary of its cache contents to web servers, e.g., in a cookie [22].

CSS, or JavaScript) can be delayed due to bandwidth contention on the client's access link with other resources (such as bulky images) from other domains. This makes the network a potential bottleneck in the client's processing of resources.

4.2.2 Combining PUSH with Dependency Hints

Given these limitations associated with relying *solely* on HTTP/2 PUSH, we leverage an additional server-side capability. When a web server receives a request for a resource, in addition to pushing the content for some dependent resources that it owns, the server can also return a list of URLs for other dependent resources—we refer to this list as *dependency hints*. Web servers can include such a list of URLs as an additional header in HTTP responses.

When a client receives dependency hints, it can fetch every resource whose URL is included in the list, without having to first process other resources on the page to discover the URLs in the list. For example, in Figure 4.1(c), based on the hint received from `a.com`, the client can fetch `img.jpg` from `b.com` without having to wait to receive and execute `foo.js`.

Legacy browsers already support dependency hints in the form of HTTP Link headers which have the preload attribute set [40]; resources listed in these headers are immediately fetched by browsers but are not evaluated until they are referenced by the page (i.e., Link preload headers are primarily used to prewarm browser caches during page loads).

Using dependency hints in addition to HTTP/2 PUSH offers several advantages:

- Any web server can safely send clients dependency hints for third-party resources. For any URL received via dependency hints, a client will fetch it directly from the respective domain, enabling the client to confirm the integrity of resources served over HTTPS and preserving that domain's ability to personalize content.
- The client need not fetch resources that are already cached locally, making it easier to minimize bandwidth waste compared to the use of HTTP/2 PUSH.

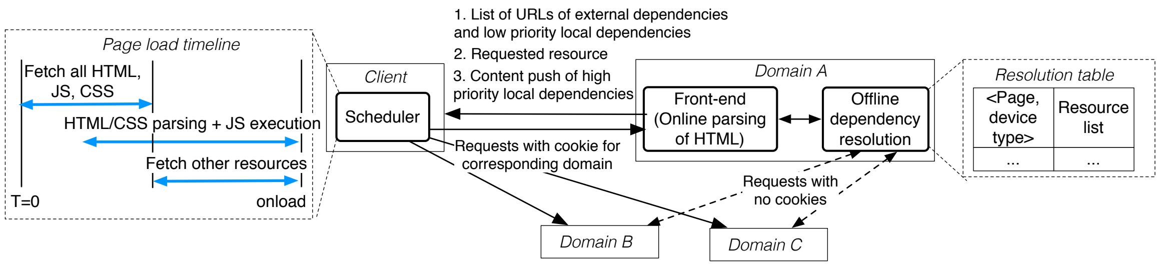

Page load timeline
Fetch all HTML, JS, CSS
HTML/CSS parsing + JS execution
Fetch other resources
T=0
onload
Client
Scheduler
1. List of URLs of external dependencies and low priority local dependencies
2. Requested resource
3. Content push of high priority local dependencies
Requests with cookie for corresponding domain
Domain A
Front-end (Online parsing of HTML)
Offline dependency resolution
Requests with no cookies
Domain B
Domain C
Resolution table
<Page, device type>
Resource list
...
...

Figure 4.2: Illustration of the components in Vroom and the interactions between them.

- The client maintains control over its concurrent fetches of resources from multiple domains; it can coordinate its downloads such that high priority resources (those that must be processed) are not delayed.

4.3 Design

Using the approach described in the previous section requires us to answer three questions:

- In response to a request from a client, how can a web server accurately identify the list of dependent resources that it should inform the client about, including ones hosted in other domains, without having clients fetch resources that are irrelevant to the ongoing page load (thereby wasting bandwidth)?
- How can a server provide a sufficient number of dependency hints to clients, without knowledge of how content is personalized by other domains that serve resources for a given page?
- At any point in time during a page load, a client's access link bandwidth is shared by resources that are explicitly fetched by the client or proactively pushed by servers. Given this contention, when should clients schedule resource fetches? What resources should servers push?

In designing Vroom to address these questions, we respect two primary constraints: 1) we do not rely on input from developers to characterize the dependencies on web pages because the resources on a page are typically spread across several domains [59], and no single developer is likely to have complete knowledge about all dependencies on a page; and 2) to preserve the integrity of content and to protect user privacy, any client will accept a resource only from the domain that serves that resource, and it will share its cookie for a domain only with web servers in that domain. Figure 4.2 illustrates the server-side and client-side components of Vroom, which we describe next.

4.3.1 Server-side Dependency Resolution

Generating an accurate list of dependent resources for a web page is challenging due to the constant flux in resources on modern pages. Moreover, unlike recent work [60, 75] focused on generating a page's stable dependency *structure*, here we need to identify the precise *URLs* of resources that a server must either push or include in its dependency hints. To appreciate the challenge in doing so, we first consider two strawman approaches before describing our solution.

4.3.1.1 Strawmans for resource discovery

Strawman 1 (Online). When a server receives a request for the HTML of a page, the server could load the entire page locally, mimicking a client browser, to identify the other resources on the page. However, many of the URLs fetched by the server during its page load will not be requested during the client's page load. On the one hand, back-to-back loads of a page often differ in the exact set of URLs fetched (e.g., ads typically insert a randomly selected identifier into the URLs they fetch). For example, 22% of the URLs fetched to load the median page in the Alexa Top 100 list change across back-to-back loads. On the other hand, loading a page at one server cannot account for the personalization performed by other domains, since the server only has the user's cookie for its domain. If servers fail to account for these discrepancies and either push to the client or ask the client to fetch unnecessary objects, the user is likely to experience *higher* load times.

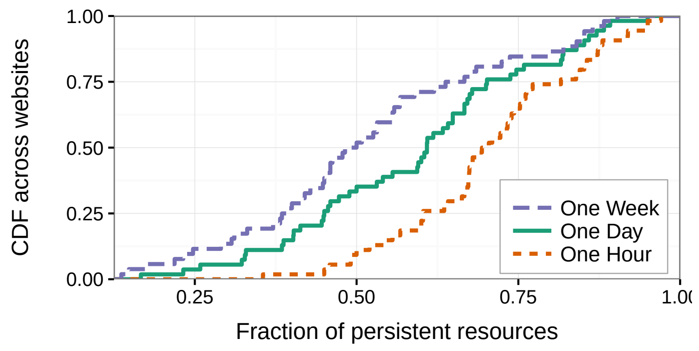


Figure 4.3: Fraction of resources, per page in the Alexa Top 100, that persist over different time scales.

Strawman 2 (Offline). Alternatively, a server can periodically load each page that it serves. This enables the server to account for the variation in resources over time; when a client requests the HTML for a page, the server can return the set of resources that it has repeatedly observed on recent loads of that page. With this approach, we risk missing a large fraction of URLs that a client will need to fetch when loading the page. For example, the set of stories or set of products on the landing page of a News or Shopping site changes often. Figure 4.3 confirms this; for the median site in the Alexa Top 100, only 70% of the resources on the landing page remain stable over one hour, and this number drops to 50% over one week.

4.3.1.2 Our solution: offline + online discovery

The two strawmen approaches for server-side resource discovery illustrate the following trade-off: servers must ensure that a client does not end up fetching unnecessary resources, but if they are too conservative (and thus let clients discover many resources on their own), the utility of input from servers will be minimal. To address this trade-off, we observe that *both* offline and online dependency resolution are necessary at the server. Figure 4.4 summarizes the techniques we employ. Periodic offline resolution of a page helps confirm which URLs are consistently fetched when loading the page, whereas online resolution helps account for flux in page content.

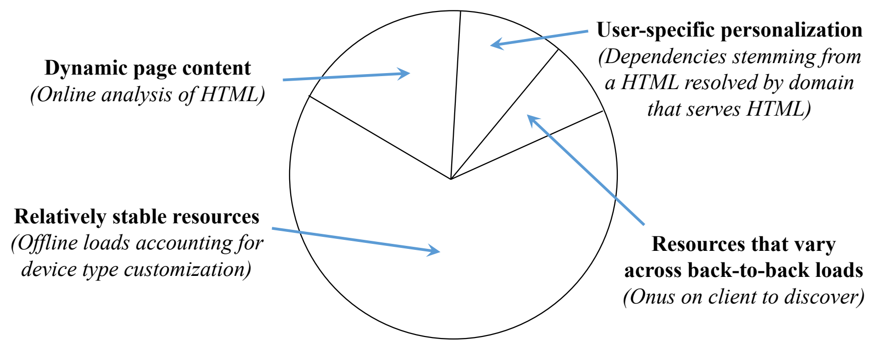

Dynamic page content
(Online analysis of HTML)
User-specific personalization
(Dependencies stemming from a HTML resolved by domain that serves HTML)
Relatively stable resources
(Offline loads accounting for device type customization)
Resources that vary across back-to-back loads
(Onus on client to discover)

Figure 4.4: Summary of techniques used in Vroom for server-side dependency resolution to account for the different types of resource on any web page.

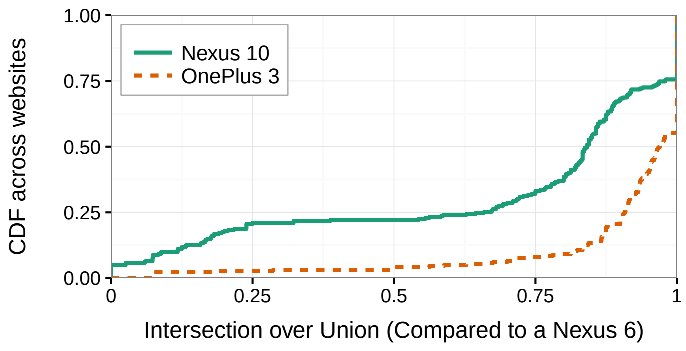


Figure 4.5: Comparison of the stable set of resources on each page when the user device is a Nexus 6 compared with when the user device is a Nexus 10 or a OnePlus 3.

Offline dependency resolution. Offline server-side determination of the resources on a page works as described previously: the page is loaded periodically (once every hour in our implementation), and at any point in time, URLs fetched in all recent loads are considered likely to also be fetched when a client loads the page. However, even the largely stable subset of resources on a page can vary across different types of client devices (Figure 4.5). For example, it is common for website providers to use CSS stylesheets and JavaScript objects that cause different clients to fetch images of different sizes on the same page depending on the client's display resolution or pixel density.

Vroom's offline server-side dependency resolution efficiently accounts for device-specific customization of resources in two ways. First, the server need not load each page on every type of device; this would be onerous given the large variety of smartphones and tablets on the market. Instead, after a few loads of a page, the server can bin all device types into a few equivalence classes. The equivalence classes can vary across pages because different pages may be customized based on different device characteristics. For example, in Figure 4.5, the stable set of URLs fetched when loading a page on a Nexus 6 smartphone matches the stable set of resources for a OnePlus 3 phone much more closely than for a Nexus 10 tablet. Second, after device type equivalence classes for a page are identified, the server need not load the page on a real device in each class. Instead, the server can leverage existing device emulation tools [20].

Online HTML analysis. In addition to offline dependency resolution, when a Vroom-compliant web server responds to a request with an HTML object, it not only informs the client of dependencies discovered from loading this object offline, but also includes all URLs seen in the HTML object by parsing it on the fly. While there can be other sources of dynamism on a page (e.g., a script on the landing page of a shopping site may fetch products currently on sale), we show later in Section 4.5 that accounting for the URLs in HTML objects suffices on most pages to capture the flux in page content. Importantly, we find that server-side parsing of HTML objects as they are being served adds a median delay of only roughly 100 ms across the landing pages of the top 1000 websites. This overhead is offset by the multi-second reduction in page load times made possible by server-aided resource discovery.

4.3.2 Accounting for Personalization

Unlike content push, dependency hints allow a server to inform clients about resources served by other domains. But, content served by one domain may be personalized in ways that other domains are unaware of (e.g., based on information stored in cookies).

A naive solution for handling personalization would be for any web server to never return dependencies derived from external content. In other words, any resource that is discovered during the page load by parsing or executing an external resource is deemed as one that could differ due to personalization. For example, in Figure 4.6, in response

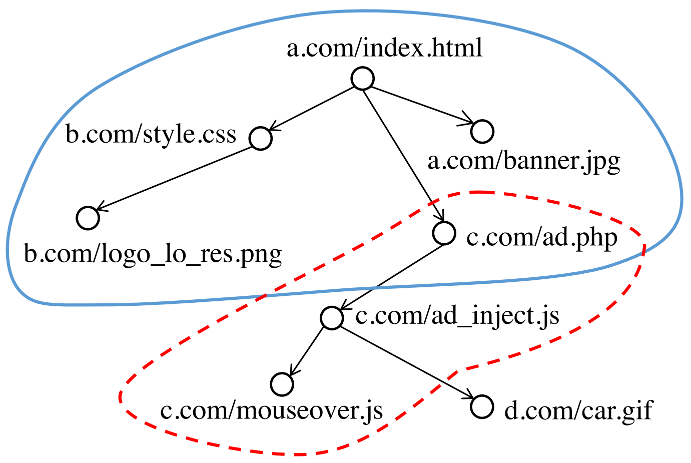


Figure 4.6: Illustration of how Vroom-compliant servers account for personalization. A client that requests for `index.html` from `a.com` only discovers the resources within the solid blue envelope, because a request for `ad.php` returns a HTML, whose content could be personalized to a specific user. The client discovers the resources in the dashed red envelope in response to its request for `ad.php` sent to `c.com`. The client will have to itself discover the need to fetch `d.com/cars.gif`.

to a request for `index.html`, `a.com` can inform a client about `b.com/style.css`, but let the client discover the need to fetch `b.com/logo_lo_res.png` only when it requests `style.css` from `b.com`. Accounting for personalization in this manner would however inflate the latency incurred by the client in discovering all resources on the page, thereby limiting its ability to fully utilize the network.

To handle content personalization while enabling low-latency resource discovery, we observe that websites are personalized primarily in two ways: by customizing the content of HTML responses,[3] and by adapting the execution of scripts. Server-side resource discovery in Vroom accounts for these two types of personalization as follows. First, web servers omit hints for dependencies derived from an external resource only when that resource is an HTML object (i.e., an embedded iframe); servers do include dependencies derived from other types of external resources (e.g., `style.css` in Figure 4.6). In comparison with the naive solution described above, our approach reduces the latency for the

[3]Content of CSS stylesheets and scripts are seldom user-specific. Personalization of images and videos (i.e., returning different versions when the same URL is fetched) is typically device-specific, which we have previously accounted for.

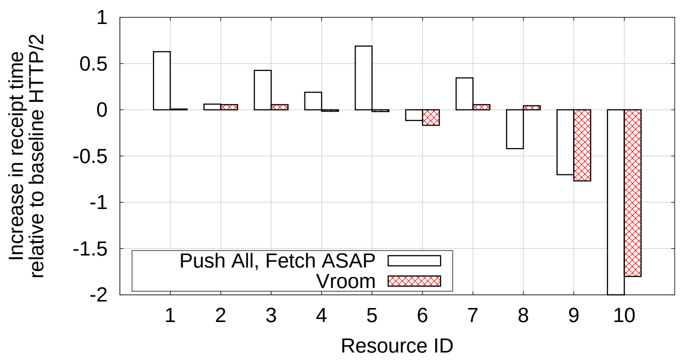


Figure 4.7: Need for and utility of careful scheduling of server-side push and client-side fetch of the resources that need to be processed (HTML, CSS, and JS) on http://eurosport.com. Resources are ordered based on the order in which they are fetched with baseline HTTP/2.

client to discover resources. Second, of the resources determined based on JavaScript execution, those affected by user-specific state (e.g., local time) are left to clients to discover on their own. JavaScript-based personalization will typically vary over time, and hence, such unstable resources will get filtered out via offline dependency resolution.

4.3.3 Cooperative Request Scheduling

Finally, we turn our attention to questions that must be answered for clients to benefit from server-side resource discovery. How should web servers combine the use of HTTP/2 PUSH and dependency hints to aid clients? How should clients utilize the dependency hints from servers?

Strawman: Push whenever possible. Fetch upon discovery. We first consider the most straightforward answers to these questions. When a web server receives a client's request for an HTML object, it can push to the client all dependent resources that it owns. The server can inform the client of all other dependencies via dependency hints. As soon as the client receives these hints, it can initiate downloads for all specified URLs.

Problem: Bandwidth contention. Though this simple strategy significantly improves utilization of the client's access link, we do not see a commensurate increase in CPU utilization. This is because, though the client receives the complete set of resources on a page well before it would without server push and dependency hints, contention on the client's access link slows down fetches of some of the resources that need to be processed. In the example in Figure 4.7, though the time to fetch the first 10 resources that need to be processed reduces by 2 seconds with the "Push All, Fetch ASAP" strawman, simultaneous use of the client's access link to transfer all of these resources delays the first few resources, causing the browser's processing to stall. Due to the resulting under-utilization of the CPU, we show later in Section 4.5 that applying this strawman solution yields no improvements in page load times.

Solution: Prioritization via selective push and staged downloads. Our solution to this problem is to prioritize the fetches of resources that need to be processed (HTML, CSS, and JS) over those that need not be processed (e.g., images and videos).[4] Classifying resources into high and low priority groups helps because, once the client has finished fetching all resources that need to be processed, utilization of the CPU and the network are largely decoupled over the rest of the page load. Moreover, the types of resources that need to be processed typically constitute a small fraction of the bytes on a page [26].

This prioritization of resources that need to be processed is achieved in Vroom via two means. First, when a web server responds to a client's request for an HTML object, out of all the dependencies the server identifies, it pushes the content of only the high priority resources served from the local domain. All other dependencies are returned to the client via dependency hints. Second, when the client receives a list of URLs, it immediately fetches only high priority resources; the server's hints specify resources in the order in which the client will need to process them, so client requests simply mimic that order. Once resource discovery from servers is complete and the client has finished fetching all high priority resources that have been discovered, it issues requests for all other resources

[4]We however consider all resources that are descendants of third-party HTML objects (i.e., HTMLs within a page's iframes) as low priority because web browsers process iframes only after the root HTML for the page has been completely downloaded and parsed. This helps minimize network contention for high priority resources referenced in the root HTML, thus reducing the fetch times for those resources.

at once. In Figure 4.7, the time by when receipt of the 10 resources shown completes is the same with Vroom's scheduling as with the strawman strategy, but without significantly delaying the receipt of any individual resource.

Note that Vroom's scheduler is tailored for the setting where web pages are loaded on a state-of-the-art mobile device connected to a LTE network; as we saw earlier in Section 2.1, the client CPU is the bottleneck in this case. Alternate scheduling strategies will likely be necessary in settings where either network bandwidth (e.g., because of many users simultaneously accessing the cellular network [61]) or latency (e.g., on 2G or 3G networks [65]) is the bottleneck.

4.4 Implementation

The realization of Vroom requires both server-side (offline and online dependency resolution; pushing resources to clients and including dependency hints in responses) and client-side (scheduling fetches of hinted objects) changes. Since this preempts evaluation in the wild, we have implemented Vroom to accelerate web page loads when Google Chrome is used to load pages from the Mahimahi [78] replay environment.

4.4.1 Server-aided Resource Discovery

Vroom-compliant servers inform clients of dependent resources via content pushes and dependency hints. To push resources, we leverage HTTP/2's PUSH capability; since Mahimahi uses Apache web servers which do not yet support HTTP/2, we run an HTTP/2 nghttpx reverse proxy [38] in front of each web server. For dependency hints, we rely on embedding additional headers in HTTP responses. For example, when a browser encounters a Link preload header [40] in an HTTP response, the browser will immediately issue a request for the URL embedded in that header.[5] Table 4.1 summarizes the headers used

[5]While Link preload headers are currently supported by Chrome, other browsers such as Firefox are actively in the process of adding support for these headers: https://bugzilla.mozilla.org/show_bug.cgi?id=

Header	Description
Link preload	Resources to be processed (e.g., JavaScript and HTML objects), fetched at the highest priority
x-semi-important	Resources to be processed that are lazily fetched, e.g., “async” JavaScript or CSS objects
x-unimportant	Resources that do not need to be parsed or executed (cannot have derived children), e.g., images

Table 4.1: HTTP headers used by Vroom-compliant servers to provide dependency hints to client browsers. Headers are listed in decreasing order of priority. Resources in each header are listed in the order they need to be processed.

by Vroom to provide dependency hints to clients.

To minimize stalls in the client browser’s processing of resources, dependency hints from any server list resources in the order the client will need to process them and the client requests hinted resources in this order; the server discovers this order during its offline and online dependency resolution. However, since the client issues these requests back-to-back, a web server may receive multiple requests near-simultaneously, causing it to respond to all requests in parallel. The resulting contention for the client’s access link bandwidth leads to the slowdown of some resources over others as seen with the “Push All, Fetch ASAP" strategy in Figure 4.7. Therefore, we modify Mahimahi so that any web server returns the content for requested resources in the same order in which it receives requests.

4.4.2 Scheduling Requests with JavaScript

To schedule client-side downloads of URLs learned via dependency hints (Section 4.3.3), Vroom uses a JavaScript-based request scheduler. For each recorded page in Mahimahi, we modify the page’s top-level HTML using Beautiful Soup [7]. Vroom’s scheduler script is added as the first tag in the HTML, thereby ensuring that the browser executes this script as soon as it begins parsing the HTML.

1222633.

Vroom's scheduler script begins its execution with two steps. First, it defines an `onload` handler, *response_handler*, which it attaches to each request that it makes. This handler maintains a list of dependency hints that it has seen and fires every time the browser receives a response for a request made by Vroom's scheduler. Second, the script issues an XHR (*XMLHttpRequest*) for the page's HTML, whose URL we embed as an attribute in the `<html>` tag.[6]

The scheduler examines Link preload headers in the response for the page's HTML to discover dependency hints.[7] It then issues requests for high priority dependencies by adding `<link>` tags (with the preload attribute set) to the DOM. Importantly, the scheduler's requests for high priority resources are served from the browser's local cache, because the browser itself immediately requests URLs included in Link preload headers. Moreover, modern browsers permit only a single outstanding request for any given URL.

Thereafter, Vroom's scheduler runs in an event-driven loop. Whenever the browser invokes *response_handler* upon receiving a resource, the scheduler marks that resource as fetched. The scheduler then examines the set of outstanding requests to determine whether it should start fetching dependencies in the next level of priority. Specifically, once all high priority resources learned via dependency hints have been received, Vroom's scheduler issues requests for all semi-important resources that it has discovered until that point. This process then repeats for low priority resources.

By using a JavaScript-based request scheduler, our implementation of Vroom can accelerate page loads on unmodified commodity browsers. However, due to the single-threaded nature of Javascript, if the browser is executing another script on the page when a response arrives, *response_handler* will not fire immediately. This delays the fetches of lower priority resources. Therefore, in the future, incorporation of the scheduling logic

[6]Note that Vroom's scheduler removes this attribute and its own DOM node from the page once the XHR for the top-level HTML is issued. Thus, subsequent accesses to the DOM are not affected.

[7]To ensure that the request scheduler can securely access headers in HTTP responses served from third-party domains, responses must include the "Access-Control-Expose-Headers" header with the values "Link," and our custom headers "x-semi-important" and "x-unimportant."

into the browser may enable greater performance gains than what we report.

4.5 Evaluation

We evaluate Vroom from two perspectives: 1) performance benefits for users, and 2) the accuracy with which servers can aid resource discovery for clients. The key highlights from our evaluation are:

- Across 100 popular News and Sports websites, we see that the adoption of Vroom would yield near-optimal performance on the median site with respect to two different metrics: page load time (PLT) and above-the-fold time (AFT). In comparison to the adoption of only HTTP/2, Vroom's use would reduce the median PLT and AFT values by 30% and 20%, respectively.
- Simple alternatives to Vroom (e.g., relying only on prior loads of the page for dependency discovery, using only HTTP/2 PUSH but not dependency hints, or not scheduling pushes and fetches) can increase the median page load time by over 2 seconds.
- On the median site, Vroom's server-side discovery of dependent resources has a false negative rate below 5%, which in turn results in a 22% decrease in client-side latency to discover all resources on the page.

4.5.1 Impact on Client Performance

Methodology. We use the setup shown in Figure 4.8 to experimentally evaluate our implementation of Vroom. We load pages in Chrome for Android on a Nexus 6 smartphone connected to a Verizon LTE hotspot. The phone is also connected via USB to a desktop, which subscribes to events exported by Chrome via the Remote Debugging Protocol (RDP). The phone has a VPN tunnel setup to the desktop, on which we host Mahimahi [78]. For every web page on which we test Vroom, we initially load the page with the desktop as a proxy to have Mahimahi record all page contents; page load times with this setup

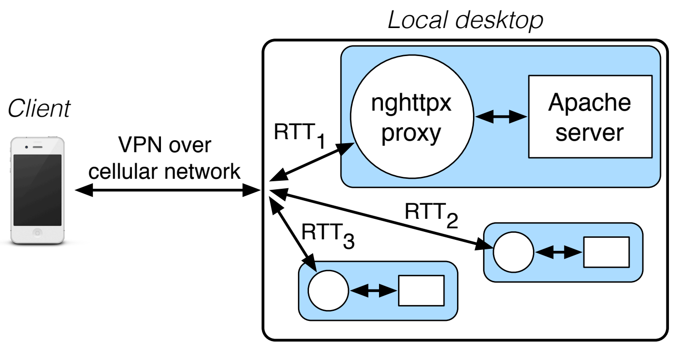
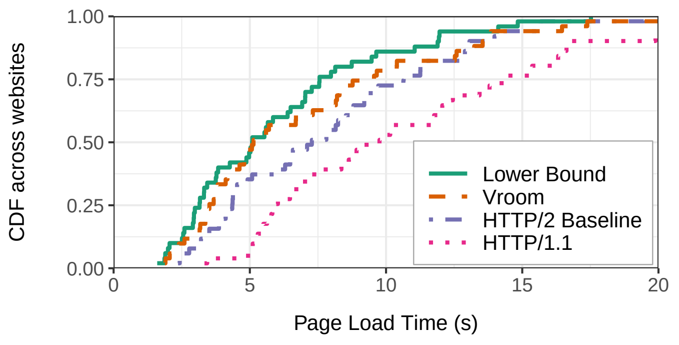


Figure 4.8: Setup to evaluate page load performance enabled by our implementation of Vroom.

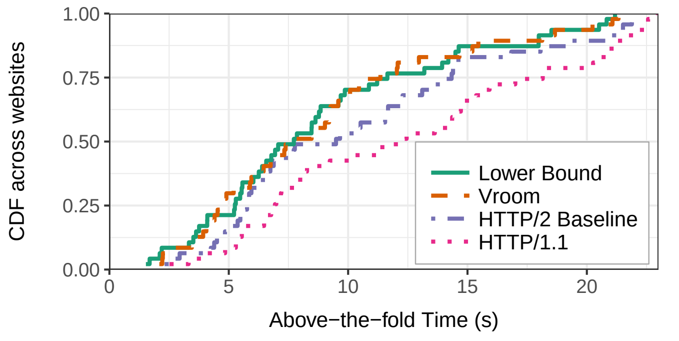
match those measured when we load pages with the phone directly communicating with web servers. Thereafter, when replaying page loads, we configure Mahimahi such that traffic between the phone and any of the web servers is subjected to not only the delay over the cellular network but also the median RTT observed between the desktop and the corresponding web server when recording page contents. The desktop on which we deploy Mahimahi is sufficiently well-provisioned so that its CPU or network is not a bottleneck, and as we mentioned earlier in Section 2.1, load times when we replay page loads using HTTP/1.1 between the client and all servers closely match load times measured when loading pages directly from the web.

We evaluate the utility of Vroom on the landing pages of the top 50 News and top 50 Sports websites as well as 100 randomly chosen sites from Alexa's top 400 sites;[8] in most of our experiments, we focus on the News and Sports sites because, as seen earlier in Section 2.1, the need for performance improvements on these sites is particularly acute with a median page load time of 10.5 seconds. We load each page 3 times in our replay setup and consider the load with the median page load time. During these loads, web servers identify the dependencies to return to clients by drawing upon three prior loads of each page gathered 1, 2, and 3 hours prior to when we recorded the page content in Mahimahi.

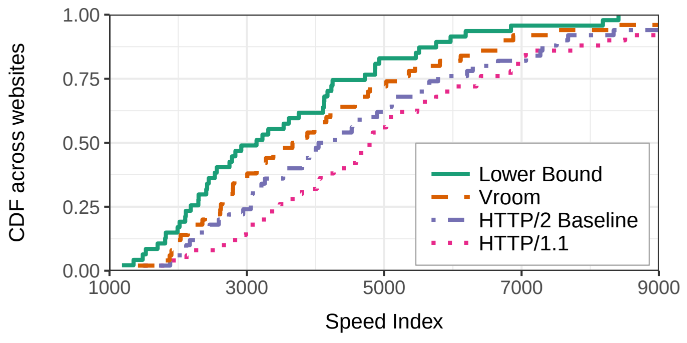
[8]The high page load times for these pages and the need to load each page multiple times to account for the variability of the cellular network limit us from running our evaluation on more web pages.



(a)



(b)



(c)

Figure 4.9: With respect to three different metrics, Vroom yields significant benefits compared to simply upgrading from HTTP/1.1 to HTTP/2, and comes close to matching the achievable lower bound.

In addition to page load times, we also evaluate Vroom's benefits with respect to additional metrics which capture the speed with which page content is rendered, as this impacts users' perception of page load performance [69]. To measure these metrics, we use *screenrecord*, a utility program which captures videos of page loads on Android devices. We pass the recorded videos to the *visualmetrics* tool [50] which outputs the metrics of interest.

Improvement in page load times. First, we compare page load performance when using Vroom with that when doing a HTTP/2 based replay (i.e., same setup as Figure 4.8, except that servers simply return requested resources), which we refer to as the HTTP/2 baseline. On the top 50 News and top 50 Sports sites, Figure 4.9(a) shows that Vroom significantly reduces the page load time on the median site—from 7.3s with the HTTP/2 baseline to 5.1s with Vroom—closely matching the lower bound (median of 5s); the lower bound corresponds to the maximum of the CPU-bound and network-bound loads described earlier in Section 2.1. On the 100 sites from the top 400, where the median page load time is significantly lower than on the News and Sports sites even with the HTTP/2 baseline (median of 4.8s), Vroom still reduces the median page load time to 4s.

The improvements in load times described above are feasible when all domains adopt the changes prescribed by Vroom. To understand Vroom's benefits as it is incrementally adopted, we evaluate Vroom in a scenario where the first party domain for every web page (i.e., the domain that serves the root HTML for the page), along with all other domains controlled by the same organization, are Vroom-compliant. All other domains contacted in each page load only conform to HTTP/2, without pushing content or providing dependency hints. While the median page load time across the News and Sports increases to 5.6s in this case, as compared to 5.1s with universal adoption of Vroom, this is still significantly lower than the 7.3s median page load time with the HTTP/2 baseline.

We also compared Vroom to Polaris [75], a state-of-the-art web accelerator which uses information about a page's dependency structure to prioritize requests for critical resources. Figure 4.10 shows that, compared to Polaris, Vroom is able to reduce the median page load time for the popular News and Sports sites from 6.4s to 5.1s. As noted in Section 2.1, the primary reason for this performance gain is that Polaris still leaves clients to discover resources on their own (i.e., the client can discover the need to fetch a resource only after fetching and evaluating another resource). Further, Polaris does not

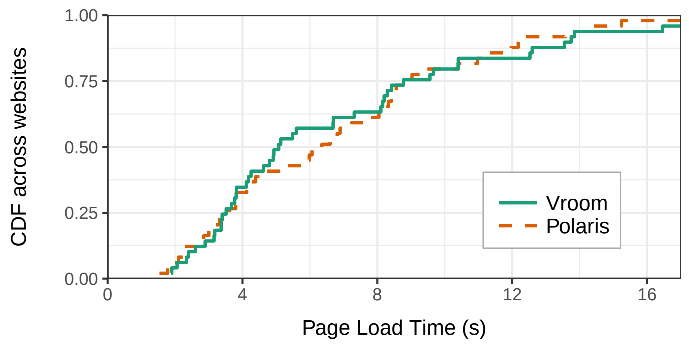
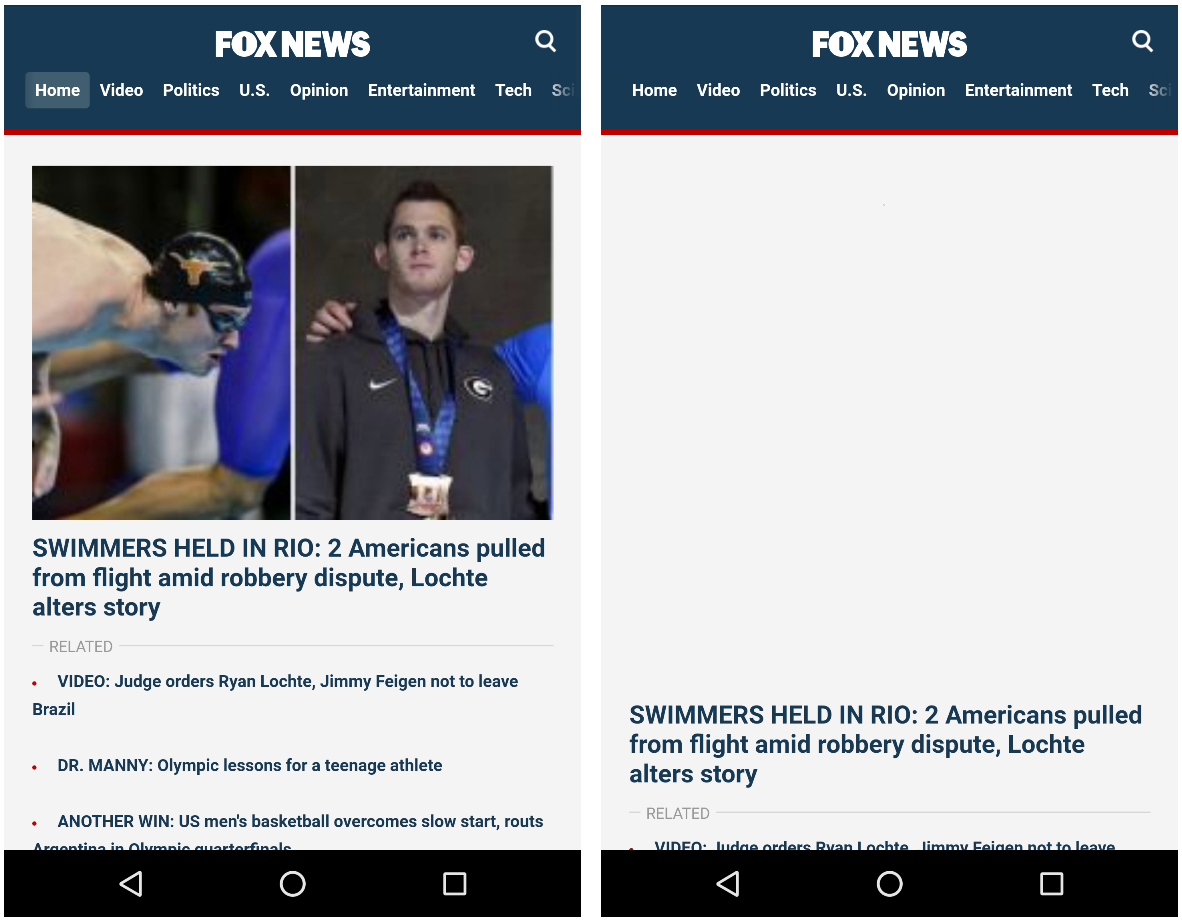


Figure 4.10: Comparison of page load times when pages are loaded with Vroom and with Polaris.

specify policies for servers to proactively push resources to clients in anticipation of future requests.

Though Figures 4.9(a) and 4.10 show that Vroom significantly improves performance for the median page, these benefits are marginal at the tail; in fact, Polaris outperforms Vroom in the tail of the load time distribution. The reasons for this are two-fold. First, certain sites include dynamic content that Vroom is unable to detect simply by online analysis of HTMLs; Vroom defers the discovery of such unpredictable resources to clients and is unable to provide hints for these resources. Second, when clients prefetch objects (specified by dependency hints) and servers in multiple domains concurrently push resources, bandwidth contention can result in high priority resources being delayed. These results illustrate that combining the complementary approaches used in Vroom and Polaris is a promising direction of future work.

Improvement in visual performance metrics. In addition to reducing page load times, Vroom also improves metrics such as Above-the-fold time and Speed Index which grade performance based on visual completeness of page loads. Above-the-fold time measures the time until all content that is “above the fold" (i.e., a user sees prior to scrolling) is rendered to its final state. Speed Index extends this metric to capture the rate at which all the content that is above the fold gets rendered. Figures 4.9(b) and (c) show that, across the popular News and Sports sites, Vroom improves the Above-the-fold time and Speed



SWIMMERS HELD IN RIO: 2 Americans pulled from flight amid robbery dispute, Lochte alters story

RELATED

- VIDEO: Judge orders Ryan Lochte, Jimmy Feigen not to leave Brazil
- DR. MANNY: Olympic lessons for a teenage athlete
- ANOTHER WIN: US men's basketball overcomes slow start, routs Argentina in Olympic quarterfinals

SWIMMERS HELD IN RIO: 2 Americans pulled from flight amid robbery dispute, Lochte alters story

RELATED

- VIDEO: Judge orders Ryan Lochte, Jimmy Feigen not to leave

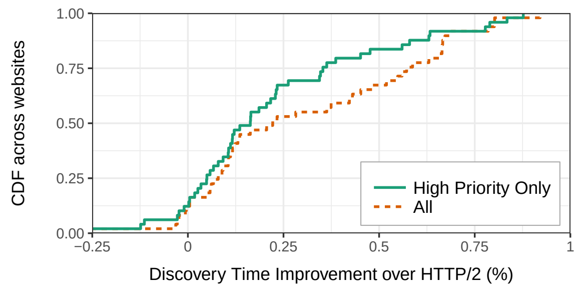
(a) (b)

Figure 4.11: For the Fox News mobile site, (a) rendering of the above the fold content completes at 9.26s with Vroom; (b) with only HTTP/2 enabled, rendering is incomplete at that time and completes only later at 13.87s.

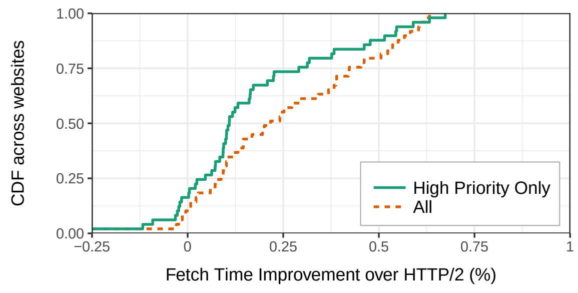
Index for the median site by 400ms and 380ms, respectively.

Figures 4.11 shows these benefits on an example site (http://m.foxnews.com/). With Vroom, rendering of all above the fold content completes at 9.26s. At that same time in the page load with the HTTP/2 baseline configuration, the main images are still missing and the rendering of the page is yet to converge. With HTTP/2, rendering of all above the fold content takes 4.6s longer, completing at 13.87s into the page load.

The above-described improvements in page load performance with Vroom are due to several reasons; we dig into each of these next. For brevity, we show results only for the popular News and Sports sites.



(a)



(b)

Figure 4.12: In comparison to HTTP/2, Vroom reduces the latency in both (a) discovering resources and (b) completing their downloads.

Latency in discovering and fetching resources. A key benefit of server-aided resource discovery is that it enables clients to discover resources and complete fetching them much sooner than in normal page loads. Figure 4.12 depicts these improvements both when considering all resources identified as dependencies by Vroom-compliant web servers, and also when considering only those dependencies which are high priority resources (i.e., HTML, CSS, and JS objects, which are the ones that need to be parsed or executed).

Reducing the time by when the client completes fetching all dependencies (a 22% reduction on the median page) is crucial because any further network activity necessary to complete the page load is only to fetch the small subset of unpredictable resources that servers fail to identify as dependencies. But, in addition, the drop in the time by when all high priority dependencies finish downloading—a 12% median reduction compared to baseline HTTP/2—is also critical since use of the CPU and the network are largely decoupled thereafter. Both of these improvements are made possible because of the speedup in the client discovering dependencies: in the median, 22% and 16% faster discovery of all dependencies and of all high priority dependencies, respectively.

Faster discovery and preemptive receipt of resources also helps reduce the time spent on the critical path waiting to receive data over the network. While the simple use of HTTP/2 led to network delays accounting for over 30% of the critical path on the median

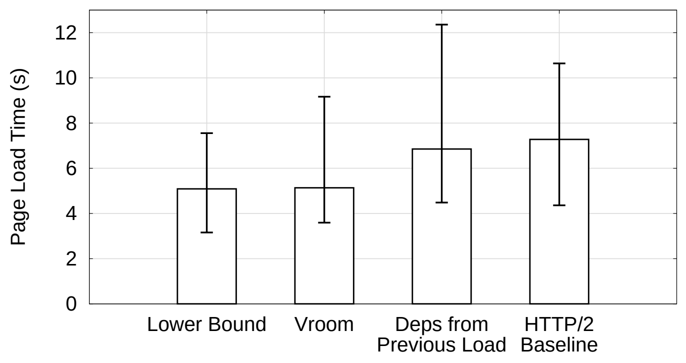


Figure 4.13: In contrast to Vroom, if servers return dependencies as all the resources seen on a prior load of the page, page load times increase on many pages. 25^{th} percentile, median, and 75^{th} percentile are shown in each case.

site (Figure 2.4), Vroom's use reduces the network wait time on the critical path by 24% on the median site.

Utility of accurate dependency inference. While input from servers enables clients to discover and fetch dependent resources sooner, the benefit of this input strongly relies upon the accuracy of the dependencies returned. To show this, we consider servers identifying the set of dependencies to return to a client simply based on a prior load of the page, i.e., all resources seen in a load within the past hour are assumed to be relevant to the new load. Figure 4.13 shows that, though the median page load time does reduce with this approach, the extraneous inaccurate dependencies returned to the client degrade performance on many sites; the 75^{th} percentile increases by over 1.5 seconds.

Need for combining HTTP/2 PUSH and dependency hints. Beyond accuracy in server-side dependency discovery, Vroom's benefits draw upon the *combined* use of HTTP/2 PUSH and dependency hints. Figure 4.14 shows that simply relying on server push is insufficient. Irrespective of whether we push all static resources or only the subset of static resources that need to be processed, median page load time remains more than 2 seconds higher than with Vroom. This stems from the preponderance of third-party resources on modern web pages; servers can inform clients of such dependencies only via dependency hints, as any server can securely push only the content that it hosts.

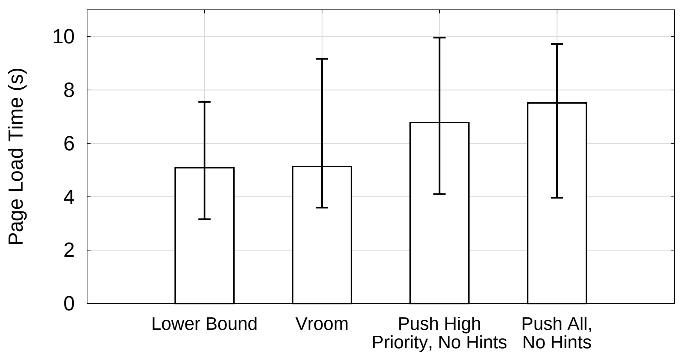


Figure 4.14: Vroom improves page load times compared to using only server-side push to inform clients of dependent resources. 25th percentile, median, and 75th percentile are shown in each case.

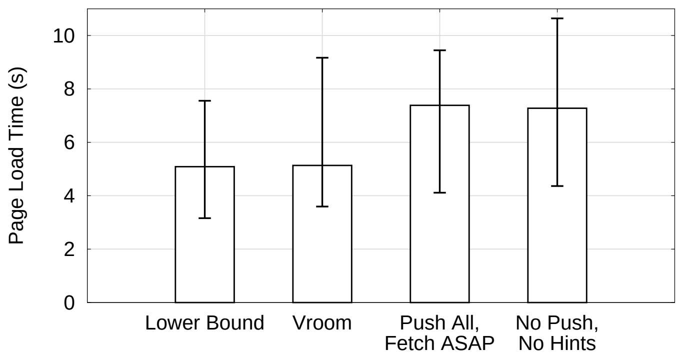


Figure 4.15: Vroom's judicious scheduling of server push and client fetch is key to enabling improvements in page load time. 25th percentile, median, and 75th percentile are shown in each case.

Utility of scheduling. While HTTP/2 PUSH and dependency hints enable faster resource discovery, performance improvements with Vroom also hinge upon judicious coordinated scheduling of pushes and downloads of discovered dependencies. Figure 4.15 illustrates the utility of the cooperative scheduling in Vroom compared to the strawman "Push All, Fetch ASAP" approach discussed in Section 4.3.3 (where servers push any resource they can and clients fetch any resource immediately upon discovery). Because only high priority resources are pushed by Vroom servers and are preferentially fetched by clients, contention for access link bandwidth has minimal impact on the processing of resources. The resultant increased utilization of the CPU enables Vroom to improve performance over baseline HTTP/2. Whereas, use of the strawman approach for servers

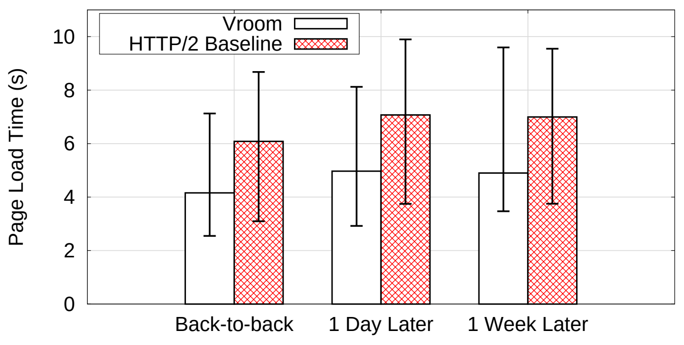


Figure 4.16: For three different delays between the load that warms the browser's cache and the load on which we evaluate page load performance, VROOM reduces page load times. 25th percentile, median, and 75th percentile are shown in each case.

to inform clients of dependencies offers minimal benefits; in fact, the median load time increases as a result of increased network contention.

VROOM **accelerates page loads with warm caches.** All of our experiments thus far have considered the browser's cache to be empty. To evaluate VROOM when the browser's cache is not empty, we first identify cacheable objects by examining the headers in HTTP responses. We mimic three different scenarios, wherein once a page is loaded by a client, it loads the page again immediately thereafter (i.e., back-to-back loads), a day later, or a week later. Importantly, to prevent wasted bandwidth, resources that were already cached at the client were not pushed by servers.

Figure 4.16 shows that VROOM significantly improves page load times in all three warm browser cache settings. When considering back-to-back loads, VROOM reduces the page load time of the median site by 1.6s. This improvement increases to 2.2s and 2.1s for the loads separated by one day and one week, respectively.

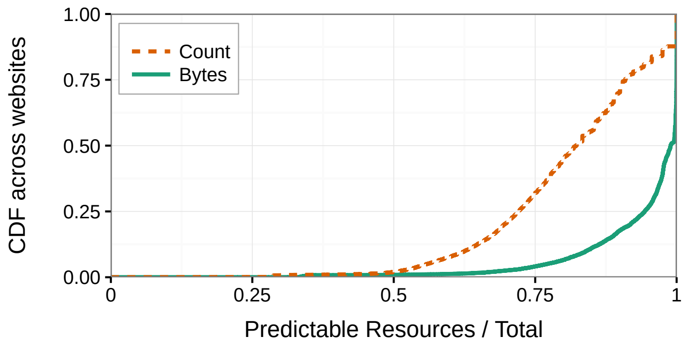
4.5.2 Accuracy of Server-side Dependency Resolution

Setup. To evaluate the accuracy of the resource dependencies that VROOM-compliant servers return to clients, we consider 265 web pages drawn from popular News and Sports websites; these pages span a variety of page types such as landing pages, individual articles, results for specific games, etc. We load these pages once every hour for a week from the perspective of four users, whose cookies are seeded by visiting the landing pages of the top 50 pages in the Business, Health, Computers, and Shopping/Vehicles Alexa categories, respectively. Every hour, we load each page twice back-to-back from every user's perspective for reasons described shortly.

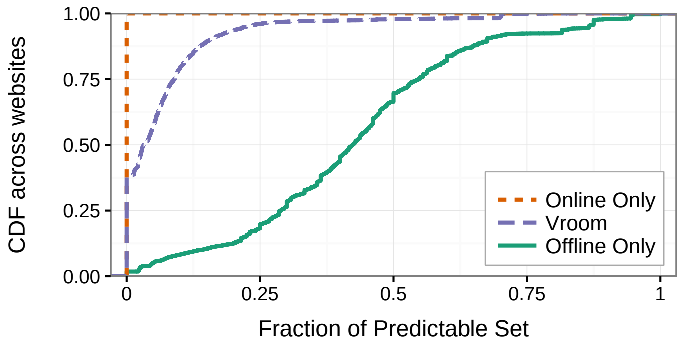
As described earlier in Section 4.3.1, server-side dependency resolution in Vroom relies upon both offline and online analysis. The dependencies identified via offline dependency resolution include the resources seen in each of the loads in the past 3 hours. For online analysis, we model the server which serves any HTML object—either the root HTML on a page or one embedded in an iframe—returning all links in that HTML. Recall that, in order to account for personalization, Vroom-compliant servers return dependencies—either via push or via dependency hints—only in response to requests for HTML objects (Section 4.3.2).

Strategies for server-side resource discovery. We compare dependency resolution in Vroom with both the strawman approaches described earlier in Section 4.3.1: *offline-only* returns URLs seen in the intersection of loads over the past 3 hours, and *online-only* loads the page on the fly at the server and returns the URLs fetched.

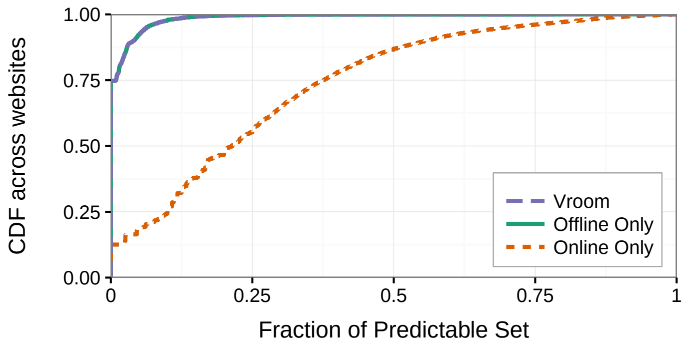
Definition of accuracy. To evaluate the accuracy with which each of these approaches can identify the set of URLs that a client must fetch during a page load, we partition the set of URLs we see in any page load into a predictable and unpredictable subset. We identify the subset of unpredictable URLs as ones that differ between back-to-back loads; these are URLs that Vroom leaves it up to the client to discover. As seen in Figure 4.17(a), out of the subset of resources on a page that a server can potentially return as dependencies in response to a request for a HTML (i.e., all the resources derived from HTML minus the ones derived from embedded iframes), the predictable subset accounts for over 80% and over 95%, respectively, in terms of the number of resources and the number of bytes. We evaluate the accuracy of each approach for server-side resource discovery with two



(a)



(b)



(c)

Figure 4.17: (a) Among the resources derived from a page's root HTML, except for those derived from embedded HTMLs, the contribution of the predictable subset to the number of resources and bytes. As a fraction of the size of this predictable subset, the resources that are either (b) missed or (c) are extraneous when using Vroom's server-side dependency resolution as compared to offline-only and online-only analyses.

metrics—the number of resources identified as dependencies by the server which do not appear in the predictable subset of the client's load (false positives), and the number of resources in the predictable subset that the server fails to identify (false negatives)—both computed as fractions of the predictable subset's size.

Results. First, Figure 4.17(b) shows that, out of the resources in the predictable subset, the fraction that Vroom-compliant servers would fail to identify is less than 5% for the median page. Whereas, offline-only dependency resolution ends up missing as many as 40% of the predictable subset of resources seen on any particular page load because of its inability to cope with changes from hour to hour. The online-only approach is perfect with respect to this metric, which validates our design decision to account for personalization by limiting the set of dependencies returned to exclude resources recursively derived from embedded HTMLs.

Second, with respect to the overhead imposed on clients by returning dependencies not in the predictable subset, Figure 4.17(c) shows that Vroom matches the offline-only approach. In both of these cases, identifying the stable set of resources seen consistently on a page helps in ignoring resources that happen to show up on a single load. Due to its inability to cope with such nondeterminism, the online-only approach identifies many extra resources, which inflate the predictable subset by as much as 20% in the median case.

4.6 Discussion

Deployability. Unlike attempts at clean-slate redesigns of the Internet's architecture, new designs for client-server interactions on the web are more amenable to deployment, as evidenced by the recent incorporation of SPDY into HTTP/2. This is possible because of several differences between the web and general communication over the Internet: 1) clients directly interact with web servers, unlike an ISP having to rely on other ISPs to forward traffic, 2) a few popular browsers and web servers are dominant, and 3) since some browsers (Chrome and IE) are controlled by popular content providers (Google and Bing), these providers can unilaterally test performance improvements enabled by a new proposal such as ours without depending on adoption by others.

Server-side overhead. Vroom-enabled servers identify dependent resources using both online and offline analyses. For popular websites that host thousands of web pages, loading each page every hour to facilitate offline dependency resolution will likely be onerous. We observe that there are typically only a few *types* of pages on each site and the stable set of resources (e.g., CSS stylesheets, fonts, logo images, etc.) are likely to be common across pages of the same type. For example, on a news site, landing pages for different news categories are likely to share similarities as will news articles about different individual stories. We defer for future work the task of leveraging the similarity across pages of the same type to improve the scalability of Vroom's server-side resource discovery.

Note that the overhead of resolving dependencies will be incurred only by the domains which serve HTML objects, i.e., the root HTML for an entire page or for an individual frame in the page. Other servers involved in a page load need only serve individual resources as they are requested. Moreover, for over 80% of sites, the landing page's HTML is served directly from origin web servers [26] and not from third-party CDNs. Thus, the overhead imposed by Vroom's resource discovery will largely be incurred by top-level domains who are only responsible for the websites they own.

Security. At first glance, it may appear that having servers push resources and having clients fetch hinted resources present new security concerns for page loads, whereby compromised web servers could push or provide hints for malware. However, of the resources that are pushed or fetched based on dependency hints, client browsers will only process those resources referenced by the page being loaded, e.g., as HTML tags. Further, page loads that include unnecessary pushes or hints will still load correctly to completion, albeit with higher load times due to the downloads of unnecessary resources. Thus, page loads with Vroom encounter the same (but not worse) security concerns as page loads do today.

4.7 Summary

The recognition that dependencies within the page load process are the dominant cause for slow page loads has led to a slew of solutions recently. However, all of these solutions either compromise security and privacy by relying on proxies to resolve dependencies or have limited ability to improve mobile web performance since they require clients to themselves discover the resources on any page. Vroom offers the best of both worlds: by having servers aid a client's discovery of resources (both via HTTP/2 PUSH and dependency hints), we decouple the client's processing and downloads of resources, but do so while preserving the end-to-end nature of the web. By improving CPU utilization, Vroom significantly decreases page load times compared to baseline HTTP/2.

CHAPTER 5

Taking the Long View: The Relationship Between Page Complexity and Performance

5.1 Overview

In this chapter, we present a longitudinal study on how websites have changed over the years and how have the changes impacted page load performance. We set out to conduct this study because rewriting web pages is the most fundamental change that a content provider can make; both proxy-based optimizations and end-to-end solutions such as CASPR are constrained by how a page is written. Understanding the effects that changes to a page have on that page's load performance will shed light on practices that other website providers can avoid or follow to ensure performant page loads.

Prior efforts have characterized every page's complexity using a variety of metrics (i.e., the features of a page which determine the impact of network and compute delays on its loads) and studied the relationship between these complexity metrics and page load performance [83, 59, 75, 81, 79]. For example, pages with more resources on them are more likely to be slow on client devices with slow CPUs and networks. This understanding has, in turn, led to a number of recommendations for web developers to follow in order to ensure that their pages load quickly for users, e.g., avoid the use of blocking JavaScript files [42] and compress images [80].

However, improving page load performance is far from the only concern when website providers modify their web pages. They may want to add resources to their pages in order to offer richer functionality (e.g., use scripts to enable personalization) [51] or to reap larger revenues (e.g., include more advertisements) [1]. On the other hand, website providers may remove content from their pages in order to make it easier for users to navigate their pages, e.g., to offer a more streamlined web experience on mobile devices [36].

In order to understand the performance impact of such changes, we flip the lens with which prior work has viewed the relationship between page complexity and performance. Instead of analyzing web pages at a particular point in time and identifying what changes *could be made* to enable faster page loads, we characterize how pages have evolved over time and study the performance impact of the changes that *have been made* to these pages. Specifically, we consider pages that are periodically crawled by the HTTP Archive project [26] and study the changes in these pages over time—primarily focusing on the period from 2016 to 2019—with respect to page complexity, performance, and the relationship between the two. Since prior work has found that the network, more than the CPU, is typically the dominant bottleneck in web page loads [75, 79], we focus on complexity metrics that capture the network's impact on page load performance: the number of resources on a page and the total number of bytes across these resources. In conducting this study, we make the following contributions.

How has page complexity changed over time? First, we observe that the manner in which pages have changed from 2016 to 2019 is distinctly different compared to the period prior to 2016. In the three years leading up to 2016, we see that the vast majority of pages that we study increased in complexity. In contrast, the number of pages that increased in complexity from 2016 to 2019 is roughly equal to the number that saw a decrease in complexity over that period. We find that substantial increases in the number of resources and number of bytes are primarily due to the inclusion of images, both in quantity and higher quality. Many of the additional resources included on web pages are due to advertisements, but these contribute little to the need for clients to fetch more bytes. Drops in page complexity, on the other hand, are enabled by lazy loading of images and the use of image optimization services.

How to analyze performance of historical versions of pages? Second, to study the impact that changes in page complexity have on page load performance, we set up a high fidelity replay environment. Our testbed lets us evaluate the network's impact on versions of a page from different years. The primary challenge here is that, to the best of our knowledge, there exists no publicly available data source which contains *all* of the content present on historical versions of a large corpus of web pages. The absence of even a single resource can significantly impact the measured performance for a page because all resources recursively referenced by that missing resource will go unfetched. To tackle this challenge, we demonstrate how to combine data from several sources (HTTP Archive [26], Internet Archive's Wayback Machine [32], and the web today), and strategically patch resources that are still missing after doing so.

What is the performance impact of increased page complexity? Third, based on page loads in our replay environment, we find that an increase in page complexity from 2016 to 2019 often does not translate to a proportionate performance slowdown. Our analysis reveals that the primary reasons for this non-intuitive relationship between complexity and performance are the presence of HTTP redirections and blocking JavaScript files. Prior work [83, 75, 81, 60, 79] has observed that the serialization of resource fetches caused by these factors results in inefficient use of a client's network. We instead find that such inefficiencies can prove to be beneficial in the long term, as they allow for more content to be added to a page without adversely impacting performance. This observation could not have been made by analyzing pages from any one point in time, reinforcing the need to consider page complexity and performance from a longitudinal perspective, as we do in this chapter.

How to maximize the impact of reduced page complexity? Lastly, we find that serialization of resource fetches does have the undesired effect that most pages that significantly decreased in complexity from 2016 to 2019 do not see a proportionate speedup. Towards enabling such pages to see higher performance gains after removing content, we characterize the various types of serialized resource fetches that exist on these pages and identify the ones which could potentially be transparently removed without impacting the page's functionality. For example, among resources fetched via JavaScript, those that consistently appear in a page over time could be inlined in the page's HTML (thereby preempting the aforementioned performance penalty of serialization), as they do not require the dynamism that JavaScript enables. We evaluate the impact of removing such inefficiencies from pages; we find that doing so increases the number of pages on which a reduction in content from 2016 to 2019 results in a proportionate performance improvement.

5.2 Dataset and Metrics

5.2.1 Dataset

Source of data. To study the complexity of web pages over time, we use data from the HTTP Archive project [27]. Prior to January 2019, HTTP Archive loaded the landing pages of the Alexa top 1 million websites [4] once every 15 days. After January 2019, HTTP Archive switched to the landing pages of the top 1.3 million domains from Chrome's User Experience report [11] and loaded them once every 30 days. In each cycle, HTTP Archive loads every page 3 times with an empty cache and records data from the load with the median page load time. For each site, HTTP Archive separately loads the desktop landing page and the mobile landing page by respectively mimicing a desktop and mobile user-agent.

HTTP Archive uses private instances of WebPageTest [53], deployed at the Internet Systems Consortium data center in Redwood City, CA, to perform page loads. WebPageTest considers the page as fully loaded and terminates a page load once the browser fetches all outstanding resources and the CPU on the test machine is idle for at least 500ms [56]. We focus our analysis of each page load on the portion that occurs before the *onload* event.

Description of data. In the snapshot that it publishes at the end of each load cycle, HTTP Archive publishes data in the HAR format [28] containing timings (of requests and responses) and response bodies for each resource fetched in every page load; HTTP Archive, however, does not record the response bodies of some resources and we describe how to handle them in Section 5.4.1. Our analysis predominantly relies on HTTP Archive's data from the July 15, 2016 and July 1, 2019 snapshots, but also examines the July 15, 2013 snapshot for reference. In all snapshots, we consider the landing pages for 600 sites that appear in both the 2016 and 2019 snapshots; we partition all sites based on their ranking – $[0, 5000)$, $[5000, 50000)$, and $[50000, \infty)$ – and randomly select 200 sites from each rank range.

5.2.2 Complexity Metrics

To better understand how web sites have changed over the years, we identify key metrics to capture the complexity of a web page. While there are a range of complexity metrics to choose from [59, 83], we take guidance from prior work, which has observed that most page loads today are constrained by the network [60, 75, 79]. Therefore, we focus on complexity metrics which capture the features of a web page which determine how the page is affected by sources of delay imposed by the network: 1) network latency, and 2) network bandwidth.

To capture the impact of these, we respectively use the *number of resources* and *number of bytes* (i.e., the amount of data transferred over the network to fetch all resources) as complexity metrics for any page. Fetching each resource incurs at least one round-trip of latency (more if a new TCP/TLS connection needs to be setup); therefore, greater the number of resources on a page, greater the slowdown the page will incur on a high-latency network. Whereas, pages that require clients to fetch a greater number of bytes will be more impacted by limited network bandwidth.

We do not consider more sophisticated complexity metrics, such as the length of the critical path in a page's load [83], because such metrics are a function of not only page content, but also the operation of the browser in which the page is loaded. For example, the maximum number of TCP connections that a client can maintain to a particular domain varies across web browsers [49], which in turn affects the client's ability to fetch resources in parallel, thereby impacting the page load's critical path. Instead, our analysis focuses on how *pages* have evolved over time, and how those changes specifically affect load performance.

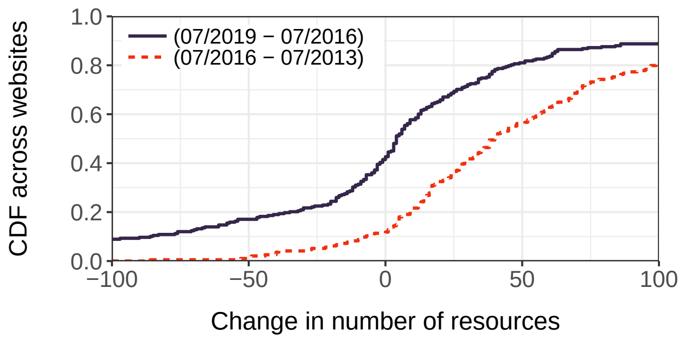
5.3 Characterizing Changes in Page Complexity

5.3.1 Change in Complexity Over Time

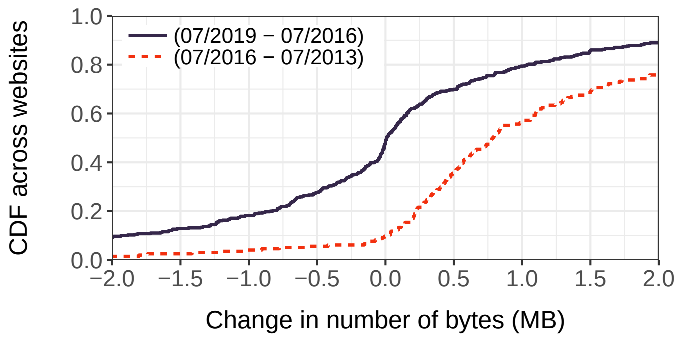
We begin our analysis by examining how the web pages in our dataset have changed over the years with respect to our two complexity metrics. For each page, we compute the difference in number of resources and bytes over two periods: from July 2016 to July 2019, and from July 2013 to July 2016. Figure 5.1 shows the distribution of these changes across the desktop landing pages; the distribution is similar for the mobile landing pages.

When considering the 3 year span between 2013 and 2016, most websites see a significant increase in both the number of resources and number of bytes. Figure 5.1(a) shows that the number of resources increased on roughly 90% of sites, with the median website observing an increase of more than 40 resources. On the other hand, Figure 5.1(b) shows that 90% of sites observe an increase in the number of bytes fetched during a page load, with an increase of more than 750KB for the median website.

In stark contrast, when considering the timeframe between 2016 and 2019, we find that page complexity did not strictly increase with respect to either metric; complexity



(a)



(b)

Figure 5.1: Distribution across sites of the change in number of (a) resources and (b) bytes on their desktop landing page from 2013 to 2016 and from 2016 to 2019.

increased for some sites and decreased for others, with the change for the median site close to 0 both in terms of number of resources and number of bytes. For example, Figure 5.1(a) shows that 20% of the sites saw an increase of more than 50 resources, but another 20% witnessed a decrease of 50 resources or more. Similarly, Figure 5.1(b) shows that the number of bytes fetched increased by more than 500KB on 35% of sites, but decreased by over 500KB for a different subset of 35% sites.

Overall, between 2013 and 2016, websites generally see an increase in page complexity, but between 2016 and 2019, the change is largely mixed. The significant increase in complexity between 2013 and 2016 can stem from a variety of reasons, e.g., mobile landing pages leveraging improved device and network speeds to include additional features on their web pages; Explaining the phenomena underlying these trends is beyond the scope of this work and is worthy of a separate study of its own.

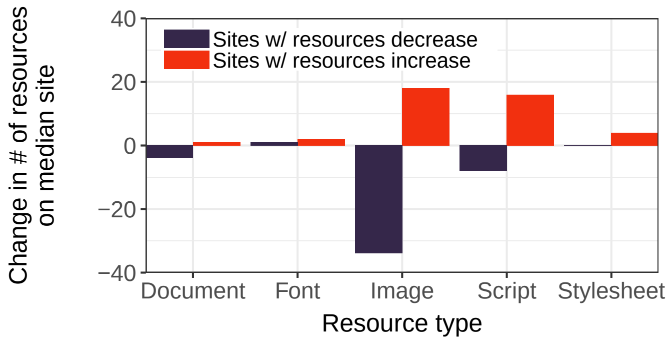


Figure 5.2: For each resource type, change in the number of requests for resources of that type on the median site.

Given the mix of complexity changes between 2016 and 2019, in the reset of the chapter, we focus on digging deeper into the changes over this period. In the rest of this section, we examine what changes underlie the increase in complexity on some sites and the decrease on others. In subsequent sections, we study the impact of these changes on page load performance.

5.3.2 Characterizing Content by Resource Type

To analyze sites with a substantial change in landing page complexity between 2016 and 2019, we focus on sites in the top 30% and bottom 30% of the distributions in Figures 5.1(a) and 5.1(b); note that pages with a substantial increase (or decrease) in the number of resources do not necessarily see a substantial increase (or decrease) in the number of bytes. Here, we examine the types of content that were added/removed from these pages. When the takeaways are similar for both desktop and mobile landing pages, we present results only for the desktop landing pages.

Sites with substantial changes in number of resources. On sites with a substantial increase or decrease in the number of resources, images and scripts are the highest contributor to these changes. Figure 5.2 shows that, among sites where the number of resources on the landing page substantially increased, the median site saw an increase of 47 resources overall, of which at least 15 were images and 15 others were scripts. For

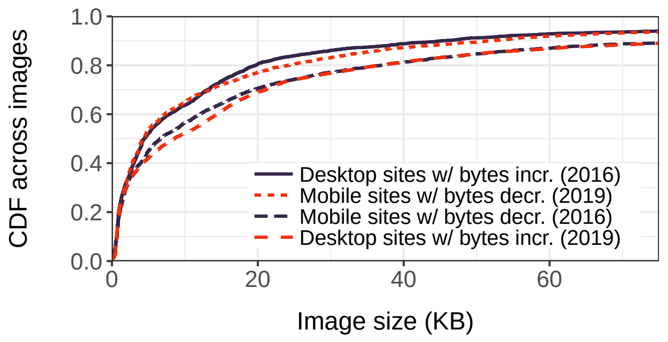


Figure 5.3: Distribution of image sizes across desktop landing pages with a substantial increase in bytes and mobile landing pages with a substantial decrease in bytes.

sites with a substantial decrease in resources, we find that the reduction in the number of images is even more pronounced; on the median site, almost 30 fewer images were fetched.

On closer inspection, we find that the cause for the reduction in number of images is *not* always because images were removed from these pages. Instead, we observe that the significant reduction in the number of images fetched is often due to the adoption of lazy loading of images; only images that appear above-the-fold (i.e., the portion of the page visible to users before they begin scrolling) are fetched prior to the *onload* event. For example, lazy loading of images helped redbubble.com reduce the number of images whose fetches impact page load time by 60, while the rest of the page had minimal other changes.

Sites with substantial changes in number of bytes. As for sites with substantial changes in the number of bytes on their landing pages, an increase/decrease in the number of images is a key reason – as one would expect given the above results – but not the sole contributor. In addition, we observe that the average image size changed significantly from 2016 to 2019 on many sites. For example, Figure 5.3 shows that, on desktop landing pages with a substantial increase in bytes, the median image size increased from 4.4KB in 2016 to 8.5KB in 2019. The underlying cause here appears to be website providers choosing to use higher quality images.

Figure 5.3 also shows that, across mobile landing pages with a substantial decrease in bytes, the median image size decreased from 6.6KB to 4.3KB. We observe that this drop in image sizes is due to websites optimizing the images they serve in a couple of ways. On some sites (e.g., marketingweek.com), the use of image optimization services such as imgix.com helped reduce image sizes. On other sites (e.g., escombray.cu), the adoption of responsive images [43] enabled the site to serve images tailored to the screen size on users' devices, thus reducing data consumption.

5.3.3 Characterizing Content by Service Type

Beyond characterizing changes in page complexity with respect to the *types of resources* being added/removed, we now analyze the *types of services* offered by the new/old resources. For example, how much of the new content is due to the addition of advertisements?

To do so, we correlate the URL of every added/removed resource against the third-party-web [14] dataset, which provides the service type of a large number of domains. If the domain for a resource is not in the dataset, we determine whether it is a first-party resource by comparing with the domain of the site whose landing page is being considered. We classify all remaining resources as *"Other"*. We extract the domain from any URL by removing the suffix of the hostname based on the public suffix dataset [41], splitting the remainder of the hostname based on the '.' delimiter, and using the last token.

Figure 5.4 shows that ads account for most of the increase in resources; on the median desktop landing page, the number of ad-based resources increased by 20. We observe that this increase is not necessarily due to the inclusion of more ads, but is often because of additional resources being requested per ad domain. For example, we find that between 2016 and 2019, the number of ad domains on nfl.com remains largely the same (45), but the average number of resources per domain increased by 4.

Figure 5.4 also shows that for websites with a substantial decrease in the number of

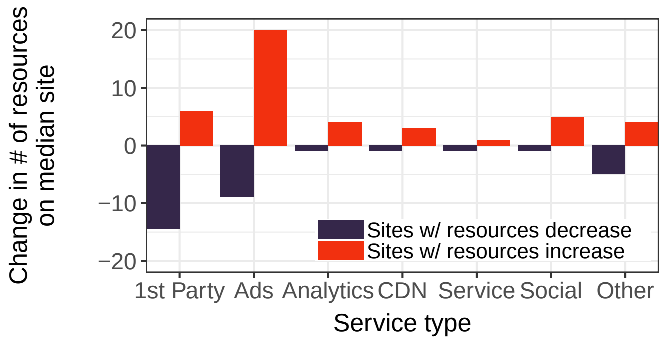


Figure 5.4: Comparison of the median change in number of resources per website for each service type.

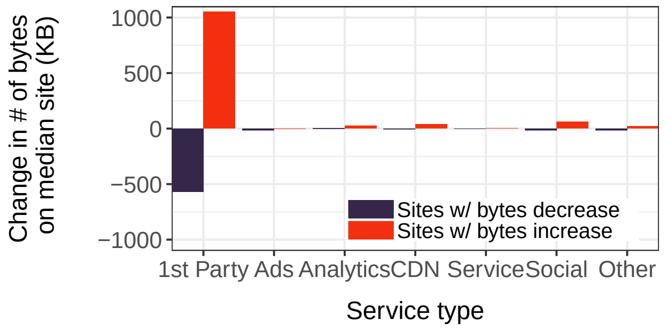


Figure 5.5: Comparison of the median change in number of bytes per website for each service type.

resources, reductions in the number of first-party resources and resources attributed to advertisements were the primary contributors. We observe that websites see a decrease in the number of ads-related requests largely because of the reduction in advertisement services adopted on the site. For example, mlssoccer.com reduced the number of ads domains from 46 in 2016 to 15 in 2019, with 5 resources fetched on average per domain in both years.

Figure 5.5 shows that ads, however, contribute little to changes in the number of bytes; changes on this front primarily stem from addition/removal of first-party content.

5.4 Relationship Between Complexity and Performance

Having characterized the changes in page complexity between 2016 and 2019, we now ask: *what has been the impact of these changes on page load performance?* To answer this question, we first present our methodology for measuring performance and then describe our results.

5.4.1 Page Load Replay Testbed

The page load time values recorded in the HTTP Archive dataset are specific to the measurement setup used by that project, which itself changed between July 2016 and July 2019 [54]. To be able to quantify page load performance in any arbitrary network condition, we use Mahimahi [78], a web record and replay tool, in conjunction with the HTTP Archive dataset. Mahimahi enables us to keep a fixed client configuration across loads of multiple snapshots, so that any performance changes we see are solely due to page modifications made over time.

Challenge. The key obstacle in using the HTTP Archive dataset to load pages in Mahimahi is that HTTP Archive records the HTTP response bodies for only a third of resources seen during its crawls. For these resources, we transform the metadata and HTTP response body stored in the HTTP Archive data into Mahimahi-compliant files. For the remaining 66% of resources, HTTP Archive only provides HTTP response headers, but not the content of the fetched resource. Most of the missing response bodies stem from the fact that HTTP Archive only stores non-binary resources [68], but we also observe some missing iframes and JavaScripts.

To get the content of those resources without response bodies, we could fetch them from the web today using the metadata stored in the dataset. However, many of the resource URLs that were previously included on web pages no longer exist or have been modified since. For example, among the resources which HTTP Archive crawled from desktop landing pages in 2016, almost half have a different size today than that recorded in the dataset.

Approach. We account for missing response bodies as follows to create our performance measurement setup.

- **Instead of fetching resources only from the web, we first check for the resource on Internet Archive's Wayback Machine [32].** For resource URLs seen in one of HTTP Archive's snapshots, we check to see if Wayback Machine had crawled that URL in the same month, as the resource size is more likely to match in that case than the copy from today's web (if available).

  Note that conducting our study using archived page loads from Wayback Machine, instead of using data from the HTTP Archive project, would not have obviated the need to patch up resources missing from the data. Our comparison of HTTP Archive and Wayback Machine crawls from the same time period show that the latter too is missing many resources; specifically, the latter ignores resources that are not worth archiving (e.g., ones related to ads and analytics). HTTP Archive's data at least provides a complete list of all resource URLs fetched in each of its page loads and is only missing the content of some resources. Whereas, from any of Wayback Machine's crawls, you would not even know that it is missing any mention of many resources! On the median desktop landing page, the number of resources seen in Wayback Machine's archive from 2016 are 35% lesser than the number of resources seen for that page in HTTP Archive's July 2016 data.

  Due to the incompleteness of Wayback Machine crawls, it does not suffice to rely solely on Wayback Machine to find resource content missing in the HTTP Archive data; on the median desktop landing page, we were still missing content for roughly 60% of resources after leveraging Wayback Machine.

  Therefore, for the remaining resources, we fetch them directly from the web. The combination of fetches from the Wayback Machine and from the live web helps us reduce the fraction of resources for which we are missing content on the median site to at most 23% across all combinations of desktop/mobile and 2016/2019.

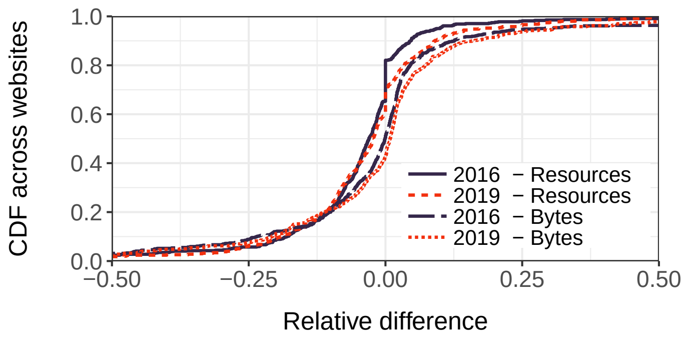
- We find that approximately 15% of the resources we fetch from Wayback Machine and the live web have a different size than that recorded in the HTTP Archive dataset, i.e., the content we obtain for these resources differs from what HTTP Archive fetched as part of its crawls. Of these modified resources, over 65% are images.

  To account for fetched images whose size differs from that recorded in the HTTP Archive data, as well as any images for which we still lack content, we observe that the speed with which a page loads is not impacted by the precise content of images that the page includes. Instead, only the image's type (e.g., jpg or png) and size matters.

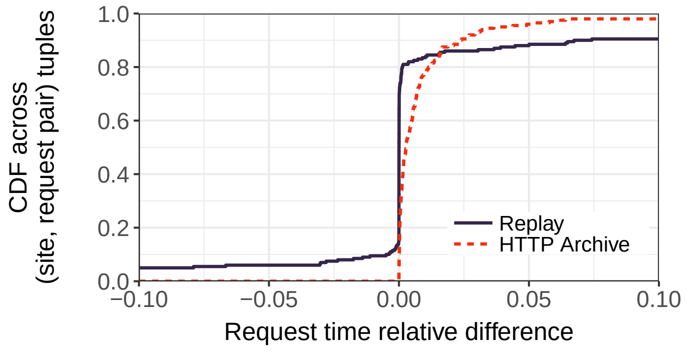
  Therefore, for any image in the HTTP Archive data for which we are unable to obtain a copy of its original size, we replace it with another image of the same type and similar size. For every image type, we amass a pool of 200 image replacement candidates as follows: for every half percentile in the distribution of all image sizes in the HTTP Archive dataset, out of all images we downloaded, we pick the image whose size most closely matches that value in the distribution. For every image for which we lack content of the recorded size, we then replace it with that image of the same type from our candidate pool which has the most similar size.

- After the previous two steps, we are missing less than 3% of resources on the median site. However, missing content for iframes (i.e., HTML documents embedded in the page) and JavaScript files can have an outsized impact, because the browser would fail to fetch all of the resources that it would discover after parsing/executing them. If any of these unfetched child resources are iframes or JavaScripts, recursively other descendants of the missing iframes and scripts will go unfetched.

  Hence, for any of a page's iframes or JavaScripts that are missing or have changed,



(a)



(b)

Figure 5.6: Distribution of (a) the relative difference with respect to number of bytes and resources between HTTP Archive dataset and our replay setup, (b) normalized difference in fetch start times between every pair of consecutive requests seen in HTTP Archive loads.

we carefully insert their descendants back into the page. First, we load every page in our Mahimahi replay environment and identify the resources mentioned in HTTP Archive's records that go unfetched. For every frame on the page, we then insert a script at the top of the frame's HTML; this script uses the `MutationObserver` API [37] to subscribe to changes to the page's DOM. Based on the order in which resources were requested in the HTTP Archive crawl, our script inserts a resource (which would otherwise go unfetched) into the DOM when it learns that the resource prior to it in the order is added to the DOM. Our script appends any resource to the <body> section of that resource's parent frame; the `referer` HTTP request header recorded in the HTTP Archive data enables us to identify which frame every resource belongs to.

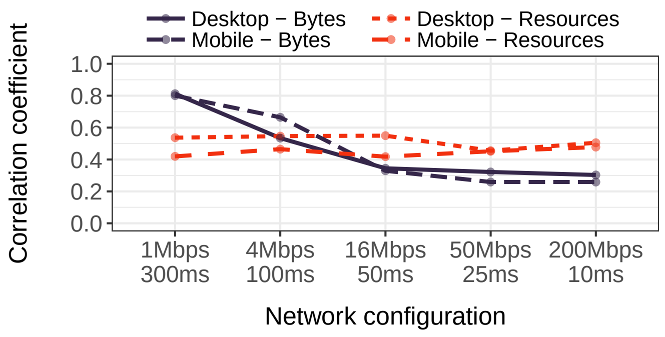
Sanity check. Whether our augmentation of the HTTP Archive data to make it amenable to replay its page loads suffices depends on two considerations: 1) do we preserve page complexity?, and 2) do we preserve the relative ordering and timing of resource fetches? We validate our setup on both fronts by replaying page loads using the network conditions used in HTTP Archive's crawls.

- **Our replay setup largely preserves page complexity.** Figure 5.6(a) shows that the number of bytes and number of resources seen in our replay loads closely match those in the original data. 80% of sites see a relative difference of no more than 20% in both bytes and resources. To maximize confidence in our subsequent correlation between page complexity and performance, we focus our analysis in the rest of the chapter on sites where the relative difference is no more than 20% for both complexity metrics.
- **We also largely preserve the sequence and relative timings of resource fetches.** For each page load, given the order in which resources were requested in the HTTP Archive crawl, we compute for every pair of consecutive requests the difference between the times at which their fetches were initiated. For landing pages in the July 2016 snapshot, Figure 5.6(b) plots the distribution across request pairs of this difference, normalized by page load time. The distribution in our replay loads closely matches that in the HTTP Archive data. We observe the same when replaying the 2019 snapshot (not shown).

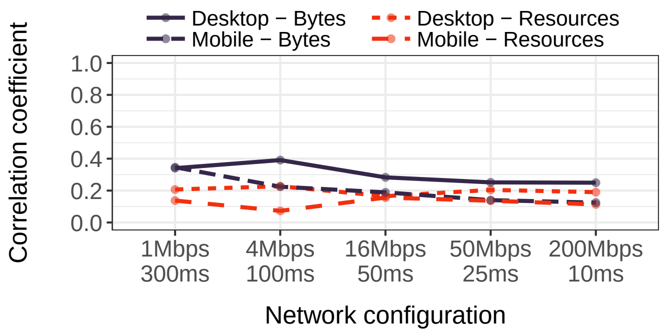
Note that, since our replay setup is missing the content for some JavaScripts, we make no claims about preserving client-side computation in our page loads. Given our focus on page complexity metrics impacted by network-based delays, this lack of fidelity is admissible.

5.4.2 Impact of Change in Complexity on Performance

Next, for every page, we correlate the change in complexity from 2016 to 2019 with the change in loading performance. To isolate changes in performance that are solely due to



(a)



(b)

Figure 5.7: For two different performance metrics – (a) page load time, and (b) Speed Index – correlation between the change in complexity metrics and the change in performance.

changes in the page, we use our replay setup to load both the 2016 and 2019 versions of every page in exactly the same client and network configuration. We load pages using Chrome v77 on a Ubuntu desktop with a quad-core processor with 16 GB of RAM. For every page load, we configure our setup to apply both per-packet client-server delays and server processing delays, both of which we extract from HTTP Archive's data. We then apply additional network shaping based on the kind of client network connection we aim to mimic.

The correlation between the change in complexity and the change in performance varies across network conditions. As listed in Figure 5.7, we vary network conditions across a very constrained network (1Mbps, 300ms)—mimicking the lower end of a 3G network [8], a moderately constrained network (16Mbps, 50ms)—mimicking a 4G network [8], and a very fast network (200Mbps, 10ms)—mimicking a cable network [23].

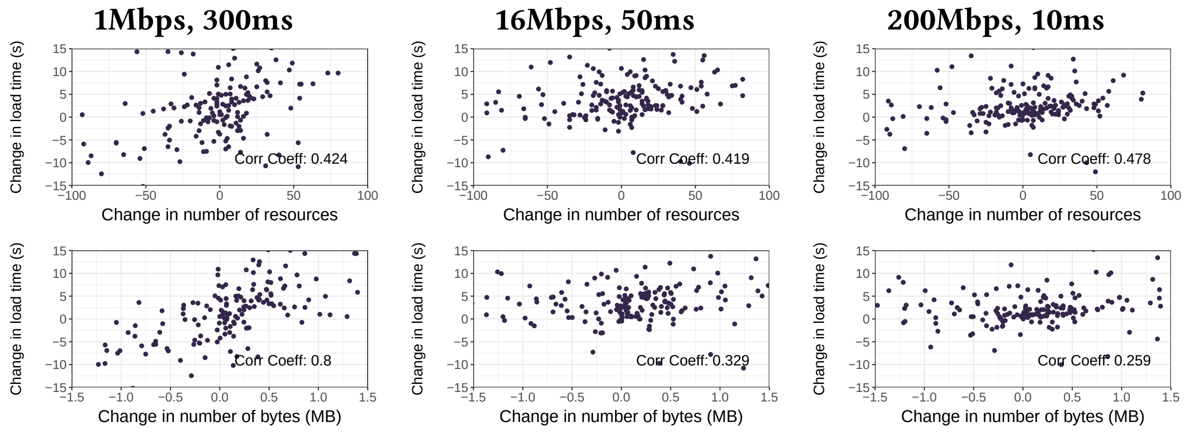


Figure 5.8: Change in complexity and the change in page load time are strongly correlated in a constrained network. The correlation becomes significantly weaker when the network is less constrained. The plots show the correlation for mobile landing pages, but we see the same patterns for desktop landing pages.

Figure 5.7 shows that, in both the desktop and mobile settings, changes in the number of bytes are strongly correlated with changes in page load time when the network is poor; for a (1Mbps, 300ms) network, the Pearson's correlation coefficient is 0.82 for both desktop and mobile landing pages. However, when the network is less constrained, the correlation is significantly weaker. With a (200Mbps, 10ms) network, the correlation between the change in number of bytes and change in page load time drops to 0.29. These differences in correlations are evident from the scatter plots in the bottom row of Figure 5.8.

When we look at the relationship between changes in the number of resources and changes in load time (top row of Figure 5.8), the correlation is weak in all network conditions. The same is true for the correlation between changes in Speed Index and changes in either complexity metric in all network conditions (Figure 5.7(b)).

The weak correlations suggest that when websites add or remove content, the change does not proportionately affect page load performance. In other words, when pages become more complex over time, they do not necessarily slow down proportionately. Similarly, pages which reduce in complexity do not experience a proportionate speedup.

Setting	Complexity Metric	Complexity ↑, but no proportionate PLT ↑	Complexity ↓, but no proportionate PLT ↓
Desktop	Resources	17/79	70/80
	Bytes	34/78	66/78
Mobile	Resources	26/70	61/72
	Bytes	29/70	58/70

Table 5.1: With a (16Mbps, 50ms) network, many of the websites with a substantial increase in complexity do not see a proportionate degradation in page load times. Among the sites with a substantial decrease in complexity, most do not see a proportionate speedup. The denominator in each cell is the count of number of sites with substantial increase/decrease in complexity.

To confirm this interpretation of the weak correlations, we focus on those landing pages which had a substantial change in their complexity from 2016 to 2019, i.e., pages in the top 30% and bottom 30% of the distributions in Figure 5.1. Among these pages, we identify ones that do not observe a change in page load time that is inline with the change in complexity. Specifically, we identify pages where the relative difference in page load time between the 2016 and 2019 versions of the page is less than 80% of the relative change in complexity.

Table 5.1 shows the results of our analysis for the network which is in the middle of the range we consider. The findings are largely the same for either complexity metric.

Among the sites where complexity of their landing page has significantly increased between 2016 and 2019, a significant fraction do not see a commensurate slowdown. For example, in the desktop setting, of the 78 sites with a substantial increase in number of bytes, 34 (or 43%) do not see a proportional increase in page load time.

This non-intuitive relationship between change in complexity and page load times is even more stark for sites where complexity of their landing pages substantially decreased. No matter how you look at the data – desktop or mobile, number of requests or number of bytes – an overwhelming majority of the sites with reduced complexity do not benefit from a proportionate decrease in page load time.

5.5 In-depth Analysis of Pages

Our results from Section 5.4 reveal many pages which observe a change in complexity over time without the (expected) proportional change in page load performance. In this section, we perform deep-dive experiments in order to uncover the origins of this counterintuitive behavior. We focus primarily on settings with well-provisioned network resources, as the above trends are most pronounced there; results are shown for the (16Mbps, 50ms) configuration, but the listed trends and takeaways also hold for the superior conditions in Figure 5.7.

Approach. Our goal here is to elucidate the impact that each facet of page complexity has on performance. One approach would be to perform this analysis using page load time (as in Section 5.4), but this metric is not well-suited for our target analysis. The reason is that page load time is a coarse metric that captures overall load performance, bundling together the effects of *all* page load tasks, including those that do not relate to (i.e., are not influenced by) the complexity factor at hand. Instead, we introduce a new set of performance metrics that aim to explicitly capture the delays that are solely induced by each complexity metric.

5.5.1 Network Latency

Recall that we capture the impact of network latency on page load performance by characterizing a page's complexity as the number of resources on the page. Section 5.4 showed a weak correlation between changes in this complexity metric and page load time. Thus, to dig deeper, we propose the following more-targetted metric.

Performance metric: connection setup time. Each resource in a page can elicit network delays by triggering the establishment of a new network connection and requiring round trips for request and response transmission. We focus our analysis on overheads stemming from TCP connection setups (and TLS connection setups for HTTPS resources)

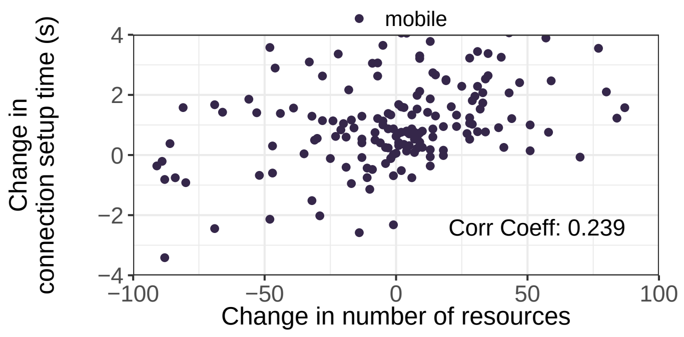


Figure 5.9: The correlation between the change in *connection setup time* and the change in number of resources is weak, which suggests that not every added or removed resource directly affects performance.

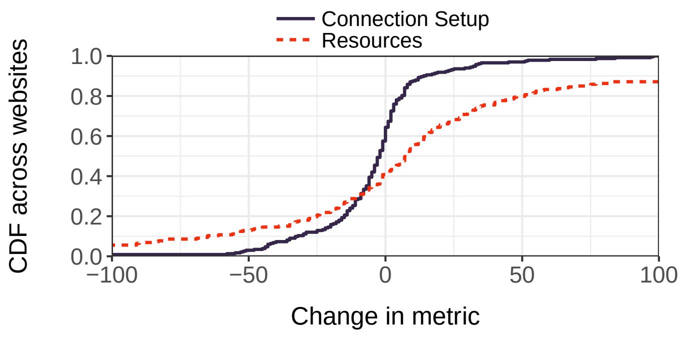


Figure 5.10: Distribution across websites of the change in number of resources is more pronounced than changes in the number of connections setup. This shows reuse of connections.

as request/response transmission delays can be influenced by factors unrelated to network latency, e.g., server processing delays in serving a response. We define our connection setup time metric as *"the amount of time in a page load during which at least one connection is being established"*. Importantly, we do not measure total connection setup time (i.e., summing the times across all established connections), as concurrent setups impose a shared effect on incurred network delays.

The correlation between the change in the number of resources on a page and the change in connection setup time is weak. Figure 5.9 illustrates the weak correlation for our mobile pages: the correlation coefficient is only 0.24. We observed a similarly weak correlation for our desktop pages as well (0.39).

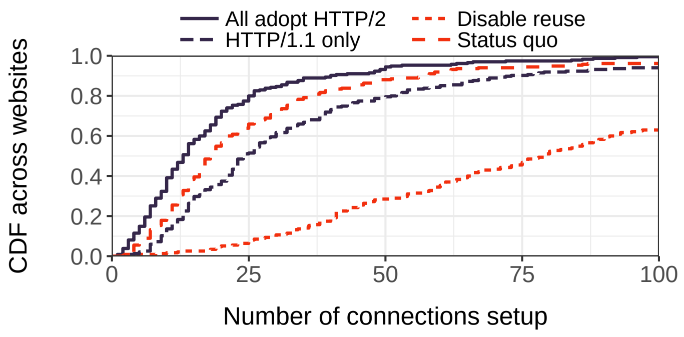


Figure 5.11: Distribution across websites shows that enabling connection reuse even without HTTP/2 yields significant reduction in connections setup. Enabling HTTP/2 further reduces the number of connection setups, but it is yet to reach the best case scenario.

Implication. The weak correlation suggests that there are pages that experience an increase/decrease in number of resources, without a (proportional) increase/decrease in connection setup time. We hypothesize that this is likely due to connections being reused, whereby each request does not necessarily incur a connection setup delay.

To test our hypothesis, we compare the change in the number of resources from 2016 to 2019 to the corresponding change in the *number* of connection setups. Figure 5.10 shows that these two properties change at very different rates. For example, 20% of mobile pages see an increase of more then 50 resources, and roughly 15% see a decrease of 50 resources or more. In contrast, the fraction of sites with either 50 more or 50 fewer connection setups is miniscule.

Enabling connection reuse. Digging deeper into the effects of connection reuse, we observe that reuse decisions are largely dictated by the HTTP protocol and browser operation. Broadly speaking, the HTTP/1.1 protocol enables connection reuse by default; most modern browsers attempt to reuse established connections, and limit the number of concurrent (open) connections per hostname [49]. The more recent HTTP/2 protocol [58] alters this by permitting only a single connection per hostname; browsers can multiplex requests onto each connection to enable even more reuse.

To understand the effects of each protocol on the above analysis, we compare the number of established connections for mobile and desktop pages in 2019 across four scenarios. First, we use only HTTP/1.1 and entirely disable connection reuse by including the `Connection: close` HTTP header in each response [13]; in this case, the number of connections equals the number of resources on a page. Second, we enable HTTP/1.1's default connection reuse, while still not using HTTP/2. Third, we consider the status quo, where HTTP/1.1 and HTTP/2 are intertwined according to origin server support. Fourth, we mandate HTTP/2 for all origins; this establishes a lower bound on the number of established connections, i.e., one per hostname.

Figure 5.11 shows that enabling HTTP/1.1 connection reuse yields the largest drop in connection setup counts. Without reuse, browsers must incur more than 75 connection setups at the median; this lowers to 24 when HTTP/1.1 connection reuse is enabled (a 66% reduction). Further, we observe that the current state of the web, in which only a subset of hostnames enable HTTP/2, marginally improves this number to 18, which is still 5 setups higher than a scenario where all hostnames adopt HTTP/2, i.e., one connection per hostname.

Implication. Recall that in Section 5.4, we find that when a page observes an increase in number of resources, page load performance does not always degrade proportionately, but it does more often than not. However, when pages observe a decrease in resource count, page load performance typically does not improve proportionately. Our findings in this section, i.e., that connection reuse mitigates the effects that a resource has on incurred network latencies, help characterize this behavior. In particular, pages that see an increase in number of resources see a greater increase in number of connections, as compared to the disproportionately small drop in number of established connections witnessed on pages with decreasing resource counts.

A potential explanation for this trend is that, when pages add resources, they also introduce new hostnames, but when pages remove resources, the corresponding hostnames remain in the page as they still contribute other resources (that have not been removed).

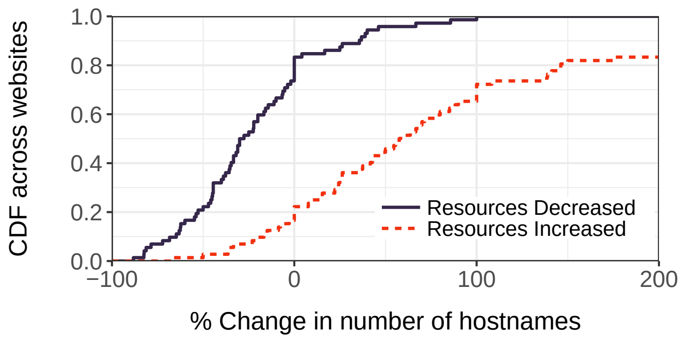


Figure 5.12: For websites with substantial increase/decrease in number of resources, relative change in number of hostnames. When websites see an increase in the number of resources, the magnitude of change in the number of hostnames is larger than that for sites that see a decrease in the number of resources.

Figure 5.12 shows that pages that experience an increase in the number of resources see a greater magnitude of change in number of hostnames. Pages that see a decrease in the number of resources do see a drop in hostname counts, but the change is much less pronounced. For example, 80% of the pages that see a decrease in the number of resources see a reduction in hostnames of at most 50%. For comparison, only 50% of pages with an increase in number of resources observe an increase in number of hostnames below 50%. From this, we conclude that when the number of resources decrease, the number of hostnames fails to decrease proportionately, precluding the expected improvements in overall load times.

5.5.2 Network Bandwidth

Next, we aim to perform a similar analysis (as in Section 5.5.1) in order to understand the reason why many pages with a substantial change in the number of bytes do not see a proportional change in performance.

Performance Metric: Network Index. Page bytes are associated with network bandwidth-related delays (Section 5.2). As such, we seek a performance metric that captures only that aspect of the overall page load performance. Page load time is not appropriate as it is susceptible to latency-based delays that are caused by small straggler requests towards the

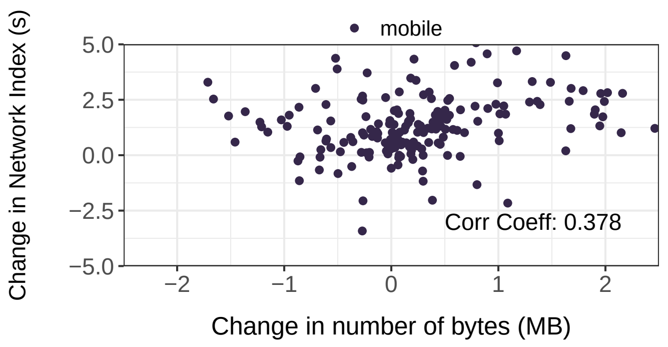


Figure 5.13: Changes in the number of bytes are not highly correlated with changes in Network Index.

end of a page load, e.g., single-pixel images used for ad or analytics services that involve many redirections [70]. Instead, inspired by the Speed Index metric [44], we introduce a new metric called *Network Index* that directly captures the efficiency at which bytes are delivered. More precisely, Network Index measures *“the average time at which a page’s bytes are fetched"*, and is defined as:

$$NetworkIndex = \int_{t=0}^{PLT} 1 - BytesCompleted(t) \tag{5.1}$$

where

$$0 \leq BytesCompleted(t) \leq 1$$

The correlation between the change in number of page bytes and change in Network Index is weak. Figure 5.13 illustrates the weak correlation for our mobile pages: the Pearson’s correlation coefficient is only 0.38. The weak correlation also holds for desktop pages (coefficient of 0.35).

Implication. The weak correlation suggests that there are pages that experience a decrease in bytes without proportional improvements in Network Index, and vice versa. We hypothesize that the reason for these seemingly counterintuitive trends stem from the

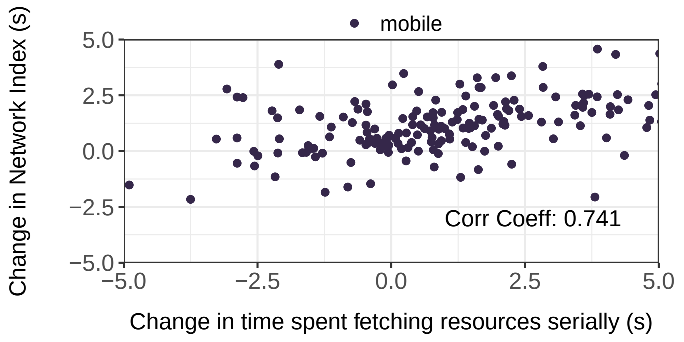


Figure 5.14: Changes in the amount of time that there is at least one ongoing serial fetch are highly correlated with changes in Network Index.

recursive nature of the page load process: not all bytes (and resources) are directly referenced by a page's HTML (and fetched without inter-object blocking delays [25]). Instead, many resources can be fetched only after the fetch and evaluation of some other resource on the page [75, 81]. Consequently, we assert that fetches of resources that cannot be discovered directly from a page's HTML file (i.e., those that would be fetched later in the page load) are the primary influencers of changes in Network Index. These fetches, which we call "serial fetches", comprise of (1) requests made directly by JavaScript code, and (2) requests that involve HTTP redirections, i.e., where a server redirects the fetch of one resource to some other URL.

To test our hypothesis, we correlate the change in the time spent performing serial fetches to the change in Network Index. Similar to the connection setup time from Section 5.5.1, we compute the time spent performing serial fetches as *"the amount of time in a page load during which at least one serial fetch is in progress"*.

Figure 5.14 shows (for mobile pages) that the correlation between the change in time spent performing serial fetches and the change in Network index is strong: the correlation coefficient is 0.74. This finding also holds for desktop pages, where the correlation coefficient is 0.73.

The takeaway from these results is that, when serialization delays exist in a page, they enable the addition of bytes without proportionately degrading performance. This is why the change in a page's Network Index is weakly correlated with the change in number of bytes on the page, but is strongly correlated with the change in serialization-induced delays.

This result is surprising as it goes against the recent flurry of web optimizations which advocate for the removal of serialization in the page load process [75, 81, 60, 79]. Instead, we find that such delays enable pages to evolve in complexity over time without taking a performance hit. This finding also clarifies the reverse trend: removal of page bytes may not improve performance unless the removed bytes also eliminate serialization delays.

5.6 Recommendations and Evaluation

One of the goals of our longitudinal study is to extract actionable insights that 1) are rooted in real changes that pages have made, and 2) enable web developers to strike their desired balance between complexity and performance. In this section, we present an analysis and methodology through which developers could act upon our findings from Section 5.5. We focus our discussion on the takeaways from Section 5.5.2, and seek to assist developers of pages that observe a drop in page bytes without a commensurate improvement in performance. Recall that the main reason for this counterintuitive behavior is the presence of serial fetches which can arise through JavaScript-induced requests or HTTP redirections; we describe how to address each factor, in turn.

5.6.1 Characterizing Redirections

In an ideal case, we would remove all redirections present on a web page. However, doing so automatically is challenging because redirections can be used for synchronizing a user's information between different domains, both within the same administrative organization (e.g., Google Analytics and Doubleclick) and across organizations (e.g., Google and Facebook) [55]; see Section 5.3.3 for how we extract the domain from any URL. We

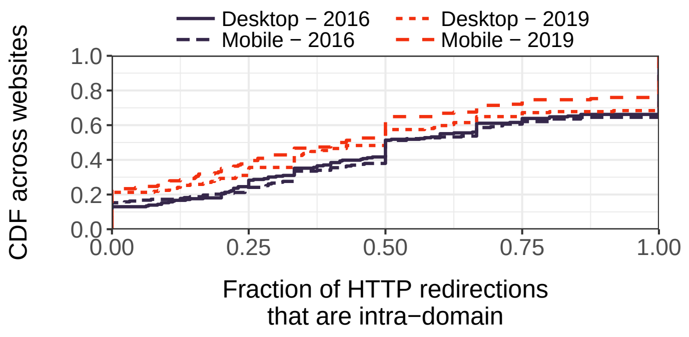


Figure 5.15: On most pages, HTTP redirections are typically within the same domain.

URL Component	Initial URL	Target URL
Scheme	http://www.livescore.com/	https://www.livescore.com/
Hostname	http://www.yohobuy.com/	http://m.yohobuy.com/
Path	https://match.adsrvr.org/track/cmf/generic	https://match.adsrvr.org/track/cmb/generic
Filename	https://luxor.mgmresorts.com/	https://luxor.mgmresorts.com/en.html
Query Params	http://www.acint.net/mc/?dp=10	http://www.acint.net/mc/?dp=10&tc=10

Table 5.2: Examples of how the URL changes within intra-domain redirections.

refer to such redirections as inter-domain redirections, and note that their removal would require coordination between the associated domains to ensure that the functionality that the redirect provided was removable or could be satisfied in other ways.

Our goal in this section is to identify redirections that can be *safely* removed without manual coordination. Given the challenges above, we shift our focus to intra-domain redirections, i.e., redirections in which the initial and target URLs belong to the same domain. As shown in Figure 5.15, on approximately half of the websites, more than 50% of the redirections are within the same domain. Furthermore, on almost 40% of the websites, *all* redirections are intra-domain.

At first glance, it appears that all intra-domain redirects can be eliminated without any risk of altering page functionality since all information in the corresponding URLs is being delivered to the same domain. However, further analysis reveals that such removals require additional care and must consider precisely how the URLs change within a redirect.

URLs comprise of five parts—a scheme (e.g., HTTP or HTTPS), hostname, path, filename, and query string—and are structured as
`scheme://hostname/path/filename?query`. Intra-domain redirects can involve modifications to any subset of these fields; Table 5.2 lists examples of each. Using this structure, we classify intra-domain redirects as removable without manual coordination (or not) according to the following rules:

- Redirects in which the path of the URL changes cannot be automatically removed. The reason is that, while a path change can simply indicate a shift in the position of a resource on the server's file system, it could also be used to point API calls to different services within the same domain (requiring coordination), e.g., match.adsrvr.org/track/cmf/generic and match.adsrvr.org/track/cmb/generic.
- Redirections which involve any combination of changes to scheme, hostname, or filename can be removed. Recall that hostname is not the same as domain, e.g., www.ilbe.com and m.ilbe.com belong to the same domain (ilbe.com), but represent different hostnames.
- Redirections with query string alterations can be removed if the hostname and path remain fixed. Since URLs with the same hostname and path likely correspond to the same service, the query string included in the target URL is generated from this common service's server-side state.

Figure 5.16 shows the breakdown of the URL components that changes between intra-domain redirection pairs. Almost 50% of the intra-domain redirections can be removed without coordination between parties. In both years, redirections with only query parameters changing contributes significantly to the number of intra-domain redirects. In fact, the fraction of redirects with query only changing increases by 5.5 percentage points.

We note that these classifications represent a *conservative* approach, even for the automatic removal of intra-domain redirections; further coordination across services within

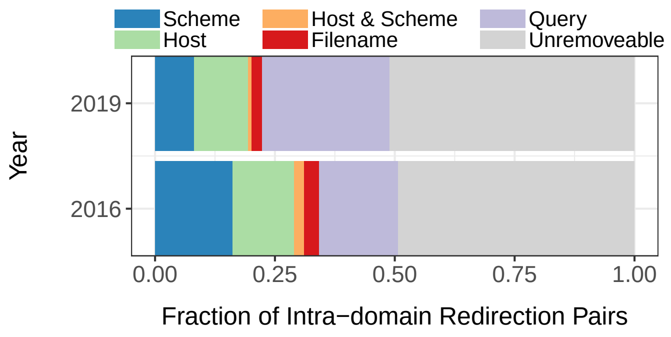


Figure 5.16: Breakdown of the URL component that changes within intra-domain redirections.

a domain could bring additional removal opportunities.

Approaches to removing redirections. At a high-level, we can think of removing redirects from two perspectives. First, for redirects that observe changes only to the query parameters, we argue that the information being passed through the redirection is already present at the server. Thus, we think that web servers should be able to eliminate such redirections by having the server respond to a request for the initial URL with the response it would return for the target URL.

For the remaining redirects that only involve changes to scheme, hostname, and filename, we envision enabling clients to skip such redirections by caching them. There already exist approaches that allow the client to directly request for HTTPS resources using HTTP Strict Transport Security (HSTS) [47]; a domain can use HSTS to inform the client browser that it should always make an HTTPS request when requesting a resource from that domain. Inspired by HSTS, we propose that HTTP response headers could also be used to eliminate hostname redirections.

For example, today, when a server receives a request for a URL such as www.yohobuy.com from a mobile device and responds with a redirection to m.yohobuy.com, this redirection is specific to this particular URL. So, even if the server marked its redirection response as cacheable, the client would again incur the redirection overhead when it issues a request for a different URL on the same hostname, e.g., www.yohobuy.com/lifestyle

Site: https://nationalgeographic.com/ (108 images fetched from JS)

- *Example of image fetched via JS:* https://nationalgeographic.com/.../tv/drk_400x600.jpg
- *Intent:* These images represent the latest shows on National Geographic, which requires up-to-date data from the server.

Site: https://www.foxnews.com/ (45 scripts fetched from JS)

- *Example of script fetched via JS:* https://global.fncstatic.com/static/isa/core-app.js?v=v26
- *Intent:* The site uses JavaScript to determine whether the served page is a "testing" page and serves scripts according to those conditions.

Figure 5.17: Examples of JavaScript-induced fetches.

redirecting to m.yohobuy.com/lifestyle.

We propose that web servers can instead indicate in the HTTP headers of their hostname-only redirections that this change in hostnames is valid for this client independent of the rest of the URL. Depending on the richness of the policy, the server's response could indicate the set of URLs (e.g., in the form of a set of regular expressions) for which this hostname-only redirection can be reused and for how long this redirection policy is cacheable.

5.6.2 Characterizing JavaScript-Induced Fetches

In order to determine how to eliminate JavaScript-induced fetches, we must first understand their operation and what functionality web pages use them for. To that end, Figure 5.17 presents a couple of representative examples, focusing on the two most common resource types fetched via JavaScript: images (44%) and other JavaScript files (28%). As shown in these examples, JavaScript-based fetches primarily enable pages to "dynamically" determine the precise URL to fetch depending on current state (client-side or server-side), e.g., the list of latest shows stored at the National Geographic server described in Figure 5.17.

In order to eliminate JavaScript-induced fetches, we advocate for web servers to directly embed the corresponding URLs into HTML tags prior to serving the main HTML to a client. However, we note that the location of the state that JavaScript-induced fetches rely on must be considered when applying this optimization. In particular, whereas servers are aware of the local (server-side) state that may affect the precise URLs to load, they are not aware of client-side state that may be considered by the JavaScript file. Thus, in order to be conservative and eliminate the risk of altering page functionality or breaking pages, we add the following criteria: only JavaScript-induced fetches that persist across loads of a page should be inlined as HTML tags, as these resources are unlikely to depend on client-side state or nondeterminism (e.g., time of day). We observe that 75% and 50% of the JavaScript-induced fetches for our pages persist across back-to-back loads and across loads separated by two weeks, respectively.

5.6.3 Evaluation: Automated Removal of Serial Fetches

We applied the aforementioned optimizations to our desktop and mobile pages that counter-intuitively (and undesirably) did not benefit from a proportionate speedup when the number of bytes on them was reduced (Table 5.1). To eliminate JavaScript-induced fetches, we listed the corresponding resources in HTTP Link preload tags (i.e., `<link rel="preload">`) [40] that were added to the beginning of the `<head>` section of the main HTML for a page. Resources listed in these headers are fetched immediately after header parsing; note that we did not alter JavaScript files, and instead let the corresponding JavaScript-induced fetches simply hit the local browser cache (which was already populated from the preloads of those resources). For HTTP redirects, we altered our replay setup to return `200 OK` responses to the first request in a sequence of redirects that we deemed could be automatically skipped.

These alterations enable additional pages to experience the desired (expected) performance improvements resulting from their reduction in bytes. For example, 8 more mobile pages experience this expected behavior, where the median site witnesses a 320ms improvement in page load time. We note that these results reflect only conservative modifications to pages in order to act upon our findings in Section 5.5; indeed, more rewarding alterations could be made to further improve performance by considering alterations that affect page functionality, e.g., handling inter-domain redirects.

5.7 Summary

Prior studies on the relationship between web page complexity and performance have focused on studying a corpus of pages gathered at a particular point in time. In this chapter, we highlighted the challenges and utility of instead viewing this relationship from a longitudinal perspective. On the one hand, we showed that studying how page complexity and page load performance have evolved over time requires careful aggregation of data from multiple data sources. On the other hand, we found that a longitudinal analysis reveals new implications of prior findings, namely that inefficiencies in page loads, specifically the use of serial fetches, can prove to be beneficial in the long-term. We hope that our work will spur future efforts to examine the evolution of the web and the corresponding implications.

CHAPTER 6

Conclusion

6.1 Thesis Contributions

In this dissertation, I support the following thesis statement: *client-side CPU and network often go underutilized in web page loads; this underutilization can be reduced to speed up page loads or leveraged to add content to pages without any performance penalty.* Specifically, this dissertation makes the following contributions.

Rendering optimizations to CASPR**.** CASPR is a solution by Google Chrome's Data Saver proxy that moves the execution of JavaScripts from the client to the cloud. When the proxy loads a web page on behalf of clients and constructs a snapshot of the web page, which is highly optimized for users with poor network conditions and slow devices. My contribution lies primarily in identifying and fixing bottlenecks in rendering web page snapshots. Specifically, I implemented progressive rendering in CASPR, allowing users to see progress during the page load. I also optimize the delivery of CSS for CASPR snapshots. Both optimizations lead to an improvement of more than 80% in the first-contentful paint metric.

A rethink of the end-to-end process of loading web pages. While CASPR is able to substantially speed up the loads of web pages, it still suffers the drawbacks of a proxy-based solution; users have to trust that proxies preserve the integrity of HTTPS resources and have to share cookies with the proxies to appropriately handle personalized content.

Next, I developed Vroom, a solution that improves client-side CPU and network utilization while preserving the end-to-end nature of web page loads. Vroom decouples the dependencies between resources on a page by having web servers identify and inform any client about dependent resources it would need in addition to any resource that it requests. Further, it includes a scheduler that appropriately prioritizes the fetches of hinted dependencies. Combining the hinted dependencies with the scheduler, Vroom improves the median load time by 50

Longitudinal study Finally, I conducted a longitudinal study of how the complexity of web pages has changed over time and what effects did the changes have on performance. I found that performance does not always change proportionally to the change in complexity (e.g., complexity increased, but performance did not proportionally degrade). For example, I found that fetches of resources that happen serially (i.e., fetches of JavaScripts and HTTP redirections) are the primary cause of the disproportionate impact of complexity changes on performance, allowing websites to add additional content and functionality without degrading performance.

6.2 Future Work

While the web is an ever changing ecosystem with new technologies constantly being introduced to improve the status quo, achieving substantial speed up in page loads is challenging without fundamental changes to how web page loads work. With the existence of the coupling of network and CPU usage, I believe that the solutions presented in this dissertation will still be applicable in the future. Further, improvements to the network or CPU usage, such as the standardization of QUIC, are still important as they will complement the solutions presented. For the remainder of this section, I outline potential future research areas that are motivated by some limitations of this dissertation.

Approximating benefits of web optimization techniques. Almost all of the evaluations in this dissertation were done in a lab setting. However, given the variety of user devices and network configurations in the wild, the improvement in page load performance when deployed in production may differ depending on a user's device and network.

To allow developers to better reason whether an optimization approach is worth pursuing, I think that a fruitful research avenue would be a system that allows web performance researchers and engineers to *estimate* the benefits of an envisioned optimization without actually implementing it. It should allow developers to swiftly estimate the change in load times in numerous client device and network configuration combinations. This research direction would be extremely useful as it helps researchers and engineers to better reason about the effort versus benefit trade-off.

Ensuring correctness of web optimization techniques. There are numerous solutions which speed up web page loads by transforming web pages into a more performant format. In an ideal world, after applying any page load optimizations, the final state of the page should be equivalent to its unmodified counterpart (e.g., the final DOM and JavaScript heap should be identical between the optimized and unmodified versions of the web page). To my knowledge, there is not any work with regards to checking the correctness of the optimized web pages. Given the abundance of web optimization techniques, I argue that exploring the notion of correctness of web pages is also equivalently important to ensure that the optimization techniques do not degrade user experience on the web.

6.3 Summary

In this dissertation, I describe CASPR, Vroom, and a longitudinal study of web pages to identify key approaches to better utilize client-side CPU and network during web page loads. However, these approaches also have trade-offs among them, for example, an end-to-end modification of page loads requires change to the web server infrastructure, which can be challenging to website owners. I hope that this dissertation can guide stakeholders in the web ecosystem, such as website owners, proxy operators, etc., through navigating the trade-offs associated with different web optimizations and allow them to deploy websites that effectively utilize the client-side CPU and network.

Bibliography

[1] 10 Ways on How to Monetize a Website. https://www.hostinger.com/tutorials/how-to-monetize-a-website/.

[2] Accelerated mobile pages project. https://www.ampproject.org/.

[3] Alexa - Top Sites. https://www.alexa.com/topsites.

[4] Alexa top 1,000,000 sites. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip.

[5] Amazon Found Every 100ms of Latency Cost them 1% in Sales. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/.

[6] Amazon Silk. https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html.

[7] Beautiful Soup. http://www.crummy.com/software/BeautifulSoup/.

[8] Brief History of the G's. https://hpbn.co/mobile-networks/#brief-history-of-the-gs.

[9] Caching | 2019 | the web almanac by http archive. https://almanac.httparchive.org/en/2019/caching.

[10] Chrome DevTools. https://developers.google.com/web/tools/chrome-devtools.

[11] Chrome User Experience Report. https://developers.google.com/web/tools/chrome-user-experience-report.

[12] The Chromium blog: Experimenting with QUIC. http://blog.chromium.org/2013/06/experimenting-with-quic.html.

[13] Connection. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Connection.

[14] Data on third party entities and their impact on the web. https://www.thirdpartyweb.today/. https://github.com/patrickhulce/third-party-web.

[15] Find out how you stack up to new industry benchmarks for mobile page speed. https://www.thinkwithgoogle.com/data/mobile-page-speed-new-industry-benchmark/.

[16] Find out how you stack up to new industry benchmarks for mobile page speed. https://developers.google.com/web/fundamentals/performance/why-performance-matters/.

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

[18] First Contentful Paint. https://w3c.github.io/paint-timing/#first-contentful-paint.

[19] First CPU Idle. https://web.dev/first-cpu-idle/.

[20] Google developers - simulate mobile devices with device mode. https://developers.google.com/web/tools/chrome-devtools/iterate/device-mode/.

[21] Google page speed. https://developers.google.com/speed/pagespeed/.

[22] H2O - the optimized HTTP/2 server. https://h2o.examp1e.net/configure/http2_directives.html#http2-casper.

[23] How fast is Fiber Optic Internet?

[24] How One Second Could Cost Amazon $1.6 Billion In Sales. https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales.

[25] How the browser pre-loader makes pages load faster. https://andydavies.me/blog/2013/10/22/how-the-browser-pre-loader-makes-pages-load-faster/.

[26] HTTP archive. http://httparchive.org/.

[27] HTTP archive project. https://httparchive.org/.

[28] HTTP archive specification. http://groups.google.com/group/http-archive-specification/web/har-1-1-spec?hl=en.

[29] HTTPS adoption *doubled* this year. https://snyk.io/blog/https-breaking-through/.

[30] Hypertext transfer protocol version 2. https://http2.github.io/http2-spec/.

[31] Hypertext Transfer Protocol Version 2 (HTTP/2): Server Push. https://tools.ietf.org/html/rfc7540#section-8.2.

[32] Internet Archive Wayback Machine. https://archive.org/.

[33] Largest Contentful Paint. https://web.dev/lcp/.

[34] Latency is everywhere and it costs you sales - how to crush it. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it.

[35] Lighthouse. https://developers.google.com/web/tools/lighthouse.

[36] Mobile-Friendly vs Mobile-Optimization vs Responsive Design for Websites. http://torspark.com/mobile-friendly-vs-mobile-optimized-vs-responsive-design/.

[37] Mutationobserver. https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver.

[38] nghttpx - HTTP/2 proxy. https://nghttp2.org/documentation/nghttpx-howto.html.

[39] Opera Mini. https://www.opera.com/mobile/mini.

[40] Preload. https://www.w3.org/TR/preload/.

[41] Public Suffix List. https://publicsuffix.org/.

[42] Remove Render-Blocking Javascript. https://developers.google.com/speed/docs/insights/BlockingJS.

[43] Responsive images. https://developer.mozilla.org/en-US/docs/Learn/HTML/Multimedia_and_embedding/Responsive_images.

[44] Speed Index. https://developers.google.com/web/tools/lighthouse/audits/speed-index.

[45] StatCounter global stats. http://gs.statcounter.com/#desktop+mobile+tablet-comparison-ww-monthly-201309-201408.

[46] State of the mobile network: India (april 2018).

[47] Strict-Transport-Security. https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Strict-Transport-Security.

[48] Time to Interactive. https://developers.google.com/web/tools/lighthouse/audits/time-to-interactive.

[49] Using Multiple TCP Connections. https://hpbn.co/http1x/#using-multiple-tcp-connections.

[50] visualmetrics. https://github.com/WPO-Foundation/visualmetrics.

[51] Web Personalization. https://developers.marketo.com/javascript-api/web-personalization/.

[52] Web Vitals. https://web.dev/vitals/.

[53] Webpagetest project. https://www.webpagetest.org/.

[54] What changes have been made to the test environment that might affect the data? https://httparchive.org/faq#what-changes-have-been-made-to-the-test-environment-that-might-affect-the-data.

[55] What is Cookie Syncing and How Does it Work? https://clearcode.cc/blog/cookie-syncing/.

[56] wptagent. https://github.com/WPO-Foundation/wptagent/blob/master/internal/desktop_browser.py#L334.

[57] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's data compression proxy for the mobile web. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, 2015.

[58] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2. http://httpwg.org/specs/rfc7540.html, 2015.

[59] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: Measurements, metrics, and implications. In *Proceedings of the 11th ACM SIGCOMM Conference on Internet Measurement Conference*, 2011.

[60] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, 2015.

[61] A. Chakraborty, V. Navda, V. N. Padmanabhan, and R. Ramjee. Coordinating cellular background transfers using LoadSense. In *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, 2013.

[62] J. L. Contreras and R. Lakshane. Patents and mobile devices in india: An empirical survey. *Vanderblit Journal of Transactional Law*, 2017.

[63] M. Dasari, S. Vargas, A. Bhattacharya, A. Balasubramanian, S. R. Das, and M. Ferdman. Impact of device performance on mobile internet qoe. In *Proceedings of the 18th ACM SIGCOMM Conference on Internet Measurement Conference*, 2018.

[64] S. Englehardt and A. Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[65] T. Everts. Rules for mobile performance optimization. *ACM Queue*, 11(6), 2013.

[66] D. F. Galletta, R. Henry, S. McCoy, and P. Polak. Web site delays: How tolerant are users? *Journal of the Association for Information Systems*, 5(1):1–28, 2004.

[67] Goran Candrlic. What is Above The Fold Time and What to Do With It. https://www.globaldots.com/fold-time/.

[68] igrigorik. Analyzing HTML, CSS, and javascript response bodies. https://discuss.httparchive.org/t/analyzing-html-css-and-javascript-response-bodies/442.

[69] C. Kelton, J. Ryoo, A. Balasubramanian, and S. R. Das. Improving user perceived page load times using gaze. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, 2017.

[70] D. Kumar, Z. Ma, Z. Durumeric, A. Mirian, J. Mason, J. A. Halderman, and M. Bailey. Security challenges in an increasingly tangled web. In *Proceedings of the World Wide Web Conference*, 2017.

[71] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security*, 2016.

[72] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, K. Papagiannaki, and P. Steenkiste. The cost of the "S" in HTTPS. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, 2014.

[73] A. Nazir, S. Raza, D. Gupta, C.-N. Chuah, and B. Krishnamurthy. Network level footprints of Facebook applications. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement Conference*, 2009.

[74] J. Nejati and A. Balasubramanian. An in-depth study of mobile browser performance. In *Proceedings of the World Wide Web Conference*, 2016.

[75] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *Proceedings of 12th USENIX Symposium on Networked Systems Design and Implementation*, 2016.

[76] R. Netravali and J. Mickens. Prophecy: Accelerating mobile page loads using final-state write logs. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[77] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring time-to-interactivity for web pages. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation*, 2018.

[78] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of the USENIX Annual Technical Conference*, 2015.

[79] R. Netravali, A. Sivaraman, J. Mickens, and H. Balakrishnan. Watchtower: Fast, secure mobile page loads using remote dependency resolution. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, 2019.

[80] G. Podjarny, T. Kadlec, M. McCall, Y. Weiss, N. Doyle, and C. Bendell. *High Performance Images.* O'Reilly Media, 2016.

[81] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the mobile web with server-aided dependency resolution. In *Proceedings of the ACM SIGCOMM Conference*, 2017.

[82] A. Sivakumar, S. P. N., V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. 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*, 2014.

[83] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with WProf. In *Proceedings of 12th USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[84] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is SPDY? In *NSDI*, 2014.

[85] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding up web page loads with Shandian. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation*, 2016.