Taming Webpage Complexity to Optimize User Experience on Mobile Devices

A Dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Michael Andrew Butkiewicz

December 2015

Dissertation Committee:

Dr. Harsha V. Madhyastha, Co-Chairperson
Dr. Srikanth V. Krishnamurthy, Co-Chairperson
Dr. Eamonn Keogh
Dr. Vagelis Hristidis
The Dissertation of Michael Andrew Butkiewicz is approved:

________________________________________

________________________________________

________________________________________

Committee Co-Chairperson

________________________________________

Committee Co-Chairperson

University of California, Riverside
Acknowledgments

I am grateful to my advisor, without whose help, I would not have been here.
To my parents for all the support.
ABSTRACT OF THE DISSERTATION

Taming Webpage Complexity to Optimize User Experience on Mobile Devices

by

Michael Andrew Butkiewicz

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, December 2015
Dr. Harsha V. Madhyastha, Co-Chairperson
Dr. Srikanth V. Krishnamurthy, Co-Chairperson

Despite web access on mobile devices becoming commonplace, users continue to experience poor web performance on these devices. Traditional approaches for improving web performance face an uphill battle due to the fundamentally conflicting trends in user expectations of lower load times and richer web content. Embracing the reality that page load times will continue to be higher than user tolerance limits for the foreseeable future, we ask: How can we deliver the best possible user experience?

To establish the positive correlation between rich webpage content and high webpage load times, we perform the first known measurement-driven study of the complexity of web pages and its impact on performance. To mirror a client-side view, we use browser-based measurements of over 2000 websites from 4 geo-diverse vantage points over a 3 week period. We find that we can accurately predict page load times using a handful of metrics, with the number of resources requested (content richness) being the most critical factor.

Given the rising amount of webpage content and webpage load time, strategic reprioritization of content offers a parallel avenue to better the user’s page load experience. To this end,
we present KLOTSKI, a system that prioritizes the content most relevant to a user’s preferences. In designing KLOTSKI, we address several challenges in: (1) accounting for inter-resource dependencies on a page; (2) enabling fast selection and load time estimation for the subset of resources to be prioritized; and (3) developing a practical implementation that requires no changes to websites. Across a range of user preference criteria, KLOTSKI can significantly improve the user experience relative to native websites.

Finally, we investigate the potential to further improve the user experience over that offered by KLOTSKI by pushing all content on a web page to any client attempting to load a page, but find that this offers little to no improvement in page load times due to limited capabilities of the client to consume the pushed content. This result reinforces KLOTSKI’s focus on limited resource reprioritization for fixed time period goals.
# Contents

List of Figures x

List of Tables xiii

1 Introduction 1

2 Webpage Complexity 3

2.1 Abstract . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
2.2 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
2.3 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
2.4 Measurement Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.5 Characterizing Complexity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
  2.5.1 Content complexity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
  2.5.2 Service complexity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
  2.5.3 What do non-origins offer? . . . . . . . . . . . . . . . . . . . . . . . . . . 22
  2.5.4 Summary of main observations . . . . . . . . . . . . . . . . . . . . . . . 27
2.6 Impact on Client Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
  2.6.1 Load times . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
  2.6.2 Variability in load times . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
  2.6.3 Impact of Firebug . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
  2.6.4 Analysis of high load times . . . . . . . . . . . . . . . . . . . . . . . . . . 39
  2.6.5 Summary of main observations . . . . . . . . . . . . . . . . . . . . . . . 41
2.7 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
2.8 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

3 Klotski 49

3.1 Abstract . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
3.2 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
3.3 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
3.4 System overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
  3.4.1 Approach and Challenges . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
  3.4.2 KLOTSKI Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
3.5 Page fingerprint generation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
  3.5.1 High-level approach . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61
3.5.2 Identifying replacements in URLs .............................................. 62
3.5.3 Generating URL patterns ....................................................... 65
3.6 Optimizing page loads ............................................................. 66
  3.6.1 Selecting resources to prioritize ........................................ 66
  3.6.2 Prioritizing selected resources ........................................... 68
3.7 Load time estimation .............................................................. 69
3.8 Implementation ....................................................................... 72
3.9 Evaluation ............................................................................. 73
  3.9.1 Evaluation setup .............................................................. 73
  3.9.2 Improvement in user experience ....................................... 74
  3.9.3 Evaluation of KLOTSKI’s components ............................... 81
3.10 Discussion ........................................................................... 86
3.11 Related work .......................................................................... 87
3.12 Conclusions ......................................................................... 89

4 Push All Proxy ................................. 90
  4.1 Abstract ............................................................................. 90
  4.2 Introduction ......................................................................... 91
  4.3 Measurement Setup ............................................................ 93
  4.4 Results ............................................................................... 94
  4.5 Related ............................................................................... 97
  4.6 Conclusion .......................................................................... 99

5 Conclusions ........................................ 100

Bibliography ........................................ 102
List of Figures

2.1 Comparison of page load times across browsers. ........................................ 9
2.2 Example snippet showing the different fields in a HAR file. From this report, we
can reconstruct the exact sequence of requests and responses, the number and size
of requests for objects of different content types, and identify the server/origin for
each request. .................................................................................................. 10
2.3 Total number of objects loaded on the base web page of websites across rank ranges. 14
2.4 Total number of objects loaded on the base web page of websites across categories. 15
2.5 Median number of requests for objects of different MIME-types across different
rank ranges. .................................................................................................. 16
2.6 Median number of requests for objects of different MIME-types for different cate-
gories. ........................................................................................................ 17
2.7 Fraction of objects accounted for by Flash objects, normalized per category. . . 18
2.8 Median normalized contribution of different MIME types to total bytes downloaded. 19
2.9 Number of distinct servers & origins contacted to load the base webpage for web-
sites across different rank ranges (a and c) and categories (b and d). ............... 20
2.10 Contribution of non-origin services with respect to number of objects, number of
bytes, and download time. ............................................................................ 21
2.11 Normalized contribution of objects from non-origin services in the median case. . 23
2.12 Contribution of non-origin services with respect to number of objects, number of
bytes, and download time. ............................................................................ 24
2.13 Contribution of non-origin services with respect to number of objects, number of
bytes, and download time. ............................................................................ 25
2.14 Comparison of types of content served from origin and non-origin servers on the
median website. ............................................................................................. 25
2.15 Distribution of RenderEnd times across all websites. ................................. 29
2.16 Correlation between RenderEnd and various complexity metrics. ............... 30
2.17 Box-and-whiskers plot confirming the correlation between RenderEnd and number
of objects. ..................................................................................................... 31
2.18 Page load times for websites of different categories. .................................. 32
2.19 Normalized mean-squared error as a function of top k coefficients. ............ 33
2.20 Scatter plot of page load times predicted by the regression model that uses the top 5
features versus the actual page load times. .................................................. 34
2.21 Correlation coefficients between the different metrics and the variability in RenderEnd. 36
2.22 Comparison of page load times with and without Firebug. .............................. 37
2.23 Correlations of complexity metrics with (a) RenderEnd, and (b) variability in RenderEnd, when not using Firebug. ................................................................. 38
2.24 Characterization of load times that are over 3 seconds: variability across time of day 39
2.25 Characterization of load times that are over 3 seconds: (a) determinism per website, and (b) contribution of non-origin content. ....................................................... 39
2.26 Non-landing pages are less complex than landing pages for most sites. ............ 42
2.27 Reduction in (a) number of objects and (b) number of servers with client-side filtering. 43
2.28 Reduction in (a) number of objects and (b) number of servers with phone-specific customization. ............................................................... 43
2.29 Page load times for mobile websites when loaded with mobile and desktop UserAgents. ................................................................. 46
2.30 Bytes saved using minify js, one optimization from Google’s Page Speed for reducing page load times. ....................................................... 47

3.1 Load time comparison for top 200 websites. ......................................................... 53
3.2 Comparison of page load times with various well-known performance optimization techniques. ................................................................. 54
3.3 Page load time when using Google’s compression proxy vs. the number of resources loaded on the page. ................................................................. 55
3.4 Overview of KLOTSKI’s architecture. ................................................................. 58
3.5 Fraction of replaced URLs, comparing loads an hour apart, a day apart, and a week apart. ................................................................. 60
3.6 Given $O_1$ and $O_2$ fetched in two loads of the same web page, we map URLs in $(O_1 - O_2)$ to ones in $(O_2 - O_1)$. ................................................................. 60
3.7 Illustrative example showing dependency structures (a) and (b) being merged into aggregated dependency structure (c). ................................................................. 60
3.8 Example snippets from two loads of youtube.com that illustrate the utility of local similarity based matching. URLs that replace each other are shown in bold. 63
3.9 Choosing a dependency-compliant subset of resources that maximizes utility within load time budget. Each node represents a resource; shaded nodes have high utility. 67
3.10 Illustrative execution of load time estimator. Shaded resources are high utility resources selected for prioritization. Times shown are for events fired in the simulator. 69
3.11 Comparison of fraction of high utility resources loaded within load time budget: Box and whisker plots showing spread across websites ................................................................. 74
3.12 Comparison of fraction of high utility resources loaded within load time budget: CDF across websites of difference between KLOTSKI and the original website. . 75
3.13 Screenshots comparing loads of an example site (http://huffpost.com), 3 seconds into the page load, without and with KLOTSKI. ................................................................. 76
3.14 Comparison of utility delivered when varying the function used to compute the utility values for high utility objects. ................................................................. 77
3.15 Comparison of utility delivered when loading full versions of web pages (by faking a desktop’s User-Agent). ................................................................. 78
3.16 Comparison of utility delivered when loading web pages on a desktop. ............ 78
3.17 Variance across users of utility perceived in a few types of objects. ............ 80
3.18 For 20 users, spread across websites of the difference between KLOTSKI and the original website in the fraction of high utility resources loaded within 2 seconds. ATF considers all above-the-fold content important. 

3.19 False positive/negative matches as a function of back-end’s aggregation window for merging dependencies. 

3.20 Comparison of no. of resources selected to load and no. of resources loaded in practice within the budget. 

3.21 (a) Absolute and (b) relative errors in KLOTSKI’s load time estimates. Comparison with intrinsic variability. 

4.1 Our experimental setup 

4.2 CDF across 100 websites of total page load time for three different load setups. 

4.3 Median fraction of throughput over 100 webpages per 100ms intervals. 

4.4 Fraction of bytes transferred over time during www.salesforce.com load. 

4.5 CDF showing three different load time perspectives on the same 100 website load set.
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Summary of spread across rank ranges of websites in our measurement dataset.</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Summary of key takeaways from our analysis with respect to various web page</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>complexity metrics.</td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>Breakdown of the types of services provided by top 200 non-origins.</td>
<td>26</td>
</tr>
<tr>
<td>2.4</td>
<td>Classification of the services provided by the top-10 non-origin service</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>providers.</td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td>Highest-impact complexity metrics in the regression model for different website</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>categories, and their intersection with aggregate regression model.</td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>Runtime of resource selection algorithm for the median web page as a function</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>of time budget.</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Web access on mobile platforms already constitutes a significant (more than 35%) share of web traffic [59] and is even projected to surpass traditional modes of desktop- and laptop-based access [42, 49]. In parallel, user expectations of web performance on mobile devices and "richness" of content are increasing. Industry analysts report that 71% of users expect websites to load as quickly as on their desktops, and 33% of annoyed users are likely to visit competing sites, resulting in lost revenues [68, 24].

To cater to this need for a faster mobile web, there are a range of proposed solutions such as customizing content for mobile devices [41, 12], specialized browsers [47], in-cloud acceleration solutions for executing scripts [4], new protocols [55], and compression/caching solutions [4, 13]. Despite these efforts, user experience on mobile devices is still woefully short of user expectations. Industry reports show that the median web page takes almost 11 seconds to load over mobile networks even on state-of-art devices [2]; this is the case even for top mobile-optimized retail websites [32]. In fact, several recent studies show that the benefits from the aforementioned optimizations are marginal [95, 156, 5], and they may even hurt performance [151].
Our conjecture is that the increasing complexity of web page content [27, 8, 83] and decreasing user tolerance will outpace the benefits from such incremental performance enhancements, at least for the foreseeable future. For instance, though RTTs on mobile networks halved between 2004 and 2009 [145], the average number of resources on a web page tripled during the same period [8]. Embracing the reality that page load times will continue to be higher than user tolerance limits for the foreseeable future, we ask: How can we deliver the best possible user experience?

Our proposed solution is broken down into 3 main chapters, each representing a distinct but related body of work. Background information on the issues discussed as well as relevant related work will be provided in each chapter separately. Chapter 2 begins our work by establishing the positive correlation between rich webpage content and high webpage load times. We achieve this by conducting a measurement-driven study of webpage complexity and its impact on performance. Given that webpage complexity is continually on the rise, any solution to the poor user experience problem must also acknowledge that the related high load times are here to stay. In Chapter 3 we build such a solution, KLOTSKI, by focusing on reducing the initial time spend waiting for the most important content, rather than total page load time (all content). KLOTSKI achieves this in part through preemptive delivery of webpage resources to the client. In chapter 4 we finish our work by experimentally determining the upper bound of webpage performance improvement possible through such preemptive delivery methods.
Chapter 2

Webpage Complexity

2.1 Abstract

Over the years, the web has evolved from simple text content from one server to a complex ecosystem with different types of content from servers spread across several administrative domains. There is anecdotal evidence of users being frustrated with high page load times. Because page load times are known to directly impact user satisfaction, providers would like to understand if and how the complexity of their websites affects the user experience.

While there is an extensive literature on measuring web graphs, website popularity, and the nature of web traffic, there has been little work in understanding how complex individual websites are, and how this complexity impacts the clients’ experience. This paper is a first step to address this gap. To this end, we identify a set of metrics to characterize the complexity of websites both at a content-level (e.g., number and size of images) and service-level (e.g., number of servers/origins).

We find that the distributions of these metrics are largely independent of a website’s popularity rank. However, some categories (e.g., News) are more complex than others. More than 60%
of websites have content from at least 5 non-origin sources and these contribute more than 35% of the bytes downloaded. In addition, we analyze which metrics are most critical for predicting page render and load times and find that the number of objects requested is the most important factor. With respect to variability in load times, however, we find that the number of servers is the best indicator.

2.2 Introduction

Over the last decade, web pages have become significantly more complex. Originally used to host text and images, web pages now include several content types, ranging from videos to scripts executed on the client’s device to “rich” media such as Flash and Silverlight. Further, a website today fetches content not only from servers hosted by its providers but also from a range of third party services such as advertising agencies, content distribution networks (CDNs), and analytics services. In combination, rendering a single web page today involves fetching several objects with varying characteristics from multiple servers under different administrative domains.

On the other hand, the ill-effects of slow websites are well-documented. Recent surveys suggest two thirds of users encounter slow websites every week [69] and that 49% of users will abandon a site or switch to a competitor after experiencing performance issues [44]. While there is plenty of anecdotal evidence that the increase in web page complexity is a key factor in slowing down websites, formal studies on this topic have been limited. Most prior work on web measurement focuses on characterizing the Web graph [73, 71], analyzing the network footprint of Web traffic [115, 117, 147, 132, 148, 100], or studying the rate of change of content on the Web [97]. While these have contributed to a better understanding of web usage, they do not analyze the websites themselves.
In this paper, we present a comprehensive measurement-driven study of the complexity of web pages today and its impact on performance. We measure roughly 1700 websites from four geographically distributed locations over a 7 week period. These websites are spread across both a wide range of popularity ranks and genre of website categories. In analyzing website complexity, we focus on a client-side view of the landing pages of these sites and not on the dependencies in the back-end server infrastructure—an interesting topic complementary to our efforts.

Understanding the complexity of web pages and its implications is vital on several fronts. With the increasing diversity of client platforms for accessing the Web, it is important for browser developers to identify the aspects of web page complexity that impact user-perceived performance. On the other hand, as website providers increasingly incorporate third-party services such as advertising, analytics, and CDNs into their webpages, they need tools and techniques to evaluate the impact of these services on users. Furthermore, beyond the perspective of any given user or web provider, understanding website complexity is a first step toward solutions for automatically customizing web pages for varying client platforms to achieve the right balance between performance, usability, and business interests.

Our study focuses on two broad questions. First, we quantify the complexity of a web page with a broad spectrum of metrics. We characterize a web page by the content fetched in rendering it—the number of objects fetched, the sizes of these objects, and the types of content. While these features remain largely the same across different rank ranges of websites, we see a marked difference across different website categories. For example, News websites load a significantly higher number of objects than others, whereas Kids and Teens websites host a higher fraction of Flash content.

In addition to characterizing this content-level complexity, we study the complexity of
web pages with respect to the services they build upon. We find that non-origin content accounts for a significant fraction of the number of objects and number of bytes fetched, an observation that holds even on low ranked websites. However, the impact on download time of non-origin content is low—the median contribution to download time is only 15%. Though the most popular third-party services are unsurprisingly analytics and advertising providers, emerging services such as social networking plugins and programming frameworks also appear on a sizeable fraction of websites. A significant difference that we observe in the types of content served from non-origins in comparison to that from website providers themselves is that Javascripts account for a much higher fraction of non-origin objects.

The second focus of our study is to identify the critical complexity metrics that have the most impact on the time to download and render a web page. We find that rather than the total number of bytes fetched to render a website, the number of objects fetched is the most dominant indicator of client-perceived load times. We corroborate this with a linear regression model that predicts page load times with a normalized mean squared error less than 0.1. We also determine that, in contrast to actual load times, variability in load times is better correlated with the number of servers from which content is fetched. Finally, we show that websites either consistently experience high load times or do not, and those that do have a disproportionate fraction of third-party content.

### 2.3 Related Work

There have been many efforts to analyze different aspects of the Web ecosystem. This includes work to understand web structure, tools to improve web performance, and measurements of emerging web applications. We describe these next. Note that most of these efforts focus either on web traffic or web protocols. There has been surprisingly little work on quantifying and understanding website
Structure and evolution: The literature on modeling the Web graph and its evolution focus on the interconnecting links between websites [73, 71] rather than the structure and content of individual websites. Related efforts have studied how the content of individual web pages evolves over time [97]. Recent efforts have also tried to “map” the hosting sites from which content is served [75].

Performance and optimization: As the usage scenarios for the Web have changed, researchers have analyzed inefficiencies in web protocols and suggested improvements [129, 77, 55]. In parallel, there are efforts toward developing better browsers [128], tools to optimize webpages [23, 14, 60], benchmarking tools [93, 58, 87], services for customizing web pages for different platforms [101, 47, 41], and techniques to diagnose performance bottlenecks in backend infrastructures [119] and to debug client performance in the wild [110].

Web traffic measurement: This includes work on measuring CDNs [115], understanding emerging Web 2.0 and AJAX-based applications [117, 147], measuring the network impact of social network applications [132, 148], and characterizing end-user behavior within enterprises [100], and longitudinal studies [105] among many others. These focus on web traffic as observed at the network-level, and not on understanding the structure and performance of individual websites.

Impact of load time on users: Several user experience studies evaluate how page load times impact user satisfaction [82, 99]. There are also commercial services that measure page load times in the wild [33]. These highlight the importance of optimizing page load times. However, there have been few attempts to understand how different aspects of website complexity impact the time to load web pages.

Privacy leakage: Krishnamurthy et al. [114] report the proliferation of third-party services. Our
measurement setup is similar to theirs and we quantify the use of third-party services as well. However, our end goals are very different. In particular, they focus on the privacy implications and observe that a small number of administrative entities (e.g., Google, Microsoft) have broad insights into web access patterns. Our focus, instead is on using the presence of third-party services as a metric to characterize website complexity and on studying their impact on page load times.

**Complexity metrics in other domains:** Other efforts present metrics to quantify the complexity of network protocols [91], network management [79], and systems more broadly [86]. In a Web context, Zhang et al. [164] present metrics to capture ease of web page navigation and Levering et al. [118] analyze the document layout structure of web pages. The high-level motivation in these efforts is the need for quantitative metrics to measure system complexity and understand its impact on performance and usability. Our study follows in the spirit of these prior efforts to quantify website complexity to understand its impact on page load times.

**Characterizing webpages:** The closest related work appears in recent industry efforts: HTTP Archive [27] and at Google [35]. While the data collection steps are similar, we extend their analysis in two significant ways. First, we consider a more comprehensive set of complexity metrics and present a breakdown across different rank ranges and categories. Second, and more importantly, we go a step further and construct models for correlating and predicting performance and variability in performance with respect to the measured complexity metrics. Furthermore, we view the presence and timing of these parallel industry efforts as further confirmation that there is a *key gap* in understanding website complexity and its performance implications. Our work is a step toward addressing this gap.
2.4 Measurement Setup

We begin by describing the measurements we gathered that serve as input for all of our analysis. All our datasets can be downloaded at http://www.cs.ucr.edu/~harsha/web_complexity/. We start with around 2000 websites at random from the top-20000 sites in Quantcast’s list of most popular websites.¹ We annotate these sites with content categories obtained from Alexa.²

To faithfully replicate the actions of a web browser when a user visits a website, we use a browser (Firefox) based measurement infrastructure. We use a “clean” Firefox instance (version 3.6.15) without any ad or pop-up blockers. (We install suitable plugins such as the Adobe Flash player to ensure that websites render properly.) To confirm that our choice of the Firefox web browser for conducting measurements does not introduce any bias, we compare it with other browsers. From a server at UCR, we load the landing pages of the 2000 websites in our dataset with

¹http://www.quantcast.com/top-sites
Chrome, Internet Explorer, Firefox (versions 3 and 9), and Safari. With every browser, we visit each web page five times. Figure 2.1 compares the distribution of median load time per site across the different browsers. We see that load times with Firefox are comparable to measurements with other browsers.

In our Firefox-based measurement agent, we use the Firebug extension (version 1.7X.0b1) with the Net:Export (version 0.8b10) and Firestarter (version 0.1.a5) add-ons to automatically export a log of all the requests and responses involved in rendering a web page. This extension generates a report in the HTTP archive record (HAR) format [28] that provides a detailed record of the actions performed by the browser in loading the page.

Figure 2.2 shows a snippet of a HAR report. It reports two page load metrics - onCon-
Table 2.1: Summary of spread across rank ranges of websites in our measurement dataset.

<table>
<thead>
<tr>
<th>Rank range</th>
<th>Number of websites</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-400</td>
<td>277</td>
</tr>
<tr>
<td>400-1000</td>
<td>298</td>
</tr>
<tr>
<td>2000-2500</td>
<td>330</td>
</tr>
<tr>
<td>5000-10000</td>
<td>443</td>
</tr>
<tr>
<td>10000-20000</td>
<td>400</td>
</tr>
<tr>
<td>All</td>
<td>1748</td>
</tr>
</tbody>
</table>

tentLoad, which is the time taken to start rendering content (we call this RenderStart), and onLoad, the time to completely render all components of the page (we call this RenderEnd). Our main field of interest is the array of request-response entries. Each entry provides a timestamp of when the request was initiated, the time it took to complete the request, the host to which the request was sent, and the size and content type of the response.

We gathered measurements from four geographically distributed vantage points. Three of these vantage points were Amazon EC2 Micro instances running Ubuntu Linux (version 11.04) located in the US-East, Europe, and Asia Pacific regions of EC2. To discount any effects specific to EC2, our last vantage point is a personal desktop at UC Riverside. We choose multiple vantage points to ensure that our choice of measurement site does not introduce any bias.

At each vantage point, we run a measurement agent that periodically (every 60 seconds) selects a website at random from the list of 2000 sites, launches a Firefox instance that loads the base URL (i.e., www.foo.com) for the website, generates the log report in the HAR format, and subsequently terminates the Firefox process. We repeat this for a period of 9 weeks (between May and August 2011) and gather roughly 30 measurements per website on average. Our primary focus is on the root or landing pages of these sites; we present a preliminary study of non-landing pages.

3There is discussion in the web community on good load time metrics; suggestions include “above-the-fold” time, time to first “paint”, time-to-first-byte, etc. [38]. Using a full spectrum of render metrics is outside the scope of our analysis. We pick the two standard metrics reported by the browser itself.
Table 2.2: Summary of key takeaways from our analysis with respect to various web page complexity metrics.

<table>
<thead>
<tr>
<th>Complexity metric</th>
<th>Key result(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Content complexity</strong></td>
<td></td>
</tr>
<tr>
<td>No. of objects</td>
<td>Across all rank ranges, median web page requests over 40 objects and 20% request over 100 objects.</td>
</tr>
<tr>
<td></td>
<td><em>News</em> websites load a significantly greater number of objects than others.</td>
</tr>
<tr>
<td>MIME type</td>
<td>Contribution of various content types is similar across rank ranges.</td>
</tr>
<tr>
<td></td>
<td>Images dominate in fraction of objects, but to a lesser extent with respect to fraction of bytes.</td>
</tr>
<tr>
<td></td>
<td><em>Kids and Teens</em> websites have a significantly greater fraction of Flash content than others.</td>
</tr>
<tr>
<td><strong>Service complexity</strong></td>
<td></td>
</tr>
<tr>
<td>No. of servers</td>
<td>25–55% of websites load content from at least 10 servers.</td>
</tr>
<tr>
<td></td>
<td><em>News</em> websites fetch content from significantly more servers than others.</td>
</tr>
<tr>
<td>Non-origin contribution</td>
<td>60% of websites fetch content from more than 5 non-origins.</td>
</tr>
<tr>
<td></td>
<td>Non-origins make significant contributions to content—30% of objects and 35% of bytes in the median case.</td>
</tr>
<tr>
<td></td>
<td>Contribution of non-origins to page load time is low (80th percentile is 35%) due to browser optimizations.</td>
</tr>
<tr>
<td></td>
<td>Images dominant object type from origins, but Javascript accounts for sizeable fraction of non-origin objects.</td>
</tr>
<tr>
<td></td>
<td>Advertising and analytics services account for most non-origin objects, but CDNs account for most bytes.</td>
</tr>
</tbody>
</table>

We perform the following pre-processing on our data. We discard individual HAR files if they have recorded the number of bytes fetched or the page load time as zero, specify the HTTP status code as not 200, or are malformed. We discard measurements from all websites for which we observe consecutive HAR files spaced less than 60 seconds apart. These correspond to corner cases where the Firebug add-on to Firefox had trouble exporting a HAR file for the web page it loaded. These steps for cleaning the dataset leave us with measurements from 1748 websites. Table 2.1 summarizes the spread of the websites that remain across five rank-ranges.

For each website, we compute the median value for various features, such as number of objects loaded or number of servers contacted, across multiple measurements of the website. We use these median values of features for most of our analysis, though we do consider variation across
samples when studying how complexity impacts variability in page load times.

2.5 Characterizing Complexity

Our analysis of our measurement dataset is two-pronged. First, in this section, we analyze web pages with respect to various complexity metrics. Next, in Section 2.6, we analyze the impact of these metrics on performance. Note that our focus is on capturing the complexity of web pages as visible to browsers on client devices; we do not intend to capture the complexity of server-side infrastructure of websites [119].

We consider two high-level notions of web page complexity. Content complexity metrics capture the number and size of objects fetched to load the web page and also the different MIME types (e.g., image, javascript, CSS, text) across which these objects are spread. Now, loading www.foo.com may require fetching content not only from other internal servers such as images.foo.com and news.foo.com, but also involve third-party services such as CDNs (e.g., Akamai), analytics providers (e.g., Google analytics), and social network plugins (e.g., Facebook). Service complexity metrics capture the number and contributions of the various servers and administrative origins involved in loading a web page.

We begin with the content-level metrics before moving on to service-level metrics. In each case, we present a breakdown of the metrics across different popularity rank ranges (e.g., top 1–1000 vs. 10000–20000) and across different categories of websites (e.g., Shopping vs. News). Here, we only show results for one of the vantage points as the results are (expectedly) similar across vantage points. Table 2.2 summarizes our key findings for the various complexity metrics.
2.5.1 Content complexity

**Number of objects:** We begin by looking, in Figure 2.3, at the total number of object requests required, i.e., number of HTTP GETs issued, to load a web page. Across all the rank ranges in Figure 2.3, loading the base web page requires more than 40 objects to be fetched in the median case. We also see that a non-trivial fraction (20%) of websites request more than 100–125 objects on their landing web page, across the rank ranges. While the top 1–400 sites load more objects, the distributions for the different rank ranges are qualitatively and quantitatively similar; even the lower rank websites have a large number of requests.

Next, we divide the sites by their Alexa categories. For clarity, we only focus on the top-two-level categories from Alexa. To ensure that our results are statistically meaningful, we consider only the categories that have at least 50 websites in our dataset. The breakdown across the categories in Figure 2.4 shows a pronounced difference between categories; the median number of objects requested on *News* sites is nearly $3 \times$ the median for *Business* sites. We suspect that this is
an artifact of News sites tending to cram in more content on their landing pages compared to other sites to give readers quick snippets of information across different news topics.

**Types of objects:** Having considered the total number of object requests, we next consider their breakdown by content MIME types. For brevity, Figure 2.5 shows only the median number of requests for the four most popular content types across websites of different rank ranges. The first order observation again is that the different rank ranges are qualitatively similar in their distribution, with higher ranked websites having only slightly more objects of each type.

However, we find several interesting patterns in the prevalence of different types of content. While it should not come as a surprise that many websites use these different content types, the magnitude of these fractions is surprising. For example, we see that, across all rank ranges, more than 50% of sites fetch at least 6 Javascript objects. Similarly, more than 50% of the sites have at least 2 CSS objects. The median value for Flash is small; many websites keep their landing pages simple and avoid rich Flash content. These results are roughly consistent with recent independent
Figure 2.5: Median number of requests for objects of different MIME-types across different rank ranges.

Figure 2.6 shows the corresponding breakdown for the number of objects requested of various content types across different categories of websites. Again, we see the News category being dominant across different content types. As previously seen in Figure 2.4, News sites load a larger number of objects overall compared to other site categories. Hence, a natural follow-up question is whether News sites issue requests for a proportionately higher number of objects across all content types. Therefore, for each website, we normalize the number of objects of each content type by the total number of objects for that site. The distribution of the median values of the normalized fraction of objects of various content types (not shown) presents a slightly different picture than that seen with absolute counts. Most categories have a very similar normalized contribution from all
content types in terms of the median value. The only significant difference we observe is in the case of Flash objects. Figure 2.7 shows that *Kids and Teens* sites have a significantly greater fraction of Flash objects than sites in other categories.

**Bytes downloaded:** The above results show the number of objects requested across different content types, but do not tell us the contribution of these content types to the total number of bytes downloaded. Again, for brevity, we summarize the full distribution with the median values for different website categories in Figure 2.8. Surprisingly, we find that Javascript objects contribute a sizeable fraction of the total number of bytes downloaded (the median fraction of bytes is over 25% across all categories). Less surprising is that images contribute a similar fraction as well. For websites in the *Kids and Teens* category, like in the case of number of objects, the contribution of
Flash is significantly greater than in other categories. As in the case of the number of objects, we see no significant difference across different rank ranges.

2.5.2 Service complexity

Anecdotal evidence suggests that the seemingly simple task of loading a webpage today requires the client-side browser to connect to multiple servers distributed across several administrative domains. However, there is no systematic understanding of how many different services are involved and what they contribute to the overall task. To this end, we introduce several service complexity metrics.

Number of distinct servers: Figure 2.9(a) shows the distribution across websites of the number of distinct web servers that a client contacts to render the base web page of each website. We identify a server by its fully qualified domain name, e.g., bar.foo.com. Across all five rank ranges, close to 25–55% of the websites require a client to contact at least 10 distinct servers. Thus, even loading simple content like the base page of websites requires a client to open multiple HTTP/TCP
connections to many distinct servers. Also, Figure 2.9(b) mirrors the earlier result from Figure 2.4; News sites have the most number of distinct servers as well.

**Number of non-origin services:** Not all the servers contacted in loading a web page may be under the web page provider’s control. For example, a typical website today uses content distribution networks (e.g., Akamai, Limelight) to distribute static content, analytics services (e.g., google-analytics) to track user activity, and advertisement services (e.g., doubleclick) to monetize visits.

Identifying non-origins, however, is slightly tricky. The subtle issue at hand is that some providers use multiple origins to serve content. For example, yahoo.com also owns yimg.com and uses both domains to serve content. Even though their top-level domains are different, we do not want to count yimg.com as a non-origin for yahoo.com because they are owned by the same entity. To this end, we use the following heuristic. We start by using the two-level domain
identifier to identify an origin; e.g., x.foo.com and y.foo.com are clustered to the same logical origin foo.com. Next, we consider all two-level domains involved in loading the base page of www.foo.com, and identify all potential non-origin domains (i.e., two-level domain not equal to foo.com). We then do an additional check and mark domains as belonging to different origins only if the authoritative name servers of the two domains do not match [114]. Because yimg.com and yahoo.com share the same authoritative name servers, we avoid classifying yimg.com as having a different origin from yahoo.com.

Figures 2.9(c) and 2.9(d) show that, across the different rank ranges and categories, clients need to contact servers in at least 10 different origins for 20–40% of websites. The presence of non-origin content is even more pronounced on News sites; more than 40% of News sites serve content
Figure 2.10: Contribution of non-origin services with respect to number of objects, number of bytes, and download time.

from over 20 non-origin providers. On further inspection, we find that because the landing pages of News sites have to provide content that spans multiple user interests (e.g., sports, weather) they provide links to non-origin affiliates that serve such content as well.

Contribution of non-origin services: The previous result simply counts the number of distinct domains contacted. Next, we quantify the contribution of the non-origin domains along three dimensions: fraction of objects, fraction of bytes, and fractional contribution to total page load time.

Figure 2.10 shows that, in the median case, over 30% of the total number of objects and over 35% of the total number of bytes downloaded are from non-origin services. At the same time, we see that the distribution is pretty heavy-tailed; for 20% of websites, non-origin services account for roughly 80% of the objects and total bytes.

The total number of objects or bytes may, however, not directly translate into download time because modern browsers can parallelize requests to multiple servers. Now, parallelization also makes it inherently difficult to exactly determine the time contribution of non-origins. In light of
this, we use three alternative ways to measure the non-origin contribution to total page load time: (1) the “wall-clock” time where the browser is retrieving content from at least one non-origin (labeled “Time: At Least 1 Non-Origin”), (2) the ratio of the sum of all time spent in downloading non-origin content to the total time spent downloading all content (labeled “Time: Total Cycle Contribution”), and (3) emulating the act of loading the page by disabling all non-origin content using custom Adblock filters (labeled “Time: Block Non-Origin”).

We see in Figure 2.10 that in the median case, content from non-origins contributes to only 15% of the page load time in terms of the At Least 1 Non-Origin and around 25% for the Total Cycle Contribution. These results suggest that though non-origin services play a significant part of the web ecosystem in terms of the fraction of content they contribute, browser optimizations (e.g., pipelining and parallelizing requests to distinct servers) lower their impact on page load times.

2.5.3 What do non-origins offer?

A natural question is what types of content and services do non-origins provide. Beyond a basic curiosity of what non-origin content includes, this also has important performance implications. For example, if most non-origin objects constitute content essential for the user experience, then it might be difficult for website providers to directly optimize their delivery or client-side blocking of non-origin content would adversely affect user experience. However, if most non-origin objects are from services for displaying ads or tracking users, they could potentially be consolidated or optimized. Therefore, in this section, we do a more in-depth analysis of the MIME-types of objects served by non-origins, how they differ from objects served from origins, and also identify the class of services that these non-origins provide.
As a first step, we want to understand what types of content are served by non-origins across different websites. Figure 2.11 shows the breakdown of the different content types served by non-origins, both in terms of the number of objects and their size in bytes. This breakdown is shown for the median website, i.e., the website that loads the median number of objects (or median number of bytes) for each content type. Interestingly, we find that while the vast fraction of the number of objects served by non-origins are images, the relative fraction in terms of number of bytes served is much lower. This is at odds with the normal expectation that the contribution of images to bytes fetched will be larger than their contribution to the number of objects, since images are typically larger than Javascript and CSS objects. Investigating this further, we find that this discrepancy is an artifact of the use of small *gifs* fetched from non-origins for...
analytics services [65]. We illustrate this point in Figure 2.13, which shows the distribution of the number of objects and object size for each MIME-type. We see that though images are the most common type of content, the median size of an image is less than 2 KB—more than an order of magnitude smaller than the median size of a Flash object.

**Origin vs. non-origin content:** Next, we proceed to analyze if the content served by non-origins differs significantly from that served by the origin sites themselves. Figure 2.14 shows the contribution of different MIME-types to the number of objects fetched from origins and non-origins on the median website. The most noticeable difference is that non-origins serve a much higher fraction of Javascript objects while origins serve a greater fraction of images than non-origins.

**Classifying non-origins:** Beyond types of content, we analyze the types of services offered by non-origins. For this study, we rank non-origin domains based on the number of websites in which they occur. Then, for the 200 most popular non-origins, we identify the services they offer by combining
Figure 2.13: Contribution of non-origin services with respect to number of objects, number of bytes, and download time.

Figure 2.14: Comparison of types of content served from origin and non-origin servers on the median website.

three sources of information. First, we look up each non-origin domain in Alexa’s categorization of web domains. Second, we use information from CrunchBase\textsuperscript{4} to identify the type of company that runs the non-origin service. Last, we manually classify the remaining non-origins based on information gleaned from their website or keyword-based heuristics on the objects being fetched from them.

Table 2.3 presents a breakdown of the types of services these top 200 non-origins offer

\textsuperscript{4}http://www.crunchbase.com
Table 2.3: Breakdown of the types of services provided by top 200 non-origins.

<table>
<thead>
<tr>
<th>Type of service</th>
<th>Number</th>
<th>Found in no. of origins</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analytics</td>
<td>65</td>
<td>593</td>
</tr>
<tr>
<td>Advertising</td>
<td>64</td>
<td>233</td>
</tr>
<tr>
<td>Tracking Cookies</td>
<td>23</td>
<td>137</td>
</tr>
<tr>
<td>Services/Widgets</td>
<td>21</td>
<td>142</td>
</tr>
<tr>
<td>CDN</td>
<td>18</td>
<td>218</td>
</tr>
<tr>
<td>Social Networking</td>
<td>5</td>
<td>98</td>
</tr>
<tr>
<td>Programming API</td>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>Total</td>
<td>200</td>
<td>669</td>
</tr>
</tbody>
</table>

and the number of origins in which each category appears. Here, we only consider the 669 origins, in which all the non-origin objects belong to one of these top 200 non-origins. Unsurprisingly, the top two categories of non-origin services are Analytics (e.g., google-analytics and quantserve) and Advertising (e.g., doubleclick and googleadservices). However, even beyond these two service types, we see that each of the non-origin service types are seen on a significant fraction of the 669 origins.

To give an example of the types of non-origin services we encounter, Table 2.4 shows the top 10 non-origins with the type of service they provide and the fraction of sites on which they appear. While many of these are very recognizable ad and analytics providers, we were surprised by some of the less recognized names appearing in a significant fraction of websites. For example, among the top 20, we found other lesser known services like bluekai.com, invitemedia.com, and imrworldwide.com, that each appeared in more than 5% of the websites (not shown).

Finally, we examine the contribution of these non-origin service types on a typical web page. For this analysis, we only consider the 669 websites where all non-origins on their base web page belong to the top 200 non-origins. On each of these 669 websites, we compute the fraction of the number of objects and bytes fetched from non-origins that are attributable to each service type.
Table 2.4: Classification of the services provided by the top-10 non-origin service providers.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Fraction of sites</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>google-analytics.com</td>
<td>0.58</td>
<td>Analytics</td>
</tr>
<tr>
<td>2</td>
<td>doubleclick.net</td>
<td>0.45</td>
<td>Ads</td>
</tr>
<tr>
<td>3</td>
<td>quantserve.com</td>
<td>0.30</td>
<td>Analytics</td>
</tr>
<tr>
<td>4</td>
<td>scorecardresearch.com</td>
<td>0.27</td>
<td>Analytics</td>
</tr>
<tr>
<td>5</td>
<td>2mdn.net</td>
<td>0.24</td>
<td>Ads</td>
</tr>
<tr>
<td>6</td>
<td>googleadservices.com</td>
<td>0.18</td>
<td>Ads</td>
</tr>
<tr>
<td>7</td>
<td>facebook.com</td>
<td>0.17</td>
<td>Social net</td>
</tr>
<tr>
<td>8</td>
<td>yieldmanager.com</td>
<td>0.16</td>
<td>Ads</td>
</tr>
<tr>
<td>9</td>
<td>atdmt.com</td>
<td>0.14</td>
<td>Analytics</td>
</tr>
<tr>
<td>10</td>
<td>googleapis.com</td>
<td>0.12</td>
<td>Prog. API</td>
</tr>
</tbody>
</table>

Figure 2.11(b) plots these fractions for the median website. In keeping with the relative popularity of non-origin services of different types in the top 200 non-origins, Analytics and Advertising account for most non-origin objects. However, content fetched from CDNs dominate with respect to the number of bytes.

2.5.4 Summary of main observations

In summary, our main observations are as follows:

- A website’s rank is not a significant indicator of the content complexity, at least within the top 20K websites.

- However, a website’s category does matter; News sites load significantly more content than others from a lot more servers and origins, while Kids and Teens sites have significantly more Flash content than others.

- Most websites load a surprisingly large number of CSS and Javascript objects.

- Content from non-origins represents a significant fraction of objects and bytes on most web pages, but their impact on download time is relatively low.
2.6 Impact on Client Performance

In the previous section, we measured a range of content-complexity and service-complexity metrics. In this section, we tackle the natural follow-up question: which of these metrics have the most impact on performance.

We consider two performance measures to characterize page load times. RenderStart measures the time at which the browser has completed parsing the HTML and has fetched sufficient content to begin rendering the page, while RenderEnd measures the total time to fetch and render all content on the web page. For each measure, we are interested in both the typical load time for each website and the variability in load time across different samples.

To put our analysis in perspective, Figure 2.15 shows the distribution of the median and 90th percentile of the RenderEnd values for each site across several measurements from one of our vantage points. Rather surprisingly, more than 50% of sites have a median RenderEnd higher than 2 seconds. (We also validated these seemingly high page load times from independent measurements from the HTTP Archive project [27].) User studies and industry surveys show that users are likely to be frustrated beyond this two second threshold [130]. Thus, it is critical to systematically understand what are the key factors affecting page load times.

We use correlation and regression based analysis to identify the key complexity metrics that are the best indicators of page load times and the variability in them. In this analysis, we use a range of complexity metrics from the previous section—the \{absolute value or fraction\} of \{objects, bytes, servers, origins, non-origins\} characterized by \{content MIME-type, service type\}, and whether loaded from \{origin, non-origin, either\}. We also use other aggregate metrics such as the size of the maximum object fetched. For brevity, we only present results for metrics that turned
out to be the most dominant indicators of either absolute load times or their variability.

2.6.1 Load times

Correlation: First, we analyze the correlation between RenderEnd (RenderStart) and various complexity metrics. For this analysis, we compute for each website the median values of RenderEnd (RenderStart) across multiple measurements of that website and the median value of various complexity metrics. Then, across all websites in our dataset, we compute the Pearson and Spearman rank correlation coefficients between the two load time measures and various complexity metrics. Since the results are similar for RenderStart and RenderEnd, we present only the results for RenderEnd. Also, the results are similar for both Pearson and Spearman correlations; hence, for brevity, we only show the Spearman values. To ensure that the correlations we observe are not artifacts of a particular measurement site, we consider each of the 4 vantage points separately. Figure 2.16 shows the Spearman correlation coefficients with respect to various complexity metrics in decreasing order of the median value across the different measurement sites. Across all 4 measurement sites, we see
that the five most correlated metrics are the total number of objects loaded, the number of these objects that are Javascripts, the total webpage size, the number of servers, and the number of origins contacted in loading the page.

Figure 2.17 further visually confirms the strong correlation between RenderEnd and the number of objects requested. Here, we bin websites based on the number of objects on their base web page. Then, for each bin, we construct a box-and-whiskers plot showing the median, $25^{th}$ percentile, and $75^{th}$ percentile plot in the “box” and the min/max values for the whiskers. Further, tying the fact that number of object requests is the most dominant indicator of load times with our observation from Section 2.5 that News sites fetch a significantly larger number of objects than other sites, Figure 2.18 shows that the page load times for News sites are indeed much higher than for other sites.
Regression: The correlation analysis tells us which metrics are good indicators of page load times. Next, we attempt to identify a minimal set of metrics to estimate load times. For this, we augment the above correlation analysis by building a linear regression model using the LASSO technique [72]. Each feature in this model represents one of the various complexity metrics presented in the previous section. We use LASSO instead of simple linear regression because it produces a sparser model; thus, models with LASSO are more robust. To further avoid overfitting, we use a $k \times 2$ cross validation technique. Here, in each run, we partition the set of websites into two halves—a training set and a testing set. For each run, we run the LASSO procedure and record the coefficients for each feature. Then, we build an aggregate model using the average values of the individual coefficients across all runs.

Figure 2.19 shows the normalized root mean-squared error (NRMSE),\textsuperscript{5} as a function of

\textsuperscript{5}If $\hat{X}$ is a vector representing estimates of page load times, and $X$ contains the true values, then NRMSE = $\sqrt{\frac{\sum (X - \hat{X})^2}{\max(X) - \min(X)}}$
the top k selected features. In this figure, we sort the features based on the magnitude of their weights in the model after the cross-validation procedure described above. Then, we emulate the effects of using a model that uses only the top k features.

As a point of comparison, we also considered a naive estimator that simply predicts the mean value of RenderStart and RenderEnd; its NRMSE was around 50% worse than the LASSO estimate (not shown).

We see two key results here. First, the set of top k features roughly mirrors the correlation result we saw earlier. One notable exception is that the total size is relegated down the list. We speculate that because the base web pages of websites are typically not that large, page load times are dominated by the number of RTTs rather than the number of bytes transferred. Thus, having chosen the number of requests and number of scripts as the top 2 features, the marginal contribution

---

6One obvious concern is whether the magnitude of the weights are meaningful if the different features and the load time are in different “scales”. A pre-processing step in LASSO re-scales all features to have zero mean/unit variance and also normalizes the load metric to have zero mean. Thus, the magnitude measures the relative importance of the metric and avoids these scale-related concerns.
of total size is low. Second, we find that the prediction error flattens after the first 4-5 features are selected. In other words, we see a natural diminishing returns effect of adding other features. Identifying a small, but predictive set of features is a critical result for both website designers and performance optimization tools.

Figure 2.20 visually confirms the goodness-of-fit. The X-axis represents the actual page load time (RenderEnd), and the Y-axis represents the load time predicted by the regression model. One interesting artifact is a horizontal cluster near the origin, where the model predicts the same value of RenderEnd for around 30 sites. On further investigation, we discovered that these websites have very sparse landing pages that have little to no content. Thus, the values of the top 5 metrics were zero and the prediction model outputs a constant value.

Having determined the top k complexity metrics for the entire population of websites in our study, we next analyze if there is a significant difference across website categories. We repeat
Figure 2.20: Scatter plot of page load times predicted by the regression model that uses the top 5 features versus the actual page load times.

the regression analysis for each category separately. Table 2.5 shows the top 2 metrics identified by the regression for each category. It also shows the cardinality of the set intersection between the top 5 metrics for each category and the top 5 metrics from the aggregate regression model across all websites (from Figure 2.19).

First, we see that the top 2 metrics are quite diverse across the different categories. For example, the number of images and requests are most critical for News sites, but the total size of Javascript objects and the number of servers are most critical for Games sites. Second, there is a significant overlap in the top 5 metrics between the per-category models and the overall model. In fact, for most categories, the metrics in the intersection are the number of objects, the number of servers, and the number of Javascript objects, which are the top 3 metrics in Figure 2.19. This suggests that while there is some value in customizing the prediction model to identify the critical metrics, the aggregate model is a very useful starting point in itself.
Table 2.5: Highest-impact complexity metrics in the regression model for different website categories, and their intersection with aggregate regression model.

<table>
<thead>
<tr>
<th>Category</th>
<th>Top 2</th>
<th>Top 5 Intersection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business</td>
<td># objects, # images</td>
<td>3</td>
</tr>
<tr>
<td>Technology</td>
<td># js, # origins</td>
<td>3</td>
</tr>
<tr>
<td>Games</td>
<td># servers, size js</td>
<td>3</td>
</tr>
<tr>
<td>Kids and Teens</td>
<td># js, # objects</td>
<td>3</td>
</tr>
<tr>
<td>News</td>
<td># images, # objects</td>
<td>3</td>
</tr>
<tr>
<td>Shopping</td>
<td>size css, # js</td>
<td>3</td>
</tr>
</tbody>
</table>

We also experimented with an enhanced regression model where we added nonlinear terms, e.g., \( \log(X) \) and \( X^2 \), for each feature \( X \). Adding these nonlinear terms does not improve the prediction error and does not change the stability of the top k feature set. We do not present these results given space limitations.

2.6.2 Variability in load times

So far, we studied the impact of various complexity metrics on median page load times. Next, we analyze if the same set of factors impact how load times vary, or if a different set of metrics are critical for predicting variability. Here, we restrict our analysis to only focus on websites where the complexity metrics such as page size and number of object requests do not change significantly across measurements. That is, the variability in RenderEnd cannot be attributed simply to a change in the content of the page across our measurements.

As a starting point, we measure variability in load times for any particular website as the difference between the 75th and 25th percentile values of RenderEnd across different measurements for that website; we consider the difference between the 75th and 25th percentiles instead of between the maximum and minimum values to discount for any client-side effects. Then, we correlate the absolute and normalized (by the median) value of this variability versus the various complexity metrics. As seen in Figure 2.21, in comparison to the earlier correlation result, we find
two differences. First, the correlations are weaker in general (e.g., the highest value is 0.65). Second, the number of servers is the most dominant factor in this case, instead of the number of objects, which was the dominant indicator of absolute load times.

### 2.6.3 Impact of Firebug

In all of our analysis thus far, we have used both complexity metrics and page load times obtained when visiting websites with a Firebug-enabled Firefox browser. While we showed earlier in Section 2.4 that page load times with Firefox are comparable to those measured with other browsers, we now inspect the impact of Firebug. To do so, we load each of the websites in our dataset 5 times each from a server at UCR. We gather these measurements first using a Firebug-enabled Firefox browser...
browser. We then repeat these measurements without Firebug, and use loadtimer\textsuperscript{7} instead to measure page load times. In this latter setup, our measurement agent repeatedly fetches the same URL that points to a coordination server at UCR. The web page fetched from this coordination server only contains a small javascript. Every time that this web page is loaded, the javascript on the page fetches the landing page of one of the websites in our dataset. The script records the times at which the request to fetch the page was issued and at which the onLoad event for that page was fired. It then communicates the difference between these times to the coordination server as the measured page load time.

Figure 2.22 shows that page load times when using Firebug are significantly greater than those measured without Firebug (i.e., using loadtimer). Both sets of measurements were gathered in the same time periods to rule out time-of-day effects. We also confirmed the higher load times with Firebug by fetching web pages with the Firebug-enabled Firefox via a web proxy and measuring the load time at the proxy; we use pcap2har\textsuperscript{8} to convert the network traffic at the proxy

\textsuperscript{7}http://code.google.com/p/loadtimer/
\textsuperscript{8}https://github.com/andrewf/pcap2har
to a HAR file and get the RenderEnd time from this HAR file. We defer for future work a deeper investigation of why Firebug adds this overhead on page load times.

Given the overhead on page load times imposed by Firebug, we repeat our correlation analysis between complexity metrics and page load times. This time we use page load time measurements obtained from a server at UCR using our **loadtimer**-based setup. For any particular website, we consider the load time as the median RenderEnd value from 10 page loads and the variability in load time as the difference between the 75th and 25th percentile values. As before, we obtain the complexity metrics from the HAR files gathered during our measurements from EC2.

Figures 2.23 (a) and (b) present the revised correlation values. We see that the top few metrics that are most correlated with page load times or variability in load times are largely the same. The number of objects is still the most correlated with load time and the number of servers/origins are among the top metrics correlated with load time variability.
2.6.4 Analysis of high load times

In the final part of our analysis, we focus on page loads that a typical user would consider slow. In 2010, Akamai found that a user will typically wait for 3 seconds before navigating away from a web page. Therefore, here, we focus on instances where a web page took longer than 3 seconds to load in our measurements. We refer to such instances as painful page loads.

First, we find that time of day has little effect on whether page load times will be over 3
seconds. Figure 2.24 shows, as a function of hour of the day, the fraction of websites for which we observed at least one painful page load during that period. We see that the occurrence of painful page loads has equal probability throughout the day.

Instead, we find that the probability of painful page loads is indeed a function of the website being fetched. For every website in our dataset, we compute the fraction of our measurements for this site that had a load time of over 3 seconds; we restrict our analysis here to those sites for which we have at least 10 measurements. Figure 2.25(a) plots the distribution of this fraction across websites. We see that 60% of websites had less than 20% of their measurements as painful, whereas 20% of sites took more than 3 seconds to load on over 80% of page loads; only 20% of sites are in the middle ground. Thus, a website that typically has painful page loads is likely to suffer from this problem often, and vice-versa.

Next, we investigate the characteristics of a website that make it more prone to consistently having load times of over 3 seconds. To do so, we divide the websites in our dataset into two partitions—based on whether a site’s median load time is under or over 3 seconds—and compare the values of various complexity metrics across these partitions. As one would expect, the number of objects/bytes in sites that have median load time over 3 seconds is significantly greater than on sites with a median load time less than 3 seconds; in the median case, sites in the former partition have 3x more bytes and 2.9x more objects than the latter set of sites. However, interestingly, we also find that websites with over 3 second load times have a significantly higher fraction of non-origin content. Figure 2.25(b) shows that, of the sites with load times consistently over 3 seconds, the median site fetches 53% of its objects from non-origins, and these non-origin objects comprise 83% of the servers from which content needs to be fetched to load the page. In contrast, the contribution
of non-origins to objects and server is 22% and 67% on sites which typically load in less than 3 seconds.

2.6.5 Summary of main observations

The key takeaways from our analysis of load times are:

- The top five complexity metrics that determine RenderStart and RenderEnd are the total number of objects loaded, the number of these objects that are Javascripts, the total webpage size, and the number of servers and origins contacted in loading objects on the page.

- We can build a sparse model for predicting page load times with a normalized mean squared error less than 0.1.

- Variability of load times is less correlated with our complexity metrics, but number of servers is its most dominant indicator rather than the number of objects.

- High load times are a function of the website being visited and not of the time of day. Sites with consistently high load times have a significantly greater fraction of non-origin content.

2.7 Discussion

The previous sections provide a good first-order understanding of how website complexity affects the user experience. We acknowledge that there are likely to be a much wider range of factors that can affect a user’s web experience: “deeper” non-landing pages, diversity in client-side platforms, use of client and provider tools for customizing websites, and other forms of personalization (e.g., services that require a login). In this section, we present a preliminary discussion of such factors.

Landing vs. non-landing pages: Our study focused on the landing pages of websites. As a preliminary study to evaluate how “deeper” non-landing pages might differ, we consider a random
Figure 2.26: Non-landing pages are less complex than landing pages for most sites.

sample of 100 sites. For each such site, we follow other links to pages within the same domain (e.g., www.foo.com has a link to www.foo.com/bar or x.foo.com/bar) and compute the various complexity metrics for each such non-landing page. For each site and metric, we then look at the difference between the base site and the median across these landing pages, and normalize this difference by the value for the base site. Figure 2.26 shows the distribution across sites of these normalized differences for five key metrics: number of requests, number of servers, fraction of non-origin objects, page size, and download time. We see that other than from the perspective of the fraction of non-origin objects metric, websites on which non-landing pages are less complex than the base site far outweigh the sites on which the opposite is true. We do, however, see a long negative tail with the non-landing pages of some sites being up to $2 \times$ more complex than the base site.

**Choice of browser:** The choice of browser does not affect our complexity analysis in Section 2.5.
Figure 2.27: Reduction in (a) number of objects and (b) number of servers with client-side filtering.

Figure 2.28: Reduction in (a) number of objects and (b) number of servers with phone-specific customization.

However, browsers may vary in the specific strategies in how they parallelize requests, optimize scripts and so on; this could affect the load time analysis in Section 2.6. One additional concern when comparing load time results across browsers is that the semantics of `onLoad` might vary [38]. (This is not a concern for our paper because all our measurements use the same version of Firefox.) We leave for future work the task of understanding these effects.

**Personalized web services:** Some services present different landing pages to users who have subscribed or logged in. For example, `facebook.com` and `igoogle.com` have content that is personalized to the specific user logged in. The key to understanding these effects is to emulate user
profiles that are representative of a broad spectrum of browsing habits (e.g., casual vs. expert users).

While this is relevant to our broader theme, it is outside the scope of this paper.

**Interaction with client-side plugins:** Users today deploy a wide range of client-side browser extensions to customize their own web browsing experience. For example, two popular Firefox extensions block advertisements (Adblock) and scripts (NoScript). We conducted a preliminary study spanning 120 randomly chosen websites from our overall list on how these extensions impact the complexity metrics. We avoid page load time here because these extensions alter the user experience (i.e., it is not showing the same content) and it is unfair to compare the time in this case.

Figures 2.27(a) and 2.27(b) compare two separate browser instances—one with the extension enabled, and the other with the browser instance from our previous measurements. Here, we install each extension with its default configuration. We only show the complexity metrics that were dominant indicators of page load time and variability in load time: number of objects and number of servers. The median number of objects requested reduces from 60 on the vanilla browser to 45 with Adblock and 35 with NoScript. The reduction in the number of servers is even more marked—the median value drops from 8 to 4 with Adblock and to 3 with NoScript. NoScript’s effect is more pronounced than that of Adblock because disabling scripts can in turn filter out objects that would have been fetched as a consequence of the script’s execution.

It is unclear though if the reduction in complexity caused by these client-side controls is entirely good, even though it likely improves page load times. We do not know how this affects the user experience that the website provider intended for the user (e.g., is some useful content being blocked?) and how these may affect the provider’s business interests (e.g., ad click/conversion rates).
**Customization for client platform:** In parallel to the evolution of web pages, the diversity of client platforms (e.g., mobile phones, tablet computers, and even televisions) used for web access has also increased. These platforms vary in their connectivity, display attributes, and user interface. Consequently, providers are interested in customizing the web experience for these platforms.

We considered 120 randomly chosen websites that offer customized web pages for mobile phones. From a desktop PC running firefox (version 3.6.15), we visited these sites once with the default browser setting and once with the browser instrumented to emulate an iPhone (by spoofing the UserAgent string). Figures 2.28(a) and 2.28(b) show that the phone-specific customization dramatically reduces the number of objects fetched and the number of servers contacted. Furthermore, we find that this reduced complexity of phone-specific web pages significantly reduces the page load times experienced on mobile devices. Figure 2.29 shows the load times measured for these websites on a HTC Sensation smartphone over a 4G connection, once with the normal UserAgent and once with a spoofed desktop UserAgent. We see that the median RenderEnd time reduces from 25 seconds with the desktop version to 11 seconds with the version of the web page customized for mobile devices. However, as in the case of client-side controls, it is not clear if this customization affects the user experience compared to a desktop-based experience (e.g., was some content dropped).

**Optimizations for improving performance:** Many tools like PageSpeed from Google [23] suggest optimizations such as compressing images, combining requests for small images and CSS files, and “minify-ing” Javascript objects to improve website performance. Based on some sample websites, we found that minify js appears as a “high priority” suggestion for several websites.9

---

9This is a code compacting tool that removes unnecessary white spaces, and comment lines to reduce the size of
To analyze how this optimization would help websites in the wild, we emulate the effect of running it on each website in our dataset. Figure 2.30 shows the savings this optimization could provide in terms of the fraction of total bytes of Javascript and the fraction of the total size of the web page downloaded. We see that for most websites, the potential savings is quite small. While this result is preliminary and does not explore all possible optimizations, it does hint that optimizations that are broadly perceived as high-priority may not yield high gains for all websites. Thus, we need to explore more systematic tools and new avenues to improve page load times.

Overall, reducing the complexity of web pages in order to improve performance for clients is not straightforward. Though client-side filtering of particular content types may reduce page load times, it can adversely impact user experience and compromise revenue for web providers. On the other hand, website providers need to understand the impact that non-origin content on their web pages has on their users in order to customize for different client platforms. Therefore, we believe an intermediate proxy service that has a view spanning users and websites is necessary to customize Javascript objects.
Figure 2.30: Bytes saved using *minify js*, one optimization from Google’s Page Speed for reducing page load times.

web pages while simultaneously optimizing page load times, retaining the user experience, and preserving the business interests of web providers.

### 2.8 Conclusions

The increasing complexity of web pages and its impact on performance has been anecdotally well-recognized, but there have been no rigorous studies of the same. In this paper, we presented a first attempt at characterizing web page complexity and quantifying its implications. We characterized the complexity of web pages both based on the content they include and the services they offer. We find that a website’s popularity is a poor indicator of its complexity, whereas its category does matter. For example, *News* sites load significantly more objects from many more servers and origins than other categories. Also, we found that though a significant fraction of objects and bytes are fetched from non-origins on most websites, the contribution of non-origins to page load time is minimal in comparison. Our correlation- and regression-based analysis showed that number of objects and
number of servers are the dominant indicators of page load time and variability in page load times, respectively.
Chapter 3

Klotski

3.1 Abstract

Despite web access on mobile devices becoming commonplace, users continue to experience poor web performance on these devices. Traditional approaches for improving web performance (e.g., compression, SPDY, faster browsers) face an uphill battle due to the fundamentally conflicting trends in user expectations of lower load times and richer web content. Embracing the reality that page load times will continue to be higher than user tolerance limits for the foreseeable future, we ask: How can we deliver the best possible user experience?

To this end, we present KLOTSKI, a system that prioritizes the content most relevant to a user’s preferences. In designing KLOTSKI, we address several challenges in: (1) accounting for inter-resource dependencies on a page; (2) enabling fast selection and load time estimation for the subset of resources to be prioritized; and (3) developing a practical implementation that requires no changes to websites. Across a range of user preference criteria, KLOTSKI can significantly improve the user experience relative to native websites.
3.2 Introduction

Web access on mobile platforms already constitutes a significant (more than 35%) share of web traffic [59] and is even projected to surpass traditional modes of desktop- and laptop-based access [42, 49]. In parallel, user expectations of web performance on mobile devices are increasing. Industry analysts report that 71% of users expect websites to load as quickly as on their desktops, and 33% of annoyed users are likely to visit competing sites, resulting in lost revenues [68, 24].

To cater to this need for a faster mobile web, there are a range of proposed solutions such as customizing content for mobile devices [41, 12], specialized browsers [47], in-cloud acceleration solutions for executing scripts [4], new protocols [55], and compression/caching solutions [4, 13]. Despite these efforts, user experience on mobile devices is still woefully short of user expectations. Industry reports show that the median web page takes almost 11 seconds to load over mobile networks even on state-of-art devices [2]; this is the case even for top mobile-optimized retail websites [32]. In fact, several recent studies show that the benefits from the aforementioned optimizations are marginal [95, 156, 5], and they may even hurt performance [151].

Our thesis is that the increasing complexity of web page content [27, 8, 83] and decreasing user tolerance will outpace the benefits from such incremental performance enhancements, at least for the foreseeable future. For instance, though RTTs on mobile networks halved between 2004 and 2009 [145], the average number of resources on a web page tripled during the same period [8]. Therefore, rather than blindly try to improve performance like prior approaches, we argue that we need to improve the user experience even if load times will be high.

Our high-level idea is to dynamically reprioritize web content so that the resources on a page that are critical to the user experience get delivered sooner. For instance, user studies show
a typical tolerance limit of 3–5 seconds [99, 82, 130]. Thus, our goal is to deliver as many high utility resources as possible within this time. Our user studies, however, suggest that the content considered high utility significantly varies across users. Therefore, point solutions that optimize for a single notion of user utility, e.g., by statically rewriting web pages or by dynamically prioritizing above-the-fold objects [30, 88] will not suffice. Instead, we want to develop a general solution that can handle arbitrary user preferences.

However, there are three key challenges in making this approach practical:

- **Inferring resource dependencies**: Scheduling the resources on a web page requires a detailed understanding of the loading dependencies between them. This is especially challenging for dynamically generated web content, which is increasingly common.

- **Fast scheduling logic**: We need a fast (tens of ms) scheduling algorithm that can generate near-optimal schedules for arbitrary user utility functions. The challenge is that this scheduling problem is NP-hard and is inefficient to solve using off-the-shelf solvers.

- **Estimating load times**: Predicting the load time for a given web page is hard due to the complex manner in which browsers parallelize the loading of resources on a web page. Our problem is much worse—we need to estimate the load times for arbitrary loading schedules for subsets of web resources. Furthermore, we need to be able to do so across heterogeneous device and network conditions.

In this paper, we present the design and implementation of KLOTSKI, a practical dynamic reprioritization layer that delivers better user experience. Conceptually, KLOTSKI consists of two parts: a back-end measurement engine and a front-end proxy. The back-end uses offline measurements to capture key invariant characteristics of a web page, while the front-end uses these
characteristics along with user preferences and client conditions to prioritize high-utility content. In
tackling the above challenges, KLOTSKI’s design makes three contributions:

• Though the specific URLs on a page vary across loads, we develop techniques to merge mul-
tiple loads of a page to extract the page’s invariant dependency structure and capture how
resource URLs vary across loads.
• We design a fast and near-optimal greedy algorithm to identify the set of resources to prioritize.
• We create an efficient load time estimator, based on the insight that the key bottleneck is the
link between the client and the KLOTSKI front-end. Thus, we can effectively simulate this
interaction to estimate load times.

We implement KLOTSKI as an integrated proxy-browser architecture [47, 4] that improves
user experience on legacy devices and web pages by using standard web protocols to implement our
reprioritization scheme. Using a range of measurements, system benchmarks, and across a variety
of user utility functions, we demonstrate that: (1) on the median web page, KLOTSKI increases
the fraction of high utility resources delivered within 2 seconds from 25% to roughly 60%; (2) our
dependency representations are robust to flux in page content and typically only need to be updated
once every 4 hours; and (3) our load time estimates achieve near-ideal accuracy.

Looking beyond our specific design and implementation, we believe that the principles
and techniques in KLOTSKI can be more broadly adopted and are well aligned with emerging web
standards [10, 29, 55]. Moreover, while our focus here is on mobile web access, we show that
KLOTSKI can also improve traditional desktop browsing as well.
3.3 Motivation

We begin by confirming that: 1) web performance on mobile devices is still below expectations, and 2) these performance issues exist even with popular optimizations. We also argue that these issues stem from the growing complexity of web content and that this growth is outpacing improvements in network performance.

Web performance on mobile devices: The growing adoption of the mobile web has been accompanied by a corresponding decrease in user tolerance—users today expect performance comparable to their desktops on their phones [68]. To understand the state of mobile web performance, we compared the page load times\footnote{We measure page load time by the time between when a page load was initiated and when the browser’s onLoad event was fired.} of the landing pages of the top 200 websites (as ranked by Alexa) under three scenarios: 1) on a HTC Sensation smartphone using a 4G connection, 2) on the same phone using WiFi, and 3) on a desktop connected to the same WiFi network. For each web page, we run these three scenarios simultaneously to avoid biases due to content variability across loads. For each page, we report the median load time across 5 loads.
Figure 3.2: Comparison of page load times with various well-known performance optimization techniques.

Figure 3.1(a) shows the CDF of load times for the three scenarios, and Figure 3.1(b) shows the normalized load times on the smartphone w.r.t. the desktop case. We see that the mobile load times are significantly worse, e.g., the median webpage is $5 \times$ worse on 4G and $3 \times$ worse even on WiFi. The tail performance is particularly bad, with the $80^{th}$ percentile $\geq 10$ s on both 4G and WiFi.

**Limitations of performance optimizations:** We study three prominent classes of web optimizations used today: split-browsers such as Opera Mini [47], Google SPDY [55], and recommended compression strategies. For the latter two cases, we relay page loads through a SPDY-enabled NodeJS proxy and through Google’s data compression proxy (DCP) [13], respectively. We use the HTC Sensation smartphone with a 4G connection for these measurements.

Initially, we loaded the top 200 websites using these performance optimizations. However, we saw no improvement in load times (not shown). To see if these optimizations can potentially help other websites, we pick the landing pages of 100 websites chosen at random from the top 2000 websites and compare the load times with and without these optimizations in Figure 3.2. While the optimizations help improve load times on some web pages, we see that they increase load times on other web pages, thus resulting in little change in the overall distribution; load times remain con-
Figure 3.3: Page load time when using Google’s compression proxy vs. the number of resources loaded on the page.

siderably higher than the 5-second tolerance threshold typically observed in usability studies [68]. These observations are consistent with other recent studies [156, 95, 151].

**Complexity vs. Performance:** A key reason why these protocol-, network-, and browser-level optimizations are largely ineffective is because web pages have become highly complex [27, 8, 83]. For instance, the number of resources included on a web page is a key factor that impacts load times [157, 83]. Indeed, even in our measurements, Figure 3.3 shows that the load time for every web page using Google’s compression proxy shows a strong correlation with the number of resources on the page; i.e., even with the optimizations, the number of resources continues to be a dominant factor.

Furthermore, past trends indicate that increase in page complexity tends to match or even outpace improvements in network performance. For example, prior studies show that the average number of resources on a web page tripled from 2004 to 2009 [8], while RTTs on mobile networks only halved over the same period [145].

**Takeaways:** In summary, we see that web page loads take a significant amount of time on mobile devices, and that common optimizations offer limited improvements for complex pages. Given
that page complexity is likely to grow at the same or faster rate than improvements in network performance, we need to rethink current approaches to improve the mobile web experience.

3.4 System overview

Embracing the reality that load times will be high despite performance optimizations, we argue that rather than purely focusing on improving performance, we should be asking a different question:

*How can we deliver good user experience under the assumption that page load times will be high?*

In this section, we start with the intuition underlying our approach and discuss practical challenges associated with realizing this goal. Then, we present an overview of the KLOTSKI system to address these challenges.

3.4.1 Approach and Challenges

Our high-level approach is to ensure that resources that the user considers important are *delivered sooner*. Note that we do not block or filter any content, so as to not risk rendering websites unusable. Based on studies showing that users have a natural frustration or tolerance limit of a few seconds [99, 82, 130], our goal is to deliver as many high utility URLs on the page as possible within a (customizable) tolerance threshold of $M$ seconds.

To see how this idea works, consider a hypothetical “oracle” proxy server whose input is the set of all URLs $O = \{o_i\}$ on a web page. Each $o_i$ has an associated load time $t_i$ and a user-perceived utility $Util_i$. The oracle picks a subset of URLs $O' \subseteq O$ that can be loaded within the time limit $M$ such that this subset maximizes the total utility $\sum_{o_i \in O'} Util_i$. The proxy will then prioritize the delivery of these selected URLs.

Using this abstract problem formulation, we highlight several challenges:

- *Page dependencies and content churn:* First, this subset selection view ignores inter-resource
dependencies, e.g., when a page downloads an image as a result of executing a script on the client, the script is a natural parent of the image. To prioritize a high-utility URL $o_i$, we must also prioritize all of $o_i$’s ancestors. Second, because dynamically generated content is common on today’s web pages, we may not even know the set $O$ of URLs before loading the page.

- **Computation time:** Selecting the subset $O'$ that maximizes utility is NP-hard even ignoring dependencies, and adding dependencies makes the optimization more complex. Since the number of URLs fetched on a typical web page is large ($\approx 100$ URLs [6]), it is infeasible to exhaustively evaluate all possible subsets. Note that running this step offline does not help as it cannot accommodate diversity across user preferences and operating conditions (e.g., 3G vs. LTE).

- **Estimating load times:** Any algorithm for selecting URLs to prioritize will need to estimate the load time for any subset of URLs, to check that it is $\leq M$. This estimation has to be reasonably accurate; under-estimation will result in some of the selected high utility URLs failing to load within the user’s tolerance threshold, whereas over-estimating and choosing additional high utility URLs to load if all the selected URLs load well within the time limit $M$ may lead to suboptimal solutions. Unfortunately, predicting the load time for a given subset of URLs is non-trivial. In addition to the dependencies described above, it is hard to model how browsers parallelize requests, parse HTML/CSS files, and execute scripts.

- **Deployment considerations:** Requiring custom features from clients or explicit support from providers reduces the likelihood of deployment and/or restricts the benefits to a small subset of users and providers. Thus, we have a practical constraint—the prioritization strategy should be realizable even with commodity clients and legacy websites.
3.4.2 KLOTSKI Architecture

To tackle the above challenges, we develop the KLOTSKI system shown in Figure 3.4. We envision KLOTSKI as a cloud-based service for mobile web acceleration. There are many players in the mobile ecosystem who have natural incentives to deploy such a service, including browser vendors (e.g., Opera Mini), device vendors (e.g., Kindle Fire’s Silk), cellular providers (offering KLOTSKI as a value-added service), and third-party content delivery platforms (e.g., Akamai). While KLOTSKI requires no changes to client devices or legacy webservers and makes minimal assumptions about their software capabilities, it can also incorporate other optimizations (e.g., caching, compression) that they offer.

We assume there is some process for KLOTSKI users to specify their preferences; we discuss some potential approaches in Section 3.10. Our focus in this paper is on building the platform for content reprioritization, and we defer the task of learning user preferences to future work.

The KLOTSKI back-end is responsible for capturing page dependencies and dynamics
via offline measurements using *measurement agents*.\(^2\) For every page fetched, the agents record and report key properties such as the dependencies between resources fetched, the size (in bytes) of every resource, the page load *waterfall*, and every resource’s position on the rendered display. The back-end aggregates different measurements of the same page (across devices and over time) to generate a compact *fingerprint* \(f_w\) per web page \(w\). At a high-level, \(f_w\) is a DAG, where each node \(i\) is associated with a *URL pattern* \(p_i\). (The role of this URL pattern will become clearer below.) In this paper, we focus specifically on the fingerprint generation algorithm (§3.5) and do not address issues such as coordinating measurements across agents.

The **KLOTSKI front-end** is an enhanced web proxy that *prioritizes* URLs that the user considers important. It uses legacy HTTP to communicate with webservers, and communicates with clients using SPDY, which is now supported by popular web browsers [56]. When a request for a page \(w\) from user \(u\) arrives (i.e., the GET for \texttt{index.html}), the front-end uses \(f_w\), the user’s preferences, and a load time estimator (§3.7) to compute the set of resources that should be prioritized (§3.6).

The front-end can preemptively *push* static resources that need to be prioritized. For other selected resources that are dynamic, however, it cannot know the URLs in the current load until the page load actually executes. Thus, when a new GET request from the client arrives, the front-end *matches* the URL requested against the URL patterns for the selected resources. If a match is found, the front-end prioritizes the delivery of the content for these URLs over other active requests.

\(^2\)The measurement agents can be KLOTSKI’s clients that occasionally run unoptimized loads, or the KLOTSKI provider can use dedicated measurement clients (e.g., [33]).
Fraction of replaced URLs, comparing loads an hour apart, a day apart, and a week apart.

Given $O_1$ and $O_2$ fetched in two loads of the same web page, we map URLs in $(O_1 - O_2)$ to ones in $(O_2 - O_1)$.

### 3.5 Page fingerprint generation

Next, we describe how the KLOTSKI back-end generates web page fingerprints. It takes as input multiple loads of a given webpage $w$ as input, and generates the fingerprint $f_w$ that captures parent-child dependencies across resources on $w$ as well as high-level URL patterns describing each resource on the page.
3.5.1 High-level approach

Prior works such as WebProphet [119] and WProf [157] infer dependencies across URLs for a single load of a web page. Unfortunately, this single-load dependency graph cannot be used as our $f_w$ because the URLs on a page change frequently. Figure 3.5 shows a measurement of the URL churn for 500 web pages. We see that at least 20% of URLs are replaced for $\geq 30\%$ of web pages over the course of an hour and for $\geq 60\%$ of web pages over a week. Due to this flux in content, the dependencies inferred from a prior load of $w$ may cause us to incorrectly prioritize URLs that are no longer on the page or fail to prioritize the new parent of a high utility URL.

Now, even though the set of URLs changes across page loads, there appears to be an intrinsic dependency structure for every web page that remains relatively static. Suppose we construct an abstract DAG from every load of a web page, with an edge from every URL $o$ to its parent $p$. Then, for 90% of the same 500 pages considered above, changes in this DAG, captured by tree edit distance, are less than 20% even after a week (not shown). That is, most pages appear to have a stable set of inter-resource dependencies, with only the URL corresponding to every resource changing across loads.

Based on this insight, we generate the fingerprint $f_w$ as follows. We take multiple mea-
measurements of $w$ over a recent interval of $\Delta$ hours. From this set of measurements, $\text{Loads}_{w,\Delta}$, we identify a reference load $\text{RefLoad}_w$. While there are many possible choices, we find using the load with the median page load time within $\text{Loads}_{w,\Delta}$ works well. Given that the dependency structure is quite stable, we use the dependency DAG for $\text{RefLoad}_w$; in Section 3.8, we describe how we obtain this dependency DAG in our implementation of $\text{KLOTSKI}$.

However, one challenge remains. To ensure that this DAG is reusable across loads, we need to annotate every node in the DAG with a $URL$ pattern that captures the variations of the URLs for this node across loads. We do this as follows. We initialize the page’s dependency structure $D$ as the DAG between URLs fetched during the reference load $\text{RefLoad}_w$. We then iteratively update $D$ by reconciling the differences between $D$ and other loads in $\text{Loads}_{w,\Delta}$. First, for every $o \in \text{RefO}$ (the set of URLs in $\text{RefLoad}_w$) that is absent in some other (non-reference) load $O$, we identify the URL $o' \in O$ that replaced $o$. Then, for every such matched URL, we update the URL pattern at the node in $D$ with a regular expression that matches both the previous URL annotation and the URL for $o'$.

Thus, we have two natural subtasks: (1) Given two different loads $\text{Load}_1$ and $\text{Load}_2$, we need to map URLs that have replaced each other; and (2) We need to capture the variations in a resource’s URL across loads to generate robust URL patterns. We describe these next.

### 3.5.2 Identifying replacements in URLs

Let $O_1$ and $O_2$ be the sets of URLs fetched in two different loads within $\text{Loads}_{w,\Delta}$. While some URLs are present in both loads, some appear only in one. Our goal here is to establish a bijection between URLs fetched only in the first load ($O_1 - O_2$) and those fetched only in the second load ($O_2 - O_1$), as shown in Figure 3.6.
To this end, we rely on three key building blocks:

- **Identical parent, lone replaced child:** We identify the parent(s) for each URL in $O_1$ and $O_2$.

  Then, we consider every URL $p$ that appears in both loads and has children in both loads. Now, if $p$ has only one unmatched child $o$ in $O_1$ and only one unmatched child $o'$ in $O_2$, i.e., $p$'s remaining children appear in both loads\(^3\), then we consider $o'$ to have replaced $o$.

- **Similar surrounding text:** In practice, a single parent resource $p$ may have several children replaced across loads; e.g., a script may fetch different URLs across executions, or the URLs referenced in a page’s HTML may be rewritten across loads. In such cases, we need to identify the mapping between an URL $p$’s children in $(O_1 - O_2)$ and its children in $(O_2 - O_1)$.

  Here, we observe that the relative position of these children in their parent’s source code is likely to be similar. Figure 3.8 shows an example snippet from the main HTML on youtube.com, which fetches different images across loads. As we can see, even as the HTML gets rewritten to fetch a different image, the text within the code surrounding the reference to either image is almost identical.

\(^3\)Or equivalently, all the other children have already been matched.
We use this observation to identify URL replacements as follows. For every URL, Klotski’s measurement agents log the location within its parent’s source code where it is referenced. Then, for every pair \((o, o')\) that have the same parent \(p\), we compute a local text similarity score between the portions of \(p\) that reference \(o\) in the first load and \(o'\) in the second load. We compute this similarity score as the fraction of common content \(^4\) across (1) 500 characters of text on either side of the URL’s reference, and (2) the lines above and below the URL’s reference. We iterate over \((o, o')\) pairs in decreasing order of this score, and declare \(o_i'\) as having replaced \(o_i\) if either URL has not already been paired up and if the score is greater than a threshold.

- **Similar position on display:** Local similarity may fail to identify all URL replacements as not all URLs are directly referenced in the page’s source code (e.g., algorithmically generated URLs). Hence, we also identify URL replacements based on the similarity in their position on the display when the page is rendered. Again, Klotski’s measurement agents log the coordinates of the top-left and bottom-right corner of the visible content of every URL (details in §3.8). We then declare \(o'\) in one load as having replaced \(o\) in another load if the sum of the absolute differences between the coordinates of their corners is less than 5 pixels.

**Putting things together:** We combine the above building blocks as follows. First, we map URLs that have an identical parent and are the only child of their parent that changes across loads. We apply this technique first since it rarely yields false positives. Then, we identify mappings based on the local similarity scores, following which we leverage similarity in screen positions. After these steps match many URLs in \(O_1\) and \(O_2\), there may now be new URLs that share a parent and are the

---

\(^4\)We apply Ratcliff and Metzener’s pattern matching algorithm [143], which returns a value in \([0, 1]\) for the similarity between two strings.
only unmatched child of their parent. Thus, we apply the first step again to further match URLs. At this point, there remain no more URL pairs that are the sole replaced children of their common parent and all URLs that can be matched based on similarity in surrounding text or screen position have already been matched.

### 3.5.3 Generating URL patterns

Now that we are able to identify how URLs fetched on a web page are replaced across loads, we next discuss how KloSkI generates the URL patterns.

While one could use complex algorithms to merge URLs into regular expressions (e.g., [162]), our empirical analysis of thousands of websites shows that over 90% of URL replacements fall in one of three categories:

- **URL argument changes:** When URL \( o \) in one load is replaced by \( o' \) in another, they often differ only in the associated arguments, i.e., the part of the URL following the ‘?’ delimiter. This is common in advertisements, as the argument is dynamically rewritten to select the ad shown. For example, \( \text{c.com/ad.php?arg1} \) in Figure 3.7(a) is replaced by \( \text{c.com/ad.php?arg2} \) in Figure 3.7(b). In such cases, we merge URLs into a regular expression that preserves the common prefix and indicates that any argument is acceptable; \( \text{c.com/ad.php?*} \) in our example.

- **Single token in the URL changes:** Second, when URLs \( o \) and \( o' \) are split into tokens using ‘/’ as the delimiter, they often differ only in one token. This happens when an image on a page is replaced by another image with the same path name, or when an URL includes a hash value that is randomly generated on every load, e.g., in Figures 3.7(a) and 3.7(b), \( \text{d.com/\%tB\#3v/a.js} \) replaces \( \text{d.com/n\#92@H/a.js} \). Here, the merged URL pattern we create is the URL
for $o$, but the token that differs from $o$’s URL is replaced with a wildcard; $d.com/*/a.js$

in our example.

- **Resources fetched from CDNs:** Last, we account for content served via CDNs. For such URLs, the hostname portion of the URL changes across loads only in the first token, when the hostname is split into tokens based on ‘.’. The regular expression that we use replaces only the portion of the hostname that changes with a wildcard, e.g., in Figure 3.7, the regular expression $*.b.com/img.jpg$ captures $cdn1.b.com/img.jpg$ replacing $cdn2.b.com/img.jpg$.

One concern is that these merging techniques may become too generic (i.e., too many wildcards), producing many false matches at the front-end. We show in §3.9 that, with a suitable choice of $\Delta$ to refresh the DAG, this is unlikely to occur.

### 3.6 Optimizing page loads

When a client loads a web page $w$ via the KLOTSKI front-end, the front-end does two things.

First, it selects the subset of resources on the page that it should prioritize. Thereafter, as the client executes the page load, the front-end alters the sequence in which the page’s content is delivered to the client, in order to prioritize the delivery of the selected subset of resources. Next, we discuss how the KLOTSKI front-end performs these tasks.

#### 3.6.1 Selecting resources to prioritize

Recall from §3.4.2 that the KLOTSKI front-end begins selecting the subset of resources to prioritize on a page $w$ once it receives the request for $w$’s main HTML.

The front-end’s resource selection for $w$ uses the *previously computed* fingerprint $f_w$ that characterizes the dependencies and features of resources on $w$. Using $f_w$ in combination with the
user’s preferences, the front-end computes per-resource utilities and constructs an annotated DAG where every node corresponds to a resource on \( w \) and is annotated with that resource’s utility.

As shown in Figure 3.9, our goal is to select a suitable \( \text{DAG-cut} \) in this structure, i.e., a cut that also satisfies the dependency constraints. Formally, given a page’s dependency structure \( D \) and a time budget \( M \) for user perceived load time, we want to select the optimal cut \( C^* \) that can be loaded within time \( M \) and maximizes the expected utility. Now, selecting the optimal cut is NP-hard and it is inefficient to solve using off-the-shelf solvers.\(^5\)

It is clear that we need a fast algorithm for resource subset selection because it is on the critical path for loading web pages—if the selection itself takes too long, it defeats our goal of optimizing the user experience. Hence, we heuristically adapt the greedy heuristic for the weighted knapsack problem as follows.

We associate every resource \( o_i \) in the page, whose utility is \( \text{Util}_i \), with an initial cost \( C_i \) equal to the sum of its size and its ancestors’ sizes in \( D \). Then, in every round of the greedy algorithm, we iterate through all the unselected resources in the descending order of \( \frac{\text{Util}_i}{C_i} \). When considering a particular resource, we estimate the time (using the technique in §3.7) that will be

---

\(^5\)We can formally prove via a reduction from the weighted knapsack problem, but do not present it for brevity.
required to load the selected DAG-cut if this resource and all of its ancestors were added to the cut. If this time estimate is within the budget $M$, then we add this resource and all of its ancestors to the selected DAG-cut; else, we move to the next resource. Every time we add a resource and its ancestors to the DAG cut, we update the cost $C_i$ associated with every unselected resource $o_i$ as the sum of its size and the sizes of all of its ancestors that are not yet in the DAG cut. We repeat these steps until no more resources can be accommodated within the budget $M$ (or all resources have been selected).

3.6.2 Prioritizing selected resources

Having selected the resources to prioritize, there are two practical issues that remain. First, the front-end does not have the actual content for these resources; $f_w$ only captures dependencies, sizes, and position on the screen. Second, the URLs for many of the resources will only be determined after the client parses HTML/CSS files and executes scripts; the KLOTSKI front-end does not parse or execute the content that it serves.

Given these constraints, the front-end prioritizes transmission of the selected resources to the client in two ways. First, for every static resource (i.e., a resource whose node in the page’s $f_w$ is represented with a URL pattern without wildcards), the front-end pre-emptively requests the resource from the corresponding web server and pushes the resource’s content to the client without waiting for the client to request it. However, the front-end cannot do this for any resource whose URL pattern is not static, as the front-end does not know which of the various URLs that match the URL pattern will be fetched in this particular load. Hence, the front-end matches every URL requested by the client against the URL patterns corresponding to the selected resources, and it prioritizes the delivery to the client of URLs that find a match over those that do not. We describe
how we implement these optimizations via SPDY features in §3.8.

3.7 Load time estimation

As discussed in the previous section, our greedy algorithm needs a load time estimator to check if a candidate subset of resources can be delivered within the load time limit $M$. In this section, we begin by discussing why some natural strawman solutions fail to provide accurate load time estimation, and then present our approach.

**Strawman solutions:** One might consider modeling the load time for a subset of resources as some function of key features such as the number of resources, the total number of bytes fetched, or the number of servers/domains contacted. Unfortunately, due to the inter-resource dependencies and the complex (hidden) ways in which browsers issue requests (e.g., interleaving HTML/CSS parsing and script execution vs. actual downloads), these seemingly natural features are poorly correlated with the effective load time. Alternatively, to incorporate the dependencies, we could try to extend the resource loading waterfall (i.e., the sequence in which URLs are fetched and the associated timings) from the reference load $RefLoad_w$. However, this approach also has two key shortcomings: (1) since we are explicitly changing the sequence of requests, the original waterfall is no longer accurate, and (2) it is fragile due to the diversity in load times across clients and network conditions.

**Our approach:** To account for the dependencies and accurately estimate the load time for a given subset of resources, we need to estimate four key timing variables for each URL $o_i$: (a) $ClientStart_i$, when the client requests $o_i$; (b) $ProxyStart_i$, when the front-end starts delivering $o_i$ to the client; (c) $ClientReady_i$, when the client can begin to render or use $o_i$; and (d) $ProxyFin_i$, 
Figure 3.10: Illustrative execution of load time estimator. Shaded resources are high utility resources selected for prioritization. Times shown are for events fired in the simulator.

when the front-end finishes delivering $o_i$. Together, this gives us all the information we need to model the complete page download process for a given subset of resources.

Intuitively, if the link between the client and the front-end is the only bottleneck and the bandwidth is shared equally across current downloads [126], then we can use a lightweight fluid-model simulation of the client-front-end interaction. Given this assumption, we use a simple analytical model to estimate the values of the four variables as described below. We explain this with the example in Figure 3.10, where we have 5 URLs with the DAG $D$ shown and everything except $o_5$ is selected to be prioritized. For clarity of presentation, we describe the case when each $o$ has only one parent.

1. $ClientStart_i$: This depends on the finish time of $o_i$’s parent as well as delays for the client to process the parent; e.g., in Figure 3.10, $o_3$ is requested some time after the completion of $o_1$. Specifically, $ClientStart_i = ClientReady_{p_i} + Gap_i$, where $p_i$ is the parent. $Gap_i$ is the processing delay between the parent and the child; we capture these parent-child gaps from KLOTSKI’s measurements, store these in $f_w$, and replay the gaps with a simple linear extrapolation to account for CPU differences between the measurement agent and the current client.

---

$^6$All times are specified in terms of the client clock.
2. **ClientReady**: In the simplest case, we can simply set ClientReady = ProxyFin; i.e., when the front-end has finished sending the object. However, there is a subtle issue if the front-end had decided to push URL o_i. In particular, the client may not have processed o_i’s parent when the front-end completes delivering o_i. This means that the client will start consuming an URL only when it is ready to issue the request for that URL. Thus, we modify the above expression to ClientReady = max(ProxyFin, ClientStart). In our example, o_2 finishes downloading at ProxyFin = t_4, but the client finishes processing the parent to issue the request for o_2 at ClientStart = t_7.

3. **ProxyStart**: The time at which the front-end can start delivering o_i depends on two scenarios. If o_i was chosen to be pushed (see §3.6.2), then it can start immediately. Otherwise, the front-end needs to wait until the request arrives (e.g., for dynamically generated URLs). If Latency is the latency between the client and the front-end, we have:

   \[
   ProxyStart = \begin{cases} 
   0, & \text{if } o_i \text{ is pushed} \\
   ClientStart + Latency, & \text{otherwise}
   \end{cases}
   \]

   In our example, the front-end has to wait until the dynamically generated URL o_4 has been requested before starting to deliver it.

4. **ProxyFin**: Finally, to compute the time for the front-end to finish delivering an URL, we model the front-end as a priority but work-conserving scheduler with fair sharing. That is, if there are no high-priority URLs to be scheduled, then the front-end will chose some available low priority URL; e.g., in [t_4, t_5], there were no high priority URLs to schedule as o_4 has not yet been requested, so the front-end tries to deliver the low priority URL o_5, but after o_4 is ready,
it preempts $o_5$. Moreover, the bandwidth between the client and the front-end is equally shared across concurrently delivered URLs, e.g., in intervals $[t_1, t_2]$ and $[t_3, t_4]$.

Together, this simple case-by-case analysis provides the necessary information to model the complete page download process for a given subset of resources. As we will see later, our assumptions on the bottleneck link and fair sharing holds reasonably well in practice and this model provides accurate load time estimations.

3.8 Implementation

**Measurement agent:** We implement Android-based measurement agents that load web pages in the Chrome browser. We use Chrome’s Remote Debugging Protocol to extract the inter-URL dependencies in any particular page load. For every URL fetched, this gives us the mime-type, size, parent, and the position within that parent’s source code where this URL is referenced. In addition, when the `onLoad` event in the browser fires, we inject a Javascript into the web page. This script traverses the DOM tree constructed by the browser while loading the page and dumps several pieces of information contained within the node for every resource, e.g., whether it is visible, and if so, its coordinates on screen.

**Front-end:** We implement the KLOTSKI front-end by modifying the NodeJS proxy with the SPDY module enabled [57]. Our front-end uses SPDY to communicate with clients and HTTP(S) to communicate with webservers. For any resource delivered by the proxy to a client, it maps the resource to one of SPDY’s 7 priority levels as follows: a web page’s main HTML is mapped to priority 0, pushed resources have priority 1, resources that are dynamically prioritized (by matching their URLs against regular expressions in the web page’s fingerprint) are assigned priority 2, and all other resources are spread across lower priority levels in keeping with the order in which the NodeJS proxy
assigns priorities by default.

In addition, we require one modification to typical client-side browser configurations in order for them to be compatible with the KLOTSKI front-end. By default, browsers accept resources delivered using SPDY PUSH only if the domain in the resource’s URL is the same as the one from which the page is being loaded [55]. We select the configuration option in Chrome for Android which makes it accept pushed resources from any domain. However, since Chrome accepts a HTTPS resource via SPDY PUSH only if it is pushed by the domain hosting it, we consider all such resources only for dynamic prioritization.

3.9 Evaluation

Our evaluation of KLOTSKI comprises two parts. First, we showcase the improvements in user experience enabled by KLOTSKI across a range of scenarios. Then, we evaluate each of KLOTSKI’s components in isolation. We begin with a description of our evaluation setup.

3.9.1 Evaluation setup

All experiments were conducted using a HTC Sensation smartphone, running Android 4.0.3, as the client. This client connected to a WiFi hotspot exported by a Mac Mini, which in turn obtained its Internet connectivity via a T-Mobile 4G dongle. We use this setup, rather than the phone directly accessing a 4G network, so as to log a pcap of the network transfers during page loads.

For most of our experiments, we use the landing pages of 50 websites chosen at random from Alexa’s top 200 websites. We load the full version of these web pages using Google Chrome version 34.0.1847.116 for Android. We host the KLOTSKI front-end in a small instance VM in Amazon EC2’s US West region.
3.9.2 Improvement in user experience

We evaluate the improvement in user experience enabled by KLOTSKI, compared to page loads that go through an unmodified proxy, in a variety of client/network settings and across a range of user preferences; note that we see little difference in load times when our client directly downloads page content from web servers and when it does so via a vanilla web proxy. In all cases, though resources not visible to the user (e.g., CSS and Javascripts) have a utility score of 0, KLOTSKI may choose to prioritize such resources if doing so is necessary in order to prioritize a high utility resource, due to dependencies.

**Prioritizing above-the-fold content:** First, we consider all resources on a page that appear “above-the-fold” (i.e., resources that are visible without the user having to scroll) as high utility. We assign a utility score of 1 for every high utility object and a score of 0 for all others.

We then load every web page on our smartphone client first without any optimization,
and then via the KLOTSKI front-end. In either case, we log the sequence in which resources were received at the client and later identify the high utility resources delivered within the load time budget.

We ran this experiment varying the load time budget value between 1 and 4 seconds; prior studies suggest most users have a tolerance of at most 5 seconds [68].

For each load time budget value, Figure 3.11 shows the utility delivered to the client within the budget, using either of the page load strategies. For each (time budget, strategy) pair, we present a box and whiskers plot that shows the 10th, 25th, 50th, 75th, and 90th percentiles across websites. We see that KLOTSKI consistently delivers a significantly better user experience. When user tolerance is 2 seconds, the fraction of high utility resources loaded within this limit on the median web page increases from 25% with the original website to roughly 60% with KLOTSKI. Similarly, we see KLOTSKI increasing the utility delivered on the median web page from 50% to almost 80% when the time budget is 3 seconds.

In addition, in Figure 3.12, we plot the distribution across websites of the difference be-
Figure 3.13 illustrates these benefits offered by KLOTSKI by comparing the screenshots 3 seconds into the page load when loading an example website.

between KLOTSKI and the original website in terms of the fraction of high utility resources loaded within the budget. KLOTSKI consistently fares better or no worse than the original website. For time budgets of 1–4 seconds, KLOTSKI manages to deliver an additional 20% of the high utility resources on roughly 20–40% of the websites.
We also compared the utility improvements offered by KLOTSKI to that obtained when loading web pages via a caching proxy. We consider the best case scenario where the proxy has cached all cacheable resources on every web page. However, we found that the user experience with a caching proxy is almost identical to that obtained with a proxy that simply relays communication between clients and web servers without any caching. Caching at the proxy does not offer any benefits because the 4G network between the client and the proxy is the bottleneck here. Hence, KLOTSKI’s proactive delivery of static high utility content to the client is critical to enabling the improvements in user experience that it offers.

**Impact of utility function:** While we assigned a utility score of 1 to all high utility resources in the above experiment, one can also consider assigning different positive scores to different high utility resources. For example, among all above-the-fold resources, the user may derive larger utility from larger objects.

To evaluate the impact on KLOTSKI’s benefits when varying the utility score across different high utility resources, we rerun the previous experiment with two utility functions. For any
Figure 3.15: Comparison of utility delivered when loading full versions of web pages (by faking a desktop’s User-Agent).

Figure 3.16: Comparison of utility delivered when loading web pages on a desktop.

above-the-fold resource that is \( B \) bytes large and occupies an area \( A \) on the display, we assign a utility score of \( \log_{10}(A) \) in one case and \( \log_{10}(B) \) in the other case. For a time budget of 2 seconds, Figure 3.14 compares the improvement in user experience offered by KLOTSKI in these two cases as well as with the binary utility function that we used above. While the precise improvements vary across the utility functions, KLOTSKI improves the utility delivered on the median web page by over 60% in all three cases.

**Utility for full versions of web pages and on desktops:** Though our primary motivation in de-
veloping KLOTSKI is to improve user experience on the mobile web, its approach of reprioritizing important content can also be beneficial in other scenarios. For example, though many websites offer mobile-optimized versions, nearly a third of users prefer the full site experience [43] and 80% of mobile-generated revenue is generated when users view the full site [52]. However, page load times for these full versions are even worse than the poor performance on the average mobile-optimized web page. Similarly, though page load times are typically within 5 seconds on desktops (Figure 3.1(a)), recent surveys [26] show that 47% of users expect a page to load within 2 seconds and that 67% of users expect page loads on desktops to be faster than on mobile devices.

We evaluate KLOTSKI’s ability to improve the web experience in these two scenarios by first loading full versions of web pages on a smartphone, and thereafter, by loading web pages on a desktop with a wired connection. We vary user tolerance from 2 to 5 seconds in the former case, and from 0.7 to 1.3 seconds in the latter. In both cases, we assign a utility score of 1 for all the above-the-fold resources and a score of 0 for other resources. Figure 3.15 and 3.16 show that KLOTSKI’s reprioritization of important content helps significantly improve the user experience even in these cases.

**Personalized preferences:** So far, we considered all above-the-fold content important. We next evaluate KLOTSKI when accounting for user-specific preferences.

To capture user-specific utility preferences, we surveyed 120 users on Mechanical Turk. On our survey site (http://object-study.appspot.com), we show every visitor snapshots of 30 web pages—the landing pages of 30 websites chosen at random. For each page, we pick one resource on the page at random and ask the user to rate their perceived utility of each resource on a range varying from “Strong No” to “Strong Yes” (i.e., on a Likert scale from -2 to 2). We only
consider data from respondents who 1) chose the correct rating for 4 objects known definitively to be very important or insignificant, and 2) gave consistent responses when they were asked again for their opinion on 5 (chosen at random) of the 30 objects that they had rated.

We observe significant variances in user preferences. For example, Figure 3.17 shows the distribution of utilities for four types of resources—has a link, in the top third of a page, larger than 100x100 pixels, or is above-the-fold. In each case, we see that the fraction of resources considered important (“Yes” or “Strong Yes”) greatly varies across users. This validates the importance of KLOTSKI’s approach of being able to account for arbitrary utility preferences, instead of existing approaches [30, 48] that can only optimize above-the-fold content.

For a given user, we use her survey responses to estimate the utilities that she would perceive from the content on other web pages as follows. We partition all URLs into 9 categories, which are such that any given user’s ratings are consistently positive or consistently negative across all the URLs that they rated in that category. For every (user, category) pair, we assign a utility score of 1/0 to all URLs in that category if a majority of the user’s ratings for URLs in the category were
We consider the data gathered from 20 random users and evaluate KLOTSKI taking their preferences into account. For each of the 20 users, Figure 3.18 shows the distribution across websites of the difference between KLOTSKI and the native unmodified page load in terms of the fraction of high utility resources delivered within 2 seconds. For almost all users, we see that KLOTSKI increases the fraction of high utility resources loaded within 2 seconds by at least 20% on over 25% of websites. Moreover, most users see an increase as high as 50% on some websites. On the flip side, only few users see a worse experience on any website.

### 3.9.3 Evaluation of KLOTSKI’s components

The improvement in user experience with KLOTSKI is made possible due to its combined use of several components. We evaluate each of these in isolation next.
Fingerprint generation

Matching replaced resources: First, we evaluate the accuracy with which the KLOTSKI back-end can map URL replacements across page loads. The primary challenge in doing so is that we do not have ground truth data (i.e., pairs of URLs which indeed replaced each other across page loads). It is very hard to identify matches manually, and no prior techniques exist for this task.

Hence, we instead use the following evaluation strategy. We gathered a dataset wherein we fetched 500 web pages once every hour for a week. For every web page, we compare every pair of loads. As mentioned earlier, we find that our first technique for mapping replaced resources – identical parent, lone replaced child – is almost always accurate. Therefore, we consider all replacements identified using this technique as the ground truth, and use this data to evaluate our other two techniques for mapping replaced resources: similar surrounding text and similar position on display. When we apply these two techniques one followed by the other, we find that the matches obtained with a threshold of 100% for the local text similarity and 5 pixels for the display position similarity yield a 96% true positive rate and a 3% false positive rate. While local text similarity and display position similarity result in reasonably high false negative rates when applied in isolation, they enable accurate detection of URL replacements when used in combination.

Aggregation of dependency structures: Recall that KLOTSKI’s back-end generates a fingerprint per web page by aggregating its measurements of that page over an aggregation window $\Delta$, i.e., it aggregates measurements from $\Delta$ hours ago until now. Here, we ask: what should be the value of $\Delta$? The smaller the value of $\Delta$, the URL patterns stored in a page’s dependency structure would not have converged sufficiently to capture the page’s dynamics. The larger $\Delta$, these URL patterns may
become too generic, resulting in many false positive matches when the KLOTSKI front-end uses these patterns for dynamic prioritization of URLs.

We examine this trade-off with the same dataset as above where we loaded 500 pages once an hour for a week. After every hour, we applied the KLOTSKI back-end to generate a dependency structure for every page by aggregating measurements of that page over the past $\Delta$ hours. We then compute the number of false positive and false negative matches when using the patterns in the aggregated dependency structure to match URLs fetched in the next load of that page. Varying $\Delta$ from 1 to 24 hours, Figure 3.19 plots the false positive rate and false negative rate on the median web page. We see that an aggregation window of 4 hours presents the best trade-off.

**Resource selection**

Having already demonstrated the utility improvements offered by KLOTSKI, we now evaluate the correctness and efficiency of its selection of resources to prioritize.

First, we evaluate KLOTSKI’s ability to accurately account for flux in page content when selecting resources. For every web page, we evaluate whether the number of resources selected by
Figure 3.20: Comparison of no. of resources selected to load and no. of resources loaded in practice within the budget.

<table>
<thead>
<tr>
<th>Time budget (sec)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median runtime (ms)</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 3.1: Runtime of resource selection algorithm for the median web page as a function of time budget.

KLOTSKI’s front-end for prioritization for a load time budget of 2 seconds match the number of high utility URLs received by the client within 2 seconds. These two values can differ due to errors in KLOTSKI’s load time estimates and due to inaccuracies in KLOTSKI’s dependency structure. Figure 3.20 plots the distribution across web pages of the absolute value of the relative difference between these two values.

We see that our error is less than 20% on roughly 80% of websites (i.e., the number of resources delivered within the budget is within 20% of the number selected by the front-end), thus validating the correctness of KLOTSKI’s fingerprints and the accuracy of its load time estimator.

Second, we examine the overhead of KLOTSKI’s greedy resource selection algorithm. Recall that the execution of this algorithm is on the critical path of loading a web page, since the front-end begins executing the algorithm only when it receives the request for the page’s main HTML. Table 3.1 shows that, across a range of budget values, the runtime of the front-end’s resource
selection is within 10 ms for the median web page. Given that the load time of the main HTML is itself greater than 500 ms on over 90% of web pages, combined with the fact that the average size of KLOTSKI’s fingerprint for a web page is 1.6 KB (more than an order of magnitude lesser than the average size of the main HTML [27]), this shows that the front-end can fetch the fingerprint and finish executing the resource selection algorithm before it completes delivering the main HTML to the client.

Load time estimation

Finally, we evaluate the accuracy of KLOTSKI’s load time estimator. We apply KLOTSKI to estimate the load time when web pages are loaded via the front-end, albeit without the front-end prioritizing any content. We compute the absolute and relative error in KLOTSKI’s load time estimates compared to measured load times. Since a source of error here is the intrinsic variability in load times, we get a distribution of load time variability as follows. We load every web page 10 times and partition these loads into two sets of 5 loads each. For each page, we then compute the difference between the median measured load times in the two partitions. Figure 3.21 shows that the errors in KLOTSKI’s
load time estimates closely match the distribution of this intrinsic variability.

3.10 Discussion

Optimizing other metrics: Though KLOTSKI maximizes the utility delivered within a time budget, our design can easily accommodate other optimization criteria. For example, to help users cope with data caps, KLOTSKI’s greedy algorithm can be modified to select a subset of resources that maximizes utility subject to a limit on the total number of bytes across the selected resources; all unselected resources can be blocked by the front-end. Similarly, given appropriate models of energy consumption, the front-end can deliver a subset of resources that keep energy consumed on the client within a limit.

Utility function: To obtain a user’s preferences, we can have the user take a one-time survey (similar to our user study in Section 3.9.2) when she first begins using KLOTSKI. Alternatively, since going over several objects and rating them upfront can be onerous, KLOTSKI can initially start with a default utility function for every user, and on any web page that the user visits, we can provide the user the option of marking objects as low utility (say, whenever the page takes too long to load); e.g., the Adblock Plus browser extension similarly lets users mark ads that the user wants it to block in the future.

Measurement scheduling and personalized websites: Since websites differ in the rate of flux in their content, KLOTSKI’s back-end can adaptively vary measurements of different web pages. For example, the back-end can more frequently load pages from news websites as compared to websites used for reference, since content in the former category changes more often than in the latter category. In addition, for any web page that personalizes content, the back-end can load the web page using different user accounts to capture which parts of the page stay the same across users.
3.11 Related work

Our contribution here is in recognizing the need for, and presenting a practical solution for, improving mobile web experience by dynamically prioritizing content important to the user, rather than trying to reduce page load times. Here, we survey prior work related to KLOTSKI.

Measuring and modeling web performance: Prior efforts have analyzed the complexity of web pages [27, 35, 83] and how it impacts page load times [83, 158, 85] and energy consumption [157, 80, 146]. Our work is motivated by such measurements.

Characterizing webpage content: Song et al. [153] develop techniques to partition webpages into logical blocks to identify blocks that users consider important. Other related efforts compare different pages in the same website [163, 120] or use the DOM tree [102]. Unlike these previous efforts, KLOTSKI associates utilities with individual resources rather than blocks. The closest works on dependency inference are WebProphet [119] and WProf [157]. KLOTSKI infers a more high-level structure robust to the flux in content across loads.

Better browsers and protocols: There are several proposals for new web protocols (e.g., SPDY [55]), guidelines for optimizing pages (e.g., [23]), and optimized browsers (e.g., [125, 123, 47]) and hardware [128, 165]. Many studies have shown that these do not suffice, e.g., two recent studies [156, 95] show that SPDY-like optimizations do not improve performance significantly and interact poorly with cellular networks. KLOTSKI pursues a complementary approach to prioritize important resources.

Cloud-based mobile acceleration: KLOTSKI’s architecture is conceptually similar to recent cloud-
based mobile web acceleration services (e.g., [4, 47]). A recent study suggests that these can hurt performance [151]. The key difference is that our objective is to maximize user-perceived utility rather than optimize page load times.

**Web prefetching:** A widely studied approach for improving web performance on mobile devices has been to prefetch content [140, 108, 96]. However, despite the large body of work on accurately predicting what content should be prefetched [141, 131, 89], prefetching is rarely used in practice on mobile devices due to the overheads on energy and data usage imposed by prefetching content that is never used [144]. KLOTSKI’s approach of pushing high utility resources on a web page to a client only once the client initiates the load of that page improves user experience without delivering unnecessary content.

**Prioritizing important content:** Concurrent to our work, some startups (e.g., InstartLogic and StrangeLoop Networks) try to deliver higher priority resources earlier. Based on public information [64, 43], these appear to optimize certain types of content such as images and Flash, and do not incorporate user preferences like KLOTSKI. We are not aware of published work that highlights how they address the challenges w.r.t. dependencies, optimization, and load time estimation that we tackle. Moreover, their approach requires website providers to use their CDN services, whereas KLOTSKI does not explicitly require any changes to web providers.

Older efforts that dynamically re-order the delivery of web content are limited to prioritizing above-the-fold resources [30, 88]. Based on the observation from our study that users significantly differ in the content that they consider important on the same page, we instead design and implement KLOTSKI to account for arbitrary utility functions.
3.12 Conclusions

Our work tackles a set of contradictory trends in the mobile web ecosystem today – users desire rich content but have decreasing tolerance, even as current performance optimizations yield low returns due to increasing website complexity. In light of these trends, KLOTSKI takes the stance that rather than blindly try to improve performance, we should try to dynamically reprioritize the delivery of a web page to deliver higher utility content within user tolerance limits.

We addressed several challenges to realize this approach in practice: dependencies across content on a page, complexity of the optimization, difficulty in estimating load times, and delivering benefits with minimal changes to clients and webservers. Our evaluation shows that KLOTSKI’s algorithms tackle these challenges effectively and that it yields up to a 60% increase in user-perceived utility. While our focus was on the imminent challenge of improving mobile web user experiences, the ideas in KLOTSKI are more broadly applicable to other scenarios (e.g., desktop) and requirements (e.g., energy).

Acknowledgment

We thank Google’s support of this work in the form of a Faculty Research Award.
Chapter 4

Push All Proxy

4.1 Abstract

Despite increased reliance on smartphones to access the web, users continue to experience poor web performance on these devices. Loading dependencies between webpage resources can cause decreased network utilization, and even dead-periods, while the browser is forced to iteratively resolve resource loading chains.

The resource push feature of the increasingly popular SPDY protocol offers a possible solution by allowing the server to effectively break resource dependency waiting requirements. However, relatively little work has been done on testing the benefits of SPDY pushing on webpage load times in a mobile environment.

In this paper, we conduct a series of experiments to quantify the expected benefits of resource pushing on a real mobile device. We find that while network utilization improves and the client is able to receive resources much earlier than normal, due to limited mobile device hardware, the client’s browser is unable to consume the resources at the same rate. We see little improvement in terms of load time benefits of pushing with a caching proxy over the use of a caching proxy.
4.2 Introduction

In our previous work we developed Klotski [84], a custom web proxy with the ability to preemptively push webpage resources to a client browser during a page load. While Klotski can deliver significant utility gain during a brief initial load time window, it is limited in its ability to optimize for total page load time. This is due to limitations on which resources it is able to push as well as its focus on a brief time frame budget.

To overcome these limitations, our new work explores a more extreme design point, with the ultimate goal being to reduce total web page load time. We will first describe the proposed design of a ”push all web proxy”, able to maximize the fraction of resources available for preemptive delivery. In order to approximate what this theoretical system could expect as an upper bound on total load time improvement, we then design and run a series of experiments which simulate the functioning of such a system.

A major challenge in preemptive delivery lies in determining which resources can confidently be expected to be present on the webpage during the client’s load. Klotski’s design allows it to identify a subset of resources as being sufficiently static to preemptively push to the client. By merging previously gathered webpage resources across time from different clients, it can filter out resources deemed too dynamic (i.e. unpredictable). This filtering limits the scope of optimization possible. To account for resource churn over time and per user, our new push all proxy would generate a resource graph in real time with the client’s load, while taking client specific state into account. This way, these previously dynamic resources become static for the purposes of a specific load by a specific client and thus become pushable.
To generate a dependency graph on the fly, the proxy would start a mirrored webpage load in a headless browser of its own when it receives the initial webpage request from the client. As the proxy’s browser receives resources from the web server, it could start pushing them down to the client. In order to make sure that the proxy’s browser receives the same content as will be requested the client, the proxy would need to share client state (i.e. cookies). If we assume that the client has made all previous webpage interaction through the proxy as well, then the proxy could simply save all proxied cookies from the client and reuse them in later webpage visits. While this method maximizes the resources which can be pushed, it may be the case that a client already has an resource in its cache. With the SPDY protocol, browsers are currently required to cancel any push resource stream if the resource is already in their cache. This solution is good, but may waste some network resources as a partial resource may be transferred before the proxy receives the client’s cancellation notice. One solution is for the client to include cache state information along with the initial webpage request.

We leave the full implementation of our proposed push all proxy to future work. In the following sections we build a simplified proxy server which is able to emulate the push all proxy in a best case scenario. This is meant to give us an upper bound on what preemptive resource delivery techniques can provide. We focus on total page load time, defined as the time until the last resource has been consumed by client browser, as our primary measurement of improvement. Our web proxy is given a fully primed cache and a complete list of pushable resources. Once the HTML request is received, the proxy is allowed to push all resources in dependency order. We compare this optimal setup to a normal, cache only, and pushing only setup to isolate the contribution from each.
4.3 Measurement Setup

Figure 4.1 illustrates our experimental setup. All experiments were conducted using a Nexus 4 smartphone running Android 4.4.4 as the client. This client connected to a WiFi hotspot exported by a Mac Mini, which in turn obtained its Internet connectivity via a residential cable internet connection. We used Dummynet on the Mac Mini to client connection to emulate a 4G connection with a 6Mbps downlink, 1.5Mbps uplink, and 100ms RTT. We chose these values based on speeds recorded using the Speedtest.net app running on the Nexus 4 client while connected to real 4G over a T-Mobile 4G dongle WiFi hotspot broadcast. They approximately mirror the speeds seen in Klotski.

We use the SPDY pushing enabled web proxy developed in Klotski, with all enhancements other than pushing disabled. For proxy-side caching support, our web proxy runs all requests through an instance of Squid 2.7. To increase the cache hit rate we enable cache query rewriting in Squid such that all URLs sharing the same domain name, filename, and number of URL tokens all map to the same cached item.

We use the landing pages of 100 websites chosen from Alexa’s top 1000 websites, such that we were able to successfully cache greater than 90% of the resources, and push greater than 80% of the resources. No SSL resources are available from the cache or for pushing due to SPDY
and Squid restrictions. We load the full version of these web pages using Google Chrome version 43 for Android. The SPDY and Squid proxies are both run on a moderately powerful Ubuntu machine on the same LAN as the Mac Mini.

Using the method detailed in Klotski, we gather a har file from Chrome during each page load. This har file contains the total page load time, as well as when Chrome consumed each resource. On the client device we capture a packet trace (pcap) during the page load in order to determine when all of the pushed content was received.

### 4.4 Results

In this section we investigate to what extent resource pushing can benefit mobile web page load times. The optimal case for pushing is when the resources are already cached on the proxy, so this setup serves as our best case scenario. We also perform pushing only loads and caching only loads to isolate the benefits from each. The cached resources and list of resources to push are both gathered from the same initial unmodified load.

Figure 4.2 shows the page load time for websites loaded normally, with proxy-side caching, or with both caching and pushing. All of these loads go through the SPDY and Squid proxy in order to keep things consistent. While the use of caching shows some improvement over the normal load, the decrease in load time between caching only and caching/pushing is only slight. To rule out the effect of any end of page load behavior as a cause for this result, we have also compared the time till 75% of the page is loaded, but found a similar result. For the rest of this section we will investigate why pushing should offer so little improvement over proxy-side caching alone.

To begin we first confirm that pushing all resources will lead to better network utilization,
since the proxy need not wait for the browser to request each resource. Figure 4.3 shows the network utilization obtained in the push only, cache only, and cache/push runs. For each 100 ms interval of a page load, we plot the median fraction of total obtainable throughput (6 Mbps emulated limit) across all webpage loads which still have at least 20% of their bytes remaining to be transferred. This cutoff was chosen to eliminate any potential low utilization periods which are common towards the end of page loads, which would throw off the median numbers. The two pushing runs behave as expected, maximizing the network utilization through their early and consistent transfer of bytes from proxy to client. The cache only load on the other hand sees only around half the network utilization, as the proxy is forced to wait for a string of client requests to send the same content. It is interesting to note that since pushing alone is enough to saturate the client-proxy link, the cache/push combination run sees almost identical utilization to that of the push only run. Figure 4.4 focuses on the load of www.salesforce.com to show this same utilization difference in terms of fraction of bytes transferred.
Finally, we focus on the client to see why the faster transfer of resources does not lead to significantly faster page loads. For this we examine our ideal caching/pushing case. Figure 4.5 shows three different views across the same webpage loads. The first view serves as a best possible case by showing the time it takes to transfer a webpage equivalent number of bytes over a plain tcp from proxy to client, using the iperf tool. The second view is from the pcap captured at the client during the page load. This shows the time it took for the client to receive all bytes in the webpage. As we would expect there is some overhead between the vanilla tcp of iperf and the higher layer SPDY with SSL load, but the times for each are similar. The third view is the total page load time as seen in the har file. This shows the time it took for the client to consume all webpage resources. While the pcap shows that the client was able to receive resources in a close to optimal time, the har shows us that the client’s browser was unable to consume the resources at the same rate due to its

Figure 4.3: Median fraction of throughput over 100 webpages per 100ms intervals.

over time.
limited hardware power.

4.5 Related

Parcel [152] and Cumulus [133] come closest to the setup described for our push all proxy. Both host a proxy side browser to mirror the client’s webpage load, preemptively delivering received resources to the client. Parcel extends Firefox to function as both proxy-side browser and proxy server, bundling then delivering content to a custom Android webview client browser. The Cumulus system on the other hand uses a proxy-side PhantomJS browser along with an Apache proxy to deliver content in-bulk to a client-local caching proxy to which any client browser can connect. While our use of SPDY for resource pushing necessitates per-resource transfers, Cumulus delivers only in bulk while Parcel begins after a certain threshold has been reached. These alternate strategies are likely to improve total page load time while negatively impacting progressive page loading. Cumulus does not discuss client specific customization concerns, whereas Parcel suggests a model
where proxies are deployed as personalized proxies for each user, running on virtual machines.

Overall, improvements showed by our push all proxy are difficult to directly compare with Parcel and Cumulus due to differing setups. For total load time improvement, Parcel sets up an experiment using the full version of 34 websites from the top 500 replayed from cache using web-page-replay, loaded by their custom browser on a Samsung Galaxy S3 phone over a real client-proxy LTE link. The proxy-server link is emulated using dummynet and preemptive delivery is done per resource. Compared to a normal Android browser, they cite a median of 3x speedup in total page load time compared to their proxy setup. Cumulus sets up an experiment using the full version of the top 500 websites, loaded by Chrome on a PC laptop over a real client-proxy LTE link, with resources delivered in bulk. They cite a median speedup of 1.3x compared to a normal Chrome load. This is similar to the 1.2x speedup for our own median case. The shared use of LTE, full website versions, and in the case of Cumulus a desktop based browser significantly affects the ability to directly compare with these experiments.
Much like how our work focuses on finding an upper bound on proxy-assisted preemptive content delivery, Z. Wang et al. [159] explore the upper bound of client-only solutions. By tracking 24 iphone users over one year, they find that: client-side caching has little impact due to a 40% cache hit rate, prefetching predicted future page loads is dangerous as 75% of pages are visited only once, and speculative loading (prefetching) of resources based on client history can provide a 1.2x decrease in page load time.

S. Wang et al. [156] have studied overall SPDY performance comparisons, including a few push specific experiments. Under conditions of an artificial browser, webserver, and network, but using replayed desktop computation times, They showed that server push can help, especially under high RTT values. Using artificially scaled values to approximate a mobile device, they note that large computational delays on mobile devices could reduce the benefits provided by SPDY and pushing. In contrast, the experiments in this paper use a real mobile device with an unmodified Chrome browser for more realistic results.

4.6 Conclusion

Previous work had shown that selective pushing of webpage resources could increase user-perceived utility in a mobile environment by prioritizing the delivery of content important to the user, such as above the fold content. In this study, we tested whether total webpage load time could be similarly improved in an ideal pushing setup. We found that despite allowing for a full cache at the proxy and foreknowledge of all requested resources to enable a full push, the total load time improvement was only slightly better than using proxy-side caching alone. While network utilization improved and the client received resources much earlier as expected, due to limited mobile device hardware the client’s browser was unable to consume the resources at the same rate.
Chapter 5

Conclusions

Despite web access on mobile devices becoming commonplace, users continue to experience poor web performance on these devices. Embracing the reality that page load times will continue to be higher than user tolerance limits for the foreseeable future, our work has focused on what can be done to deliver the best possible user experience. We first established the positive correlation between rich webpage content and high webpage load times with the first known measurement-driven study of the complexity of web pages and its impact on performance. We found that we could accurately predict page load times using a handful of metrics, with the number of resources requested (content richness) being the most critical factor.

To better the user’s page load experience, we presented KLOTSKI, a system that prioritizes the content most relevant to a user’s preferences. With KLOTSKI we addressed several challenges: (1) accounting for inter-resource dependencies on a page; (2) enabling fast selection and load time estimation for the subset of resources to be prioritized; and (3) developing a practical implementation that requires no changes to websites. Across a range of user preference criteria, KLOTSKI proved that it could significantly improve the user experience relative to native websites.
Finally, we investigated the potential to further improve the user experience over that offered by KLOTSKI by pushing all content on a web page to any client attempting to load a page, but found that this offered little to no improvement in page load times due to limited capabilities of the client to consume the pushed content. This result reinforced KLOTSKI’s focus on limited resource reprioritization for fixed time period goals.
Bibliography


