Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices

Michael Butkiewicz and Daimeng Wang, ***University of California, Riverside;***
Zhe Wu and Harsha V. Madhyastha, ***University of California, Riverside, and University of Michigan;*** **Vyas Sekar,** ***Carnegie Mellon University***

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/butkiewicz

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Open Access to the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15) is sponsored by USENIX

KLOTSKI: Reprioritizing Web Content to Improve User Experience on Mobile Devices

Michael Butkiewicz*, Daimeng Wang*, Zhe Wu*‡, Harsha V. Madhyastha*‡, and Vyas Sekar†
*UC Riverside †CMU ‡University of Michigan

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.

1 Introduction

Web access on mobile platforms already constitutes a significant (more than 35%) share of web traffic [28] and is even projected to surpass traditional modes of desktop- and laptop-based access [19, 23]. 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 [30, 10].

To cater to this need for a faster mobile web, there are a range of proposed solutions such as customizing content for mobile devices [18, 7], specialized browsers [21], in-cloud acceleration solutions for executing scripts [2], new protocols [25], and compression/caching solutions [2, 8]. 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 [1]; this is the case even for top mobile-optimized retail websites [15]. In fact, several recent studies show that the benefits from the aforementioned optimizations are marginal [37, 59, 3], and they may even hurt performance [56].

Our thesis is that the increasing complexity of web page content [12, 5, 33] 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 [54], the average number of resources on a web page tripled during the same period [5]. 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 [39, 32, 48]. 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 [14, 35] 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 multiple 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 [21, 2] 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.

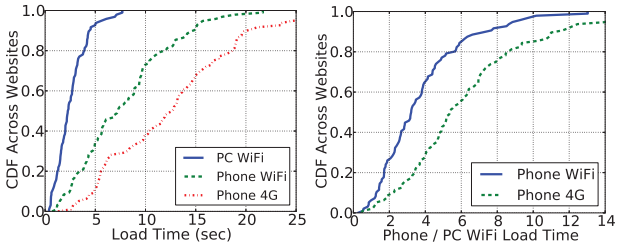
Looking beyond our specific design and implementation, we believe that the principles and techniques in KLOTSKI can be more broadly adopted and are well aligned with emerging web standards [6, 13, 25]. Moreover, while our focus here is on mobile web access, we show that KLOTSKI can also improve traditional desktop browsing as well.

2 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 [30]. To understand the state of mobile web performance, we compared the page load times[1] 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

[1]We measure page load time by the time between when a page load was initiated and when the browser's `onLoad` event was fired.



(a) (b)

Figure 1: ***Load time comparison for top 200 websites.***

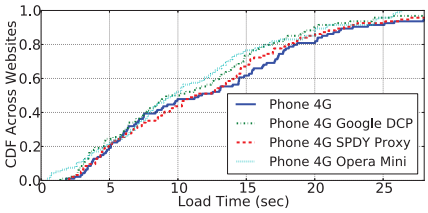


Figure 2: ***Comparison of page load times with various well-known performance optimization techniques.***

variability across loads. For each page, we report the median load time across 5 loads.

Figure 1(a) shows the CDF of load times for the three scenarios, and Figure 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**× worse on 4G and **3**× worse even on WiFi. The tail performance is particularly bad, with the 80^{th} percentile $\geq$ 10s 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 [21], Google SPDY [25], 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) [8], 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 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 considerably higher than the 5-second tolerance threshold typically observed in usability studies [30]. These observations are consistent with other recent studies [59, 37, 56].

Complexity vs. Performance: A key reason why these protocol-, network-, and browser-level optimizations are largely ineffective is because web pages have

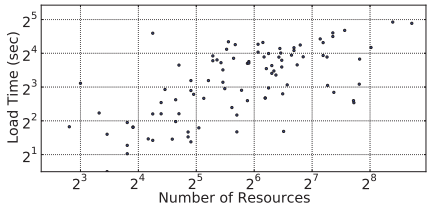


Figure 3: ***Page load time when using Google's compression proxy vs. the number of resources loaded on the page.***

become *highly complex* [12, 5, 33]. For instance, the number of resources included on a web page is a key factor that impacts load times [58, 33]. Indeed, even in our measurements, Figure 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 [5], while RTTs on mobile networks only halved over the same period [54].

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 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.

3.1 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 [39, 32, 48], 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 [4]), 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; underestimation 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.

3.2 KLOTSKI Architecture

To tackle the above challenges, we develop the KLOTSKI system shown in Figure 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.,

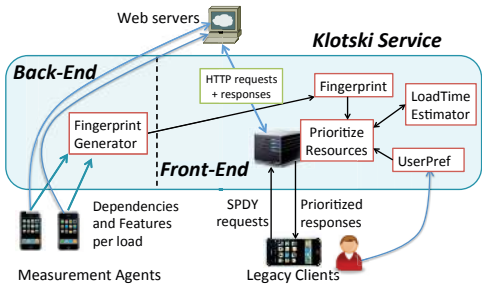


Figure 4: ***Overview of* KLOTSKI*'s architecture.***

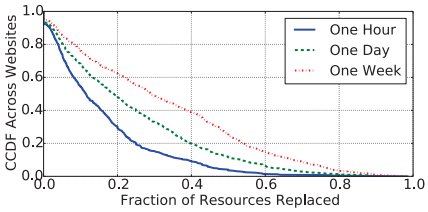


Figure 5: ***Fraction of replaced URLs, comparing loads an hour apart, a day apart, and a week apart.***

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 9. 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 (§4) 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 [26]. When a request for a page w from user u arrives (i.e., the GET for `index.html`), the front-end uses f_w, the user's preferences, and a load time estimator (§6) to compute the set of resources that should be prioritized (§5).

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., [16]).

4 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-children dependencies across resources on w as well as high-level URL patterns describing each resource on the page.

4.1 High-level approach

Prior works such as WebProphet [42] and WProf [58] 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 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 measurements of w over a recent interval of Δ hours. From this set of measurements, $Loads_{w,\Delta}$, we identify a *reference load* $RefLoad_w$. While there are many possible choices, we find using the load with the median page load time within $Loads_{w,\Delta}$ works well. Given that the dependency structure is quite stable, we use the dependency DAG for

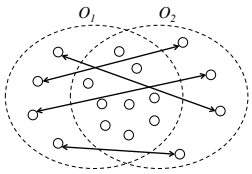
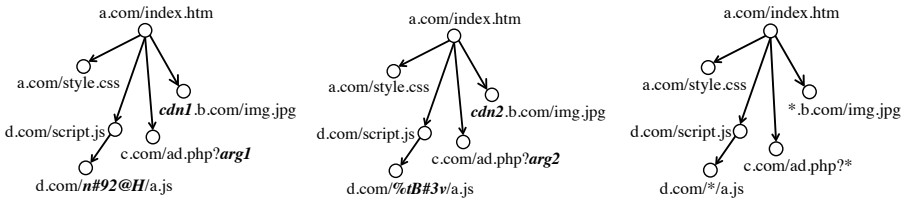


Figure 6: ***Given*** O_1 ***and*** O_2 ***fetched in two loads of the same web page, we map URLs in*** $(O_1 - O_2)$ ***to ones in*** $(O_2 - O_1)$.



(a) (b) (c)

Figure 7: ***Illustrative example showing dependency structures (a) and (b) being merged into aggregated dependency structure (c).***

$RefLoad_w$; in Section 7, we describe how we obtain this dependency DAG in our implementation of 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 $RefLoad_w$. We then iteratively update D by reconciling the differences between D and other loads in $Loads_{w,\Delta}$. First, for every $o \in RefO$ (the set of URLs in $RefLoad_w$) 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 $Load_1$ and $Load_2$, 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.

4.2 Identifying replacements in URLs

Let O_1 and O_2 be the sets of URLs fetched in two different loads within $Loads_{w,\Delta}$. 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 6.

```
<span class="yt-thumb-default">
<span class="yt-thumb-clip">
<img aria-hidden="true" alt="" width="175"
src="//i1.ytimg.com/vi_webp/k2waw0wZ7VA/mqdefault.webp">
<span class="vertical-align">
</span></span></span></span>
```

(a) Load 1

```
<span class="yt-thumb-default">
<span class="yt-thumb-clip">
<img aria-hidden="true" alt="" width="175"
src=" //i1.ytimg.com/vi/GSjA3voJydk/mqdefault.jpg">
<span class="vertical-align">
</span></span></span></span>
```

(b) Load 2

Figure 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.***

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 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.
  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 iden-

[3] Or equivalently, all the other children have already been matched.

[4] We apply Ratcliff and Metzener's pattern matching algorithm [52], which returns a value in $[0, 1]$ for the similarity between two strings.

tify 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 §7). 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 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.

4.3 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 KLOTSKI generates the URL patterns.

While one could use complex algorithms to merge URLs into regular expressions (e.g., [61]), 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, `c.com/ad.php?arg1` in Figure 7(a) is replaced by `c.com/ad.php?arg2` in Figure 7(b). In such cases, we merge URLs into a regular expression that preserves the common prefix and indicates that any argument is acceptable; `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 7(a) and 7(b), `d.com/%tB#3v/a.js` replaces `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.

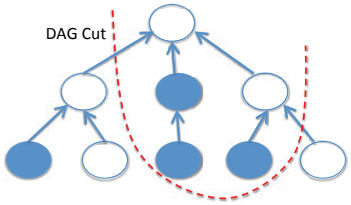


Figure 9: ***Choosing a dependency-compliant subset of resources that maximizes utility within load time budget. Each node represents a resource; shaded nodes have high utility.***

- **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 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 §8 that, with a suitable choice of Δ to refresh the DAG, this is unlikely to occur.

5 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.

5.1 Selecting resources to prioritize

Recall from §3.2 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 9, our goal is to select a suitable *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 ex-

pected 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 $Util_i$, 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 $\frac{Util_i}{C_i}$. When considering a particular resource, we estimate the time (using the technique in §6) that will be 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).

5.2 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 §7.

[5] We can formally prove via a reduction from the weighted knapsack problem, but do not present it for brevity.

6 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) $ProxyFin_i$, when the front-end finishes delivering o_i.[6] 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 [46], then we can use a lightweight fluid-model simulation of the client-frontend 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 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 10, o_3 is requested

[6] All times are specified in terms of the client clock.

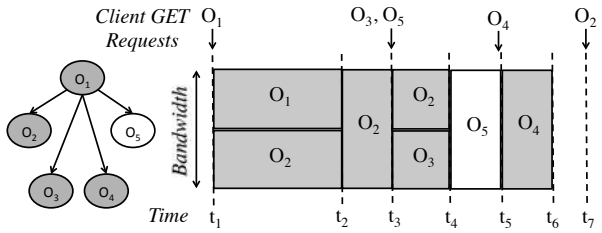


Figure 10: ***Illustrative execution of load time estimator. Shaded resources are high utility resources selected for prioritization. Times shown are for events fired in the simulator.***

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.

2. *$ClientReady_i$:* In the simplest case, we can simply set $ClientReady_i = ProxyFin_i$; 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 $ClientReady_i = \max(ProxyFin_i, ClientStart_i)$. In our example, o_2 finishes downloading at $ProxyFin_i = t_4$, but the client finishes processing the parent to issue the request for o_2 at $ClientStart_i = t_7$.

3. *$ProxyStart_i$:* 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 §5.2) , 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:

$$ProxyStart_i = \begin{cases} 0, \text{if } o_i \text{ is } pushed \\ ClientStart_i + Latency, \text{otherwise} \end{cases}$$

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. *$ProxyFin_i$:* 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.

7 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 [27]. 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 [25]. 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.

8 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,

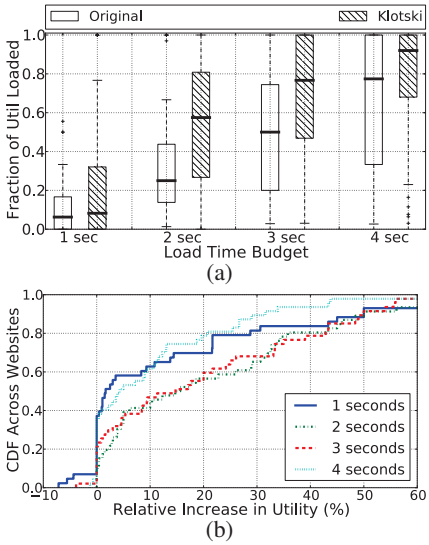
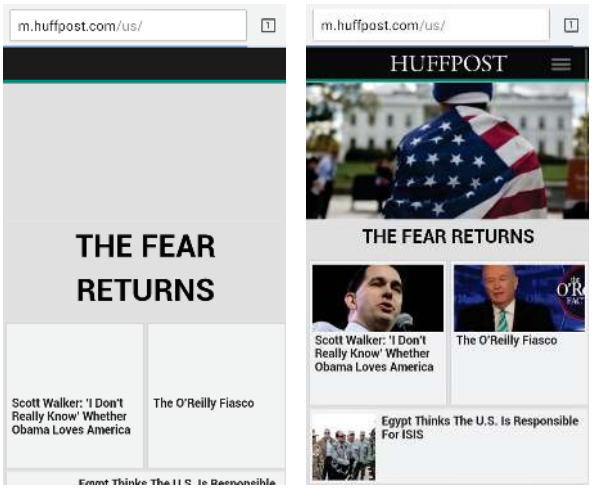


Figure 11: ***Comparison of fraction of high utility resources loaded within load time budget: (a) Box and whisker plots showing spread across websites, and (b) CDF across websites of difference between*** KLOTSKI ***and the original website.***



(a) Original page load (b) Page load with KLOTSKI

Figure 12: ***Screenshots comparing loads of an example site (`http://huffpost.com`), 3 seconds into the page load, without and with*** KLOTSKI.

we evaluate each of KLOTSKI's components in isolation. We begin with a description of our evaluation setup.

8.1 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.

8.2 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 webservers 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.

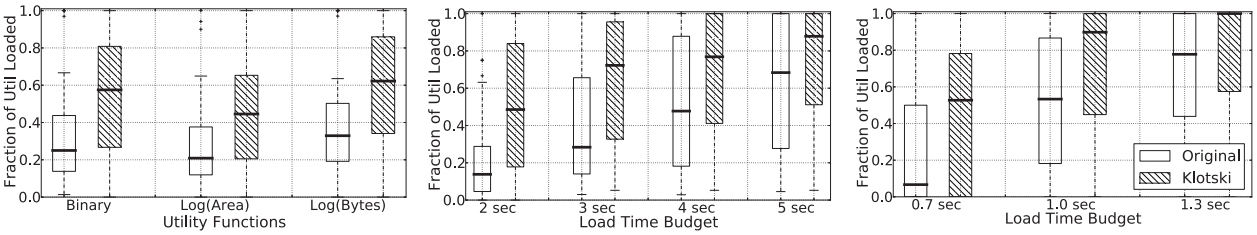
Prioritizing above-the-fold content: First, we consider all resources on a page that appear "above-the-fold" (i.e., resources that are visible without the user having to scroll) as high utility. We assign a utility score of 1 for every high utility object and a score of 0 for all others.

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 [30].

For each load time budget value, Figure 11(a) 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 10^{th}, 25^{th}, 50^{th}, 75^{th}, and 90^{th} 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 11(b), we plot the distribution across websites of the difference 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.

Figure 12 illustrates these benefits offered by KLOTSKI by comparing the screenshots 3 seconds into the page load when loading an example website.



(a) (b) (c)

Figure 13: ***Comparison of utility delivered when (a) varying the function used to compute the utility values for high utility objects, (b) loading full versions of web pages (by faking a desktop's User-Agent), and (c) loading web pages on a desktop.***

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 webservers 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 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 13(a) 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.

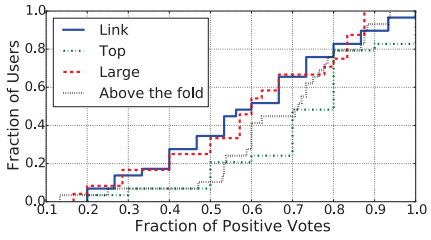
Utility for full versions of web pages and on desktops: Though our primary motivation in developing KLOTSKI is to improve user experience on the mobile web, its approach of reprioritizing important content can also be beneficial in other scenarios. For example, though many websites offer mobile-optimized versions, nearly a third of users prefer the full site experience [20] and 80% of mobile-generated revenue is generated when users view the full site [24]. 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 1(a)), recent surveys [11] 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 13(b) and 13(c) show that KLOTSKI's reprioritization of important content helps significantly improve the user experience even in these cases.

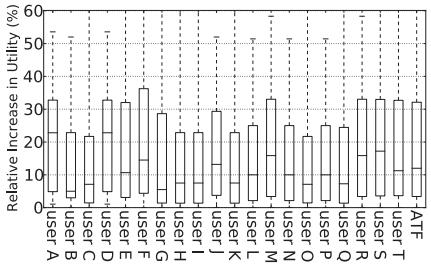
Personalized preferences: So far, we considered all above-the-fold content important. We next evaluate KLOTSKI when accounting for user-specific preferences.

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 14 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 [14, 22] that can only optimize above-the-fold content.



(a)



(b)

Figure 14: ***(a) Variance across users of utility perceived in a few types of objects. (b) 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.***

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 positive/negative.

We consider the data gathered from 20 random users and evaluate KLOTSKI taking their preferences into account. For each of the 20 users, Figure 14 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.

8.3 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.

8.3.1 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.

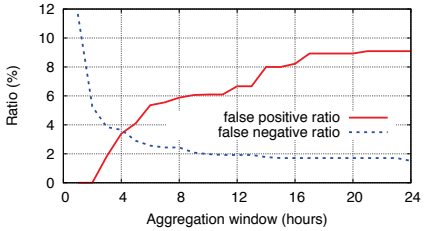


Figure 15: ***False positive/negative matches as a function of back-end's aggregation window for merging dependencies.***

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 Δ, i.e., it aggregates measurements from Δ hours ago until now. Here, we ask: what should be the value of Δ? The smaller the value of Δ, the URL patterns stored in a page's dependency structure would not have converged sufficiently to capture the page's dynamics. The larger Δ, 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 Δ 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 Δ from 1 to 24 hours,

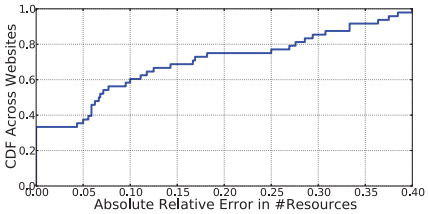


Figure 16: ***Comparison of no. of resources selected to load and no. of resources loaded in practice within the budget.***

Time budget (sec)	1	2	3	4	5
Median runtime (ms)	1	3	4	6	7

Table 1: ***Runtime of resource selection algorithm for the median web page as a function of time budget.***

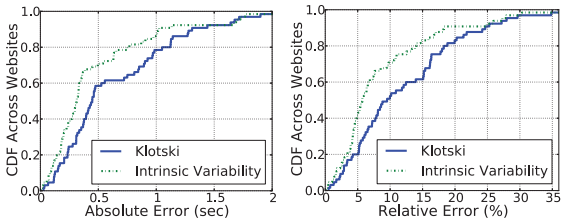
Figure 15 plots the false positive rate and false negative rate on the median web page. We see that an aggregation window of 4 hours presents the best trade-off.

8.3.2 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 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 16 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 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 [12]), 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.



(a) (b)

Figure 17: ***(a) Absolute and (b) relative errors in*** KLOTSKI***'s load time estimates. Comparison with intrinsic variability.***

8.3.3 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 17 shows that the errors in KLOTSKI's load time estimates closely match the distribution of this intrinsic variability.

9 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 8.2) 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 and which parts change.

10 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 [12, 17, 33] and how it impacts page load times [33, 60, 34] and energy consumption [58, 31, 55]. Our work is motivated by such measurements.

Characterizing webpage content: Song et al. [57] 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 [62, 43] or use the DOM tree [40]. Unlike these previous efforts, KLOTSKI associates utilities with individual resources rather than blocks. The closest works on dependency inference are WebProphet [42] and WProf [58]. 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 [25]), guidelines for optimizing pages (e.g., [9]), and optimized browsers (e.g., [45, 44, 21]) and hardware [47, 63]. Many studies have shown that these do not suffice, e.g., two recent studies [59, 37] 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., [2, 21]). A recent study suggests that these can hurt performance [56]. 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 [50, 41, 38]. However, despite the large body of work on accurately predicting what content should be prefetched [51, 49, 36], 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 [53]. 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 [29, 20], 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 [14, 35]. 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.

11 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.

References

[1] 2012 state of mobile ecommerce performance. http://www.strangeloopnetworks.com/resources/research/state-of-mobile-ecommerce-performance.

[2] Amazon Silk. http://amazonsilk.wordpress.com/.

[3] Amazon's Silk browser acceleration tested: Less bandwidth consumed, but slower performance. http://www.anandtech.com/show/5139/amazons-silk-browser-tested-less-bandwidth-consumed-but-slower-performance.

[4] Average number of web page objects breaks 100. http://www.websiteoptimization.com/speed/tweak/average-number-web-objects/.

[5] Average web page size triples since 2008. http://www.websiteoptimization.com/speed/tweak/average-web-page/.

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

[7] Convert any site into mobile format with Google Mobilizer. http://www.geek.com/articles/mobile/convert-any-site-into-mobile-format-with-google-mobilizer-20060117/.

[8] Data compression proxy. https://developer.chrome.com/multidevice/data-compression.

[9] Google page speed. http://code.google.com/speed/page-speed/.

[10] Google research: No mobile site = lost customers. http://www.forbes.com/sites/roberthof/2012/09/25/google-research-no-mobile-site-lost-customers/.

[11] How loading time affects your bottom line. https://blog.kissmetrics.com/loading-time/.

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

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

[14] Imageloader utility: Loading images below the fold. http://www.yuiblog.com/yui/md5/yui/2.8.0r4/examples/imageloader/imgloadfold.html.

[15] Keynote: Mobile retail performance index. http://www.keynote.com/performance-indexes/mobile-retail-us.

[16] Keynote systems. http://www.keynote.com.

[17] Let's make the web faster. http://code.google.com/speed/articles/web-metrics.html.

[18] Mobify. http://mobify.me.

[19] Mobile Internet will soon overtake fixed Internet. http://gigaom.com/2010/04/12/mary-meeker-mobile-internet-will-soon-overtake-fixed-internet/.

[20] Mobile site optimization. http://www.strangeloopnetworks.com/assets/PDF/downloads/Strangeloop-Site-Optimizer-Whitepaper.pdf.

[21] Opera Mini & Opera Mobile browsers. http://www.opera.com/mobile/.

[22] PageSpeed service: Cache and prioritize visible content. https://developers.google.com/speed/pagespeed/service/PrioritizeAboveTheFold.

[23] Pew research Internet project: Cell Internet use 2013. http://www.pewinternet.org/2013/09/16/cell-internet-use-2013/.

[24] Rules for mobile performance optimization. http://queue.acm.org/detail.cfm?id=2510122.

[25] SPDY: An experimental protocol for a faster web. http://www.chromium.org/spdy/spdy-whitepaper.

[26] SPDY coming to Safari, future versions of IE. http://zoompf.com/blog/2014/06/spdy-coming-to-safari-future-versions-of-ie.

[27] SPDY server for node.js. https://github.com/indutny/node-spdy.

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

[29] Web application streaming: A radical new approach. http://go.instartlogic.com/rs/growthfusion/images/Updated_White_Paper_Instart_Logic.pdf.

[30] What users want from mobile. http://www.slideshare.net/Gomez_Inc/2011-mobile-survey-what-users-want-from-mobile.

[31] L. Bent and G. M. Voelker. Whole page performance. In *WCW*, 2002.

[32] A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the eye of the beholder: Meeting users' requirements for internet quality of service. In *CHI*, 2000.

[33] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: Measurements, metrics, and implications. In *IMC*, 2011.

[34] M. Butkiewicz, Z. Wu, S. Li, P. Murali, V. Hris-

tidis, H. V. Madhyastha, and V. Sekar. Enabling the transition to the mobile web with WebSieve. In *HotMobile*, 2013.

[35] T.-Y. Chang, Z. Zhuang, A. Velayutham, and R. Sivakumar. Client-side web acceleration for low-bandwidth hosts. In *BROADNETS*, 2007.

[36] X. Chen and X. Zhang. A popularity-based prediction model for web prefetching. *IEEE Computer*, 36(3):63–70, 2003.

[37] J. Erman, V. Gopalakrishnan, R. Jana, and K. Ramakrishnan. Towards a SPDY'ier mobile web. In *CoNEXT*, 2013.

[38] L. Fan, P. Cao, W. Lin, and Q. Jacobson. Web prefetching between low-bandwidth clients and proxies: Potential and performance. In *SIGMETRICS*, 1999.

[39] D. Galletta, R. Henry, S. McCoy, and P. Polak. Web site delays: How tolerant are users? *Journal of the Association for Information Systems*, 2004.

[40] S. Gupta, G. Kaiser, D. Neistadt, and P. Grimm. DOM-based content extraction of HTML documents. In *WWW*, 2003.

[41] Z. Jiang and L. Kleinrock. Web prefetching in a mobile environment. *IEEE Personal Communications*, 5(5):25–34, 1998.

[42] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. WebProphet: Automating performance prediction for web services. In *NSDI*, 2010.

[43] S. H. Lin and J. M. Ho. Discovering informative content blocks from web documents. In *KDD*, 2002.

[44] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas. PocketWeb: Instant web browsing for mobile devices. In *ASPLOS*, 2012.

[45] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A case for parallelizing web pages. In *HotPar*, 2012.

[46] R. Margolies, A. Sridharan, V. Aggarwal, R. Jana, N. K. Shankaranarayanan, V. A. Vaishampayan, and G. Zussman. Exploiting mobility in proportional fair cellular scheduling: Measurements and algorithms. In *INFOCOM*, 2014.

[47] L. Meyerovich and R. Bodik. Fast and parallel web page layout. In *WWW*, 2010.

[48] F. Nah. A study on tolerable waiting time: How long are Web users willing to wait? *Behaviour & Information Technology*, 23(3), May 2004.

[49] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A data mining algorithm for generalized web prefetching. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1155–1169, 2003.

[50] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review*, 26(3):22–36, 1996.

[51] T. Palpanas. Web prefetching using partial match prediction. Technical Report CSRG 376, University of Toronto, 1998.

[52] J. W. Ratcliff and D. E. Metzener. Pattern matching: The gestalt approach. *Dr Dobbs Journal*, 13(7):46, 1988.

[53] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. Give in to procrastination and stop prefetching. In *HotNets*, 2013.

[54] P. Romirer-Maierhofer, F. Ricciato, A. D'Alconzo, R. Franzan, and W. Karner. Network-wide measurements of TCP RTT in 3G. In *Traffic Monitoring and Analysis*, 2009.

[55] A. Sampson, C. Cascaval, L. Ceze, P. Montesinos, and D. S. Gracia. Automatic discovery of performance and energy pitfalls in HTML and CSS. In *IISWC*, 2012.

[56] A. Sivakumar, V. Gopalakrishnan, S. Lee, S. Rao, S. Sen, and O. Spatscheck. Cloud is not a silver bullet: A case study of cloud-based mobile browsing. In *HotMobile*, 2014.

[57] R. Song, H. Liu, J.-R. Wen, and W.-Y. Ma. Learning block importance models for web pages. In *WWW*, 2004.

[58] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with WProf. In *NSDI*, 2013.

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

[60] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why are web browsers slow on smartphones? In *HotMobile*, 2011.

[61] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: Signatures and characteristics. In *SIGCOMM*, 2008.

[62] L. Yi and B. Liu. Eliminating noisy information in web pages for data mining. In *KDD*, 2003.

[63] Y. Zhu and V. J. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *HPCA*, 2013.