Enabling the Transition to the Mobile Web with WebSieve

Michael Butkiewicz*, Zhe Wu*, Shunan Li*, Pavithra Murali*
Vagelis Hristidis*, Harsha V. Madhyastha*, Vyas Sekar†
* University of California, Riverside † Stonybrook University

1 Introduction

Web access on mobile platforms already constitutes a significant (> 20%) share of web traffic [3]. Furthermore, this share is projected to even surpass access from laptops and desktops [11]. In conjunction with this growth, user expectations for the performance of mobile applications and websites is also growing rapidly [15]. Surveys show that 71% of users expect websites to load almost as quickly as their desktops and 33% of annoyed users are likely to go to a competitor's site leading to loss of ad- and click-based revenue streams [1].

However, the performance of the mobile web today is quite poor. Industry reports show that the median web page takes almost 11 seconds to load over 3G networks even on state-of-art devices such as iPhone5 and the Samsung Galaxy S3 [2]; LTE is only marginally better at improving the latency. The key challenge here is that, unlike traditional devices, mobile devices are fundamentally constrained in several ways in terms of networking, compute, and storage capabilities that can cause high page load times [27, 26].

We are far from being alone or the first to identify these trends. In fact, there has been renewed interest in optimizing web performance focused specifically on mobile devices as evidenced by the proliferation of: a) public measurement reports and repositories (e.g., [7]), b) new optimized protocols (e.g., [13]), c) startups that help providers to generate mobile-friendly web pages (e.g., [10]) and to increase mobile performance (e.g., [14]), d) proprietary optimizations (e.g., [4, 12]), and e) better browsers (e.g., [24, 28]).

Despite the growing realization and recognition of these issues, surveys shows that over 90% of websites are not mobile friendly today [8]. We speculate that this disconnect between the need to customize for mobile devices and the actual adoption of proposed solutions stems from two related factors. First, mobile-specific customization seems to be expensive and often involves manual intervention, thereby restricting its adoption only to high-end website providers. For example, the fraction of websites with mobile-optimized versions drops from 35% in the top 200 to 15% among the top 2000.

The second, more fundamental, issue is that, the desire to deliver rich services (and associated ads and analytics) has, over the last few years, dramatically increased the complexity of websites; rendering a single web page involves fetching several objects with varying characteristics from multiple servers under different administrative domains [16]. This complexity leads to poor interactions with mobile-specific constraints due to several factors such as the need to spawn many connections, high RTTs on wireless links, and time to download large objects on low-bandwidth links. Furthermore, this is accompanied by a corresponding increase in the complexity of website generation (especially for dynamic content); thus, re-architecting them for mobile-friendly designs would require complete overhauls or parallel workflows, further moving the mobile web out of the reach of low-end website providers.

Our overarching vision is to democratize the ability to generate mobile friendly websites, enabling even small web providers to support mobile devices without investing significant resources to do so. While others have focused on automatically adapting web page layouts for mobile devices [17] and on optimizing the load times of Javascript-heavy websites [22], our focus is on reducing the high load times seen on mobile devices for generic web pages. Given the concerns surrounding website complexity and the need to avoid overhauling existing content management workflows, we take a pragmatic approach and cast the goal of customizing websites for mobile devices as an *utility maximization* problem. Specifically, we can view this as a problem of selecting a subset of *high utility* objects from the *original* website that can be rendered within some load time budget for user tolerance (say 2–5 seconds [18, 19]). We can then either block or de-prioritize the loads of low utility objects to reduce user-perceived page load times [9].

While this approach sounds intuitively appealing, there are three high-level requirements that need to be addressed before the benefits can be realized in practice:

- *Structure-awareness:* Given the complex inter-dependencies between objects in most web pages today, blocking or delaying the load of one object may result in several other dependent objects also being filtered out or delayed, e.g., if a Javascript is not fetched, neither will any of the images that the script would have fetched. Thus, even though an object may not directly contribute to the user experience (not visible to users), it may be critical for downloading useful content.
- *Utility-awareness:* Indiscriminate filtering of objects from a web page may prune out content critical to the website's functionality and render the mobile version of the web page useless. We need mechanisms for predicting the expected utility that an user gets from different objects on a given web page. Two concerns arise: (1) we may not know the utility a user perceives in advance before actually downloading the object, and (2) users may differ in their preferences, e.g., some users may dislike ads and images but others may perceive value in these.
- *Practical optimization:* Object selection problems to maximize some utility subject to budget/dependency constraints are typically NP-hard. Additionally, due to the complex policies involved in how browsers parallelize the loading of objects on a web page, it is hard to estimate the resulting page load time when a particular subset of objects on a web page are loaded.

Corresponding to each of these requirements, we describe key practical challenges and preliminary results from our current efforts in designing a *WebSieve* prototype: (1) Naive solutions for depen-

ACM HotMobile'13, February 26–27, 2013, Jekyll Island, Georgia, USA.

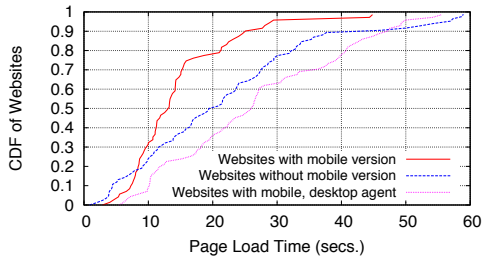

Figure 1: *Page load times on sites with/without mobile versions.*

dency extraction are unlikely to work in face of dynamic content and needs systematic solutions to extract causal relationships, but a practical "block-and-infer" strategy appears promising (§4); (2) We report experiences from a user study suggesting that any framework for assigning utilities to objects needs to account for user-specific preferences (§5); and (3) Despite the theoretical intractability, we can find practical near-optimal solutions in conjunction with approximate load time estimates (§6).

We do acknowledge that blocking low utility objects to reduce page load times may affect the page's functionality; e.g., a button may not be functional if an associated Javascript has not been loaded or the layout may be undesirable if the CSS has not been loaded. The main challenge here is the need to automatically capture the complex inter-dependencies that exist on today's web pages. While we discuss potential approaches here to reduce the likelihood of breaking web page functionality, striking the right balance between load time, utility, and functionality forms the crux of our ongoing work.

2 Motivation

Opportunity to reduce load times: We consider a dataset of 2000 websites from Quantcast's list of the top million websites—400 each, chosen at random, from the rank ranges 1–400, 400–1000, 2000–2500, 5000–10000, and 10000–20000. We identify which of these websites have mobile-optimized web pages. Figure 1 compares the load times [1] on the Sony Xperia smartphone for randomly chosen subsets of 100 websites that have mobile versions and 100 websites that do not currently have mobile versions. (The measurements were made over a 3G connection in a residential location at Riverside.) First, we see that the sites that have mobile versions have significantly lower load times compared to those do not. Second, the load time distribution for websites that do not have mobile versions is comparable to those for the normal/desktop version for the websites that have mobile-optimized versions. In other words, these unoptimized sites have not intentionally chosen to avoid customizing their websites because their load times are already low—there is significant room for reducing the load times. Third, 60% of the mobile-optimized websites still take more than 10 seconds to load, suggesting that even these could benefit from our proposed optimizations. *These results show that there is significant opportunity for reducing the load times of web pages on mobile devices.*

Website complexity causes high load times: A key contributing factor to high page load times is the increasing complexity of web pages. Our recent work [16] showed that, on average, loading a web page requires the browser to fetch over 50 objects from more than 10 servers. Such complexity is not restricted to top-ranked websites, but it exists across web pages in all rank ranges—even among sites in the 10000 to 20000 rank range. In fact, our prior work showed that the number of objects on a page is the most correlated with load time [16, 27]. *Therefore, in order to reduce page load times on smartphones, a key step is to have a systematic solution to "tame" this web page complexity.*

[1] A page's load time is the time at which the `onLoad` event is fired when the page is loaded on the default Android browser.

Version	% responses citing significant loss of useful information			
	Set1	*Set2*	*Set3*	**Aggregate**
Flashblock	20	20	20	**20**
NoScript	0	20	70	**40**

Table 1: *User study to quantify usability impact of naive customization techniques. Numbers are reported with one significant digit given dataset's size.*

Naive approaches to tame complexity do not work: To reduce the impact of a web page's complexity on page load times, we need to either load only a subset of the objects on the page or prioritize the loads of "important" objects. A strawman solution is to filter all objects of a particular type that users may consider to be of low utility. For example, we can use browser extensions such as *Flashblock* and *NoScript* to block all flash and script objects, and all other objects that these cause to be loaded. To analyze how well this would work, we ran a preliminary user study over several websites. We chose three subsets of 10 websites from the top 1000 websites ranked by Quantcast—*Set1* comprised 10 websites chosen at random, *Set2* was the top 10 sites based on the number of unique origins contacted, and *Set3* consists of 10 randomly chosen mobile-optimized websites.

We conducted a survey across 33 participants by hosting the website `http://website-comparison.appspot.com`. We asked users to compare a screenshot of the default version of these 30 sites with screenshots for two alternatives—those obtained with the *Flashblock* or the *NoScript* extension enabled (We use the extensions with their default configurations.) Table 1 shows that the use of either *Flashblock* or *NoScript* would significantly impact user experience. While users may not consider scripts included in a web page as important, blocking those scripts impacts user experience since the objects fetched by executing those scripts are blocked as well. *Thus, to block or de-prioritize low utility content on a web page, we need to take into account the role played by every object on that page as well as the dependencies between objects.*

On the other hand, though ads and objects related to analytics may not be critical to the user experience, blocking these objects can be detrimental to the interests of website providers. *Therefore, reducing web page complexity by blocking objects also needs to take into account the implications of doing so on provider interests, e.g., the impact on their revenue.*

3 Vision and Roadmap

Next, we present a high-level overview of our envisioned WebSieve architecture to instantiate the approach of reducing web page complexity to reduce page load times on mobile devices. Our focus here is primarily to achieve the right balance between page load time, user experience, and website providers' interests. We do not focus on orthogonal problems in optimizing web pages for mobile devices such as customizing the page's layout to suit the screen size, form factor, and UI capabilities of the devices [17].

The problem statement that lies at the heart of WebSieve is as follows. Given a budget on load time, our goal is to select a subset of objects on a web page that will maximize the user experience while satisfying the load time constraint. To keep the discussion simple, consider for now that (1) there is only one type of client device, and (2) that the website's content does not significantly change over time; we discuss how to address these issues in practice in Section 7. Consider a web page W that has a set of objects O. Each

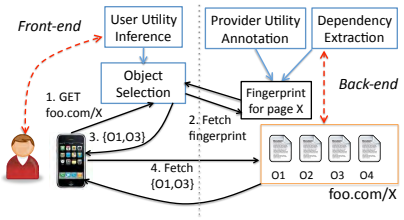

Figure 2: *Overview of WebSieve architecture.*

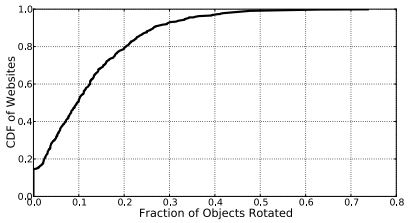

Figure 3: *Quantifying change in objects across page loads.*

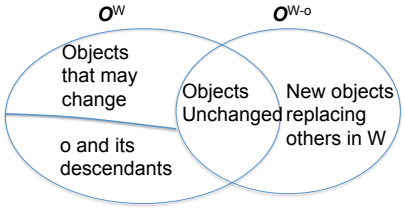

Figure 4: *Visualizing our intuition for mapping objects.*

object $o_i \in O$ takes time t_i to fetch from the server and offers *utility* $Util_i$ to users and website providers. Given a maximum allowed load time of M (e.g., user studies suggest a tolerance around 2–5 seconds [19]), our goal is to select, from all subsets of objects $O' \subseteq O$ whose load time is less than the allowed maximum (i.e., $\leq M$), the subset that maximizes the total utility $\sum_{o_i \in O'} Util_i$. WebSieve can then reduce the user-perceived load time for page W either by loading only the selected subset of objects or by loading this subset before other objects on the page.

This abstract view of the problem highlights three key design challenges that we need to address.

- **Dependencies:** There are natural loading dependencies between the objects in a web page. For example, many web pages download images as a result of executing Javascripts on the client; in this case, the script is a natural *parent* of the resultant image. This means that we cannot select to load an object without choosing to load its parent as well.
- **Utility inference:** The notion of utility perceived by users and providers is a complex issue. For instance, objects in a web page that are not "visible" may not directly contribute to the user experience but are indirectly critical to download interesting content. Moreover, users may differ in their interest and tolerance to web objects, e.g., some users may hate ads and images but others may perceive value in these.
- **Object selection:** The algorithm to select a subset of objects needs to be very efficient; otherwise the runtime of the algorithm may be better spent loading more objects. A key challenge here is that predicting the load time when a subset of objects is fetched is itself non-trivial. In addition to the parent-child relationships described above, web browsers employ parallelization techniques to accelerate page loads. For example, a default Firefox desktop installation can maintain up to 16 connections in parallel and at most 6 connections open to any particular hostname. Consequently, the time to load a set of objects cannot be simply modeled as a simple combination of the individual load times. Furthermore, the load time also depends on the specific device and operating conditions, e.g., 3G vs. WiFi connection.

Figure 2 depicts how these three components fit into WebSieve's architecture, which can be logically partitioned into a frontend and a backend. For every web page, the backend generates a compact *fingerprint* that summarizes the key properties of the page. This fingerprint includes a) the dependency structure of the web page, b) load time information for objects on the page, and c) the object utilities as perceived by the website provider. Since these features change infrequently, WebSieve's fingerprint generation can be performed offline. A website provider can host the backend for fingerprint generation of all pages on his site, or this task can be deferred to any third-party server-side infrastructure. The frontend, which customizes web pages on the fly, can be implemented either as a browser extension or in a proxy that supports dynamic page rewriting and Javascript execution capabilities.

The typical steps involved in fetching and rendering a page with a WebSieve-enabled client will be as follows. The client requests the base HTML file for the web page via the frontend. Along with the HTML file, the frontend also fetches in parallel the fingerprint for the web page from the backend. By combining this fingerprint with the utilities expressed by the local user, the frontend determines which subset of objects on the page it should load. When the client's browser issues subsequent requests for the remaining objects on the page via the frontend, the frontend either sends an empty response or defers the load for objects that are not in its selected subset.

4 Dependency Extraction

As websites increasingly contain dynamic content loaded by scripts and customizable widgets, a specific object can be identified and downloaded only after its logical parents have already been processed. Consequently, any attempt at choosing a high-value subset of objects must account for these logical dependencies. Our goal is to automatically infer the load dependency graph for any given web page; manual specification of cross-resource dependencies by the web page's developer is impractical since 30% of the objects on the median web page are fetched from third-party domains [16].

4.1 Strawman solutions

Consider a page W consisting of the set of objects $O^W = \{o_1 \ldots o_n\}$. Each object o_i has a logical dependency on its *parent* p_i. We consider two intuitive solutions to infer this parent-child dependency structure within a web page. Note that these dependencies cannot be directly inferred from the document structure of the page, as some objects may be loaded after executing dynamic scripts.

HTTP Referer: The first option is to rely on HTTP *referer* tags to identify the causal relationships between object loads. We can load the web page once, and then extract the referer tags during this load to infer parent-child relationships. While this seems intuitively simple, this is not robust. A dominant fraction of dynamic content is loaded via Javascript which does not yield useful referer tags.

Structure inference via blocking: The high-level idea here is to infer potential causal relationships by blocking objects in the web page, similar to the idea proposed in WebProphet [21]. Suppose that the set of objects on a page do not change across multiple loads. For each object $o_i \in O^W$, we can load the webpage when this object is blocked; we perform these page loads with individual objects blocked on a server using the Firefox browser (in its default configuration) with an empty cache. If the page observed when blocking o_i is W_{-o_i}, then we know that every object in the *set difference* between the object sets $O^W - O^{W-o_i}$ is a logical descendant of this blocked object o_i. Though this one step may not be able to distinguish between immediate descendants and indirect descendants, by repeating this experiment for every object, we can reconstruct the exact dependency graph between objects.

In practice, however, web pages are not static even within a short window of time; the set of objects loaded across back-to-back loads

of the same page can be different, e.g., because every refresh yields a different ad or causes the website provider to return a different banner image. For example, in our dataset of 2000 websites, Figure 3 shows that over 10% of objects on the page change across page refreshes in 40% of sites. In our context, this implies that the set of objects loaded after blocking o_i will not be a strict subset of the original set of objects O^W. Specifically, some object from O^W could be missing from W_{-o_i} either because it is a logical descendant of o_i or because it was replaced with a different object when we reloaded the page (see Figure 4). Because of this ambiguity, we may potentially infer false dependencies (i.e., claim x is a parent of y, even though they are unrelated) using the above approach.

4.2 Proposed approach

To handle the constant flux in a web page's content, we propose the following approach. As before, let O^W be the set of objects in the original webpage and the set of objects seen after blocking o_i be O^{W-o_i}. At a high-level, we want to distinguish between the objects that are genuinely missing (i.e., descendants) vs. objects that have been replaced.

As a simplifying assumption, we assume that the number of objects in the web page does not change over the period of time it takes to infer the page's dependency structure; our preliminary measurements confirm that this is indeed the case. Then, we try to infer a *one-to-one* mapping between the set of objects in $O^{W-o_i} - O^W$ and $O^W - O^{W-o_i}$; note that this cannot be a bijection since the sizes of the two sets are different. The intuition behind our approach is that, when we reload the page after blocking o_i, some of the objects in the original web page have been subsequently replaced by the content provider. These new objects are the ones in $O^{W-o_i} - O^W$. Our goal then is to *match* each such object with a corresponding object in the original web page (i.e., without any blocking). Once we have this matching, we know the true set of "missing" objects as the ones that appear in $O^W - O^{W-o_i}$ but do not match up with any object in $O^{W-o_i} - O^W$. These are the true descendants of o_i.

We infer this correspondence between blocked objects and objects in the original web page with a two-step approach. The first step is to find where the object appears in the source files downloaded and match with the object that originally appeared in its place. In our measurements, we observe that this simple mapping step is able to accurately match over 80% of objects that change across page loads. Some objects remain unmapped after this step, for example, because their URLs are generated algorithmically by an embedded script. To address such cases, we map objects using object attributes (e.g., file type and file size) with a simple nearest neighbor like algorithm. With this two-stage approach, we obtain a comprehensive procedure for mapping objects across page loads.

5 Utility Inference

Next, we focus on inferring the utility of individual objects in a webpage. First, we consider the user-perceived utility of different web objects. Ideally, for every object on a web page, we want to run controlled user studies across a sufficiently large sample of users to evaluate the expected value that users perceive from that object.

Since it is infeasible to do so for every single web object, we explore the possibility of learning a classifier that can estimate utilities. Though the number of objects on the Web is potentially unbounded and growing, the utility of any object will likely depend on a few important *characteristic features* of that object. For example, some of the candidate features may include attributes such as the location of the object on the web page (e.g., providers are likely to place interesting objects on top), the type of object (e.g., advertisement vs. image), whether the object has a clickable link, whether the object is visible on the page or hidden, and so on.

Our goal is to learn a *predictive model* that takes as input such object attributes and estimate the potential utility. More formally, if we have features $F_1 \ldots F_j \ldots$ (e.g., location, type) and we have an object where the values of the features are $\langle F_1 = f_1^i, f_2^i \ldots f_j^i \ldots \rangle$ (e.g., location=top-left, bottom-right) [25], the prediction model $Util(\{f_j^i\})$ takes as input the values of these features for a particular object and outputs the object's utility score.

User study to infer utilities: To gain initial insights into the feasibility of inferring such a predictive model, we ran a *user study* using the website `http://object-study.appspot.com`. On this site, we show every visitor snapshots of 15 web pages—the landing pages of 15 websites chosen at random from our list of 2000 sites (see Section 2). For each of these 15 web pages, we pick one object on the page at random and ask the user: *Would removing the 'Object of Interest' greatly impact a user's experience on the website?* We ask users to report the perceived "value" of each object on a Likert scale from -2 to 2, which correspond to an answer varying from "Strong No" to "Strong Yes" in response to our question. We collect responses to this survey from 120 users on Amazon Mechanical Turk.[2] An examination of the responses from our user study shows that simple heuristics such as categorizing all objects "below the fold" as low utility do not work; irrespective of where we consider the fold within the range of 600 to 900 pixels, we find that roughly 35% of objects below the fold were marked as important.

Need for personalization: We use the responses from our user study to train several types of classifiers (with five-fold cross validation) such as decision tree, SVM, and linear regression. Each sample of the training data comprises the features associated with a particular object as the attributes and a user's utility for that object as the value. We associate every object with various features that capture its size, its type of content, if it is an image, whether it is part of a sprite, the object's location on the page, whether it is visible and if so, whether it is in the foreground or the background of the web page, and if the object has a link, whether that link points to third-party content. However, we find that none of these features are well correlated with the responses in our user study; as a result, the best prediction accuracy that we were able to obtain is 62%.

Surprised by the low accuracy across all classifiers, we analyzed the responses. Specifically, we looked at different types of object features, and for each, we analyzed the user responses within that specific feature (e.g., button, ad, location). It became immediately evident that the problem was that we were trying to build a global model across *all users*. We observed that there is considerable variability in user responses within each feature. For instance, 20% of users felt background images were important while 60% did not, while only 50% of users thought links on the bottom were important. What was striking, however, was that any given user was *consistent* in her responses. That is, across all websites, a user typically rates over 80% of objects with a given feature as either important or unimportant.

Hence, we foresee the need for *personalization* in WebSieve. In other words, WebSieve needs to learn and use a classifier customized for a specific user. For example, after a user first installs the WebSieve frontend, the user can mark selected objects on any web page she visits as low utility (say, whenever the page takes too long to load); the Adblock browser extension similarly lets users mark ads that the user wants it to block in the future. Based on

[2] As a sanity check, we only pick respondents who pass our validation stage where we show 4 objects known to be extremely relevant or irrelevant and filter out users who respond incorrectly.

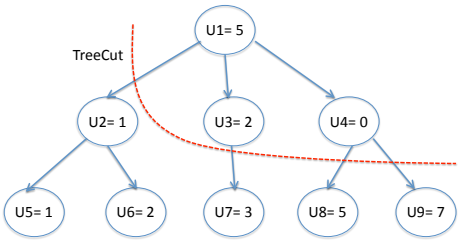

Figure 5: *Each arrow represents a logical parent-child dependency. We want to pick a subset of objects respecting the dependencies that maximizes the utility given a bound on load time.*

the user's responses, we can then learn a classifier over time that is specific to the user. However, a particular user's utility of objects with similar features may vary significantly across different types of websites. For example, small objects with links are likely to be important on a shopping website (e.g., the "shopping cart" button) but not as important on news sites (e.g., the *Like* and *+1* buttons). Therefore, WebSieve may need to consider different categories of websites, and even for a specific user, train a different classifier for each website category. A natural question here is the trade-off between increased accuracy of inferred utilities and overhead for the user as we need larger training sets.

Accounting for functional dependencies: If functional dependencies between objects are not accounted for, blocking objects can potentially break the web page's functionality and re-ordering object loads may not reduce user-perceived page load times even if high utility objects are loaded upfront. For example, delaying the load of a CSS object until after other high utility objects may result in a flash of unstyled content (FOUC) [6]. Similarly, if the Javascript that has registered an event listener with a button is not loaded, that button may not be functional. These are dependencies that our proposed approach in Section 4 will fail to detect. Hence, we directly account for such functional dependencies by associating CSS objects and Javascripts that have event listeners (which we conservatively detect via static analysis of Javascript code) with the highest utility. Based on our previous measurements [16], we estimate that these objects typically account for a small fraction of the objects on a web page.

Accounting for provider utilities: In addition to accounting for user-perceived utilities, it is important to ensure that the interests of website providers are preserved. Prioritizing only popular/useful content can hurt business interests of web providers because analytics or ads may get filtered out. To take this into account, WebSieve can allow for web providers to specify in any web page's source code the objects that are considered important by the provider of that page. For example, these prioritization hints can be added via `META` HTML tags. WebSieve can then take these priorities into account in combination with its estimates for user-perceived utilities. Thus, WebSieve can ensure that the interests of web providers are respected, while minimizing the burden on them for customizing their web pages for mobile devices.

6 Optimal object selection

We describe the abstract formulation of the object selection problem to highlight the key parameters involved, discuss practical challenges, and present our roadmap to address these.

6.1 Problem Formulation

The object selection module in the frontend receives the *fingerprint* from the backend which captures the dependency structure (§4) and annotations to specify key object features (§5). Using these features in conjunction with the user's preferences, it can compute the expected utility that each object provides. Combining these, it constructs a logical representation of the webpage as a *tree* where each node in the tree is annotated with its utility, as shown in Figure 5.

Our goal is to select a suitable *tree cut* in this tree structure; i.e. a cut that also satisfies the dependency constraints. Formally, we are given as input the page tree dependency T for a website W and the time budget M. If C denotes a cut, we want to select the cut C^* that, out of all cuts that can be loaded within time M, maximizes the expected utility.

It is evident that we need a fast algorithm that can solve this problem because object selection is on the critical path for loading the webpage. If the optimization itself takes too much time, then it defeats the purpose of reducing the page load time.

6.2 Practical Challenges

There are two key stumbling blocks. First, the dependencies between objects make this problem NP-hard.[3] Second, any optimization framework will need to model the time to load arbitrary subsets of objects. It is difficult enough to model the $LoadTime(C)$ function even for a specific fixed subset of objects, let alone for all possible subsets! This challenge arises from browser optimizations and the use of parallel connections in loading a web page. In particular, it is challenging to find a closed form function for $LoadTime(C)$. For example, some intuitive solutions like using the sum of the load times or dividing this sum by the expected number of parallel connections turn out to have very high ($\approx$ 3–4 seconds) estimation errors. Thus, we have a chicken-or-egg problem here—in order to pick the optimal subset we need to estimate the load time, but we cannot estimate this before picking a specific subset. In other words, without explicitly enumerating all possible subsets and physically loading them, it appears we cannot solve this optimization.

6.3 Proposed Approach

Dependency Modeling: To address the first problem of dependencies, we propose to use compact integer linear programming formulations. Let d_i be a $\{0, 1\}$ variable that indicates if we have selected the object o_i. Let p_i denote the logical parent of the object o_i in the page tree. Then the dependencies become a simple linear constraint of the form: $\forall i : \ d_i \leq d_{p_i}$.

Load time approximation: We see two practical approaches to address the load time estimation challenge. The key idea in both cases is to leverage the load time "waterfall" for the original web page annotated with the finish time t_i^f for each object i. This information can be included in the web page's fingerprint.

The first approach is to obtain a *conservative* load time estimate. Specifically, given a set of objects O, we can use the *maximum* finish time: $LoadTime(O) = \max_{i \in O} t_i^f$. This is conservative because blocking some objects will have scheduled this max-finish-time object much earlier. Given this context, we can write the page tree cut as an compact integer linear program (ILP). We do not show the full formulation due to space constraints. While we are still solving a NP-hard discrete optimization problem, we can leverage efficient solvers such as CPLEX. We find that it takes $\leq$ 30ms to solve the optimization with real dependency graphs for pages with $\approx$ 100 objects (but with synthetic utilities). Thus, despite the theoretical intractability, we have reasons to be optimistic.

The second, is to heuristically estimate the load time for a given subset of objects by using the timeline of object loads. The main idea is to look for "holes" in the waterfall after blocking and move

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

all objects whose parents have already been loaded to occupy these holes greedily. While this does not give a closed form equation, it gives us a practical handle on estimating the load time for a subset, and we find it works well ($< 20\%$ error). With this estimator tool, we can use greedy "packing" algorithms; iteratively pick the object with highest utility and select it along with its ancestors as long as this choice does not violate the time budget.

We can also combine these two approaches to improve the optimality. For example, we can first run the ILP and then use the greedy approach to exploit the residual time left because of the conservativeness of the max-estimator. A natural concern is how close to the *optimal* solution our conservative ILP and greedy solutions are. In particular, we need to come up with mechanisms for getting tight upper bounds on the optimal solution given that the problem is intractable. We plan to investigate these in future work.

7 Discussion

Website stability: A web page's fingerprint needs to be regenerated as the set of objects on the web page and the structure of the web page changes. To gauge how often this regeneration of a web page's fingerprint will be necessary, we performed a preliminary study with 500 websites (chosen at random from our dataset of 2000 websites). We loaded the landing page of each of these websites once every six hours for a week. Our analysis seems to indicate that, though we saw previously that a significant fraction of objects on a web page change across repeated loads, the subset of stable objects and the dependencies between them appear to persist for several days. Hence, it will likely suffice for WebSieve to regenerate dependency information once a week. In practice, we can consider an adaptive scheme based on information provided by the website provider—refresh information more (less) frequently for web pages that have more (less) flux in their content.

Extrapolating across clients: Apart from dependency information, a web page's fingerprint also includes object load time information. The load times for individual objects however depend on client device capabilities, e.g., mobile phones vs. tablets or different versions of smartphone OS. For example, recent studies show that page load times significantly vary based on the device's storage [20]. Since it is impractical to gather load time information for every web page on every type of client device in every possible network condition, we need the ability to *extrapolate* load time across clients. This algorithm should take two inputs: 1) load time measurements on a reference device type in a specific network setting, and 2) a characterization of a target device and its network. Given these inputs, the algorithm should extrapolate measured load times to the setting of the target device. At the time of loading a web page, WebSieve's frontend can then use this algorithm to appropriately tailor load time information included in the web page's fingerprint for the local client.

Balancing user-provider utilities: One obvious issue here is the tension between users and providers; e.g., users may not want ads but providers do. Note that this problem is not intrinsic to WebSieve and exists today with Adblock-like solutions [5] and tracking [23]. While we cannot speculate how this tussle will play out, our utility maximization framework provides a technical solution to deal with this tussle more explicitly, in contrast to today's binary measures that pick extreme points catering to only one side.

Other applications: While we have focused here on the problem of reducing web page load times on mobile devices, our approach also has other applications. For example, blocking low utility objects can reduce the energy consumption associated with web browsing on mobile devices. Similarly, blocking low utility objects can help users of mobile devices cope with ISP-imposed caps on the amount of data they can receive over the network.

8 Conclusions

Our common mode of access to the Web is slowly transitioning from desktops/laptops connected to wired networks to mobile devices connected with access to wireless networks. While this client-side revolution is already under way, the ability to cope with this change is currently restricted to the top websites.

Our overarching vision is to democratize the ability to generate mobile-friendly websites, enabling even small web providers to transition to support mobile devices without investing significant resources to do so. For this, we present the WebSieve architecture, whose design is motivated by the observation that the Web performance problems on mobile devices stem from the increasing complexity of websites. To tame this complexity, the WebSieve architecture takes into account the intrinsic dependency structure of webpages and user-perceived utilities, and uses these to optimally select a subset of objects to render on mobile devices given a budget on the load time. While we have highlighted several open issues that need to be addressed to translate our vision into reality, our early approaches and results give us reasons to be hopeful.

References

[1] Google Research: No Mobile Site = Lost Customers. http://www.forbes.com/sites/roberthof/2012/09/25/google-research-no-mobile-site-lost-customers/.

[2] 2012 state of mobile ecommerce performance. http://www.strangeloopnetworks.com/resources/research/state-of-mobile-ecommerce-performance.

[3] 25 percent use smartphones, not computers, for majority of Web surfing. http://www.technolog.msnbc.msn.com/technology/technolog/25-percent-use-smartphones-not-computers-majority-web-surfing-122259.

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

[5] Firefox adblock foe calls for mozilla boycott. http://www.informationweek.com/firefox-adblock-foe-calls-for-mozilla-bo/201805865.

[6] Flash of unstyled content (fouc). http://bluerobot.com/web/css/fouc.asp/.

[7] HTTP archive beta. http://httparchive.org/.

[8] Less than 10% of the web in 2012 is mobile ready. http://searchengineland.com/less-than-10-of-the-web-in-2012-is-mobile-ready-112101.

[9] Minimize initial display time to improve perceived web page speed. http://answers.oreilly.com/topic/498-minimize-initial-display-time-to-improve-perceived-web-page-speed/.

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

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

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

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

[14] Strangeloop: Speed up your website. http://www.strangeloopnetworks.com.

[15] Survey Report: What Users Want From Mobile. http://www.gomez.com/resources/whitepapers/survey-report-what-users-want-from-mobile/.

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

[17] Y. Chen, W.-Y. Ma, and H.-J. Zhang. Detecting web page structure for adaptive viewing on small form factor devices. In *WWW*, 2003.

[18] D. Galletta, R. Henry, S. McCoy, and P. Polak. Web Site Delays: How Tolerant are Users? *Journal of the Association for Information Systems*, 2004.

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

[20] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. In *FAST*, 2012.

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

[22] B. Livshits and E. Kiciman. Doloto: Code splitting for network-bound web 2.0 applications. In *FSE*, 2008.

[23] J. Mayer, A. Narayanan, and S. Stamm. Do not track: A universal third-party web tracking opt out. http://datatracker.ietf.org/doc/draft-mayer-do-not-track/.

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

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

[26] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. Singh. Who killed my battery: Analyzing mobile browser energy consumption. In *WWW*, 2012.

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

[28] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How far can client-only solutions go for mobile browser speed. In *Proc. WWW*, 2012.