Speeding up Web Page Loads with Shandian

Xiao Sophia Wang and Arvind Krishnamurthy, ***University of Washington;***
David Wetherall, ***University of Washington and Google***

https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/wang

This paper is included in the Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16).

March 16–18, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-29-4

Open access to the Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16) is sponsored by USENIX.

Speeding up Web Page Loads with Shandian

Xiao Sophia Wang[*] *Arvind Krishnamurthy*[*] *and David Wetherall*[*†]

Abstract

Web page loads are slow due to intrinsic inefficiencies in the page load process. Our study shows that the inefficiencies are attributable not only to the contents and structure of the Web pages (e.g., three-fourths of the CSS resources are not used during the initial page load) but also the way that pages are loaded (e.g., 15% of page load times are spent waiting for parsing-blocking resources to be loaded).

To address these inefficiencies, this paper presents `Shandian` (which means lightening in Chinese) that restructures the page load process to speed up page loads. `Shandian` exercises control over what portions of the page gets communicated and in what order so that the initial page load is optimized. Unlike previous techniques, `Shandian` works on demand without requiring a training period, is compatible with existing latency-reducing techniques (e.g., caching and CDNs), supports security features that enforce same-origin policies, and does not impose additional privacy risks. Our evaluations show that `Shandian` reduces page load times by more than half for both mobile phones and desktops while incurring modest overheads to data usage.

1 Introduction

Web pages have become the de-facto standard for billions of users to get access to the Internet. The end-to-end Web page load time (PLT) has consequently become a key metric as it affects user experience and thus is associated with business revenues [6, 4]. Reports suggest that Shopzilla increased its revenue 12% by reducing PLT from 6 seconds to 1.2 seconds and that Amazon found every 100ms of increase in PLT cost them 1% in sales [27].

Despite its importance and various attempts to improve PLT, the end-to-end PLT for most pages is still a few seconds on desktops and more than ten seconds on mobile devices [9, 39]. This is because modern Web pages are often complex. Previous studies show that Web pages contain more than fifty Web objects on average [9], and exhibit complex inter-dependencies that result in inefficient utilization of network and compute resources [39]. In our own experiments, we have identified three types of inefficiencies associated with Web pages and the page load process. The first inefficiency comes from the content size of Web pages. Many Web pages use JavaScript libraries such as jQuery [21] or include large customized JavaScript code in order to support a high degree of user interactivity. The result is that a large portion of the code conveyed to a browser is never used on a page or is only used when a user triggers an action. The second inefficiency stems from how the different stages of the page load process are scheduled to ensure semantic correctness in the presence of concurrent access to shared resources. This results in limited overlap between computation and network transfer, thus increasing PLT. The third and related inefficiency is that many resources included in a Web page are often loaded sequentially due to the complex dependencies in the page load process, and this results in sub-optimal use of the network and increased PLTs.

Reducing PLT is hard given these inefficiencies. Human inspection is not ideal since there is no guarantee that Web developers adhere to the ever-changing best practices prescribed by experts [35]. Thus, it is widely believed that the inefficiencies should be transparently mitigated by automated tools and techniques. Many previously proposed techniques focus on improving the network transfer times. For example, techniques such as DNS pre-resolution [22], TCP pre-connect [19], and TCP fast open [28] reduce latencies, and the SPDY protocol improves network efficiency at the application layer [32]. Other techniques lower computation costs by either exploiting parallelism [25, 12] or adding software architecture support [41, 13]. While these techniques are moderately effective at speeding up the individual activities corresponding to a page load, they have had limited impact in reducing overall PLT, because they still communicate redundant code, stall in the presence of conflicting operations, and are constrained by the limited parallelism in the page load process.

The key and yet unresolved issue with page loads is that the page load process is suboptimally prioritized as to *what* portions of a page get loaded and *when*. In this paper, we advocate an approach that precisely prioritizes resources that are needed during the initial page load (load-time state) and those that are needed only after a page is loaded (post-load state). Unlike SPDY (or HTTP/2) server push and Klotski [10], which only prioritize network transfers at the granularity of Web objects, our approach prioritizes both network transfers and computation at a fine granularity (e.g., HTML elements and CSS rules), directly tackling the three inefficiencies listed above.

[*]University of Washington
[†]Google Inc.

A key challenge addressed by our approach is to ensure that we do not break static Web objects (e.g., external JavaScript and CSS), because caching and CDNs are commonly used to improve PLT. We make design decisions to send unmodified static contents in the post-load state thereby incurring the cost of sending a small portion of redundant content that is already included in the load-time state.

To deploy this approach transparently to Web pages, we choose a split-browser architecture and fulfill part of the page load on a proxy server, which can be either part of the web service itself (e.g., reverse proxies) or third-party proxy servers (e.g., Amazon EC2). A proxy server is set up to preload a Web page up to a time, e.g., when the *load* event is fired; the preload is expected to be fast since it exploits greater compute power at the proxy server and since all the resources that would normally result in blocking transfers are locally available. When migrating state (logics that determine a Web page and the stage of the page load process) to the client, the proxy server prioritizes state needed for the initial page load over state that will be used later, so as to convey critical information as fast as possible. After all the state is fully migrated, the user can interact with the page normally as if the page were loaded directly without using a proxy server.

Note that Opera mini [26] and Amazon Silk [3] also embrace a split-browser architecture but differ in terms of how the rendering process is split between the client and the proxy server. Their client-side browsers only handle display, and thus JavaScript evaluation is handled by the proxy server. This process depends on the network, which is both slow and unreliable in mobile settings [30], and encourages the proxy server to be placed near users. We have a fully functioning client-side browser and encourages the proxy server to be placed near front-end Web servers (e.g., edge POPs) for the most performance gains.

Our contributions are as follows:

- We conduct a measurement study that identifies the inefficiencies of Web pages that can be fixed by better page structures. We find that three-fourths of the CSS is not used during a page load and that parsing-blocking CSS and JavaScript slow down page loads by 20%.
- We design and implement `Shandian`, which significantly reduces end-to-end PLTs. `Shandian` uses a proxy server to preload a Web page, quickly communicates an initial representation of the page's DOM to the client, and loads secondary resources in the background. `Shandian` also ensures that the Web page functionality in terms of user interactivity is preserved and that the delivery process is compatible with latency-reducing techniques such as caching and CDNs and security features such as the enforcement of same-origin policies. The resulting system is thus both efficient and practical.
- We evaluate `Shandian` on the top 100 Alexa Web pages which have been heavily optimized by other technologies. Our evaluations still show that `Shandian` reduces PLT by more than half with a reasonably powerful proxy server on a variety of mobile settings with varied RTT, bandwidth, CPU power, and memory. For example, `Shandian` reduces PLT by 50% to 60% on a mobile phone with 1GHz CPU and 1GB memory by exploiting the compute power of a proxy server with a multicore 2.4GHz server. Unlike many techniques that only improve network or computation, `Shandian` shows consistent benefits on a variety of settings. We also find that the amount of load-time state is decreased while the total amount of traffic is increased moderately by 1%.

In the rest of this paper, we first review the background of Web pages and the page load process by identifying the inefficiencies associated with page loads (§2). Next, we present the design of `Shandian` (§3) and its implementations and deployment (§4). We evaluate `Shandian` in §5, discuss in §6, review related work in §7, and conclude in §8.

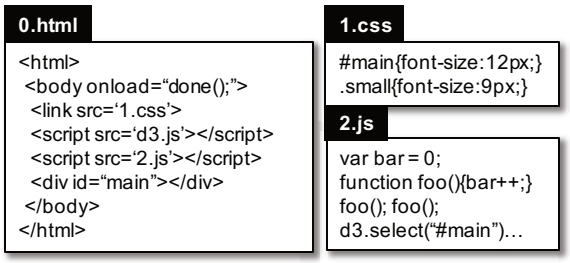
2 An analysis of page load inefficiencies

This section reviews the background on the Web page load process (§2.1), identifies three inefficiencies in the load process, and quantifies them using a measurement study (§2.2).

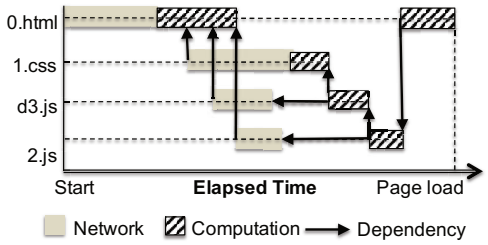
2.1 Background: Web page loads

Web page compositions. A Web page consists of several Web objects that can be HTML, JavaScript, CSS, images, and other media such as videos and Flash. HTML is the language (also the root object) that describes a Web page; it uses a markup to define a set of tree-structured elements such as headings, paragraphs, tables, and inputs. Cascading style sheets (CSS) are used for specifying presentation attributes such as colors and fonts of the HTML elements and is expressed as a set of rules. Processing the CSS involves identifying the HTML elements that match the given rules (referred to as CSS selector matching) and adding the specified styles to matched elements. JavaScript is often used to add dynamic content to Web pages; it can manipulate the HTML, say by adding new elements, modifying existing elements, or changing elements' styles, and can define and handle events. CSS and JavaScript are embedded in a Web page as HTML elements (i.e., `script`, `style`, and `link`) and can be either a standalone Web object or inline HTML.

Web page load process. First, when a user inputs or clicks a URL, the browser initiates an HTTP request to



(a) Web page contents.



(b) Dependency graph of loading the page.

Figure 1: An example of loading a page.

that URL. Upon receiving the request, the server either responds with a static HTML file, or runs server-side code (e.g., Node.js or PHP) to generate the HTML contents on the fly and sends it to the browser. The browser then starts to parse the HTML contents; it downloads embedded files (e.g., CSS and JavaScript) until the page is fully parsed. The result of the parsing process is a document object model (DOM) tree, an in-memory representation of the Web page. The DOM tree provides a common interface for JavaScript to manipulate the page. The browser progressively renders the page during the load process; it converts the DOM tree to a layout tree and further to pixels on the screen.

The browser fires a `load` event when it finishes loading the DOM tree. The `load` event is commonly used for prioritizing Web page contents to improve user experience [9, 39]. For example, websites commonly use the `load` event to defer loading JavaScript that is not used in the initial page display. Such a design makes Web pages more responsive and provides better user-perceived page load times.

Dependencies in Web page loads. Ideally, the browser should fetch Web objects of a page fully in parallel, but in practice the process is often blocked by dependencies among Web objects. One type of dependencies stems from coordinating access to shared resources [39]. For example in Figure 1, when the parser encounters the `script` tag that references 2.js, it stops parsing, loads the corresponding JavaScript, evaluates the script (i.e., compilation and execution), and then resumes parsing. As both HTML and JavaScript can modify the DOM, this ensures that the DOM is modified in the order specified in the Web page. When a CSS appears ahead of this JavaScript (e.g., 1.css), evaluating the JavaScript needs to wait until the CSS is loaded and evaluated. (CSS evaluation includes parsing CSS rules, matching CSS selectors, and computing element styles.) This is because both JavaScript and CSS can modify the elements' styles in the DOM. As is shown in Figure 1(b), HTML parsing is often blocked to ensure the correctness of execution, thus significantly slowing down page loads.[1]

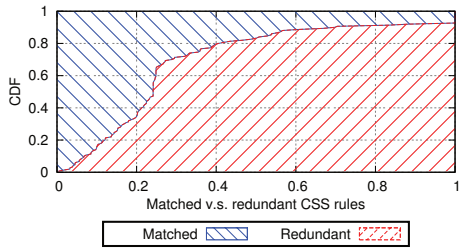


Figure 2: Fraction of redundant CSS rules of top 100 pages in bytes.

Unlike CSS and JavaScript, other Web objects (e.g., images) do not block HTML parsing or any task other than rendering. Therefore, the composition of HTML, CSS, and JavaScript resources and how they are organized are often the factors that affect PLT.

The dependencies not only slow down page loads but also prevent optimizations from being more effective. For example, the SPDY protocol would significantly improve PLT if all the objects in a page were fetched and processed in parallel; but this improvement is largely nullified by the page dependencies in real browser contexts [40]. This is because the optimization technologies often just improve one aspect of page loads (e.g., network utilization), but the overall page load process remains constrained by dependencies and the marginal improvements are not significant.

Critical paths of Web page loads. Not all the object loads on a Web page affect the PLT. The bottlenecks can be identified by performing a *critical path analysis* on the dependency graphs obtained when a page is being loaded. For example in Figure 1(b), loading 0.html and 1.css and evaluating all the objects are on the critical path. Figure 1(b) shows that the time spent on the network comes not only from time to load the HTML, but also from blocking load of JavaScript or CSS (e.g., 1.css), which significantly slows down PLT.

2.2 Page load inefficiencies

To understand inefficiencies in the Web page load process, we conduct a study on top 100 Alexa [2] pages by using Chrome (which is a highly optimized browser). To

[1] These dependencies are enforced by popular browsers including Chrome, Firefox, Safari, and IE.

provide a controlled network environment, we download all pages to our own server, and use Dummynet [11] to provide a stable network connection of 20ms RTT and 10Mbps bandwidth. Our client is a machine with a 2GHz dual core CPU and 4GB memory. We clear the cache for every page load, and define PLT as the time between when the page is requested and when the `load` event is fired. We then identified the following factors that slow down the page load.

Unused CSS in page loads. The first observation is that CSS files often contain rules that are either *never* used in a page or at least *not* used during the initial page load. Such CSS rules incur unnecessary network traffic and parsing efforts. We quantify the amount of used versus unused CSS rules in initial page loads (see Figure 2). In particular, 75% of CSS rules are unused in the median case. Surprisingly, 80% and 96% of CSS rules are unused for `google.com` and `facebook.com` respectively. This suggests that CSS is likely to be redundant for interactive pages, because interactive pages tend to load lots of CSS rules for future interactions, at the cost of increased PLT.

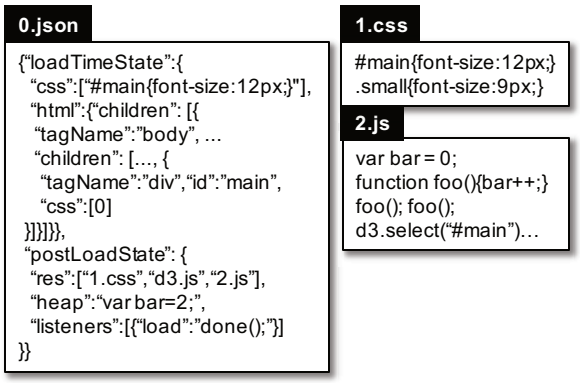
Blocking JS/CSS. JavaScript and CSS often block parsing on the critical path. We extend WProf [39] to measure the amount of additional round trips and parsing-blocking object downloads and evaluations. We find that 15% of the PLT for top pages is spent waiting for JavaScript or CSS to be loaded on the critical path, and 5% of PLT is used for evaluating CSS and JavaScript. Compared to the time to first byte, which is difficult to reduce, there are significant potential gains from optimizing CSS and JavaScript.

Additional round trips. Web objects are not loaded in a batch, but are often loaded sequentially due to the above reason. The result is that loading a page usually incurs many round trips, since loading an object often triggers a sequence of latency-inflating operations such as redirections, DNS lookups, TCP connection setups, and SSL handshakes. We find that 80% of pages have sequentially loaded Web objects on the critical path.

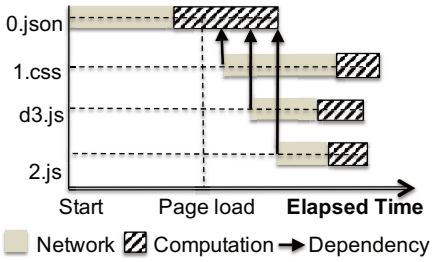
3 Design

Our design aims to reduce PLTs by restructuring the page load process to remove the inefficiencies measured in §2.2. We pre-process Web pages on the proxy server and migrate page state to the client in order to streamline the client-side page load process.

The key to our design is the state that we capture and migrate. On the one hand, page state needs to be captured at an appropriate processing stage in order to minimize the network and computational costs; on the other hand, the captured state should be comprehensive and ensure that the rendered page on the client displays and



(a) Web page contents. This is to provide an overview of the load process and we skipped some of the details in the interest of saving space.



(b) Dependency graph of loading the page.

Figure 3: An example of loading a page using `Shandian`.

functions correctly. Figure 3 shows an example of loading a page with `Shandian`. We reorganize the state in the root object (e.g., 0.json) while keeping the integrity of other objects (e.g., 1.css, d3.js, and 2.js). Figure 3(b) shows a dependency graph of loading the page with `Shandian`. The page is loaded when the load-time state is loaded and evaluated (e.g., the `#main` element is rendered), which is much faster than Figure 1(b) with object inter-dependencies. Processing post-load state does not involve any complex inter-dependencies as the evaluation of objects can happen in parallel.

The challenges are detailed in §3.1. Next, we describe the *load-time state* (§3.2) that is captured for fast page loads, and the *post-load state* (§3.3) that is captured for interactivity and compatibility. In addition to the state that is migrated from the server to the client, we discuss the state that needs to be migrated from the client to the server (§3.4).

3.1 Challenges

We identify three challenges in designing `Shandian`.

First, precisely identifying state that is needed during a page load (load-time state) is nontrivial since load-time state and post-load state are largely mingled. For example, some Web pages use a small portion of jQuery [21] to construct HTML elements while leaving a large por-

tion of jQuery unused. Precisely identifying the load-time state and migrating them to the client in the first place is key to reducing PLT.

Second, we need to ensure that the Web page rendered using `Shandian` is functionally equivalent to one that is computed solely on the client. On the one hand, as page state processing happens on both proxy servers and clients, we need to carefully design the split process to ensure that we do not break the pages. On the other hand, the server needs proper client-side state to function properly. For example, some Web pages that adopt a responsive Web design provide layouts specific to browser size. The server needs information about browser size, cookies or HTML5 local storage in order to function properly.

Third, completely recording and migrating the Web page state computed by the server is nontrivial. After the initial load process, the state computed by the server is largely dispersed across various JavaScript code fragments that comprise the Web page. This state needs to be retrieved and then migrated to their equivalent locations on the client in order to ensure that the user has a seamless experience in interacting with the Web page.

In the rest of the section, we discuss how we address these challenges in designing `Shandian`.

3.2 Load-time state

The load-time state is designed primarily for facilitating display and is captured at a processing stage that minimizes the amount of work required for rendering on the client. To this end, design decisions regarding the load-time state focus on *how much we can eliminate JavaScript/CSS evaluations while keeping the communicated state small*. As a result, the load-time state contains only HTML elements and their styles, but not JavaScript or post-load CSS. Below, we explain this in greater detail and also describe the state that is migrated to reflect JavaScript/CSS evaluations performed at the server.

3.2.1 Load-time state in JavaScript

JavaScript itself does not directly reflect on display, but the result of JavaScript evaluation can. As JavaScript evaluation is slow and blocks rendering, a design decision is to avoid both communicating JavaScript as part of the load-time state and evaluating JavaScript on the client device. Instead, the load-time state includes the result of JavaScript evaluation on the server, and this result is reflected in the HTML elements and their styles. For example, instead of transmitting a piece of D3 JavaScript [14] to construct an SVG graphic on the client, the JavaScript is evaluated at the server to generate the load-time state of HTML elements that represent the SVG. This design minimizes the computation time used in JavaScript evaluation for an initial page load and also avoids blocking executions on the client, but this is at the cost of potentially increased size of migrated state.

3.2.2 Load-time state in CSS

CSS evaluation is also slow, blocks rendering, and thus should be avoided in the initial page load as much as possible. The result of CSS evaluation is, however, a detailed and potentially unwieldy list of styles for each HTML element. Including the detailed list of styles in the load-time state would fully eliminate the CSS evaluation but incur a significant amount of time transferring the state.

Here, we seek an intermediate representation for the CSS state that incurs little additional time to finish the CSS evaluation while keeping the state small. CSS evaluation involves a sequential process of CSS parsing, CSS selector matching, and style computation. The CSS selector matching step matches the selectors of *all* the CSS rules to *each* HTML element, requiring more than a linear amount of time. The style computation process applies matched CSS properties in a proper order to generate a list of styles for rendering.

Our design decision here is to perform CSS parsing and matching on the server but leave style computations to be performed on the client. We migrate all the inputs required by style computations as part of load-time state. The required inputs are largely determined by the W3C algorithm that specifies the order according to which CSS properties are applied [37]. In addition to matched CSS selectors and properties for a given HTML element, the state also includes the importance (marked as important or not), the origin (from website, device, or user), and the specificity (calculated from CSS selectors) that determines this order. The resulting migrated state is compact compared to the detailed list of styles, and at the same time, it eliminates CSS selector matching on the client.

3.2.3 Serialization and deserialization

In order to obtain the load-time state, the proxy server first loads a Web page using a browser that has sufficient capabilities to handle HTML, CSS, and JavaScript. When the page load event or any other defined event is fired, the proxy browser serializes the load-time state from the memory. The server-side `Shandian` recursively serializes each HTML element in the DOM (excluding CSS and JavaScript elements), its attributes, and references to matched CSS rules. Then, the details of the matched CSS rules are serialized. Each CSS rule includes a CSS selector, a list of CSS properties, the importance, and the origin. Note that we do not add CSS rules to each matched HTML element, but use references to link HTML elements to their matched CSS rules (e.g., Figure 3(a) references the index of the matched CSS rule in the CSS array). This is because a CSS rule is likely

to match with many HTML elements. The HTML elements and matched CSS rules together provide complete information for a page to be displayed properly on the client.

Deserializing the load-time state, which is both simple and fast, determines the page load time on the client. The client-side `Shandian` linearly scans the load-time state and uses HTML elements and attributes to construct the DOM. Instead of running a full CSS evaluation, `Shandian` computes styles from already matched CSS, requiring just a linear amount of time. `Shandian` does not require any client-side JavaScript evaluations because the state already contains the results of JavaScript evaluation. Compared to the page load process, the deserialization process does not block, does not incur additional network interactions, and avoids parsing of unused CSS or JavaScript, thereby significantly speeding up page loads on the client as is demonstrated by Figure 3(b).

3.3 Post-load state

Following the load-time state, the post-load state needs to be processed transparently in the background in order to ensure that: (a) users can further interact with the page as if it were delivered normally and not through `Shandian` (*interactivity*), and (b) latency-reduction techniques are still viable (*compatibility*). To ensure interactivity, the post-load state should include the portion of JavaScript that was not used in the load-time state, together with unused CSS, because they might be required later in user interactions. To ensure that complementary latency-reduction techniques such as caching and CDNs can be used in `Shandian`, we need to an unmodified version of external objects in the post-load state.

The most direct approach would be to migrate unmodified JavaScript/CSS snippets, which both ensures integrity (and thus compatibility) and includes all the information for post-load state (interactivity). Our design here focuses on examining the feasibility of migrating unmodified snippets and processing unmodified snippets while excluding the effects of load-time state.

3.3.1 Post-load state in CSS

Attaching unmodified CSS snippets (copies of inline CSS and links to external CSS) in the post-load state is both feasible and simple. We can just evaluate all the CSS rules here regardless of whether they had appeared in the load-time state. This is because CSS evaluation is idempotent—evaluating the same CSS rule any number of times would give the same results. In our design, the CSS rules in load-time state will be evaluated twice (one on the proxy server, and the other on the client) while post-load CSS is evaluated once.

This design is simple and satisfies the constraints, but at the cost of repeating the evaluation of load-time state. For example, if a snippet of external CSS is already being cached, our design does not require loading any portion of this snippet from anywhere else. The price to pay is the additional energy consumption and latencies that result from the repeated evaluation of load-time state. But, since these computations happen after the initial load-time version has been rendered, the additional cost does not impact user's perception of the page load time.

3.3.2 Post-load state in JavaScript

Attaching unmodified JavaScript snippets in the post-load state incurs a complex processing procedure, because not all JavaScript evaluation is idempotent. On the one hand, we need to ensure that JavaScript evaluation has equivalent results as if `Shandian` weren't used; on the other hand, we need to completely record all the state of JavaScript. Other approaches such as migrating the entire heap would incur significantly larger state (10x) and break the integrity of JavaScript objects (consequently caching), and are thus not an option here.

Ensuring equivalent results from JavaScript evaluations. If we include the original unmodified JavaScript code in the post-load state, it is hard to ensure that JavaScript evaluation gives equivalent results as if `Shandian` weren't used. This is because the order in which JavaScript appears determines the results of JavaScript evaluation, but unfortunately this order is not preserved as a result of isolating load-time and post-load state. If we do not keep unmodified JavaScript in the post-load state, the compatibility would be compromised, so is the size of the communicated state.

Our approach uses unmodified JavaScript, together with a bit of the memory state that we call heap (referred to as partial heap), to reconstruct the whole heap. To keep the partial heap small, the key is to extract as much information as possible from the unmodified JavaScript.

To this end, we further break down the unmodified JavaScript snippets into statements, and reuse as many idempotent statements as possible. A JavaScript statement can be a function declaration, a function call, a variable declaration, and so forth. Evaluating function declarations is idempotent, but evaluating other statements is not necessarily idempotent. To avoid double evaluating non-idempotent statements, the client-side `Shandian` only evaluates function declarations in post-load JavaScript, and directly applies the partial heap—the results of JavaScript evaluation that are migrated from the proxy server.

The contents of the partial heap largely depend on how function declarations would be extracted from unmodified JavaScript. However, isolating function declarations from other JavaScript is nontrivial because they are often largely mixed. Below, we discuss the situations under

Events	Event state
DOM events	event name, callback and its arguments
`XmlHttpRequest`	internal fields of the object
`setTimeout`	time to fire, callback and its arguments
`setInterval`	time to fire, interval, callback and its arguments

Table 1: Summary of events and their states.

which function declarations are hard to isolate and also describe the use of the partial heap when necessary.

(i) Recursively embedded instance variables. JavaScript does not distinguish between functions and objects, and thus a function declaration can recursively embed other function declarations and instance variables. To this end, the server-side `Shandian` recursively captures all the instance variables as the partial heap even if they are embedded in a function declaration. When the client-side `Shandian` evaluates unmodified JavaScript, it first only evaluates function declarations by ignoring these instance variables, and then applies the partial heap to restore the instance variables.

(ii) Self-invoking functions. A self-invoking function combines a function declaration and a function call in a single statement. For example, `(function(n){alert(n);})(0)` is a self-invoking function. Our approach is to split up the single statement into a function declaration and a function call, and evaluate them differently.

(iii) `eval` and `document.write` can convert strings to JavaScript code that embeds function declarations. The use of `eval` and `document.write` is considered as bad practices for both performance and security. We disable the use of `Shandian` for Web pages that have invoked `eval` and `document.write` before a page is loaded.

Recording all the state of JavaScript. Recording all the state is challenging, because some state such as those in function closures and event callbacks are hard to capture.

(i) Instance variables in function closures. A function closure is often used for isolating code execution environments (referred to as scopes). We instrument the JavaScript engine with the ability to refer to function closures and serialize the instance variable for each closure respectively. Unlike other techniques that handle function closures by rewriting JavaScript [23], instrumenting the JavaScript engine allows us to handle function closures efficiently.

(ii) State in event callbacks. Besides function closures, event callbacks are also hard to capture. Here, we consider three kinds of events that can be added in an initial page load, which are summarized in Table 1. Serializing the event callbacks requires us to capture all the state in the event queue.

3.3.3 Serialization and deserialization

Serialization and deserialization of the post-load state happens in the background while users interact with load-time state and is more complex than that of the load-time state.

The server-side `Shandian` first serializes unmodified CSS or JavaScript snippets if they are inline (their links instead if they are external), ensuring compatibility. Next, `Shandian` serializes the event callbacks and the partial heap excluding those that can be restored from function declarations in the unmodified JavaScript (e.g., `listeners` and `heap` fields in Figure 3(a)). The post-load state together with load-time state provides complete information for a Web page to function correctly. Note that the size of the load-time and post-load state together exceeds that of the original Web page. The extra portions include the matched CSS rules, the partial heap, and event state. Because they are computed from the original Web page and are thus repetitive, they can be compressed.

The client-side `Shandian` first deserializes and parses unmodified CSS and JavaScript, fetching corresponding objects if they are external. Unlike in a Web page where fetching CSS and JavaScript has to comply with the dependency model [39], here JavaScript and CSS objects can be fetched completely in parallel. After all the objects are fetched, CSS is completely evaluated, and the function declarations in JavaScript are evaluated to avoid duplicate evaluations. Then, the partial heap is applied and events start to get fired. At this point, the Web page state on the client is restored as if the entire page load process happened on the client.

3.4 Client-side state

Website information stored in browsers. In addition to migrating state from the server to the client, some state stored in browsers needs to be first migrated from the client to the server. While constructing the DOM, the browser uses long-term storage including cookies, HTML5 local storage, and Web database. Because the server does not keep a copy of this state, lacking client-side state might break the Web page. `Shandian` handles client-side state by migrating them from the client to the server along with the page request. But this has the potential to increase the uplink transfers and thus slow down page loads. To this end, we conduct a measurement study on client-side state and have confirmed in §5.4 that the client-side state that needs to be migrated is small.

Other sources of inconsistencies. Besides browser storage, there can be differences in obtaining timestamps (`Date.now`), geolocation, and browser information from the client and the server [8]. The absolute timestamps should be the same on the client and the

server as if they are both synchronized to the global clock. However, the time zone could be different and thus needs to be sent to the server. The geolocation can only be obtained by asking users for an explicit consent. Once this happens, we send the geolocation to the server. Browser information includes the window size and user agents. As user agents are always sent to the server, we do not need to explicitly handle it. We send the window size to the server because it can be used to adjust the size of the layout (e.g., in a responsive design).

4 Deployment and Implementation

4.1 Deployment

`Shandian` can be deployed either in the reverse proxy (co-located with front-end servers) or as a separate globally distributed proxy service (similar to Opera mini [26] and Amazon Silk [3]).

Conventional wisdom suggests deployments near clients in order to make better use of edge caching and CDNs and to offer low latencies (e.g., when JavaScript offloading is needed). To the best of our knowledge, all page rewriting techniques (e.g., Opera Mini [26] and Amazon Silk [3]) are intended to be deployed near clients. Unfortunately, such a deployment slows down the preload process on the proxy server, because it adds additional round trips to the Web server, which is a key inefficiency especially for parsing-blocking object downloads in current Web pages (§2.2).

In our design of `Shandian`, we find that exploiting caching/CDNs and reducing round trip delays to the origin server are not at odds and that a carefully designed system can achieve both. We only require the resources that are used as part of the initial page load to go through the proxy server, while the resources accessed after the initial load (e.g., images and videos) can still be cached or be fetched from CDNs. Therefore, we consider deployments wherein the proxy server is located near the Web content server and is ideally co-located with the reverse proxy of the Web service in order to reduce the preprocessing time in the proxy server.

4.2 Implementation

State format. We represent the migrated state in JSON format, because it is simple and compact. Note that other formats such as XML or HTML are also viable.

Server-side `Shandian`. We implement the server-side browser as a webserver extension based on Chrome's `content_shell` with most modifications to Blink and few to V8. We chose the lightweight browser `content_shell` because it includes only page-specific features such as HTML5 and GPU acceleration, but not browser-specific features such as extensions, autofill, and spell checking [18].

Our instrumentation is primarily for state serialization, and is minimal before state serialization starts: we turn off downloads of images and other media because they are not part of the migrated state; we also block objects that are hosted on other domains because we mandate downloading all the required CSS and JavaScript to the Web server. While most of the state resides in Blink [5], some also resides in V8 [36] (e.g., event callbacks and function closures). The server extension can be added to any webserver that allows process invocation.

Client-side `Shandian`. The client-side browser is also based on Chrome, and we modify it as little as possible. We implemented a JSON lexer to parse the migrated state, and this lexer is invoked instead of the HTML lexer. After obtaining the HTML elements from the JSON lexer, we perform DOM construction using unmodified Blink. Given that the migrated state contains matched CSS rules, we skip CSS selector matching and directly apply the CSS properties to compute the element styles. We modify V8 a little to selectively evaluate function statements in JavaScript and to apply server-side results of JavaScript evaluations. We modify Blink to create event listeners and timers from our serialized state instead of executing the load-time JavaScript. The client-side browser can opt in to using `Shandian` using an HTTP header and thus can easily fallback to loading the original pages that `Shandian` does not support.

Note that we chose to modify the browsers instead of implementing state migration using JavaScript because JavaScript evaluation is time consuming and because it does not provide the appropriate APIs necessary for all the low-level manipulations. For example, JavaScript does not allow access to the matched CSS rules for an HTML element. By operating inside the browser code base, we have easy in-memory access to all the desired information, and we also avoid JavaScript execution at the client prior to the page load event.

5 Evaluation

The evaluation aims to demonstrate that: (i) `Shandian` significantly improves PLT under a variety of scenarios (§5.2), (ii) `Shandian` does not significantly hurt data usage §5.3), and (iii) the amount of client-side state that needs to be transferred to the server is small (§5.4).

5.1 Experimental setup

We conduct the experiments by setting up a client that loads Web pages using our modified Chrome and a server that hosts pages using our server extension. We detail the experimental setup below.

Our server is a 64-bit machine with 2.4GHz 16 core CPU and 16GB memory, and has an Ubuntu 12.04 installation with the 3.8.0-29 kernel. To ensure all the Web objects are co-located with `Shandian`, we download the

mobile home pages of the Alexa top 100 websites to our server and use Apache to host them. We download all the Web objects for a page, ensuring that the page loads by our server extension do not issue external network requests. In practice, only Web objects that are used in initial page loads need to be hosted on the server. For example, synchronous JavaScript needs to be placed on the server, but images and videos can be placed anywhere else. Our experimental setup emulates a deployment setting where `Shandian` executes on a front-end server of a Web service.

Our clients include mobile phones (Nexus S with 1GHz Cortex A8 CPU, 512MB RAM) and virtual machines with varying CPU and memory. We experimented with a 3G/4G cellular network, WiFi, and Ethernet in a LAN. We focus on the results from LAN because results from the cellular network are similar to simulated LAN settings.

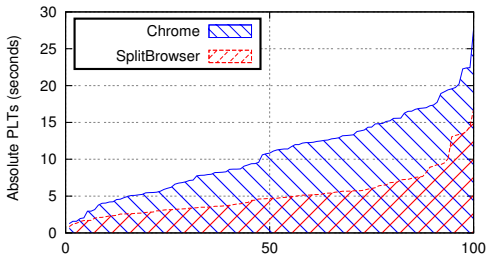
We define page load time (PLT) as the time to display page contents that are rendered before the W3C `load` event [38] is fired. Note that our approach works with any metrics of page load times, though we use the W3C `load` metric to evaluate our prototype. Alternatives to PLT such as the above-the-fold time (AFT) [7] and speed index [34] represent user-perceived page load times, but they require cumbersome video recordings and analysis and are thus out of the scope of this paper. We clear browser cache between any two page loads and do not consider client-side state that requires a login. We report the median page load times out of five runs for all the experiments.

5.2 Page load times

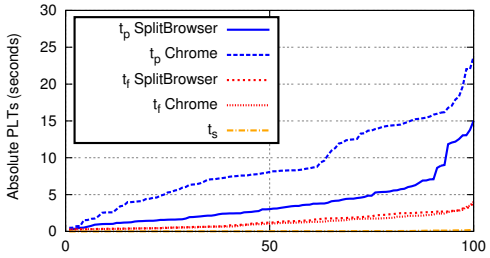
One source of benefits of `Shandian` comes from reducing the dependencies between network and computation, which in turn eliminates network operations that block rendering. For example in Figure 3(b), the network interaction is minimized to just fetching the load-time state in 0.json (before the page is loaded). Another source of benefits comes from reducing the amount of computation needed for the initial page loads (evaluating just the load-time state instead of evaluating all the dependent resources). We now demonstrate the performance benefits under a wide variety of scenarios.

5.2.1 PLT on mobile devices

We use a mobile phone, Nexus S with 1GHz Cortex-A8 CPU, 16GB internal memory (512MB RAM), and Android 4.1.2. The mobile phone is connected to the Internet via WiFi. We install our modified Android Chrome and automate experiments using `adb` shell. We load the Web pages with `Shandian` and with unmodified Chrome on the mobile phone. Figure 4(a) shows that the PLTs with `Shandian` are significantly reduced compared to those with unmodified Chrome. The reduction is as much as 60% in the median case.



(a) Overall page load times



(b) Breakdowns of page load times

Figure 4: Page load times (seconds) on Nexus S with 1GHz Cortex A8 CPU, 512MB RAM, and Android 4.1.2, and with WiFi. `Shandian` reduces page load times by 60% compared to Chrome in the median case.

Source of benefits: To identify the source of benefits, we further break down PLTs into time spent by the `Shandian` server extension t_s, time to fetch the first chunk of the page t_f, and time to parse the page (including parsing-blocking network fetch time) t_p. Figure 4(b) shows that `Shandian`'s server extension uses little time to pre-process pages (22ms in the median case, and 250 ms in the maximum case). Compared to client-side page loads that take a few seconds, server-side page loads have negligible overheads, due to the benefits accrued from more compute power (especially memory), a lightweight server browser, and mitigated network inefficiencies by deploying the cloud server near the Web server. The benefits together suggest that migrating page load computations to the server is effective. By comparing the client-side parsing times t_p of `Shandian` and Chrome, we find that the benefits of `Shandian` stem mainly from client-side parsing. This is because `Shandian` requires no JavaScript evaluations, eliminates redundant CSS, and increases network utilization by eliminating blocking operations.

5.2.2 PLT on desktop VM

To demonstrate how much `Shandian` helps PLTs on a variety of scenarios, we use a desktop VM with Ubuntu 12.04 kernel 3.8.0-29 installed and connected to the net-

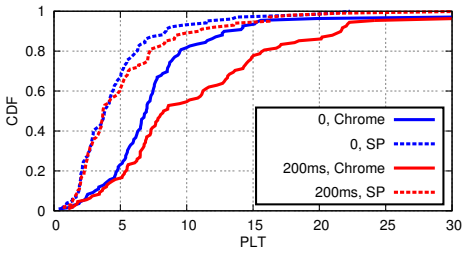


Figure 5: Varying RTT with fixed 1GHz CPU and 1GB memory.

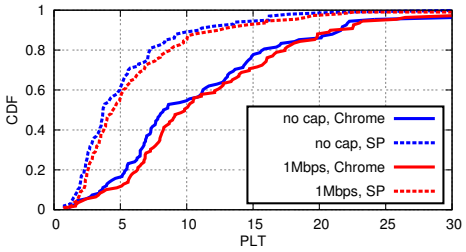


Figure 6: Varying bandwidth with fixed 1GHz CPU, 1GB memory, and 200ms RTT.

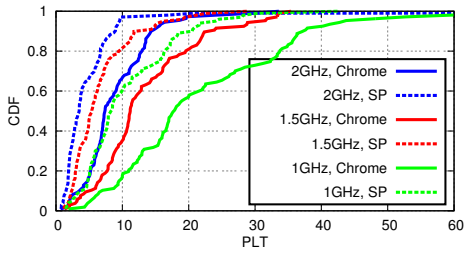


Figure 7: Varying CPU speed with fixed 1GB memory, no bandwidth cap, and no RTT insertion.

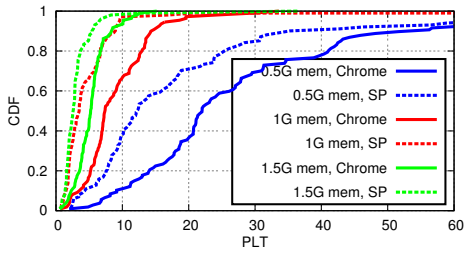


Figure 8: Varying memory size with fixed 2GHz CPU, no bandwidth cap, and no RTT insertion.

work using Ethernet. We use Dummynet [11] to emulate varying bandwidths and RTTs.

Varying RTT: We vary RTT from the minimal of the LAN to 200 milliseconds with fixed 1GHz CPU and 1GB memory, which are representative of current mobile devices. We do not cap the bandwidth. The scenario of minimal RTT approximates the scenario of having caching always enabled. Figure 5 shows the cumulative distributions of PLTs of the 100 Web pages. The increased RTT affects much of PLT with Chrome but affects little of PLT with `Shandian`, meaning that `Shandian` is insensitive to RTT. This is because among the breakdowns of PLT only t_f which is a small fraction of PLT is affected by RTT.

Varying bandwidth: We experiment with a 1Mbps bandwidth and with no bandwidth cap using fixed 1GHz CPU, 1GB memory, and 200ms RTT. Figure 6 shows that PLTs are affected little by bandwidths, which is consistent with previous findings [33, 31] that bandwidth is not a limiting factor of PLTs. We also run experiments in a cellular network but find similar results to simulated links.

Varying CPU: We vary CPU speed from 1GHz to 2GHz while fixing memory size to 1GB. We do not tune RTT or bandwidth, meaning that PLT is dominated by computation. Figure 7 shows that the PLT improvement is linear to CPU increase. It also shows that CPU speed has the same amount of impact for both `Shandian` and Chrome, because processing load-time state in `Shandian` still incurs lots of CPU cycles. As PLT is dominated by computation, the results here approximate the situations when objects are inlined or cached. Clearly, `Shandian` significantly improves PLTs than simply inlining objects since JavaScript evaluations and most of CSS evaluations are removed from the page load process.

Varying memory: We vary memory size from 0.5GB to 1.5GB with fixed 1GHz CPU and no network tuning. Figure 8 suggests that memory size has the same amount of impact for both `Shandian` and Chrome, but a decrease in memory size has a more than linear negative impact on PLT.

In summary, `Shandian` significantly improves PLT compared to Chrome under a variety of realistic mobile scenarios. This is rare since most techniques are specific to improve one of computation and network. But `Shandian` improves both.

Note that we do not evaluate page interactivity metrics, e.g., the time until interaction is possible, because users spend time on the contents of a Web page before interacting with it and it is difficult to model this delay. `Shandian` could improve the time until interaction since all external resources are loaded and evaluated in parallel, but it can also hurt if the load-time state is too large and blocks the transfer of the post-load state that is required for page interactivity.

5.3 Size of transferred data

We evaluate the transferred data size as to (i) whether it hurts latencies and (ii) whether it hurts data usage. To understand whether the size of transferred data helps

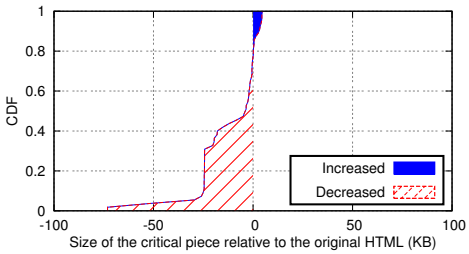


Figure 9: Size of the critical piece relative to the original HTML (KB).

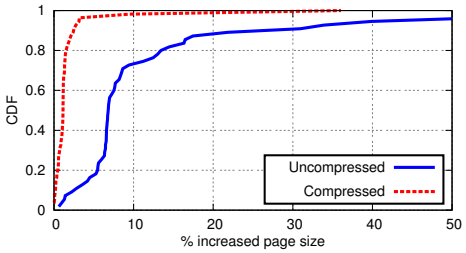


Figure 10: Percentage of increased page size (uncompressed v.s. compressed).

or hurts latencies, we consider the size of the load-time state, because the post-load state and other objects do not affect the page load time. Figure 9 shows the size of the load-time state when a standard gzip compression is applied. The size of the load-time state relative to the original HTML decreases for most pages, and increases by a small amount only for less than 20% of the pages. This means that our migrated load-time state improves latencies in overall.

To evaluate whether the size of migrated state hurts data usage which is important for mobile browsers, we consider the total size of Web pages transferred to the client including all the embedded objects. Figure 10 shows that the transferred data increases by 7% before compression, but it drops to 1% with standard gzip compression. This indicates that the overheads introduced by our approach are minimal.

5.4 Client-side state

We obtain client-side state from the browsers of a group of people, totaling 2,435 domains. The majority of the client-side state is HTML5 localStorage and cookies. We find that 90% of the websites use less than 460 bytes of localStorage, while 2% of the websites use more than 100KB of localStorage. Large localStorage is almost always used for caching. For example, websites that use CloudFlare keep many caches in their localStorage; social networking websites such as Facebook store friends lists in the localStorage; and location-based services maintain the points of interest in the localStorage. Lack of such localStorage does not break Web pages because cache misses can be remedied by fetching from the server. Unlike localStorage, cookies are widely used but are always small, and Web databases are used in less than 1% of the websites and are small. The use of sessionStorage is also not much, likely because it only persists per-session state and cannot cache as long as localStorage. The study of client-side state suggests that migrating all the necessary client-side state (e.g., cookies) to the server has a negligible effect on page load time.

6 Discussion

We believe that `Shandian` is an important first step in mitigating dependencies that are the key bottleneck of latencies in page loads. While future advances in JavaScript or the Web might require us to patch `Shandian` so as to ensure that `Shandian` does not break Web pages, there are no fundamental obstacles that prevent us from patching `Shandian` to track changes to the web page formats. Next, we discuss privacy, compatibility, and further optimizations that can be added to `Shandian`.

Latency-reducing techniques. `Shandian` is compatible with existing latency-reduction techniques with notable examples of caching and CDNs. Both caching and CDNs use a URL as the key to store a Web object. To preserve the use of caching and CDNs, we need to preserve the integrity of both the Web object itself and its corresponding URL. We leave images and other media unmodified because they do not block HTML parsing, and we make the design decisions to migrate unmodified CSS and JavaScript in the post-load state. All the resources that are typically cached or served from CDNs are kept unmodified, meaning that all the caching and CDN abilities are preserved.

Privacy. We consider the additional information that users have to sacrifice in order to use `Shandian`. Even without `Shandian`, websites already have access to user information revealed as part of the page load process (e.g., access patterns, user locations), and `Shandian` does not result in the exposure of additional user information. This is because: (i) website information stored in browsers in the form of cookies or localStorage comes from the website itself; (ii) current browsers expose geolocation to websites upon receiving consent from users; (iii) websites have already had access to the browser information (using JavaScript). To sum up, the client-side state either comes from the website or is already exposed to the website in the absence of `Shandian`, and thus users do not have to expose any additional information to servers in order to use `Shandian`.

Our design is compatible with HTTPS if it is deployed on a reverse proxy that has terminated SSL, but requires additional trust when deployed as a globally distributed proxy. Similar to Amazon Silk and Opera mini, we

would need to trust the proxy. The connection between the Web server and the proxy and the connection between the proxy and the client use two separate HTTPS connections. To handle SSL certificates, we need to route the requests to the proxy so that the proxy can fetch Web pages on behalf of the client.

Security techniques backed by same-origin policies. Same-origin policy (SOP) is used to protect third-party scripts from accessing first-party assets such as cookies. Our design is compatible with SOP when third-party scripts are embedded using an `iframe`, because frames and parent document are isolated from each other. When third-party scripts are embedded using a `script` tag, they are given full permissions to access the first-party assets, in which case our design respects SOP.

Scaling the proxy servers. `Shandian` adds computation costs to proxy servers, making it hard to scale. We discuss the scalability issues of `Shandian` in three adoption scenarios: by browsers, by third-party proxy vendors, and by websites respectively. If browsers or third-party proxy vendors were to adopt `Shandian`, they can rent private cloud instances (e.g., Amazon EC2) to users, which is similar to the scenarios of Opera Mini and Amazon Silk.

If websites were to adopt `Shandian`, additional work has to be done to increase scalability. A possible approach is to exploit the similarities of the Web pages within a website. For example, when the same Web page is sent to different users, most portions of the page are the same except for personalized data. The server side of `Shandian` can cache intermediate representations that are generated from loading one Web page for one user. These intermediate representations, if used smartly, can reduce the computation of loading the same page for a different user, or for loading a different page (if there are similarities across pages). The technical details of this extension are outside of the scope of this paper.

Using a cloud-based proxy for compression. `Shandian` is orthogonal to existing cloud-based proxy approaches that do not restructure the page load process. This means that even if a proxy is already placed near the server for `Shandian`, another proxy can be placed near the client for other purposes (e.g., Android Chrome Beta [1] for data compression, SPDY proxies for rewriting connections between the proxy and the device). However, approaches that restructure the page load process at clients (e.g., Opera mini [26] and Amazon Silk [3]) cannot be used together with `Shandian`.

Extending the definition of PLT. Currently, `Shandian` is designed for improving page load times defined by the W3C `load` event. But it would be trivial to extend `Shandian` to improve any definition of page load times. The key is to capture the state of event listeners and the progress of HTML parsing for a given definition of page load time. The flexibility of PLT definitions is important because reports have shown that user-perceived page load times matter more than when the `load` event is fired [7].

7 Related Work

Cloud browsers for mobile devices. The closest related work is cloud browsers for mobile devices. Opera mini [26] and Amazon Silk [3] only handle display in client-side browsers. Therefore, evaluating JavaScript depends on the network which is demonstrated to be both slow and unreliable in mobile settings [30]. Different from these browsers, we provide a fully functioning client-side browser that reduces latencies. Mobile Chrome [1] compresses Web pages through a proxy server to reduce network traffic. Our work is orthogonal to Mobile Chrome.

Mitigating page load dependencies. To mitigate the impact of page load dependencies, SPDY server push, Klotski [10], and techniques developed by Instart Logic [20] provide means to prioritize Web contents at the object level on front-end servers, proxies, and browsers respectively. These solutions require knowledge of dependencies between Web objects within a page beforehand to build a prioritization plan. `Shandian` prioritizes Web contents at a finer granularity and does not require the system to obtain any knowledge of the Web pages beforehand. Best practices for Web authoring also aim at mitigating page load dependencies [31]. For example, a common advice is to place CSS at the beginning of a Web page and to place JavaScript at the end of a Web page. But, such advice is hard to execute since the construction of many Web pages depends on using JavaScript libraries such as jQuery [21] and D3.js [14], which need to appear above where they are used in a page. `Shandian` is the first that automatically enforces this best practice.

Improving computation. Much work has been done to improve page load computations, with a focus on exploiting parallelism. Meyerovich et al. proposed a parallel architecture for computing Web page layout by parallelizing CSS evaluations [25]. The Adrenaline browser exploits parallelism by splitting up a Web page into many pieces and processing each piece in parallel [24]. The ZOOMM browser further parallelizes the browser engine by preloading and preprocessing objects and by speeding up computation in sub-activities [12]. Due to the dependencies that are intrinsic in browsers, the level of parallelism is largely limited. `Shandian` removes dependencies for initial page loads on the client and thus provides opportunities for more parallelism. Besides in-

creasing parallelism, other efforts focus on adding architectural support. Zhu et al. [41] specialized the processors for fast DOM tree and CSS access. Choi et al. [13] proposed a hybrid-DOM to efficiently access the DOM nodes. These approaches are orthogonal to Shandian.

Improving network. There are several efforts that improve page load latencies at the networking level. This includes speculations inside browsers (e.g., TCP preconnect [19], DNS pre-resolution [22]), using new protocols (e.g., SPDY [32], QUIC [29]), and improving TCP for Web traffic (e.g., TCP fast open [28], proportional rate reduction [16], and Tail loss probe [15, 17]). While being effective in reducing latencies for a single Web object, these techniques have a limited impact in reducing page load times; these techniques reduce the network costs but do not reduce the amount of HTTP requests for the initial page load nor do they eliminate the inefficiencies associated with dependencies. Instead, we first identify page load dependencies as the primary bottleneck for PLT, and then propose Shandian to mitigate the dependencies.

8 Summary

In this paper, we presented Shandian that improves PLT by simplifying the client-side page load process through an architecture that splits the page load process between a proxy server and the client. By performing preprocessing in the proxy server with more compute power, Shandian largely reduces the inefficiencies of page loads on the client. Shandian is fast for displaying Web pages, ensures that users are able to continue interacting with the page, and is compatible with caching, CDNs, and security features that enforce same-origin policies. Our evaluations show that Shandian reduces PLTs by more than half on a variety of mobile settings with varied RTT, bandwidth, CPU power, and memory size.

Acknowledgements

We thank our shepherd, Vyas Sekar, and the anonymous reviewers for their feedback. This work was supported by the National Science Foundation under grants CNS-1318396, CNS-1420703, and CNS-1518702.

References

[1] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Googles data compression proxy for the mobile web. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2015.

[2] Top sites in United States. http://www.alexa.com/topsites/countries/US.

[3] Amazon silk browser. http://amazonsilk.wordpress.com/.

[4] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into Web server design. In *Computer Networks Volume 33, Issue 1-6*, 2000.

[5] Blink: Chrome's Rendering Engine. http://www.chromium.org/blink.

[6] A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the eye of the beholder: meeting users' requirements for Internet quality of service. In *Proc. of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI), 2000*.

[7] J. Brutlag. Above the fold time: Measuring web page performance visually, Mar. 2011. http://en.oreilly.com/velocity-mar2011/public/schedule/detail/18692.

[8] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive Record/Replay for Web Application Debugging. In *Proc. of the ACM UIST, 2013*.

[9] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2011*.

[10] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2015.

[11] M. Carbone and L. Rizzo. Dummynet revisited. *ACM SIGCOMM Computer Communication Review*, 40(2):12–20, Mar. 2010.

[12] C. Cascaval, S. Fowler, P. M. Ortego, W. Piekarski, M. Reshadi, B. Robatmili, M. Weber, and V. Bhavsar. ZOOMM: A Parallel Web Browser Engine for Multicore Mobile Devices. In *Proc. of the ACM PPoPP, 2013*.

[13] R. H. Choi and Y. Choi. Designing a high-performance mobile cloud web browser. In *Proc. of the International World Wide Web Conference (WWW)*, 2014.

[14] D3.js. http://d3js.org/.

[15] N. Dukkipati, N. Cardwell, Y. Cheng, and M. Mathis. Tail Loss Probe (TLP): An algorithm for fast recovery of tail losses, Feb. 2013.

http://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01.

[16] N. Dukkipati, M. Mathis, Y. Cheng, and M. Ghobadi. Proportional rate reduction for TCP. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC)*, 2011.

[17] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan. Reducing web latency: the virtue of gentle aggression. In *Proc. of the ACM SIGCOMM*, 2013.

[18] Google. Content module. http://www.chromium.org/developers/content-module.

[19] I. Grigorik. Chrome networking: DNS prefetch & TCP preconnect, June 2012. http://www.igvita.com/2012/06/04/chrome-networking-dns-prefetch-and-tcp-preconnect/.

[20] Instart logic. https://www.instartlogic.com/.

[21] jquery. https://www.jquery.com/.

[22] E. Lawrence. Internet Explorer 9 network performance improvements, Mar. 2011. http://blogs.msdn.com/b/ie/archive/2011/03/17/internet-explorer-9-network-performance-improvements.aspx.

[23] J. Lo, E. Wohlstadter, and A. Mesbah. Imagen: Runtime Migration of Browser Sessions for JavaScript Web Applications. In *Proc. of the International World Wide Web Conference (WWW), 2013*.

[24] H. Mai, S. Tang, S. T. King, C. Cascaval, and P. Montesinos. A Case for Parallelizing Web Pages. In *Proc. of HotPar, 2012*.

[25] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. In *Proc. of the international conference on World Wide Web (WWW), 2010*.

[26] Opera mini browser. http://www.opera.com/mobile/.

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

[28] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. In *Proc. of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2011.

[29] J. Roskind. QUIC, a multiplexed stream transport over UDP. http://www.chromium.org/quic.

[30] 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 *Proc. of HotMobile, 2014*.

[31] S. Souders. *High Performance Web Sites*. O'Reilly Media, 2007.

[32] Spdy. http://dev.chromium.org/spdy.

[33] SPDY whitepaper. http://www.chromium.org/spdy/spdy-whitepaper.

[34] Speed index. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

[35] Apache module for rewriting web pages to reduce latency and bandwidth. http://www.modpagespeed.com/.

[36] V8: Chrome's JavaScript Engine. https://developers.google.com/v8/.

[37] Cascading Style Sheets level 2 revision 1 (CSS 2.1) specification, June 2011. http://www.w3.org/TR/CSS21/.

[38] Document Object Model (DOM) Level 3 Events specification, Sept. 2014. http://www.w3.org/TR/DOM-Level-3-Events/.

[39] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with WProf. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI), 2013*.

[40] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is SPDY? In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2014.

[41] Y. Zhu and V. J. Reddi. WebCore: Architectural Support for Mobile Web Browsing. In *Proc. of the 41st International Symposium on Computer Architecture (ISCA)*, 2014.