Demystifying Page Load Performance with WProf

Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall
University of Washington

Abstract

Web page load time is a key performance metric that many techniques aim to reduce. Unfortunately, the complexity of modern Web pages makes it difficult to identify performance bottlenecks. We present WProf, a lightweight in-browser profiler that produces a detailed dependency graph of the activities that make up a page load. WProf is based on a model we developed to capture the constraints between network load, page parsing, JavaScript/CSS evaluation, and rendering activity in popular browsers. We combine WProf reports with critical path analysis to study the page load time of 350 Web pages under a variety of settings including the use of end-host caching, SPDY instead of HTTP, and the mod_pagespeed server extension. We find that computation is a significant factor that makes up as much as 35% of the critical path, and that synchronous JavaScript plays a significant role in page load time by blocking HTML parsing. Caching reduces page load time, but the reduction is not proportional to the number of cached objects, because most object loads are not on the critical path. SPDY reduces page load time only for networks with high RTTs and mod_pagespeed helps little on an average page.

1 Introduction

Web pages delivered by HTTP have become the de-facto standard for connecting billions of users to Internet applications. As a consequence, Web page load time (PLT) has become a key performance metric. Numerous studies and media articles report its importance for user experience [5, 4], and consequently to business revenues. For example, Amazon increased revenue 1% for every 0.1 second reduction in PLT, and Shopzilla experienced a 12% increase in revenue by reducing PLT from 6 seconds to 1.2 seconds [23].

Given its importance, many techniques have been developed and applied to reduce PLT. They range from caching and CDNs, to more recent innovations such as the SPDY protocol [28] that replaces HTTP, and the mod_pagespeed server extension [19] to re-write pages. Other proposals include DNS pre-resolution [9], TCP pre-connect [30], Silo [18], TCP fast open [24], and ASAP [35].

Thus it is surprising to realize that the performance bottlenecks that limit the PLT of modern Web pages are still not well understood. Part of the culprit is the complexity of the page load process. Web pages mix resources fetched by HTTP with JavaScript and CSS evaluation. These activities are inter-related such that the bottlenecks are difficult to identify. Web browsers complicate the situation with implementation strategies for parsing, loading and rendering that significantly impact PLT. The result is that we are not able to explain why a change in the way a page is written or how it is loaded has an observed effect on PLT. As a consequence, it is difficult to know when proposed techniques will help or harm PLT.

Previous studies have measured Web performance in different settings, e.g., cellular versus wired [13], and correlated PLT with variables such as the number of resources and domains [7]. However, these factors are only coarse indicators of performance and lack explanatory power. The key information that can explain performance is the dependencies within the page load process itself. Earlier work such as WebProphet [16] made clever use of inference techniques to identify some of these dependencies. But inference is necessarily time-consuming and imprecise because it treats the browser as a black-box. Most recently, many "waterfall" tools such as Google's Pagespeed Insight [22] have proliferated to provide detailed and valuable timing information on the components of a page load. However, even these tools are limited to reporting what happened without explaining why the page load proceeded as it did.

Our goal is to demystify Web page load performance. To this end, we abstract the dependency policies in four browsers, i.e., IE, Firefox, Chrome, and Safari. We run experiments with systematically instrumented test pages and observe object timings using Developer Tools [8]. For cases when Developer Tools are insufficient, we deduce dependencies by inspecting the browser code when open source code is available. We find that some of these dependency policies are given by Web standards, e.g., JavaScript evaluation in script tags blocks HTML parsing. However, other dependencies are the result of browser implementation choices, e.g., a single thread of execution shared by parsing, JavaScript and CSS evaluation, and rendering. They have significant impacts on PLT and cannot be ignored.

Given the dependency policies, we develop a lightweight profiler, WProf, that runs in Webkit browsers (e.g., Chrome, Safari) while real pages are loaded. WProf generates a dependency graph and identifies a load bottleneck for any given Web page. Unlike existing tools that produce waterfall or HAR reports [12], our profiler discovers and reports the dependencies between the browser activities that make up a page load. It is this information that pinpoints why, for example, page parsing took unexpectedly long and hence suggests what may be done to improve PLT.

To study page load performance, we run WProf while fetching pages from popular servers and apply critical path analysis, a well-known technique for analyzing the performance of parallel programs [25]. First, we identify page load bottlenecks by computing what fraction of the critical path the activity occupies. Surprisingly, we find that while most prior work focuses on network activity, computation (mostly HTML parsing and JavaScript execution) comprises 35% of the critical path. Interestingly, downloading HTML and synchronous JavaScript (which blocks parsing) makes up a large fraction of the critical path, while fetching CSS and asynchronous JavaScript makes up little of the critical path. Second, we study the effectiveness of different techniques for optimizing web page load. Caching reduces the volume of data substantially, but decreases PLT by a lesser amount because many of the downloads that benefit from it are not on the critical path. Disappointingly, SPDY makes little difference to PLT under its default settings and low RTTs because it trades TCP connection setup time for HTTP request sending time and does not otherwise change page structure. Mod_pagespeed also does little to reduce PLT because minifying and merging objects does not reduce network time on critical paths.

We make three contributions in this paper. The first is our activity dependency model of page loads, which captures the constraints under which real browsers load Web pages. The second contribution is WProf, an in-browser profiling tool that records page load dependencies and timings with minimal runtime overhead. Our third contribution is the study of extensive page loads that uses critical path analysis to identify bottlenecks and explain the limited benefits of SPDY and mod_pagespeed.

In the rest of this paper, we describe the page load process (§2) and our activity dependency model (§3). We then describe the design and implementation of WProf (§4). We use WProf for page load studies (§5) before presenting related work (§6) and concluding (§7).

2 Background

We first provide background on how browsers load Web pages. Figure 1 shows the workflow for loading a page. The page load starts with a user-initiated request that triggers the *Object Loader* to download the corresponding root HTML page. Upon receiving the first chunk of the root page, the *HTML Parser* starts to iteratively parse the page and download embedded objects within the page, until the page is fully parsed. The embedded objects are *Evaluated* when needed. To visualize the page, the *Rendering Engine* progressively renders the page on the browser. While the HTML Parser, Evaluator, and Rendering Engine are computation processes, the Object Loader is a network process.

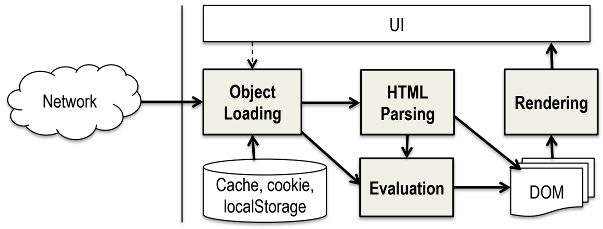


Figure 1: The workflow of a page load. It involves four processes (shown in gray).

HTML Parser: The Parser is key to the page load process, and it transforms the raw HTML page to a document object model (DOM) tree. A DOM tree is an intermediate representation of a Web page; the nodes in the DOM tree represent HTML tags, and each node is associated with a set of attributes. The DOM tree representation provides a common interface for programs to manipulate the page.

Object Loader: The Loader fetches objects requested by the user or those embedded in the HTML page. The objects are fetched over the Internet using HTTP or SPDY [28], unless the objects are already present in the browser cache. The embedded objects fall under different mime types: HTML (e.g., IFrame), JavaScript, CSS, Image, and Media. Embedded HTMLs are processed separately and use a different DOM tree. Inlined JavaScript and inlined CSS do not need to be loaded.

Evaluator: Two of the five embedded object types, namely, JavaScript and CSS, require additional evaluation after being fetched. JavaScript is a piece of software that adds dynamic content to Web pages. Evaluating JavaScript involves manipulating the DOM, e.g., adding new nodes, modifying existing nodes, or changing nodes' styles. Since both JavaScript Evaluation and HTML Parsing modify the DOM, HTML parsing is blocked for JavaScript evaluation to avoid conflicts in DOM modification. However, when JavaScript is tagged with an *async* attribute, the JavaScript can be downloaded and evaluated in the background without blocking HTML Parsing. In the rest of the paper, JavaScript refers to synchronous JavaScript unless stated.

Cascading style sheets (CSS) are used for specifying the presentational attributes (e.g., colors and fonts) of the HTML content and is expressed as a set of rules. Evaluating a CSS rule involves changing styles of DOM nodes. For example, if a CSS rule specifies that certain nodes are to be displayed in blue color, then evaluating the CSS involves identifying the matching nodes in the DOM (known as CSS selector matching) and adding the style to each matched node. JavaScript and CSS are commonly embedded in Web pages today [7].

Rendering engine: Browsers render Web pages progressively as the HTML Parser builds the DOM tree. Rendering involves two processes–Layout and Painting. Layout converts the DOM tree to the layout tree that encodes the size and position of each visible DOM node. Painting converts this layout tree to pixels on the screen.

3 Browser Dependency Policies

Web page dependencies are caused by various factors such as co-dependence between the network and the computation activities, manipulation of common objects, limited resources, etc. Browsers use various policies to enforce these dependencies. Our goal is to abstract the browser dependency policies. We use this policy abstraction to efficiently extract the dependency structure of any given Web page (§4).

3.1 Dependency definition

Ideally, the four processes involved in a page load (described in Figure 1) would be executed in parallel, so that the page load performance is determined by the slowest process. However, the processes are inter-dependent and often block each other. To represent the dependency policies, we first discretize the steps involved in each process. The granularity of discretization should be fine enough to reflect dependencies and coarse enough to preserve semantics. We consider the most coarse-grained atomic unit of work, which we call an *activity*. In the case of HTML Parser, the activity is parsing a single tag. The Parser repeatedly executes this activity until all tags are parsed. Similarly, the Object Loader activity is loading a single object, the Evaluator activity is evaluating a single JavaScript or CSS, and the activity of the Rendering process is rendering the current DOM.

We say an activity a_i is *dependent* on a previously scheduled activity a_j, if a_i can be executed only after a_j has completed. There are two exceptions to this definition, where the activity is executed after only a *partial* completion of the previous activity. We discuss these exceptions in §3.3.

3.2 Methodology

We extract browser dependency policies by (i) inspecting browser documentation, (ii) inspecting browser code if open-source code is available, and (iii) systematically instrumenting test pages. Note that no single method provides a complete set of browser policies, but they complement each other in the information they provide. Our methodology assumes that browsers tend to parse tags sequentially. However, one exception is *preloading*. Preloading means that the browser preemptively loads objects that are embedded later in the page, even before their corresponding tags are parsed. Preloading is often used to speed up page loads.

Below, we describe how we instrument and experiment with test pages. We conduct experiments in four browsers: Chrome, Firefox, Internet Explorer, and Safari. We host our test pages on controlled servers. We observe the load timings of each object in the page using Developer Tools [8] made available by the browsers. We are unable to infer dependencies such as rendering using Developer Tools. Instead, for open-source browsers, we inspect browser code to study the dependency policies associated with rendering.

Instrumenting test pages (network): We instrument test pages to exhaustively cover possible loading scenarios: (i) loading objects in different orders, and (ii) loading embedded objects. We list our instrumented test pages and results at `wprof.cs.washington.edu/tests`. Web pages can embed five kinds of objects as described in §2. We first create test pages that embed all combinations of object pairs, e.g., the test page may request an embedded JavaScript followed by an image. Next, we create test pages that embed more than two objects in all possible combinations. Embedded objects may further embed other objects. We create test pages for each object type that in-turn embeds all combinations of objects. To infer dependency policies, we systematically inject delays to objects and observe load times of other objects to see whether they are delayed accordingly, similar to the technique used in WebProphet [16]. For example, in a test page that embeds a JavaScript followed by an image, we delay loading the Javascript and observe whether the image is delayed.

Instrumenting test pages (computation): Developer tools expose timings of network activities, but not timings of computational activities. We instrument test pages to circumvent this problem and study dependencies during two main computational activities: HTML parsing and JavaScript evaluation. HTML parsing is often blocked during page load. To study the blocking behavior across browsers, we create test pages for each object type. For each page, we also embed an IFrame in the end. During page load, if the IFrame begins loading only after the previous object finishes loading, we infer that HTML parsing is blocked during the object load. IFrames are ideal for this purpose because they are not preloaded by browsers. To identify dependencies related to JavaScript evaluation, we create test pages that contain scripts with increasing complexity; i.e., scripts that require more and more time for evaluation. We embed an IFrame at the end. As before, if IFrame does not load immediately after the script loads, we infer that HTML parsing is blocked for script evaluation.

Dependency	Name	Definition
Flow	F1	Loading an object $\rightarrow$ Parsing the tag that references the object
	F2	Evaluating an object $\rightarrow$ Loading the object
	F3	Parsing the HTML page $\rightarrow$ Loading the first block of the HTML page*
	F4	Rendering the DOM tree $\rightarrow$ Updating the DOM
	F5	Loading an object referenced by a JavaScript or CSS $\rightarrow$ Evaluating the JavaScript or CSS*
	F6	Downloading/Evaluating an object $\rightarrow$ Listener triggers or timers
Output	O1	Parsing the next tag $\rightarrow$ Completion of a previous JavaScript download and evaluation
	O2	JavaScript evaluation $\rightarrow$ Completion of a previous CSS evaluation
	O3	Parsing the next tag $\rightarrow$ Completion of a previous CSS download and evaluation
Lazy/Eager binding	B1	[Lazy] Loading an image appeared in a CSS $\rightarrow$ Parsing the tag decorated by the image
	B2	[Lazy] Loading an image appeared in a CSS $\rightarrow$ Evaluation of any CSS that appears in front of the tag decorated by the image
	B3	[Eager] Preloading embedded objects does not depend on the status of HTML parsing. (breaks F1)
Resource constraint	R1	Number of objects fetched from different servers $\rightarrow$ Number of TCP connections allowed per domain
	R2	Browsers may execute key computational activities on the same thread, creating dependencies among the activities.This dependency is determined by the scheduling policy.

* An activity depends on *partial* completion of another activity.

Table 1: Summary of dependency policies imposed by browsers. $\rightarrow$ represents "depends on" relationship.

3.3 Dependency policies

Using our methodology, we uncover the dependency policies in browsers and categorize them as: Flow dependency, Output dependency, Lazy/Eager binding dependency, and dependencies imposed by resource constraints. Table 1 tabulates the dependency policies. While output dependencies are required for correctness, the dependencies imposed by lazy/eager binding and resource constraints are a result of browser implementation strategies.

Flow dependency is the simplest form of dependency. For example, loading an object depends on parsing a tag that references the object (F1). Similarly, evaluating a JavaScript depends on loading the JavaScript (F2). Often, browsers may load and evaluate a JavaScript based on triggers and timeouts, rather than the content of the page (F6). Table 1 provides the complete set of flow dependencies. Note that dependencies F3 and F5 are special cases, where the activity only depends on the partial completion of the previous activity. In case of F3, the browser starts to parse the page when the first chunk of the page is loaded, not waiting for the entire load to be completed. In case of F5, an object requested by a JavaScript/CSS is loaded immediately after evaluation starts, not waiting for the evaluation to be completed.

Output dependency ensures the correctness of execution when multiple processes modify a shared resource and execution order matters. In browsers, the shared resource is the DOM. Since both JavaScript evaluation and HTML parsing may write to the DOM, HTML parsing is blocked until JavaScript is both loaded and evaluated (O1). This ensures that the DOM is modified in the order specified in the page. Since JavaScript can modify styles of DOM nodes, execution of JavaScript waits for the completion of CSS processing (O2). Note that *async* JavaScript is not bounded by output dependencies because the order of script execution does not matter.

Lazy/Eager binding: Several lazy/eager bindings techniques are used by the browser to trade off between decreasing spurious downloads and improving latency. Preloading (B3) is an example of an eager binding technique where browsers preemptively load objects that are embedded later in the page. Dependency B1 is a result of a lazy binding technique. When a CSS object is downloaded and evaluated, it may include an embedded image, for example, to decorate the background or to make CSS sprites. The browser does not load this image as soon as CSS is evaluated, and instead waits until it parses a tag that is decorated by the image. This ensures that the image is downloaded only if it is used.

Resource constraints: Browsers constrain the use of two resources—compute power and network resource. With respect to network resources, browsers limit the number of TCP connections. For example, Firefox limits the number of open TCP connections per domain to 6 by default. If a page load process needs to load more than 6 embedded objects from the same domain simultaneously, the upcoming load is blocked until a previous load completes. Similarly, some browsers allocate a single compute thread to execute certain computational activities. For example, WebKit executes parts of rendering in the parsing thread. This results in blocking parsing until rendering is complete (R2). We were able to observe this only for the open-source WebKit browser because

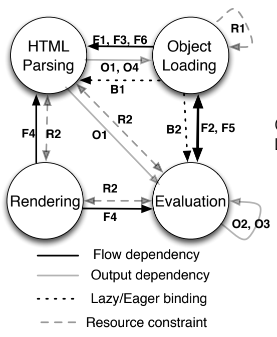


Figure 2: Dependencies between processes.

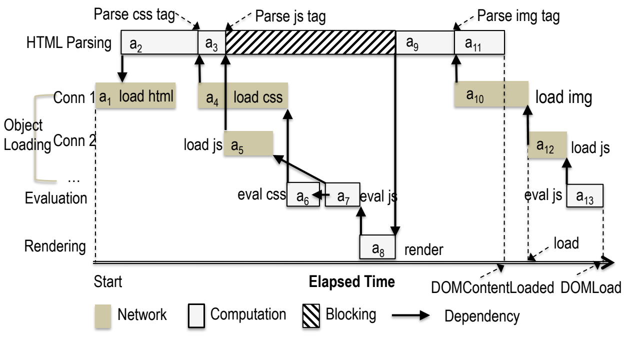
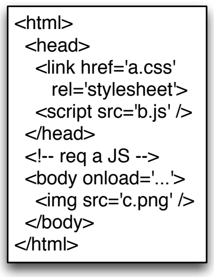


Figure 3: Dependency graph. Arrows denote“depends on” relation and vertices represent activities in a page load.



```
<html>
  <head>
    <link href='a.css'
      rel='stylesheet'>
    <script src='b.js' />
  </head>
  <!-- req a JS -->
  <body onload='...'>
    <img src='c.png' />
  </body>
</html>
```

Figure 4: Corresponding example code.

Dependency	IE	Firefox	WebKit
Output	all	no O3	no O3
Late binding	all	all	all
Eager Binding	*Preloads img, JS, CSS	Preloads img, JS, CSS	Preloads JS, CSS
Resource (R1)	6 conn.	6 conn.	6 conn.

Table 2: Dependency policies across browsers.

we directly instrumented the WebKit engine to log precise timing information of computational activities.

Figure 2 summarizes how these dependencies affect the four processes. Note that more than one dependency relationship can exist between two activities. For example, consider a page containing a CSS object followed by a JavaScript object. Evaluating the JavaScript depends on both loading the JavaScript (F2) and evaluating the previously appearing CSS (O3). The timing of these two dependencies will determine which of the two dependencies occur in this instance.

3.4 Dependency policies across browsers

Table 2 show the dependency policies across browsers. Only IE enforces dependency O3 that blocks HTML parsing to download and evaluate CSS. The preloading policies (i.e., when and what objects to preload) also differ among browsers, while we note that no browser preloads embedded IFrames. Note that flow dependency is implicitly imposed by all browsers. We were unable to compare the compute dependency (R2) across browsers because it requires modification to the browser code.

3.5 Dependency graph of an example page

Figure 3 shows a dependency graph of an example page. The `DOMContentLoaded` refers to the event that HTML finishes parsing, `load` refers to the event that all embedded objects are loaded, and `DOMLoad` refers to the event that DOM is fully loaded. Our example Web page (Figure 4) contains an embedded CSS, a JavaScript, an image, and a JavaScript triggered on the `load` event. Many Web pages (e.g., `facebook.com`) may load additional objects on the `load` event fires.

The page load starts with loading the root HTML page (a_1), following which parsing begins (a_2). When the Parser encounters the CSS tag, it loads the CSS (a_4) but does not block parsing. However, when the Parser encounters the JavaScript tag, it loads the JavaScript (a_5) and blocks parsing. Note that loading the CSS and the JavaScript subjects to a resource constraint; i.e., both CSS and JavaScript can be loaded simultaneously only if multiple TCP connections can be opened per domain. CSS is evaluated (a_6) after being loaded. Even though JavaScript finishes loading, it needs to wait until the CSS finishes evaluating, and then the JavaScript is evaluated (a_7). The rendering engine renders the current DOM (a_8) after which HTML parsing resumes (a_9). The Parser then encounters and loads an image (a_{10}) after which HTML parsing is completed and fires a `load` event. In our example, the triggered JavaScript is loaded (a_{12}) and evaluated (a_{13}). When all embedded objects are evaluated and the DOM is updated, the `DOMLoad` event is fired. Even our simple Web page exhibits over 10 dependencies of different types. For example, $a_2 \rightarrow a_1$ is a flow dependency; $a_7 \rightarrow a_6$ is an output dependency; and $a_9 \rightarrow a_8$ is a resource dependency.

The figure also illustrates the importance of using dependency graphs for fine-grained accounting of page load time. For example, both activity a_4 and a_5 are network loads. However, only activity a_4 contributes to page load time; i.e., decreasing the load time of a_5 does not decrease total page load time. Tools such as HTTP Archival Records [12] provide network time for each activity, but this cannot be used to isolate the bottleneck activities. The next section demonstrates how to isolate the bottleneck activities using dependency graphs.

4 WProf

We present WProf, a tool that captures the dependency graph for any given Web page and identifies the delay bottlenecks. Figure 5 shows WProf architecture. The primary component is an in-browser profiler that instruments the browser engine to obtain timing and depen-

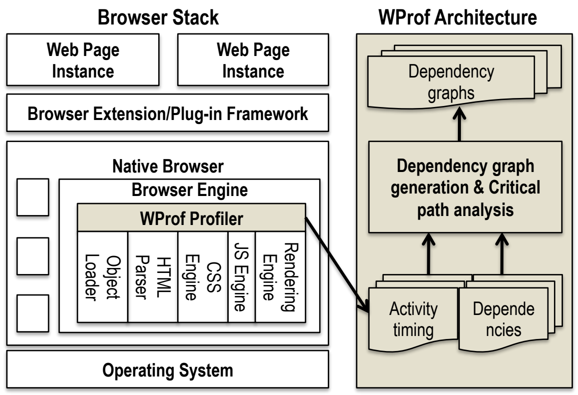


Figure 5: The WProf Architecture. WProf operates just above the browser engine, allowing it to collect precise timing and dependency information.

dency information at runtime. The profiler is a shim layer inside the browser engine and requires minimal changes to the rest of the browser. Note that we do not work at the browser extension or plugin level because they provide limited visibility into the browser internals. WProf then analyzes the profiler logs offline to generate the dependency graphs and identify the bottleneck path. The profiler is lightweight and has negligible effect on page load performance (§4.4).

4.1 WProf Profiler

The key role of WProf profiler is to record timing and dependency information for a page load. While the dependency information represents the structure of a page, the timing information captures how dependencies are exhibited for a specific page load; both are crucial to pinpoint the bottleneck path.

Logging Timing: WProf records the timestamps at the beginning and end of each activity executed during the page load process. WProf also records network timing information, including DNS lookup, TCP connection setup, and HTTP transfer time. To keep track of the number of TCP connections being used/re-used, WProf records the IDs of all TCP connections.

Logging dependencies: WProf assumes the dependency policies described in §3 and uses different ways to log different kinds of dependencies. Flow dependencies are logged by attaching the URL to an activity. For example in Figure 3, WProf learns that the activity that evaluates b.js depends on the activity that loads b.js by recording b.js. WProf logs resource constraints by IDs of the constrained resources, e.g., thread IDs and TCP IDs.

To record output dependencies and lazy/eager bindings, WProf tracks a DOM-specific ordered sequence of processed HTML tags as the browser loads a page. We maintain a separate ordered list for each DOM tree associated with the page (e.g., the DOM for the root page and the various IFrames). HTML tags are recorded when they are first encountered; they are then attached to the activities that occur when the tags are being parsed. For example, when objects are preloaded, the tags under parsing are attached to the preloading activity, not the tags that reference the objects. For example in Figure 4, the Parser first processes the tag that references a.css, and then processes the tag that references b.js. This ordering, combined with the knowledge of output dependencies, result in the dependency $a_7 \rightarrow a_6$ in Figure 3. Note that the HTML page itself provides an implicit ordering of the page activities; however, this ordering is static. For example, if a JavaScript in a page modifies the rest of the page, statically analyzing the Web page provides an incorrect order sequence. Instead, WProf records the Web page processing order directly from the browser runtime.

Validating the completeness of our dependency policies would require either reading all the browser code or visiting all the Web pages, neither of which is practical. As browsers evolve constantly, changes in Web standard or browser implementations can change the dependency policies. Thus, WProf needs to be modified accordingly. Since WProf works as a shim layer in browsers that does not require knowledge about underlying components such as CSS evaluation, the effort to record an additional dependency policy would be minimal.

4.2 Critical path analysis

To identify page load bottlenecks, we apply critical path analysis to dependency graphs. Let the dependency graph $G = (V, E)$, where V is the set of activities required to render the page and E is the set of dependencies between the activities. Each activity $a \in V$ is associated with the attribute that represents the duration of the activity. The critical path P consists of a set of activities that form the longest path in the dependency graph such that, reducing the duration of any activity *not* on the critical path ($a \notin P$), will not change the critical path; i.e., optimizing an activity not on the critical path will not reduce page load performance. For example, the critical path for the dependency graph shown in Figure 3 is (a_1, a_2, a_4, a_6, a_7, a_8, a_9, a_{11}, a_{12}, a_{13}).

We estimate the critical path P of a dependency graph using the algorithm below. We treat preloading as a special case because it breaks the flow dependency. When preloading occurs, we always add it to the critical path.

4.3 Implementation

We implement WProf on Webkit [33], an open source Web browser engine that is used by Chrome, Safari, Android, and iOS. The challenge is to ensure that the WProf shim layer is lightweight. Creating a unique identifier for each activity (for logging dependency relationship) is memory intensive as typical pages require several thou-

```
$P \leftarrow \emptyset$; $a \leftarrow$ Activity that completes last;
$P \leftarrow P \cup a$;
while $a$ is not the first activity do
    $A \leftarrow$ Set of activities that $a$ is dependent on;
    if $a$ is preloaded from an activity $a'$ in $A$ then
        $a \leftarrow a'$;
    else
        $a \leftarrow$ Activity in $A$ that completes last;
    end
    $P \leftarrow P \cup a$;
end
return $P$
```

WProf	CPU %	Memory %
on	58.5	65.5
off	53.5	54.9

Table 3: Maximum sampled CPU and memory usage.

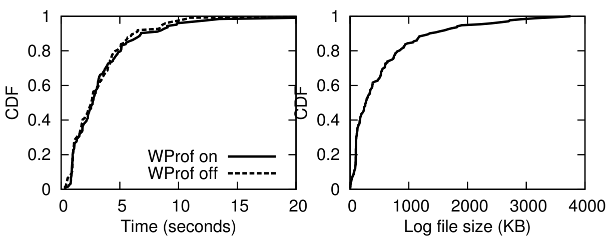
sands of unique identifiers (e.g., for HTML tags). Instead, we use pointer addresses as unique identifiers. The WProf profiler keeps all logs in memory and transfers them to disk after the DOM is loaded, to minimize the impact on page load.

We extended Chrome and Safari by modifying 2K lines of C++ code. Our implementation can be easily extended to other browsers that build on top of Webkit. Our dependency graph generation and critical path analysis is written in Perl and PHP. The WProf source code is available at `wprof.cs.washington.edu`.

4.4 System evaluation

In this section, we present WProf micro-benchmarks. We perform the experiments on Chrome. We use a 2GHz CPU dual core and 4GB memory machine running MacOS. We load 200 pages with a five second interval between pages. The computer is connected via Ethernet, and to provide a stable network environment, we limit the bandwidth to 10Mbps using DummyNet [10].

Figure 6(a) shows the CDF of page load times with and without WProf. We define page load time as the time from when the page is requested to when the `DOMLoad` (Figure 3) event is fired. We discuss our rationale for the page load time definition in §5.1. The figure shows that WProf's in-browser profiler only has a negligible effect on the page load time. Similarly, Figure 6(b) shows the CDF of size of the logs generated by the WProf profiler. The median log file size is 268KB even without compression, and is unlikely to be a burden on the storage system. We sample the CPU and memory usage at a rate of 0.1 second when loading the top 200 web pages with and without WProf. Table 3 shows that even in the maximum case, WProf only increases the CPU usage by 9.3% and memory usage by 19.3%.



(a) CDF of page load time with and without WProf.
(b) CDF of log file sizes.

Figure 6: WProf evaluation.

5 Studies with WProf

The goal of our studies is to use WProf's dependency graph and critical path analysis to (i) identify the bottleneck activities during page load (§5.2), (ii) quantify page load performance under caching (§5.3), and (iii) quantify page load performance under two proposed optimization techniques, SPDY and mod_pagespeed (§5.4).

5.1 Methodology

Experimental setup: We conduct our experiments on default Chrome and WProf-instrumented Chrome. We automate our experiments using the Selenium Webdriver [27] that emulates browser clicks. By default, we present experiments conducted on an iMac with a 3GHz quad core CPU and 8GB memory. The computer is connected to campus Ethernet at UW Seattle. We use DummyNet [10] to provide a stable network connection of 10Mbps bandwidth; 10Mbps represents the average broadband bandwidth seen by urban users [6]. By default, we assume page loads are *cold*, i.e., the local cache is cleared. This is the common case, as 40%–60% of page loads are known to be cold loads [18]. We report the minimum page load time from a total of 5 runs.

Web pages: We experiment with the top 200 most visited websites from Alexa [3]. Because some websites get stuck due to reported hang bugs in Selenium Webdriver, we present results from the 150 websites that provide complete runs.

Page load time metric: We define Page Load Time (PLT) as the time between when the page is requested and when the `DOMLoad` event is fired. Recall that the `DOMLoad` event is fired when all embedded objects are fetched and added to the DOM (see Figure 3). We obtain all times directly from the browser engine. There are a few alternative definitions to PLT. The *above-the-fold* time [1] metric is a user-driven metric that measures the time until the page is shown on the screen. However, this metric requires recording and manually analyzing page load videos, a cumbersome and non-scalable process. Other researchers use `load` [7] or `DOMContentLoaded` [18] events logged in the HTTP

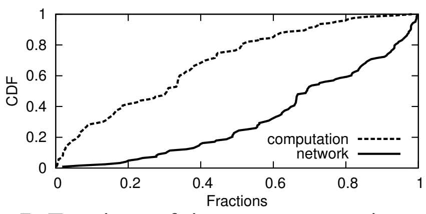


Figure 7: Fractions of time on computation v.s. network on the critical path.

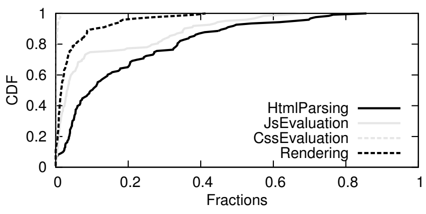


Figure 8: A breakdown of computational time to total page load time on the critical path.

Archival Record (HAR) [12] to indicate the end of the page load process. Since we can tap directly into the browser, we do not need to rely on the external HAR.

We perform additional experiments with varying location, compute power, and Internet speeds. To exclude bias towards popular pages, we also experiment on 200 random home pages that we choose from the top 1 million Alexa websites. We summarize our results of these experiments in §5.5.

5.2 Identifying load bottlenecks (no caching)

The goal of this section is to characterize bottleneck activities that contribute to the page load time. Note that all of these results focus on delays on critical paths. In aggregate, a typical page we analyzed contained 32 objects and 6 activities on the critical path (all median values).

Computation is significant: Figure 7 shows that 35% of page load time in the critical path is spent on computation; therefore computation is a critical factor in modeling or simulating page loads. Related measurements [14] do not estimate this computational component because they treat the browser engine as a black box.

We further break down computation into HTML parsing, JavaScript and CSS evaluation, and rendering in Figure 8. The fractions are with respect to the total page load time. Little time on the critical path is spent on firing timers or listeners, and so we do not show them. Interestingly, we find that HTML parsing costs the most in computation, at 10%. This is likely because: (i) many pages contain a large number of HTML tags, requiring a long time to convert to DOM; (ii) there is a significant amount of overhead when interacting with other components. For example, we find an interval of two milliseconds between reception of the first block of an HTML page and parsing the first HTML tag. JavaScript evaluation is also significant as Web pages embed more and more scripts. In contrast, CSS evaluation and rendering only cost a small fraction of page load time. This suggests that optimizing CSS is unlikely to be effective at reducing page load time.

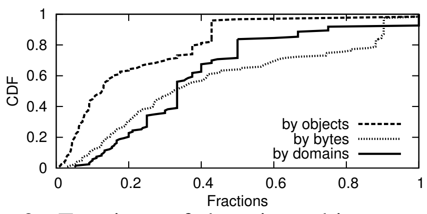
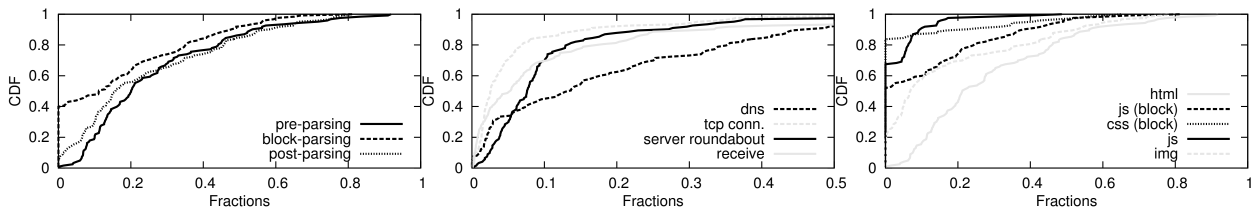


Figure 9: Fractions of domains, objects, and bytes on critical paths.

Network activities often block parsing. First, we break down network time by how it interacts with HTML parsing in Figure 10(a): pre-parsing, block-parsing, and post-parsing. As before, the fractions are with respect to the total page load time. The pre-parsing phase consists of fetching the first chunk of the page during which no content can be rendered. 15% of page load time is spent in this phase. The post-parsing phase refers to loading objects after HTML parsing (e.g., loading a_{10} and a_{12} in Figure 3). Because rendering can be done before or during post parsing, post parsing is less important, though significant. Surprisingly, much of the network delay in the critical path blocks parsing. Recall that network loads can be done in parallel with parsing unless there are dependencies that block parsing, e.g., JavaScript downloads. Parsing-blocking downloads often occur at an early stage of HTML parsing that blocks loading subsequent embedded objects. The result suggests that a large portion of the critical path delay can be reduced if the pages are created carefully to avoid blocked parsing.

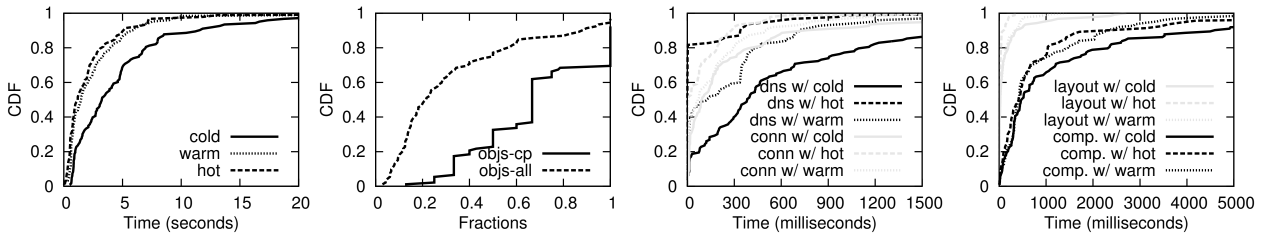
Second, we break down network time by functionality in Figure 10(b): DNS lookup, TCP connection setup, server roundabouts, and receive time. DNS lookup and TCP connection setup are summed up for all the objects that are loaded, if they are on the critical path. Server roundabout refers to the time taken by the server to process the request plus a round trip time; again for every object loaded on the critical path. Finally, the receive time is the total time to receive each object on the critical path. DNS lookup incurs almost 13% of the critical path delay. To exclude bias towards one DNS server, we repeated our experiments with an OpenDNS [21] server, and found that the lookup time was still large. Our results suggest that reducing DNS lookup time alone can reduce page load time significantly. The server roundabout time is 8% of the critical path.

Third, we break down network time by MIME type in Figure 10(c). We find that loading HTML is the largest fraction (20%) and mostly occurs in the pre-parsing phase. Loading images is also a large fraction on critical



(a) By phase. (b) By functionality. (c) By mime type.

Figure 10: A breakdown of fractions of network time on the critical path.



(a) Comparing PLT. (b) Object downloads reduced due to hot load. (c) TCP setup and DNS lookup. (d) Layout and computation time.

Figure 11: Warm and hot loads results. All results are a fraction of total page load time.

paths. Parsing-blocking JavaScript is significant on critical paths while asynchronous JavaScript only contributes a small fraction. Interestingly, we find a small fraction of CSS that blocks JavaScript evaluation and thus blocks HTML parsing. This blocking can be reduced simply by moving the CSS tag after the JavaScript tag. There is almost no non-blocking CSS on critical paths, and therefore we omit it in Figure 10(c).

Last, we look at the page load time for external objects corresponding to Web Analytics and Ads but not CDNs. We find that one fourth of Web pages have downloads of external objects on their critical path. Of those pages, over half spends 15% or more page load time to fetch external objects and one even spends up to 60%.

Most object downloads are non-critical: Figure 9 compares all object downloads and the object downloads only on the critical path. Interestingly, only 30% bytes of content is on critical paths. This suggests that minifying and caching content may not improve page load time, unless they reduce content downloaded along the critical path. We analyze this further in the next section. Note that object downloads off the critical paths are not completely unimportant. Degrading the delay of some activities that are not on the critical path may cause them to become critical.

5.3 Identifying load bottlenecks (with caching)

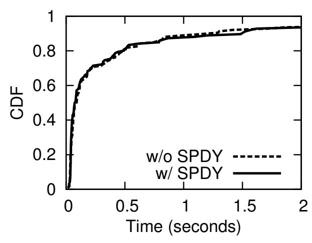
We analyze the effects of caching under two conditions: hot load and warm load. Hot loads occur when a page is loaded immediately after a cold load. Warm loads occur when a page is loaded a small amount of time after a cold load when the immediate cache would have expired. We set a time interval of 12 minutes for warm loads. Both cases are common scenarios, since users often reload pages immediately as well as after a short period of time.

Caching gains are not proportional to saved bytes. Figure 11(a) shows the distributions of page load times under cold, warm, and hot loads. For 50% of the pages, caching decreases page load time by over 40%. However, further analysis shows that 90% of the objects were cached during the experiments. In other words, the decrease in page load time is not proportional to the cached bytes. To understand this further, we analyze the fraction of cached objects that are on and off the critical path, during hot loads. Figure 11(b) shows that while caching reduces 65% of total object loads (marked *objs-all*), it only reduces 20% of object loads on the critical path (marked *objs-cp*). Caching objects that are not on the critical path leads to the disproportional savings.

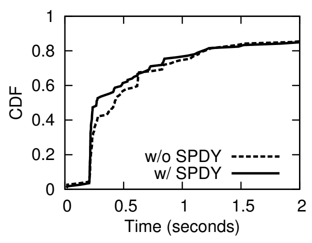
Caching also reduces computation time. Figures 11(c) and 11(d) compare the network times and computation times of hot, warm, and cold loads, on the critical path. As expected, hot and warm loads reduce DNS lookup time and TCP connection setup. Especially during hot loads, DNS lookup time is an insignificant fraction of the page load time in contrast to cold loads. Interestingly, caching not only improves network time, but also computation time. Figure 11(d) shows the time taken by compute and layout activities during page load time. The figure suggests that modern browsers cache intermediate computation steps, further reducing the page load time during hot and warm loads.

5.4 Evaluating proposed techniques

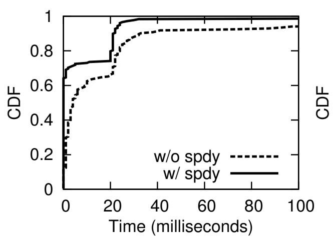
This section evaluates two Web optimization techniques, SPDY [28] and mod_pagespeed [19].



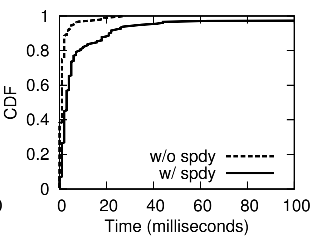
(a) PLT when RTT=20ms.



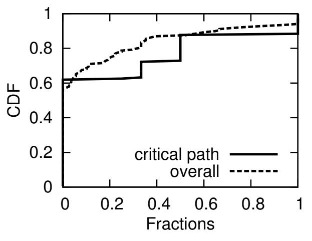
(b) PLT when RTT=200ms.



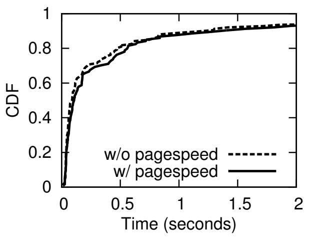
(c) TCP conn. setup time.



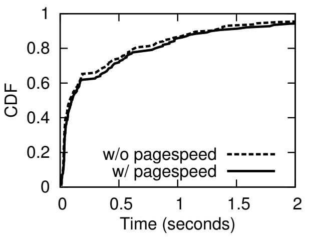
(d) Request sending time.



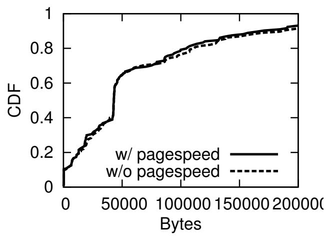
(e) # of TCP connection set ups when using SPDY.



(f) PLT for top web pages.



(g) PLT for random pages.



(h) # bytes on critical path.

Figure 12: SPDY and mod_pagespeed results.

5.4.1 SPDY

SPDY is an application-level protocol in place of HTTP. The key ideas in SPDY are—(i) Multiplexing HTTP transactions into a single TCP connection to reduce TCP connection setup time and to avoid slow start, (ii) prioritizing some object loads over others (e.g., JavaScript over images), and (iii) reducing HTTP header sizes. SPDY also includes two optional techniques—Server Push and Server Hint. Exploiting these additional options require extensive knowledge of Web pages and manual configurations. Therefore, we do not include them here.

We evaluate SPDY with controlled experiments on our own server with 2.4GHz 16 core CPU 16GB memory and Linux kernel 2.6.39. By default, we use a link with a controlled 10Mbps bandwidth, and set the TCP initial window size to 10 (increased from the default 3, as per SPDY recommendation). We use the same set of pages as the real-world experiments and download all embedded objects to our server[1] to avoid domain sharding [20]. We run SPDY version 2 over SSL.

SPDY only improves performance at high RTTs. Figure 12(a) and Figure 12(b) compares the page load time of SPDY versus non-SPDY (i.e., default HTTP) under 20ms and 200ms RTTs, respectively. SPDY provides few benefits to page load time under low RTTs. However, under 200ms RTT, SPDY improves page load time by 5%–40% for 30% of the pages. We conduct additional experiments by varying the TCP initial window size and packet loss, but find that the results are similar to the default setting.

[1]Because we are unable to download objects that are dynamically generated, we respond to these requests with a HTTP 404 error.

SPDY reduces TCP connection setup time but increases request sending time. To understand SPDY performance, we compare SPDY and non-SPDY network activities on the critical path and break down the network activities by functionality; note that SPDY does not affect computation activities. For the 20ms RTT case, Figure 12(c) shows that SPDY significantly reduces TCP connection setup time. However, since SPDY delays sending requests to create a single TCP connection, Figure 12(d) shows that SPDY increases the time taken to send requests. Other network delays such as server roundabout time and total receive time remained similar.

Further, Figure 12(e) shows that although SPDY reduces TCP setup times, the number of TCP connection setups in the critical path is small. Coupled with the increase in request sending time, the total improvement due to SPDY cancels out, resulting in no net improvement in page load time.

The goal of our evaluation is to explain the page load behavior of SPDY using critical path analysis. Improving SPDY's performance and leveraging SPDY's optional techniques are part of future work.

5.4.2 mod_pagespeed

mod_pagespeed [19] is an Apache module that enforces a number of best practices to improve page load performance, by minifying object sizes, merging multiple objects into a single object, and externalizing and/or inlining JavaScripts. We evaluate mod_pagespeed using the setup described in §5.4.1 with a 20ms RTT.

Figure 12(f) compares the page load times with and without mod_pagespeed on top 200 Alexa pages. mod_pagespeed provides little benefits to page load time.

Since the top 200 pages may be optimized, we load 200 random pages from the top one million Alexa Web pages. Figure 12(g) shows that mod_pagespeed helps little even for random pages.

To understand the mod_pagespeed performance, we analyze minification and merging multiple objects. The analysis is based on loading the top 200 Web pages. Figure 12(h) shows that the total number of bytes downloaded on the critical path remains unchanged with and without mod_pagespeed. In other words, minifying object does not reduces the size of the objects loads on the critical path, and therefore does not provide page load benefits. Similarly, our experiments show that merging objects does not reduce the network delay on the critical path, because the object loads are not the bottleneck (not shown here). These results are consistent with our earlier results (Figure 9) that shows that only 30% of the object loads are on the critical path. We were unable to determine how mod_pagespeed decides to inline or externalize JavaScripts, and therefore are unable to conduct experiments on how inlining/externalizing affects page load performance.

5.5 Summarizing results from alternate settings

In addition to our default experiments, we conducted additional experiments: (i) with 2 machines, one with 2GHz dual core 4GB memory, and another with 2.4GHz dual core 2GB memory, (ii) in 2 different locations, one at UMass Amherst with campus Ethernet connectivity, and another in a residential area in Seattle with broadband connectivity, and (iii) using 200 random Web pages chosen from the top 1 million Alexa Web pages. All other parameters remained the same as default.

Below, we summarize the results of our experiments:

- The fraction of computation and network times on the critical path were quantitatively similar in different locations.
- The computation time as a fraction of the total page load time increased when using slower machines. For example, computation time was 40% of the critical path for the 2GHz machine, compared to 35% for the 3GHz machine.
- The 200 random Web pages experienced 2x page load time compared to the top Web pages. Of all the network components, the server roundabout time of random pages (see Figure 10(b)) was 2.3x of that of top pages. However, the total computation time on the critical path was 12% lower compared to the popular pages, because random pages embed less JavaScript.

6 Discussion

In this work, we have demonstrated that WProf can help identify page load bottlenecks and that it can help evaluate evolving techniques that optimize page loads. WProf can be potentially used in several other applications. Here, we briefly discuss two applications: (i) Conducting what-if analysis, and (ii) identifying potential Web optimizations.

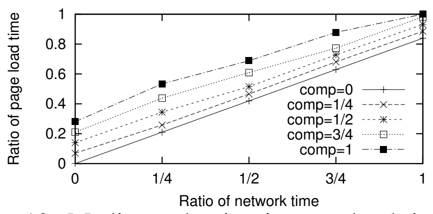


Figure 13: Median reduction in page load times if computation and network speeds are improved.

What-if analysis: As CPUs and networks evolve, the question we ask is—how can page load times benefit from improving the network speed or CPU speed? We use the detailed dependency graphs generated by WProf to conduct this what-if analysis. Figure 13 plots the reduction in page load time for a range of <network speedup, CPU speedup> combinations. When computational time is zeroed but the network time is unchanged, the page load time is reduced by 20%. If the network time is reduced to one fourth, but the computational time is unchanged, 45% of the page load time is reduced. Our results suggest that speeding up both network and computation together is more beneficial than just improving one of them. Note that our what-if analysis is limited as it does not capture all lower-level dependencies and constraints, e.g., TCP window sizes. We view our results here as estimates and leave a more extensive analysis for future work.

Potential optimizations: Our own experience and studies with WProf suggest a few optimization opportunities. We find that synchronous JavaScript significantly affects page load time, not only because of loading and evaluation, but also because of block-parsing (Figure 10(a)). Asynchronous JavaScript or in-lining the scripts can help reduce page load times. To validate this opportunity, we manually transform synchronous JavaScript to `async` on top five pages and find that this helps page load time. However, because asynchronous JavaScript may alter Web pages, we need further research to understand how and when JavaScript transformation affects Web pages. Another opportunity is with respect to prioritizing object loading according to the dependency graph. Prioritization can be either implemented at the application level such as SPDY or reflected using latency-reducing techniques [31]. For example, prioritizing JavaScript loads can decrease the time that HTML parsing is blocked. Recently, SPDY considers applying the dependency graph to prioritize object loading in version 4 [29]. Other opportunities include parallelizing page load activities and more aggressive preloading strategies, both of which require future exploration.

7 Related Work

Profiling page load performance: Google Pagespeed Insight [22] includes a (non open-source) critical path explorer. The explorer presents the critical path for the specific page load instance, but does not extract all dependencies inherent in a page. For example, the explorer does not include dependencies due to resource constraints and eager/late binding. Also, the explorer will likely miss dependencies if objects are cached, if the network speed improves, if a new CDN is introduced, if the DNS cache changes, and many other instances. Therefore, it is difficult to conduct what-if-analysis or to explain the behavior of Web page optimizations using the explorer.

The closest research to WProf is WebProphet [16] that identifies dependencies in page loads by systematically delaying network loads, a technique we borrow in this work. The focus of WebProphet is to uncover dependencies only related to object loading. As a result,, WebProphet does not uncover dependencies with respect to parsing, rendering, and evaluation. WebProphet also predicts page load performance based on its dependency model; we believe that WProf's more complete dependency model can further improve the page load performance prediction.

Tools such as Firebug [11], Developer Tools [8], and more recently HAR [12] provide detailed timings information about object loading. While their information is important to augment WProf profiling, they do not extract dependencies and only focus on the networking aspect of page loads.

Web performance measurements: Related measurement studies focus on either network aspects of Web performance or macro-level Web page characteristics. Ihm and Pai [14], presented a longitudinal study on Web traffic, Ager et al. [2] studied the Web performance with respect to CDNs, and Huang et al. [13] studied the page load performance under a slower cellular network. Butkiewicz et al. [7] conducted a macro-level study, measuring the number of domains, number of objects, JavaScript, CSS, etc. in top pages. Others [26, 15] studied the effect of dynamic content on performance. Instead, WProf identifies internal dependencies of Web pages that let us pinpoint bottlenecks on critical paths.

Improving page load performance: There has been several efforts to improve page load performance by modifying the page, by making objects load faster, or by reducing computational time. At the Web page level, mod_pagespeed [19] and Silo [18] modify the structure and content of the Web page to improve page load. At the networking level, SPDY [28], DNS pre-resolution [9], TCP pre-connect [30], TCP fast open [24], ASAP [35] and other caching techniques [34, 32] reduce the time taken to load objects. At the computation level, inline JavaScript caching [18] and caching partial layouts [18, 17] have been proposed. While these techniques provide improvement for certain aspects of page loads, the total page load performance depends on several inter-related factors. WProf helps understand these factors to guide effective optimization techniques.

8 Conclusion

In this paper, we abstract a model of browser dependencies, and design WProf, a lightweight, in-browser profiler that extracts dependency graphs of any given page. The goal of WProf is to identify bottleneck activities that contribute to the page load time. By extensively loading hundreds of pages and performing critical path analysis on their dependency graphs, we find that computation is a significant factor and makes up as much as 35% of the page load time on the critical path. We also find that synchronous JavaScript evaluation plays a significant role in page load time because it blocks parsing. While caching reduces the size of downloads significantly, it is less effective in reducing page load time because loading does not always affect the critical path. We conducted experiments over SPDY and mod_pagespeed. While the effects of SPDY and mod_pagespeed vary over pages, surprisingly, we find that they help very little on average. In the future, we plan to extend our dependency graphs with more lower-level dependencies (e.g., in servers) to understand how page loads would be affected by these dependencies.

Acknowledgements

We thank our shepherd, Rebecca Isaacs, and the anonymous reviewers for their feedback. We thank Xiaozheng Tie for setting up machines at UMass.

References

[1] Above the fold time. `http://www.webperformancetoday.com/`.

[2] B. Ager, W. Muhlbauer, G. Smaragdakis, and S. Uhlig. Web Content Cartography. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2011.*

[3] Alexa - The Web Information Company. `http://www.alexa.com/topsites/countries/US`.

[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] 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.*

[6] National Broadband Map. `http://www.broadbandmap.gov/`.

[7] 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*.

[8] Chrome Developer Tools. `https://developers.google.com/chrome-developer-tools/docs/overview`.

[9] DNS pre-resolution. `http://blogs.msdn.com/b/ie/archive/2011/03/17/internet-explorer-9-network-performance-improvements.aspx`.

[10] Dummynet. `http://info.iet.unipi.it/~luigi/dummynet/`.

[11] Firebug. `http://getfirebug.com/`.

[12] HTTP Archive. `http://httparchive.org/`.

[13] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proc. of the international conference on Mobile systems, applications, and services (Mobisys), 2010*.

[14] S. Ihm and V. S. Pai. Towards understanding modern web traffic. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2011*.

[15] E. Kiciman and B. Livishits. AjaxScope: A Platform for Remotely Monitoring the Client-side Behavior of Web 2.0 Applications. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP), 2007*.

[16] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. WebProphet: automating performance prediction for web services. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI), 2010*.

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

[18] J. Mickens. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads. In *Proc. of USENIX conference on Web Application Development (WebApps), 2010*.

[19] mod_pagespeed. `http://www.modpagespeed.com/`.

[20] Not as SPDY as you thought. `http://www.guypo.com/technical/not-as-spdy-as-you-thought/`.

[21] Open DNS. `http://www.opendns.com/`.

[22] Google Pagespeed Insights. `https://developers.google.com/speed/pagespeed/insights`.

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

[24] 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*.

[25] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989.

[26] F. Schneider, S. Agarwal, T. Alpcan, and A. Feldmann. The New Web: Characterizing AJAX Traffic. In *Proc. of Passive and Active Measurement Conference (PAM), 2008*.

[27] Selenium. `http://seleniumhq.org/`.

[28] SPDY. `http://dev.chromium.org/spdy`.

[29] Proposal for Stream Dependencies in SPDY. `https://docs.google.com/document/d/1pNj2op5Y4r1AdnsG8bapS79b11iWDCStjCNHo3AWD0g/edit`.

[30] TCP pre-connect. `http://www.igvita.com/2012/06/04/chrome-networking-dns-prefetch-and-tcp-preconnect/`.

[31] A. Vulimiri, O. Michel, P. B. Godfrey, and S. Shenker. More is Less: Reducing Latency via Redundancy. In *Proc. of the ACM Workshop on Hot Topics in Networks (HotNets-XI), 2012*.

[32] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How far can client-only solutions go for mobile browser speed? In *Proc. of the international conference on World Wide Web (WWW), 2012*.

[33] The Webkit Open Source Project. `http://www.webkit.org/`.

[34] K. Zhang, L. Wang, A. Pan, and B. B. Zhu. Smart caching for web browsers. In *Proc. of the international conference on World Wide Web (WWW), 2010*.

[35] W. Zhou, Q. Li, M. Caesar, and B. Godfrey. ASAP: A Low-Latency Transport Layer. In *Proc. of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT), 2011*.