IMPROVING WEB APPLICATIONS WITH FINE-GRAINED DATA Flows

by

RAVI ARUN NETRAVALI

Master of Science, Massachusetts Institute of Technology (2014)
Bachelor of Science, Columbia University (2012)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2018

Copyright 2018 Ravi Arun Netravali.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created. This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, please visit http://creativecommons.org/licenses/by/4.0/.

Signature redacted

Author

Department of Electrical Engineering and Computer Science
August 30, 2018

Signature redacted

Certified by

Hari Balakrishnan
Fujitsu Chair Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Signature redacted

Certified by

James Mickens
Associate Professor of Computer Science, Harvard University
Thesis Supervisor

Signature redacted

Accepted by

Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students
IMPROVING WEB APPLICATIONS WITH FINE-GRAINED DATA FLOWS

by

RAVI ARUN NETRAVALI

Submitted to the Department of Electrical Engineering and Computer Science on August 30, 2018, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

Web applications have significantly increased in complexity over the past several decades to support the wide range of services and performance requirements that users have come to expect. On the client-side, browsers are multi-process systems that can handle numerous content formats, rich interactivity, and asynchronous patterns. On the server-side, applications are distributed across many machines and employ multi-tier architectures to implement application logic, caching, and persistent storage. As a result, web applications have become complex distributed systems that are difficult to understand, debug, and optimize.

This dissertation presents fine-grained data flows as a new mechanism for understanding and optimizing complex web applications. Fine-grained data flows comprise the set of low-level reads and writes made to distributed application state during execution. We explain how fine-grained data flows can be tracked efficiently in production systems. We then present four concrete systems that illustrate how fine-grained data flows enable powerful performance optimizations and debugging primitives.

Polaris dynamically reorders client HTTP requests during a page load to maximally overlap network round trips without violating data flow dependencies, reducing page load times by 34% (1.3 seconds). Prophecy uses data flow logs to create a snapshot of a mobile page’s post-load state, which clients can process to elide intermediate computations, reducing bandwidth consumption by 21%, energy usage by 36%, and load times by 53% (2.8 seconds). Vesper is the first system to accurately and automatically measure page time-to-interactivity, without using heuristics or developer annotations. Vesper-guided optimizations improve time-to-interactivity by 32%, generating more satisfaction in user studies than systems targeting past metrics. Cascade is the first replay debugger to support distributed, fine-grained provenance tracking. Cascade also enables speculative bug fix analysis, i.e., replaying a program to a point, changing program state, and resuming replay, using data flow tracking and causal analysis to evaluate potential bug fixes.

Thesis Supervisor: Hari Balakrishnan
Title: Fujitsu Chair Professor of Electrical Engineering and Computer Science

Thesis Supervisor: James Mickens
Title: Associate Professor of Computer Science, Harvard University
To my father, Dr. Arun Netravali
Contents

Previously Published Material .................................................. 9
Acknowledgments ...................................................................... 11

1 Introduction ........................................................................... 13
  1.1 Motivation ......................................................................... 13
  1.2 Primary Contributions .................................................... 14
  1.3 Key Takeaways .................................................................. 18

2 Background ............................................................................ 19
  2.1 Basic Page Load Process .................................................. 19
  2.2 Loading More Complicated Pages ...................................... 19

3 Scout: Tracking Fine-grained Data Flows in Web Page Loads ..... 23
  3.1 Tracking JavaScript heap dependencies ............................. 23
  3.2 Tracking DOM dependencies ........................................... 24
  3.3 Missing Dependencies ...................................................... 25
  3.4 Implementation .............................................................. 25
  3.5 Performance .................................................................. 26

4 Polaris: Faster Page Loads Using Fine-grained Dependency Track-
ing ....................................................................................... 27
  4.1 Overview .......................................................................... 27
  4.2 The Pitfalls of Lexical Dependencies ................................. 29
  4.3 Dependency Tracking ....................................................... 30
    4.3.1 Dependency Types .................................................. 30
    4.3.2 Dependency Graphs: Scout vs. Prior Tools .................. 31
    4.3.3 Dependency Graphs: Structural Differences for Real Pages 33
  4.4 Polaris: Dynamic Client-side Scheduling ......................... 34
  4.5 Page Load Time Improvements with Polaris ....................... 37
    4.5.1 Methodology .......................................................... 37
    4.5.2 Reducing Page Load Times ....................................... 37
    4.5.3 Browser Caching .................................................... 40
    4.5.4 SPDY .................................................................. 41
  4.6 Polaris: Related Work ....................................................... 43
## Vesper: Measuring Time-to-Interactivity for Web Pages

### 5.1 Overview

### 5.2 Existing Web Performance Metrics

### 5.3 Ready Index

### 5.4 Vesper: Design

#### 5.4.1 Phase 1

#### 5.4.2 Phase 2

##### 5.4.2.1 Measuring Functionality Progress

##### 5.4.2.2 Measuring Visibility Progress

### 5.4.3 Discussion

### 5.5 Evaluating Ready Index with Vesper

#### 5.5.1 Methodology

#### 5.5.2 Cross-metric Comparisons

#### 5.5.3 Mobile Page Loads

#### 5.5.4 Browser Caching

#### 5.5.5 Cross-Browser Comparisons

#### 5.5.6 Case Studies

### 5.6 Optimizing for Interactivity

## Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs

### 6.1 Overview

### 6.2 Prophecy: Design

#### 6.2.1 Generating a Prophecy Frame

#### 6.2.2 Loading a Prophecy Frame

#### 6.2.3 Caching, Personalization, and Cookies

#### 6.2.4 Online versus Offline Transformation

#### 6.2.5 Optimizing for Different Definitions of Load Time

#### 6.2.6 Privacy

#### 6.2.7 Discussion

### 6.3 Implementation

### 6.4 Evaluating Prophecy

#### 6.4.1 Reducing PLT

#### 6.4.2 Reducing Bandwidth

#### 6.4.3 Energy savings

#### 6.4.4 Reductions in SI and RI

#### 6.4.5 The Sources of Prophecy's Benefits

#### 6.4.6 Server-side overhead

#### 6.4.7 Warm Browser Caches

#### 6.4.8 Diff Characteristics

#### 6.4.9 Additional Sites

### 6.5 Prophecy: Related Work

#### 6.5.1 Prepack

#### 6.5.2 Shandian

#### 6.5.3 Split browsers
7 Cascade: Using Data Flow Analysis For Speculative “What If?” Debugging of Web Applications 97
7.1 Overview ............................................ 97
7.2 Background ............................................ 99
  7.2.1 Traditional debuggers ............................. 100
  7.2.2 Deterministic replay .............................. 100
  7.2.3 Program slicing .................................... 100
  7.2.4 Data Provenance ................................... 101
  7.2.5 Speculative edit-and-continue .................... 101
7.3 Cascade: Design ...................................... 102
  7.3.1 Overview of JavaScript ........................... 102
  7.3.2 Analyzing Value Provenance ....................... 104
  7.3.3 Speculative Edit-and-Continue Debugging .......... 105
  7.3.4 Debugging Across the Wide Area ................... 109
  7.3.5 Limitations ....................................... 112
7.4 Implementation ........................................ 113
7.5 Evaluation ............................................. 113
  7.5.1 The Tractability of Data Flow Analysis .......... 114
  7.5.2 Debugging Case Studies .......................... 115
    7.5.2.1 Mailpile ..................................... 115
    7.5.2.2 EtherCalc .................................... 118
    7.5.2.3 An nba.com Article ......................... 120
    7.5.2.4 Bleacher Report .............................. 120
    7.5.2.5 Wide-area debugging ......................... 121
    7.5.2.6 CPU-bound code .............................. 123
  7.5.3 Server-side Overheads ............................ 123
  7.5.4 Replay Subtleties ................................ 124
  7.5.5 Pruning Data Flow Graphs ....................... 125
8 Conclusion ............................................. 127
  8.1 Future Work ........................................ 127
  8.2 Takeaways and Broader Impact ....................... 128
Previously Published Material


Acknowledgments

I was very fortunate to work with two fantastic advisors, Hari Balakrishnan and James Mickens. Hari and James provided infinite amounts of support, both technical and career-wise, at every stage of my PhD. They each had a unique and profound influence on who I am as a researcher. Hari taught me the importance of emphasizing the key takeaways of a research project at every step, including during project formulation, system design, paper writing, and presentations. James has been instrumental to my technical growth as a systems researcher, and has also taught me how to present research ideas and technical details in coherent (and effectively unmistakable) ways. Most importantly, both Hari and James have taught me the importance of continual patience and optimism in research settings.

I am especially grateful to Mohammad Alizadeh, not only for serving on my thesis committee, but also for being a wonderful collaborator. I have benefitted immensely from working with Mohammad. In particular, Mohammad has really taught me the value of digging into a challenge, either through mathematical reasoning or highly focused experiments, in order to understand it without any doubts.

I am especially grateful to my two (former) labmates, Keith Weinstein and Anirudh Sivaraman, for their willingness and patience in teaching me how to carefully build software systems. In addition, Anirudh has always made himself available to me, whether it be for a casual chat, technical feedback, or career advice.

I thank my labmates and members of the ninth floor, both past and present–Katrina LaCurts, Raluca Ada Popa, Shuo Deng, Lenin Ravindranath, Peter Iannucci, Tiffany Chen, Shuo Deng, Amy Ousterhout, Jonathan Perry, Vikram Nathan, Swarun Kumar, Vibhav Sivaraman, Hongzi Mao, Ravichandra Addanki, Mehrdad Khani, Songtao He, Venkat Arun, Prateesh Goyal, Frank Cangialosi, Akshay Narayan, Srinivas Narayana, Ameesh Goyal, Eugene Wu, Alvin Cheung, Frank Wang, Deepak Vasisht, Guha Balakrishnan, Jen Gong, Amy Zhao, and Hariharan Rahul—for always being available for discussions and for making the Stata Center a pleasant work environment. I will definitely miss the lab outings and the table tennis games.

I would like to thank Sheila Marian for all of her help through my PhD, including booking travel, filing reimbursements, scheduling meetings, and providing general life advice. I will certainly miss our early morning conversations.

Marcella Lusardi has been a constant source of support and happiness both academically and outside of work. I am truly lucky that we were able to complete our PhDs together.

Finally, I am forever grateful to my family for their unconditional support and
guidance throughout my academic career along with their never-fading belief in me:

My sister, Ilka Netravali, for her ability to talk to me as an academic, sister, and friend, and for truly being an inspiration to me.

My mother, Chitra Netravali, for her unbounded desire to help me focus on my academics and for her unwavering love.

Lastly, my father, Arun Netravali, for being a terrific bouncing board for research ideas and for always teaching me the importance of focused perseverance in research and in life.
Chapter 1

Introduction

1.1 Motivation

Web applications are a primary mechanism for connecting users with digital content. These applications support numerous services that are integral to today's society including banking, e-commerce, and social media. As a result, users expect these applications to function efficiently and correctly. For example, numerous studies have found that application delays of just milliseconds can result in user dissatisfaction and early abandonment of pages, costing content providers millions of dollars in lost revenue [19, 23, 50]. Search engines have also begun to assign lower search rankings to pages that load slowly or include bugs that affect functionality [55].

Unfortunately, modern web services are extremely complex. In the 1990s, a web page was a simple collection of static text and images. Today, a web page is an interactive multimedia program that executes on a rich client-side platform, and is generated by multiple server-side components. As a result, developers have increasingly struggled to understand, debug, and optimize modern web applications.

When a user loads a web page, her browser issues an HTTP request for the page's HTML, which is sent to the application's backend. The server-side of web applications are now commonly distributed across many machines and employ multi-tier architectures to respond to user requests. Typically, a front-end or application server is responsible for fielding user requests, and generating responses using logic central to the application. To create responses, these servers pull and process data from multiple storage layers, including persistent storage systems like databases, and intermediate caching layers. Rendering even a single web page often requires hundreds or even thousands of storage-layer accesses [135]. After generating the requested HTML document, the application server returns the constructed object to the client's browser.

When the client browser receives the HTML, it begins to parse it tag by tag, discovering external objects to load. Modern browsers contain multiple subsystems that are responsible for loading web pages and supporting post-load user interactions with those pages. These components cooperate to handle numerous content formats (e.g., HTML, images) and asynchronous IO patterns from the disk, user-input devices,
and the network. Additionally, a significant amount of the client-side complexity in web browsing sessions results from the use of JavaScript code that enables rich interactivity. JavaScript code can interact with other page state, in particular the DOM tree (i.e., the browser’s internal representation of the page’s HTML tree), to affect the visual appearance of a page. These interactions, coupled with JavaScript’s asynchronous and event-driven execution patterns, are used to support features such as dropdown menus, animations, and search features (e.g., autocompletion).

These changes have collectively resulted in a more intricate page load process, as well as more complex post-load interactions. Figure 1-1 illustrates this increased complexity, as compared to page loads from the 1990s. Indeed, the loading of a page, or even a single user interaction (e.g., the clicking of a link) on a loaded page, often triggers many distributed, asynchronous events. As a result, understanding, debugging and optimizing web page loads has become far more difficult.

1.2 Primary Contributions

The main contribution of this thesis is the introduction of fine-grained data flows as a new mechanism for understanding and optimizing web applications. The goal of this mechanism is to address the growing complexity in web applications by providing clarity into the complex execution of these systems. Unlike prior taint-tracking [40] and provenance systems [39, 143] which identify coarse-grained influences on program state, our work tracks fine-grained data history. For instance, instead of logging that variable \( x \) has somehow been derived from \( y \), our systems efficiently capture the exact list of code lines that accessed \( x \). In other words, fine-grained data flows comprise the set of low-level reads and writes that are made to application state (on both the client-side and server-side) during the execution of the application. Figure 1-2 provides a concrete example that compares our work to prior approaches.

At first glance, tracking fine-grained data flows throughout an entire distributed web application might seem prohibitively expensive. However, this thesis describes why web services are actually well-suited to efficiently collect and manipulate fine-grained data flows. The reasons are twofold.

- First, many components in web services are implemented in managed languages such as JavaScript and Python. For these components, we can track data flows involving high-level managed state (e.g., the JavaScript heap), instead of low-level state like registers and memory locations. This significantly reduces the overall number of reads and writes that must be tracked, lowering overheads.

- Second, though many application components use managed languages, web services can contain subsystems that are implemented in native languages like C++ (e.g., Redis). However, these native components are often single-threaded and event-driven from the application’s perspective. Thus, to track application data flows, we can treat the internal state of native components as black-box state, and only log interactions with that black-box state form the event-loop interface.
(a) Summary of the page load process from the nascency of the web (in the early 1990s). Browsers fetch static, preconstructed HTML documents from individual web servers, display the text in those documents on the screen, and load additional static objects (e.g., images) that those documents reference.

(b) Summary of the page load process today. Browsers still fetch HTML documents from servers, but those documents are often dynamically constructed by backends that include numerous subsystems like caches and databases. Browsers also must support pages that include dynamic content (e.g., JavaScript), in addition to static content (e.g., images), and complex user interactions (e.g., search with autocompletion).

Figure 1-1: Comparing the page load process of the 1990s (a) to that of today (b).
Figure 1-2: Comparing the fine-grained data flow information captured by Scout, and the coarse-grained “influence” information logged by prior tracking approaches. (a) shows the source code for a web page with two JavaScript files. (b) Fine-grained data flow information logged by Scout. Scout is able to log all of the reads and writes to page state made during a page load, as well as the fine-grained influence (and location of that influence) that each write has on another. (c) shows The information logged by prior tracking approaches (i.e., taint tracking and provenance systems). These tools only log coarse-grained “influence”, or the set of variables that affect the value of each variable.

Tracking data flows at the granularity of individual pieces of program state, like client-side JavaScript and DOM variables, and server-side key/value storage pairs, yields significant benefits. In particular, fine-grained data flows explain the causality between application tasks and state changes. Understanding those causal relationships enables more powerful performance optimizations and richer debugging tools.

Scout: automatically tracking fine-grained data flows in web pages. To capture the precise data flows in a web page load, we built Scout [128]. Scout has three main goals: to track all of the data flows in a page load, to be efficient in tracking those flows, and to work on unmodified browsers. We show in Chapter 3 that Scout is able to achieve all three of these goals. To use Scout, developers can simply pass the JavaScript and HTML code for their pages into an automatic compiler, which will return an instrumented page that, when loaded, will output data flow information. Specifically, the instrumented page will log all fine-grained reads and writes to both the JavaScript heap and the DOM tree, which are the two primary types of client-side state associated with web page loads (Chapter 2). Performing this tracking imposes only a 3%-5% overhead on page load times, making Scout usable in production settings. We have developed multiple systems (briefly summarized below) that leverage Scout to improve different aspects of web applications.

Polaris: faster page loads. A web page dependency graph specifies the load order for a page’s objects (e.g., JavaScript files). Unfortunately, browsers are unaware of the true data flow dependencies between those objects, and instead estimate dependencies
from HTML structure. As a result, browsers must use conservative loading algorithms to avoid violating hidden dependencies. For example, upon parsing the first of two HTML `<script>` tags, a browser must halt parsing, and synchronously fetch and execute the referenced JavaScript object since it may create state read by the second object. Synchronous loading ensures correctness, but is often too cautious; if objects do not modify mutually observable state, the browser should be free to load them in any order. Polaris \[128\] allows unmodified browsers to exploit the data dependencies discovered by Scout and avoid conservative load assumptions. When a client requests a page, the server responds with a Polaris scheduling stub rather than the page’s HTML. The stub includes the original HTML, a Scout dependency graph, and a small JavaScript library that determines which objects to load next using the browser’s limited resources. By prioritizing the deepest unresolved paths in the Scout graph, Polaris reduces page load times by 34\%-59\%.

**Prophecy: optimizing the mobile web.** Mobile page loads are important to optimize along multiple axes: bandwidth usage, energy consumption, and page load time. Prophecy \[129\] is a mobile web accelerator that improves all three aspects. With Prophecy, a server analyzes Scout’s data flow logs to determine the final write to each JavaScript variable and DOM object that is live at the end of a page load. When a browser requests a page, the server returns a write log (with one entry per live value) instead of the traditional HTML, CSS, and JavaScript. By applying this write log, the browser elides slow, energy-intensive computations involving JavaScript execution and rendering. Further, Prophecy’s write logs are smaller than a page’s original content, and can be fetched in fewer RTTs. Prophecy reduces load times by 53\%, energy usage by 36\%, and bandwidth cost by 21\%.

**Vesper: measuring web page time-to-interactivity.** Web pages have sophisticated GUIs which support rich interactivity like search boxes and animations. Unfortunately, existing page load metrics do not precisely capture interactivity. The traditional “page load time” metric records when all of a page’s objects have been loaded, but only some may be needed for a user to properly interact with the initial browser viewport. Newer metrics like Speed Index measure the initial viewport’s rendering time, but ignore the loading of JavaScript state that supports interaction. To address these limitations, we created a new metric called Ready Index (RI) \[131\], which directly captures page interactivity. RI defines a page to be loaded when (1) the initial viewport has fully rendered, and (2) the state supporting interactivity for each viewport element has loaded. Existing systems are not equipped to measure RI since developers do not explicitly annotate the state that supports interactivity. Thus, we built Vesper, which automatically identifies a page’s interactive state and measures RI by firing the page’s event handlers, analyzing the generated data flows, and evaluating rendering progress. Using Vesper, we found that prior metrics inaccurately measure time-to-interactivity by 39\%. Further, Vesper-guided optimizations improve RI by 32\%, resulting in more satisfaction in user studies than pages which optimize for prior metrics.
Cascad: insightful debugging for distributed applications. Debugging large-scale distributed web services is difficult. Traditional debugging primitives like breakpoints only crudely capture the temporal flows necessary to reconstruct a buggy value’s provenance across multiple asynchronous codepaths. To solve this problem, we developed Cascade, the first distributed replay debugger that makes fine-grained data flows explicit and queryable. Cascade provides three novel features to enable a fundamentally more powerful debugging experience. First, Cascade tracks precise value provenance, or the exact set of reads and writes that produce each program value, enabling queries like “How did a mouse click affect variable z?” Second, Cascade enables speculative bug fix analysis. Developers can replay a program to a certain point, change program state, and resume replay, using data flow tracking and Cascade’s causal analysis to determine the impact of hypothesized bug fixes. Third, Cascade supports wide-area debugging; by tracking data flows across clients and servers, Cascade can reason about wide-area causality and support speculative replay of distributed applications.

1.3 Key Takeaways

There are two main takeaways from this dissertation. First, capturing fine-grained data flow information at the managed runtime and event-loop level is efficient enough to run in production. The reason is that these abstractions for tracking granularity sufficiently reduce the number of data flows to consider, while ensuring complete tracking coverage for application state. Second, the low-level system information provided by fine-grained data flows enables more aggressive optimizations and debugging tools. The reason is that knowledge of data flows highlights the exact causal relationships between system tasks and state, eliminating uncertainty and the need for analyses driven by heuristics. We present four different practical systems that leverage fine-grained data flows to analyze and optimize distributed web applications. Moving forward, we believe that the capture and analysis of fine-grained data flows will have even broader implications for web applications. For example, knowledge of data flows can influence how security policies are defined and enforced, and how applications should be partitioned across distributed components.
Chapter 2

Background

As background for the rest of this dissertation, this chapter provides an overview of the page load process, with a primary focus on client-side intricacies. Subsequent chapters will discuss more background and related work regarding critical tasks and goals for the web: performance optimization (Chapters 4 and 6), performance characterization (Chapter 5), and debugging (Chapter 7).

2.1 Basic Page Load Process

In a conventional page load, the browser first downloads the page’s top-level HTML. For now, we assume that the HTML does not reference any JavaScript, CSS, or multimedia files. As the browser parses the HTML tags, it generates a data structure called the Document Object Model (DOM) tree. Each HTML tag has a corresponding node in the DOM tree; the overall structure of the DOM tree mirrors the hierarchical tag structure of the HTML. Once the HTML parse is finished and the DOM tree is complete, the browser constructs a render tree, which only contains the DOM nodes to be displayed. For example, a <text> node is renderable, but a <head> node is not. Each node in the render tree is tagged with visual attributes like background color, but render nodes do not possess on-screen positions or sizes. To calculate those geometric properties, the browser traverses the render tree and produces a layout tree, which determines the spatial location of all renderable tags. Finally, the browser traverses the layout tree and updates (or “paints”) the screen. Modern browsers try to pipeline the construction of the various trees, in order to progressively display a page.

2.2 Loading More Complicated Pages

JavaScript: Using <script> tags, HTML can include JavaScript code. A script tag blocks the HTML parser, halting the construction of the DOM tree and the derivative data structures. Script tags block HTML parsing because JavaScript can use interfaces like document.write() to dynamically change the HTML after a <script> tag; thus, when the HTML parser encounters a <script> tag, the parser cannot know what the post-<script> HTML will look like until the JavaScript code in the tag has executed.
As a result, script tags inject synchronous JavaScript execution delays into a page load. If a script tag does not contain inline source code, the browser also incurs network latencies to download the JavaScript code. While inlining a tag’s content eliminates an HTTP-level RTT to fetch the associated data, it also prevents a browser from caching the object. The reason is that the browser cache interposes on explicit HTTP requests and responses, using the URL and the headers in the HTTP request as the key for storing and retrieving the associated object.

To reduce these synchronous latencies, modern browsers allow developers to mark a `<script>` tag with the `async` or `defer` attribute. An async script is downloaded in parallel with the HTML parse, but once it is downloaded, it will execute synchronously, in a parse-blocking manner. A defer script is only downloaded and executed once HTML parsing is complete.

By default, a `<script>` tag is neither `async` nor `defer`. For example, such scripts represent 98.3% of all JavaScript files in the Alexa top 200 pages in the US [4]. When the HTML parser in a modern browser encounters a synchronous `<script>` tag, the parser enters *speculation mode*. The parser initiates the download of the JavaScript file, and as that download completes in the background, the parser continues to process the HTML after the script tag, fetching the associated objects and updating a speculative version of the DOM. The browser discards the speculative DOM if it is invalidated by the execution of the upstream JavaScript code.

**CSS:** A page may use CSS to define the visual presentation of HTML tags. The browser represents those stylings using a CSS Object Model (CSSOM) tree. The root of the CSSOM tree contains the general styling rules that apply to all HTML tags. Different paths down the tree apply additional rules to particular types of nodes, resulting in the “cascading” aspect of cascading style sheets.

A browser defines a default set of CSS rules known as the user agent styles. A web page provides additional rules, either by incorporating CSS `<link>` tags or by inlining CSS information using `<style>` tags. To create the render tree, the browser uses the DOM tree to enumerate a page’s visible HTML tags, and the CSSOM tree to determine what those visible tags should look like.

CSS tags do not block HTML parsing, but they do block rendering, layout, and painting. The reason is that unstyled pages are visually unattractive and potentially non-interactive, so style computations should be handled promptly. Best practices encourage developers to place CSS tags at the top of pages, to ensure that the CSSOM tree is built quickly. Since JavaScript code can query the CSS properties of DOM nodes, the browser halts JavaScript execution while CSS is being processed; doing so avoids race conditions on CSS state.

**Images:** Browsers do not load `<img>` tags synchronously. Thus, a page can be completely rendered and laid out (and partially painted) even if there are outstanding image requests. However, browsers are still motivated to load images as quickly as possible, since users do not like pages with missing images.
Table 2.1: Per-page object distributions for the top 200 sites in the US.

<table>
<thead>
<tr>
<th>Object Type</th>
<th>Median</th>
<th>95th Percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML</td>
<td>11.8%</td>
<td>26.2%</td>
</tr>
<tr>
<td>JavaScript</td>
<td>22.9%</td>
<td>43.0%</td>
</tr>
<tr>
<td>CSS</td>
<td>3.7%</td>
<td>16.7%</td>
</tr>
<tr>
<td>Images</td>
<td>44.9%</td>
<td>77.4%</td>
</tr>
<tr>
<td>Fonts</td>
<td>0.0%</td>
<td>7.8%</td>
</tr>
<tr>
<td>JSON</td>
<td>0.4%</td>
<td>5.0%</td>
</tr>
<tr>
<td>Other</td>
<td>0.0%</td>
<td>7.8%</td>
</tr>
</tbody>
</table>

Other media files: Besides images, a page can include various types of video and audio files. However, in this dissertation, we focus on the loading of HTML, JavaScript, CSS, and image files, which are the most common types of web objects (see Table 2.1). Optimizing the loading process for rich multimedia files requires complex, media-specific techniques (e.g., [51, 80, 95]).

Frames: A single web page consists of one or more frames. Each frame is defined by an associated HTML file, which is added to a page with an `<iframe>` tag. Each frame can contain any number of objects, and objects can be of any of the aforementioned file types. Additionally, each frame’s JavaScript heap and DOM tree are isolated from those of other frames [119].

Page State: Objects in a web page interact with each other via two kinds of state. The JavaScript heap contains the code and the data that are managed by the JavaScript runtime. There are two kinds of JavaScript objects: application-defined and native. Application-defined objects are composed of pure JavaScript-level state. In contrast, native objects are JavaScript wrappers around native code functionality defined by the JavaScript engine, the HTML renderer, or the network engine. Examples of native objects include RegExps (which implement regular expressions) and XMLHttpRequests (which expose HTTP network connections).

DOM nodes [123] are another important type of native object. As the HTML parser scans a frame’s HTML, the parser builds a native code representation of the HTML tree. There is a 1-1 correspondence between HTML tags and DOM nodes.

The JavaScript runtime interacts with the rest of the browser through the DOM interface. The DOM interface reflects internal, C++ browser state into the JavaScript runtime. However, the reflected JavaScript objects do not directly expose the rendering and layout trees. Instead, the DOM interface exposes an extended version of the DOM tree in which each node also has properties for style information and physical geometry. By reading and writing this DOM state, JavaScript code interacts with the browser’s rendering, layout, and painting mechanisms. DOM changes often require the browser to recalculate the layout and styles of DOM nodes, and then repaint the DOM nodes. These calculations are computationally expensive (and therefore energy-intensive as well) [79, 88, 190, 25]. The DOM interface also allows JavaScript
Figure 2-1: A simple frame’s JavaScript heap and DOM tree. The JavaScript heap is red; the DOM tree is blue.

code to dynamically fetch new web objects, either indirectly, by inserting new HTML tags into the DOM tree, or directly, using XMLHttpRequests or WebSockets.

As shown in Figure 2-1, native code objects like the DOM tree can reference application-defined objects, and vice versa. For example, a DOM node becomes interactive via JavaScript calls like DOMnode.addEventListener(eventType, cb), where cb is an application-defined JavaScript function which the browser will invoke upon the reception of an event.

**Browser Storage:** Browsers define two main types of client-side storage. A cookie [14] is a small, per-origin file that can store up to 4 KB of data. When a browser issues an HTTP request to origin X, the browser includes any cookie that the browser stores on behalf of X. When the server receives the cookie, the server can generate personalized content for the HTTP response. The server can also use special HTTP response headers to modify the client-side cookie. Cookies are often used to hold personal user information, so cookie sharing has privacy implications.

DOM storage is the other primary type of client-side storage. DOM storage is also siloed per origin, but allows each origin to store MBs of key/value data. DOM storage can only be read and written by JavaScript code, and is separate from the browser cache (which is automatically managed by the browser itself).
Chapter 3

Scout: Tracking Fine-grained Data Flows in Web Page Loads

To capture the fine-grained dependencies in a real web page, we built Scout. Scout rewrites each JavaScript and HTML file in the page, adding instrumentation to log fine-grained data flows across the JavaScript heap and the DOM. Scout then loads the instrumented page in a regular browser. As the page loads, it emits a dependency log listing all reads and writes made to page state. Scout works with commodity browsers and imposes low overheads on the page load process, making it usable in production settings.

3.1 Tracking JavaScript heap dependencies

To track dependencies in which both actors are JavaScript code, Scout leverages JavaScript proxy objects [114]. A proxy is a transparent wrapper for an underlying object, allowing custom event handlers to fire whenever external code tries to read or write the properties of the underlying object.

In JavaScript, the global namespace is explicitly nameable via the window object; for example, the global variable x is also reachable via the name window.x. Scout’s JavaScript rewriter transforms undorned global names like x to fully qualified names like window.x. Also, for each JavaScript file (whether inline or externally fetched), Scout wraps the file’s code in a closure which defines a local alias for the window variable. The aliasing closures, in combination with rewritten code using fully qualified global names, forces all accesses to the global namespace to go through Scout’s window proxy. Using that proxy, Scout logs all reads and writes to global variables.

Scout’s window proxy also performs recursive proxying for non-primitive global values. For example, reading a global object variable window.x returns a logging proxy for that object. In turn, reading a non-primitive value y on that proxy would return a proxy for y. By using recursive proxying and wrapping calls to new in proxy generation code, Scout can log any JavaScript-issued read or write to JavaScript state. Each read or write target is logged using a fully qualified path to the window object, e.g., window.x.y.z. Log entries also record the JavaScript file that issued the operation.
Scout’s proxy generation code tags each underlying object with a unique, non-enumerable integer id. The proxy code also stores a mapping between ids and the corresponding proxies. When a proxy for a particular object is requested, Scout checks whether the object already has an id. If it does, Scout returns the preexisting proxy for that object, creating proxy-level reference equalities which mirror those of the underlying objects.

Some objects lack a fully-qualified path to window. For example, a function may allocate a heap object and return that object to another function, such that neither function assigns the object to a variable that is recursively reachable from window. In these cases, Scout logs the identity of the object using the unique object id.

### 3.2 Tracking DOM dependencies

JavaScript code interacts with the DOM tree through the `window.document` object. For example, to find the DOM node with a particular id, JavaScript calls `document.getElementById(id)`. The DOM nodes that are returned by `document` provide additional interfaces for adding and removing DOM nodes, as well as changing the CSS properties of those nodes.

To track dependencies involving JavaScript code and DOM state, Scout’s recursive proxy for `window.document` automatically creates proxies for all DOM nodes that are returned to JavaScript code. For example, the `DOMNode` returned by `document.getElementById(id)` is wrapped in a proxy which logs reads and writes to the object via interfaces like `DOMNode.height`.

Developers do not assign ids to most DOM nodes. Thus, Scout’s logs identify DOM nodes by their paths in the DOM tree. For example, the DOM path `<1,5,2>` represents the DOM node that is discovered by examining the first child of the HTML tag, the fifth child of that tag, and then the second child of that tag.

A write to a single DOM path may trigger cascading updates to other paths; Scout must track all of these updates. For example, inserting a new node at a particular DOM path may shift the subtrees of its new DOM siblings to the right in the DOM tree. In this case, Scout must log writes to the rightward DOM paths, as well as to the newly inserted node. Similar bookkeeping is necessary when DOM nodes are deleted or moved to different locations.

The DOM tree can also be modified by the evaluation of CSS objects that change node styles. Scout models each CSS tag as reading all of the DOM nodes that are above it in the HTML, and then writing all of those DOM nodes with new style information. To capture the set of affected DOM nodes, Scout’s HTML rewriter prepends each CSS tag with an inline JavaScript tag that logs the current state of the DOM tree (i.e., all of the live DOM paths) and then deletes itself from the DOM tree.

In Scout logs, we represent DOM operations using the `window.$$dom` pseudovariable. For example, the identifier `window.$$dom.1` represents the first child of the topmost `<html>` node. We also use the `window.$$xhr` pseudovariable to track network reads and writes via `XMLHttpRequests`. These pseudovariables allow us to use a single analysis engine to process all dependency types.
3.3 Missing Dependencies

To generate a page’s dependency graph, Scout loads an instrumented version of the page on a server-side browser, and collects the resulting dependency information. Later, when Polaris loads the page on a client-side browser (§7.3), Polaris assumes that Scout’s dependency graph is an accurate representation of the dependencies in the page. This might not be true if the page’s JavaScript code exhibits nondeterministic behavior. For example, suppose that a page contains three JavaScript files called a.js, b.js, and c.js. At runtime, a.js may call Math.random(), and use the result to invoke a function in b.js or c.js (but not both). During some executions, Scout will log a dependency between a.js and b.js; during other executions, Scout will log a dependency between a.js and c.js. If there is a discrepancy between the dependency logged by Scout, and the dependency generated by the code on the client browser, then Polaris may evaluate JavaScript files in the wrong order, breaking correctness.

We have not observed such nondeterministic dependencies in our corpus. However, if a page does include such dependencies, Scout must create a dependency graph which contains the aggregate set of all possible dependencies. Such a graph overconstrains any particular load of the page, but guarantees that clients will load pages without errors. The sources of nondeterministic JavaScript events are well-understood [106], so Scout can use a variety of techniques to guarantee that nondeterministic dependencies are either tracked or eliminated. For example, Scout can rewrite pages so that calls to Math.random() use a deterministic seed [106], removing nondeterminism from calls to the random number generator.

For a given page, a web server may generate a different dependency graph for different clients. For example, a web server might personalize the graph in response to a user’s cookie; as another example, a server might return a smaller dependency graph in response to a user agent string which indicates a mobile browser. The server-side logic must run Scout on each version of the dependency graph. We believe that this burden will be small in practice, since even customized versions of a page often share the same underlying graph structure (with different content in some of the nodes).

3.4 Implementation

To build Scout, we used Esprima [73], Estraverse [161], and Escodegen [160] to rewrite JavaScript code, and we used Beautiful Soup [148] to rewrite HTML. We loaded the instrumented pages in a commodity Firefox browser (version 40.0). Each page sent its dependency logs to a dedicated analysis server; logs were sent via an XMLHttpRequest that was triggered by the onload event.

Our implementation of Scout handles the bulk of the JavaScript language. However, our implementation does not currently support the eval(sourceCode) statement, which pages use to dynamically execute new JavaScript code. To support this statement, Scout would need to shim eval() and dynamically rewrite the sourceCode argument so that the rewritten code tracked dependencies.

Our current implementation also does not support the with(obj) statement, which
places \texttt{obj} at the beginning of the scope chain that the JavaScript runtime uses to resolve variable names. To support this statement, Scout merely needs to wrap the \texttt{obj} argument in code which checks whether \texttt{obj} is a proxy; if not, the wrapper would return one.

### 3.5 Performance

We evaluated the overhead that Scout's data flow tracking imposes on page load times using Mahimahi [132], a web record-and-replay tool. Our experiments used the Alexa top 500 pages in the United States [4]. For each page, we compared the page load times of the original page and a version of the page that was instrumented with Scout. Pages were loaded over a variety of emulated network conditions, with link rates in \{1, 5, 10, 20\} Mbits/s and round trip times (RTTs) in \{10, 20, 50, 100\} ms. Our experiments revealed that Scout's data flow tracking imposes a 3\%-5\% overhead on page load times.
Chapter 4

Polaris: Faster Page Loads Using Fine-grained Dependency Tracking

4.1 Overview

To load a page, a browser must resolve the page’s dependency graph [27, 89, 175]. The dependency graph captures “load-before” relationships between a page’s HTML, CSS, JavaScript, and image objects. For example, a browser must parse the HTML <script> tag for a JavaScript file before that file can be fetched. Similarly, the browser must execute the JavaScript code in that file to reveal which images should be dynamically fetched via XMLHttpRequests. The overall load time for a page is the time that the browser needs to resolve the page’s dependency graph, fetch the associated objects, and evaluate those objects (e.g., by rendering images or executing JavaScript files). Thus, fast page loads require efficient dependency resolution.

Unfortunately, a page’s dependency graph is only partially revealed to a browser. As a result, browsers must use conservative algorithms to fetch and evaluate objects, to ensure that hidden load-before relationships are not violated. For example, consider the following snippet of HTML:

```html
<script src='http://x.com/first.js'/>
<script src='http://y.com/second.js'/>
<img src='http://z.com/photo.jpg'/>
```

When a browser parses this HTML and discovers the first <script> tag, the browser must halt the parsing and rendering of the page, since the evaluation of first.js may alter the downstream HTML [98]. Thus, the browser must synchronously fetch and evaluate first.js; this is true even if first.js does not modify the downstream HTML or define JavaScript state required by second.js. Synchronously loading JavaScript files guarantees correctness, but this approach is often too cautious. For example, if first.js and second.js do not modify mutually observable state, the browser should be free to download and evaluate the files in whatever order maximizes the utilization of the network and the CPU. However, pages do not expose such fine-grained dependency information to browsers. This forces browsers to make conservative assumptions about safe load orders by using coarse-grained relationships.
(a) The dependencies captured by traditional which tracks fine-grained data flows. New edges are shown in red.

Figure 4-1: Dependency graphs for weather.com.

(b) The dependencies captured by Scout.

Figure 4-1: Dependency graphs for weather.com.

between HTML tags to guide object retrieval. As a result, pages load more slowly than necessary.

This chapter makes two contributions. First, we describe how Scout can be used to track data dependencies between web objects like JavaScript and HTML files. For example, Scout can track read/write dependencies for an individual JavaScript variable that is accessed by multiple JavaScript files. The resulting dependency graphs are more accurate than those of prior frameworks. As shown in Figure 4-1, our graphs also have dramatically different structures than those of previous approaches. In particular, for 81% of the 500 real-world pages that we examined, our new graphs have different critical paths than those of graphs from prior work (§4.3.3). The critical path defines the set of object evaluations which, if delayed, will always delay the end-to-end load time for a page. Thus, the fact that our new graphs look different is not just an academic observation: our graphs imply a faster way to load web pages.

Our second contribution is Polaris, a dynamic client-side scheduler which uses Scout’s fine-grained dependency graphs to reduce page load times. Figure 4-2 provides an overview of how Polaris works. When a user makes a request for a Polaris-enabled page, the server returns a scheduler stub instead of the page’s original HTML.
The scheduler stub includes the Polaris JavaScript library, the page’s fine-grained dependency graph (as generated by Scout), and the original HTML. The Polaris library uses the Scout graph, as well as dynamic observations about network conditions, to load objects in an order that reduces page load time.

As shown in Figure 4-1, our fine-grained data tracking adds new constraints to standard dependency graphs. However, and perhaps counterintuitively, the Polaris scheduler has more opportunities to reduce page load times. The reason is that, since Polaris has a priori knowledge of the true data dependencies in a page, Polaris can aggressively fetch and evaluate objects “out-of-order” with respect to lexical constraints between HTML tags. In contrast, prior scheduling frameworks lack knowledge of many dependencies, and are forced to make conservative assumptions that are derived from the lexical HTML relationships (§4.2). Those conservative assumptions guarantee the correctness of an assembled page in the face of hidden dependencies, but they often leave a browser’s CPU and network connections underutilized. By using fine-grained dependency graphs, Polaris can ensure both correctness and high utilization of processors and network connections.

Because Polaris’ scheduler is implemented in JavaScript, Polaris can reduce page load times on unmodified commodity browsers; this contrasts with load optimizers like Klotski [27], Amazon Silk [7], and Opera Mini [142], which require modified browsers to interact with a server-side component. Polaris is also complementary to previous load optimizers that use data compression (§4.6) or multiplex several HTTP requests atop a single TCP connection (§4.5.4).

We evaluated Polaris using 500 popular web pages and a variety of network conditions, with latencies ranging from 25 ms to 500 ms, and bandwidths ranging from 1 Mbit/s to 25 Mbits/s. Polaris reduced page load times by 34% at the median, and 59% for the 95th percentile sites.

4.2 The Pitfalls of Lexical Dependencies

As described in Chapter 2, the traditional approach for loading a page is constrained by uncertainty. For example:

- A script tag might read CSS style properties from the DOM tree, so CSS evaluation must block JavaScript execution.
- A script tag might change downstream HTML, so when the browser encounters a script tag, either HTML parsing must block (increasing page load time), or HTML parsing must transfer to a speculative thread (a thread which, if aborted, will have wasted network and computational resources).
- In the example from Section 4.1, two script tags that are lexically adjacent might exhibit a write/read dependency on JavaScript state. Thus, current browsers must execute the script tags serially, in lexical order, even if a different order (or parallel execution) would be more efficient.

These inefficiencies arise because HTML expresses a strict tag ordering that is based on lexical dependencies between tags. In reality, a page’s true dependency graph is a partial ordering in which edges represent true semantic dependencies like write/read
dependencies on JavaScript state. Since HTML does not express all of the true semantic dependencies, the browser is forced to pessimistically guess those dependencies, or use optimistic speculation that may waste resources.

In Section 4.3, we enumerate the kinds of true semantic dependencies that pages can have, and introduce a new framework to extract them. In Section 4.4, we describe how developers can expose true dependencies to the browser, allowing the browser to load pages faster.

4.3 Dependency Tracking

In a traditional dependency graph \[27, 56, 89, 110, 113\], a vertex represents an object like an image or a JavaScript file. An edge represents a load-before relationship that is the side-effect of parsing activity. For example, if a page incorporates an image via an `<img>` tag, the image’s parent in the dependency graph will be the HTML file which contains the tag; if an image is fetched via an XMLHttpRequest, the image’s parent will be the associated JavaScript file.

By emphasizing fetch initiation contexts, i.e., the file whose parsing causes an object to be downloaded, traditional dependency graphs mimic the lexical restrictions that constrain real browsers (§4.2). However, fetch initiation contexts obscure the fine-grained data flows that truly govern the order in which a page’s objects must be assembled. In this section, we provide a taxonomy for those fine-grained dependencies, and describe how Scout (Chapter 3) can be used to capture those dependencies. The resulting dependency graphs have more edges than traditional graphs (because finer-grained dependencies are included). However, as we show in Section 4.5, fine-grained dependency graphs permit more aggressive load schedules, because browsers are no longer shackled by conservative assumptions about where hidden dependencies might exist.

4.3.1 Dependency Types

We are interested in capturing three types of data flows that involve the JavaScript heap and the DOM state belonging to HTML and CSS.

**Write/read dependencies** arise when one object produces state that another object consumes. For example, `a.js` might create a global variable in the JavaScript heap; later, `b.js` might read the variable. When optimizing the load order of the two scripts, we cannot evaluate `b.js` before `a.js` (although it is safe to `fetch b.js` before `a.js`).

**Read/write dependencies** occur when one object must read a piece of state before the value is updated by another object. Such dependencies often arise when JavaScript code must read a DOM value before the value is changed by the HTML parser or another JavaScript file. For example, suppose that the HTML parser encounters a JavaScript tag that lacks the `async` or `defer` attributes. The browser
must synchronously execute the JavaScript file. Suppose that the JavaScript code reads the number of DOM nodes that are currently in the DOM tree. The DOM query examines a snapshot of the DOM tree at a particular moment in time; as explained in Chapter 2, a browser progressively updates the DOM tree as HTML is parsed. Thus, any reordering of object evaluations must ensure value equivalence for DOM queries—regardless of when a JavaScript file is executed, its DOM queries must return the same results. This guarantees deterministic JavaScript execution semantics [106] despite out-of-order evaluation.

Write/write dependencies arise when two objects update the same piece of state, and we must preserve the relative ordering of the writes. For example, CSS files update DOM state, changing the rules which govern a page's visual presentation. The CSS specification states that, if two files update the same rule, the last writer wins. Thus, CSS files which touch the same rule must be evaluated in their original lexical ordering in the HTML. However, the evaluation of the CSS files can be arbitrarily reordered with respect to the execution of JavaScript code that does not access DOM state.

Output devices are often involved in write/write dependencies. As described in the previous paragraph, CSS rules create a write/write dependency on a machine's display device. Write/write dependencies can also arise for local storage and the network. For example, the localStorage API exposes persistent storage to JavaScript using a key/value interface. If we shuffle the order in which a page evaluates JavaScript objects, we must ensure that the final value for each localStorage key is the same value that would result from the original execution order of the JavaScript files.

Traditional dependencies based on HTML tag constraints can often be eliminated if finer-grained dependencies are known. For example, once we know the DOM dependencies and JavaScript heap dependencies for a <script> tag, the time at which the script can be evaluated is completely decoupled from the position of the <script> tag in the HTML—we merely have to ensure that we evaluate the script after its fine-grained dependencies are satisfied. Similarly, we can parse and render a piece of HTML at any time, as long as we ensure that we have blocked the evaluation of downstream objects in the dependency graph.

Images do need to be placed in specific locations in the DOM tree. However, browsers already allow images to be fetched and inserted asynchronously. So, images can be fetched in arbitrary orders, regardless of the state of the DOM tree, but their insertion is dependent on the creation of the associated DOM elements. We model this using write/write dependencies on DOM elements: the HTML parser must write an initially empty <img> DOM node, and then the network stack must insert the fetched image bitmap into that node.

### 4.3.2 Dependency Graphs: Scout vs. Prior Tools

To capture the fine-grained dependency graph for a real web page, we first record the content of the page using Mahimahi [132]. Next, we use Scout to rewrite each
Figure 4-3: Comparing the order in which different tools declare that a simple page's objects must be evaluated. The notation HTML[i:j] refers to HTML lines i up to and including j. The notation obj@HTML[k] refers to the object whose corresponding tag is at HTML[k].

(a) The HTML for a simple page. graph generated by the dependency graph created by Scout. (b) The dependency graph created by WProf [175].

JavaScript and HTML file in the page, adding instrumentation to log fine-grained data flows across the JavaScript heap and the DOM. Scout then loads the instrumented page in a regular browser. As the page loads, it emits a log of reads and writes to page state to a Scout analysis server. The server uses the log to generate the fine-grained dependency graph for the page by extracting the dependencies described in Section 4.3.1.

Figure 4-3(a) depicts a simple web page with two JavaScript files and one CSS file. Figures 4-3(b), (c), and (d) show the dependency graphs that are produced by Scout, Klotski [27], and WProf [175].

- Scout allows second.js and the first chunk of HTML to be evaluated in parallel, since second.js does not access DOM state or JavaScript state defined by prior JavaScript files. first.js does access DOM state from upstream HTML tags, but Scout allows the evaluation of first.js to proceed in parallel with the parsing of downstream HTML. Scout treats CSS as a read and then a write to all upstream HTML, so the CSS file must be evaluated before the evaluation of downstream HTML and downstream scripts which access DOM state.
- Klotski [27] cannot observe fine-grained data flows, so its dependency graphs are defined by lexical HTML constraints (§4.2). Given a dependency graph like the one shown in Figure 4-3(c), Klotski uses heuristics to determine which objects a server should push to the browser first. However, Klotski does not know the page’s true data dependencies, so Klotski cannot guarantee that prioritized objects can actually evaluate ahead of schedule with respect to their evaluation times in the original page. It is only safe to evaluate an object (prioritized or not) when its ancestors in the dependency graph have been evaluated. So, Klotski’s prioritized pushes can safely warm the client-side cache, but in general, it is unsafe for those pushes to synchronously trigger object evaluations.
- By instrumenting the browser, WProf observes the times at which a browser...
Figure 4-4: How traditional dependency graphs change when updated with information from fine-grained data flows. The updated graphs have additional edges which belong to previously untracked dependencies. The new edges often modify a page’s critical paths. Note that a slack node is a node that is not on a critical path.

is inside the network stack or a parser for HTML, CSS, or JavaScript. Thus, WProf can track complex interactions between a browser’s fetching, parsing, and evaluation mechanisms. However, this technique only allows WProf to analyze the critical path for the lexically-defined dependency graph. This graph does not capture true data flows, and forces conservative assumptions about evaluation order (§4.2). As shown in Figure 4-3(d), WProf overconstrains the order in which objects can be evaluated (although WProf may allow objects to be fetched out-of-lexical-order).

In summary, only Scout produces a dependency graph which captures the true constraints on the order in which objects can be evaluated. Polaris uses these fine-grained dependencies to schedule object downloads—by prioritizing objects that block the most downstream objects, Polaris reduces overall page load times (§4.4).

4.3.3 Dependency Graphs: Structural Differences for Real Pages

We used Mahimahi [132], an HTTP record-and-replay tool, to record the content from 500 sites in the Alexa list [4]. The corpus spanned a variety of page categories, including news, commerce, and social media. The corpus also included five mobile-optimized sites. Since our Scout prototype does not support the eval() and with() statements, we selected pages which did not use those statements.

Figure 4-4 summarizes the differences between Scout’s dependency graphs and the traditional ones that are defined by Klotski [27] and the built-in developer tools from Chrome [56], Firefox [113], and IE [110]. As shown in Figure 4-4(a), traditional graphs are almost always incomplete, missing many edges that can only be detected via data flow analysis. That analysis adds 29.8% additional edges at the median, and 118% more edges at the 95th percentile.

Those additional edges have a dramatic impact on the characteristics of dependency graphs. For example, adding fine-grained dependencies alters the critical path length for 80.8% of the pages in our corpus (Figure 4-4(b)). The set of objects on
Figure 4-5: An example of dynamic critical paths during the load of a simple page. Dynamic critical paths are shown in red. Numbers represent the order in which Polaris requests the objects. Shaded objects have been received and evaluated; numbered but unshaded objects have been requested, but have no responses yet. We assume that all objects are from the same origin, and that only two outstanding requests per origin are allowed.

those paths often changes, with old objects being removed and new objects being added. Furthermore, as shown in Figure 4-4(d), 86.6\% of pages have a smaller fraction of slack nodes when fine-grained dependencies are considered. Slack nodes are nodes that are not on a critical path. Thus, a decrease in slack nodes means that browsers have fewer load schedules which result in optimal page load times.

4.4 Polaris: Dynamic Client-side Scheduling

Polaris is a client-side scheduler for the loading and evaluation of a page’s objects. Polaris is written completely in JavaScript, allowing it to run on unmodified commodity browsers. Polaris accepts a Scout graph as input, but also uses observations about current network conditions to determine the dynamic critical path for a page. The dynamic critical path, i.e., the path which currently has the most unresolved objects, is influenced by the order and latency with which network fetches complete; importantly, the dynamic critical path may be different than the critical path in the static dependency graph.\(^1\) Polaris prioritizes the fetching and evaluation of objects along the dynamic critical path, trying to make parallel use of the client’s CPU and network, and trying to keep the client’s network pipe full, given browser constraints on the maximum number of simultaneous network requests per origin.

Figure 4-5 shows how a page’s dynamic critical path can change over time. In Figure 4-5(a), Polaris has evaluated object 0, and issued requests for objects 1 and 2, because those objects are the roots for the deepest unresolved paths in the dependency graph. In Figure 4-5(b), Polaris has received and evaluated object 1, although object 2 is still in-flight. Polaris has one available request slot, so it requests object 3, because that object is the root of the deepest unresolved path. In Figure 4-5(c), Polaris has

\(^1\)This is why a dynamic client-side scheduler is better than a static client-side scheduler that ignores current network conditions and deterministically fetches objects from a server-provided URL list.
received and evaluated object 3; Polaris uses the available request slot to fetch object 4. Then, object 2 is received and evaluated. The critical path changes—the deepest chain is now beneath object 2, so Polaris requests object 5 next.

To use Polaris with a specific page, a web developer runs Scout on that page to generate a dependency graph and a Polaris scheduler stub. The developer then configures her web server to respond to requests for that page with the scheduler stub’s HTML instead of the page’s regular HTML (see Figure 4-2). The stub contains four components.

- The **scheduler** itself is just inline JavaScript code.
- The **Scout dependency graph** for the page is represented as a JavaScript variable inside the scheduler.
- **DNS prefetch hints** indicate to the browser that the scheduler will be contacting certain hostnames in the near future. DNS prefetch hints are expressed using `<link>` tags of type `dns-prefetch`, e.g.,
  
  `<link rel="dns-prefetch" href="http://domain.com">`

  DNS hints allow Polaris to pre-warm the DNS cache in the same way that the browser does during speculative HTML parsing (Chapter 2).
- Finally, the stub contains the **page’s original HTML**, which is broken into chunks as determined by Scout’s fine-grained dependency resolution (see Chapter 3 and Figure 4-3). When Scout generates the HTML chunks, it deletes all `src` attributes in HTML tags, since the external objects that are referenced by those attributes will be dynamically fetched and evaluated by Polaris.

Polaris adds few additional bytes to a page’s original HTML. Across our test corpus of 500 sites, the scheduler stub was 3% (36.5 KB) larger than a page’s original HTML at the median.

The scheduler uses `XMLHttpRequests` to dynamically fetch objects. To evaluate a JavaScript file, the scheduler uses the built-in `eval()` function that is provided by the JavaScript engine. To evaluate HTML, CSS, and images, Polaris leverages DOM interfaces like `document.innerHTML` to dynamically update the page’s state.

In the rest of this section, we discuss a few of the subtler aspects of implementing an object scheduler as a JavaScript library instead of native C++ code inside the browser.

**Browser network constraints:** Modern browsers limit a page to at most six outstanding requests to a given origin. Thus, Polaris may encounter situations in which the next missing object on the dynamic critical path would be the seventh outstanding request to an origin. If Polaris actually generated the request, the request would be placed at the end of the browser’s internal network queue, and would be issued at a time of the browser’s choosing. Polaris would lose the ability to precisely control the in-flight requests at any given moment.

To avoid this dilemma, Polaris maintains per-origin priority queues. With the exception of the top-level HTML (which is included in the scheduler stub), each object in the dependency graph belongs to exactly one queue. Inside a queue, objects that
are higher in the dependency tree receive a higher priority, since those objects prevent
the evaluation of more downstream objects. At any given moment, the scheduler tries
to fetch objects that reside in a dynamic critical path for the page load. However,
if fetching the next object along a critical path would violate a per-origin network
constraint, Polaris examines its queues, and fetches the highest priority object from
an origin that has available request slots.\(^2\)

**Frames:** A single page may contain multiple iframes. Scout generates a scheduler
stub for each one, but the browser’s per-origin request cap is a page-wide limit. Thus,
the schedulers in each frame must cooperate to respect the limit and prevent network
requests from getting stuck inside the browser’s internal network queues.

The scheduler in the top frame coordinates the schedulers in child frames. Using
`postMessage()` calls, children ask the top-most parent for permission to request
particular objects. The top-most parent only authorizes a fetch if per-origin request
limits would not be violated.

**URL matching:** A page’s coarse-grained dependency graph has a stable structure [27]. In other words, the edges and vertices that are defined by lexical HTML con-
straints change slowly over time. However, the URLs for specific vertices change more
rapidly. For example, if JavaScript code dynamically generates an `XMLHttpRequest`
URL, that URL may embed the current date in its query string. Across multiple page
loads, the associated object for the URL will have different names, even though all of
the objects will reside in the same place in the dependency graph.

To handle any discrepancies between the URLs in Scout’s dependency graphs and
the URLs which `XMLHttpRequests` generate on the client, Polaris uses a matching
heuristic to map dynamic URLs to their equivalents in the static dependency graph.
Our prototype implementation uses Mahimahi’s matching heuristic [132], but Polaris
is easily configured to use others [27, 31, 157].

**Page-generated XHRs:** When Polaris evaluates a JavaScript file, the executed
code might try to fetch an object via `XMLHttpRequest`. Assuming that a page has
deterministic JavaScript code (Chapter 3), Scout will have included the desired object
in the page’s dependency graph. However, during the loading of the page in a real client
browser, Polaris requires control over the order in which objects are fetched. Thus,
Polaris uses an `XMLHttpRequest` shim [106] to suppress autonomous `XMLHttpRequests`.
Polaris issues those requests using its own scheduling algorithm, and manually fires
`XMLHttpRequest` event handlers when the associated data has arrived.

\(^2\) Browsers allow users to modify the constraint on the maximum number of connections per origin;
Polaris can be configured to respect user-programmed values.
4.5 Page Load Time Improvements with Polaris

In this section, we demonstrate that Polaris can decrease page load times across a variety of web pages and network configurations: performance improves by 34% and 59% for the median and 95th percentile sites, respectively. Polaris’ benefits grow as network latencies increase, because higher RTTs increase the penalty for bad fetch schedules. Thus, Polaris is particularly valuable for clients with cellular or low-quality wired networks. However, even for networks with moderate RTTs, Polaris can often reduce load times by over 20%.

4.5.1 Methodology

We evaluated Polaris using the 500 site corpus that is described in Section 4.3.3. We used Mahimahi [132] to capture site content and later replay it using emulated network conditions. To build Polaris-enabled versions of each page, we post-processed the recorded web content, generating Polaris scheduler stubs for each site. We then compared the load times of the Polaris sites and the original versions of those sites. All experiments used Firefox v40.0. Unless otherwise specified, all experiments used cold browser caches and DNS caches.

Page load time is normally defined with respect to the navigationStart and loadEventEnd JavaScript events. However, loadEventEnd is inaccurate for Polaris pages, since the event only indicates that the scheduler stub has been loaded; the rest of the page’s objects remain to be fetched by the dynamic scheduler. So, to define the load time for a Polaris page, we loaded the original version of the page and logged the objects that were fetched between navigationStart and loadEventEnd. We then defined load time for the Polaris page as the time needed to fetch all of those objects.

4.5.2 Reducing Page Load Times

Figure 4-6 demonstrates Polaris’ ability to reduce load times. There are two major trends to note. First, for a given link rate, Polaris’ benefits increase as network latency increases. For example, at a link rate of 12 Mbits/s, Polaris provides an average improvement of 10.1% for an RTT of 25 ms. However, as the RTT increases to 100 ms and 200 ms, Polaris’ benefits increase to 27.5% and 35.3%, respectively. The reason is that, as network latencies grow, so do the penalties for not prioritizing the fetches of objects on the dynamic critical path. Polaris does prioritize the fetching of critical path objects. Furthermore, Polaris never has to wait for an object evaluation to reveal a downstream dependency—Polaris knows all of the dependencies at the beginning of the page load, so Polaris can always keep the network pipe full.

The second trend in Figure 4-6 is that, for a given RTT, Polaris’ benefits increase as network bandwidth grows. This is because, if bandwidth is extremely low, transfer times dominate fetch costs. As bandwidth increases, latency becomes the dominant factor in download times. Since Polaris prioritizes the fetch orders for critical path objects (but does nothing to reduce those objects’ bandwidth costs), Polaris’ gains are most pronounced when latencies govern overall download costs.
Figure 4-6: Polaris’ average reduction in page load times, relative to baseline load times with Firefox v40.0. Each bar is the average reduction in load time across the entire 500 site corpus. Error bars span one standard deviation in each direction of the average.

Figure 4-6 describes Polaris’ gains in relative terms. Table 4.1 depicts absolute gains, describing how many raw milliseconds of load time Polaris removes. Even on a fast network with 25 ms of latency, Polaris eliminates over 250 ms of load time. Those results are impressive, given that web developers strive to eliminate tens of milliseconds from their pages’ load times [19, 23, 50].

The error bars in Figure 4-6 are large. The reason is that, for a given network bandwidth and latency, Polaris’ benefits are determined by the exact structure of a page’s dependency graph. To understand why, consider the three sites in Figure 4-7.

- The homepage for apple.com has a flat dependency graph, as shown in Figure 4-8. This means that, once the browser has the top-level HTML, the other objects can be fetched and evaluated in an arbitrary order; all orders will result in similar end-to-end page load times. Thus, for low RTTs, Polaris loads the apple.com homepage 1–2% slower than the baseline, due to computational overheads from Polaris’ scheduling logic.
- In contrast, the ESPN homepage has a dependency path of length 5, and several paths of length 4. Also, 48% of the page’s content is loaded from only two origins (a.espncdn.com and a1.espncdn.com), magnifying the importance of optimally scheduling the six outstanding requests for each origin (§7.3). In ESPN’s dependency graph, many of the long paths consist of JavaScript files. However, the standard Firefox scheduler has no way of knowing this. So, when
Table 4.1: Polaris’ raw reduction in median page load times for a subset of the parameter values in Figure 4-6.

<table>
<thead>
<tr>
<th>RTT (ms)</th>
<th>25 ms</th>
<th>100 ms</th>
<th>500 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Mbit/s</td>
<td>256.3 ms</td>
<td>883.9 ms</td>
<td>1857.5 ms</td>
</tr>
<tr>
<td>12 Mbit/s</td>
<td>309.1 ms</td>
<td>1274.1 ms</td>
<td>2935.0 ms</td>
</tr>
<tr>
<td>25 Mbit/s</td>
<td>382.5 ms</td>
<td>1385.3 ms</td>
<td>3188.3 ms</td>
</tr>
</tbody>
</table>

Figure 4-7: Polaris’ average reduction in page load times, relative to baseline load times, for three sites with diverse dependency graph structures. Each experiment used a link rate of 12 Mbits/s.

Firefox loads the standard version of the page, it initially requests a small number of JavaScript objects, and then fills the rest of its internal request queue with 32 image requests. As a result, when a JavaScript file evaluates and generates a request for another JavaScript file on the critical path, the request is often stuck behind image requests in the browser’s internal network queue. In contrast, Polaris has a priori knowledge of which JavaScript files belong to deep dependency chains. Thus, Polaris prioritizes the fetching of those objects, using its knowledge of per-origin request caps to ensure that the fetches for critical path objects are never blocked.

- As shown in Figure 4-1(b), the weather.com homepage is even more complicated than that of ESPN. Deep, complex dependency graphs present Polaris with the most opportunities to provide gains. Thus, of the three sites in Figure 4-7,
weather.com enjoys the largest reductions in load time.

Figure 4-9 depicts the order in which requests issue for the normal version of the StackOverflow site, and the Polaris version. In general, Polaris issues requests earlier; by prioritizing the fetches of objects on the dynamic critical path, Polaris minimizes the overall fetch time needed to gather all objects. However, as shown in Figure 4-9, Polaris briefly falls behind the default browser scheduler after fetching the tenth object. The reason is that, in our current Polaris implementation, HTML is rendered in large chunks. While that HTML is being rendered, Polaris cannot issue new HTML requests, because executing Polaris’ JavaScript-level scheduler would block rendering (Chapter 2). In contrast, a native browser scheduler can issue new requests in parallel with HTML rendering. Thus, the default Firefox scheduler has a lower time-to-first paint than Polaris, and Polaris falls behind the default scheduler after the tenth object fetch. However, after Polaris renders the bulk of the HTML, Polaris quickly regains its lead and never relinquishes it. To minimize Polaris’ time-to-first-paint, future versions of Polaris will render HTML in smaller increments; this will not affect Polaris’ ability to optimize network utilization.

### 4.5.3 Browser Caching

Up to this point, our experiments have used cold browser caches. In this section, we evaluate Polaris’ performance when caches are warm. To do so, we examined the HTTP headers in our recorded web pages, and, for each object that was marked as cacheable, we rewrote the caching headers to ensure that the object would remain cacheable for the duration of our experiment. Then, for each page, we cleared the browser’s cache, and loaded the page twice, recording the elapsed time for the second load.

Figure 4-10 depicts Polaris’ benefits with warm caches; the improvements are normalized with respect to Polaris’ improvements when caches are cold. In general, Polaris’ benefits decrease as cache hit rates increase, because there are fewer opportunities for Polaris to optimize network fetches. For example, Ebay caches 92% of all objects, including most of the JavaScript files involved in deep dependency chains; thus, Polaris provides little advantage over the standard scheduling algorithm.

That being said, there are many instances in which caching does not touch objects along a page’s critical path. For example, on ESPN’s site, 76% of objects are cacheable, but only one object on the deepest dependency chain is cached. Furthermore, a.espncdn.com serves many uncachable images and JavaScript objects, leading Firefox’s standard scheduler to bury critical path JavaScript files behind images that
Figure 4-9: Request initiation times for the regular and Polaris-enabled versions of StackOverflow. These results used a 12 Mbits/s link with an RTT of 100 ms.

are not on the critical path (§4.5.2). So, even though ESPN caches 76% of its objects, Polaris still provides 71% of its cold-cache benefits.

Note that the Apple site is an outlier: it caches 93% of its objects, but Polaris provides little benefit in the cold cache case (§4.5.2), so Polaris provides most of that negligible benefit in the warm cache case as well.

4.5.4 SPDY

Google proposed SPDY [100], a transport protocol for HTTP messages, to remedy several problems with the HTTP/1.1 protocol. SPDY differs from HTTP/1.1 in four major ways:

- First, SPDY uses a single TCP connection to multiplex all of a browser’s HTTP requests and responses involving a particular origin. This allows HTTP requests to be pipelined, and reduces the TCP and TLS handshake overhead that would be incurred if a browser opened multiple TCP connections to an origin.
- SPDY also allows a browser to prioritize the fetches of certain objects (e.g., JavaScript files which block HTML parsing). Priorities give servers hints about how to allocate limited bandwidth to multiple responses.
- SPDY compresses HTTP headers. HTTP is a text-based protocol, so compression can result in non-trivial bandwidth savings.
- Finally, SPDY allows a server to proactively push objects to a browser if the server believes that the browser will request those objects in the near future.
SPDY was a major influence on the HTTP/2 protocol [17] whose deployment is currently starting.

Mahimahi supports SPDY page loads using the mod_spdy Apache extension [99]. Thus, we could use Mahimahi to explore how SPDY interacts with Polaris. We loaded each page in our test corpus using four different schemes: HTTP/1.1 (which all of our previous experiments used), Polaris over HTTP/1.1, SPDY, and Polaris over SPDY. In our experiments, SPDY used TCP multiplexing, object prioritization, and HTTP header compression, but not server push, since few of the sites in our test corpus defined SPDY push policies.

Figure 4-11 compares load times using the four schemes on a 12 Mbits/s link with various RTTs; the performance baseline is the load time using HTTP/1.1. On average, load times using SPDY are 1.74%-3.98% faster than those with HTTP/1.1. Load times using Polaris over SPDY are 2.05%-4.03% faster than those with Polaris over HTTP/1.1. These results corroborate prior work [176] which found that object dependencies limit the ability of SPDY to maximize network utilization. For example, a SPDY-enabled browser may prioritize a JavaScript file in hopes of minimizing the stall time of the HTML parser. However, without Polaris, the SPDY-enabled browser is still limited by conservative lexical dependencies (§4.2), meaning that it cannot aggressively fetch objects “out-of-order” with respect to lexical constraints. In contrast, both Polaris over HTTP/1.1 and Polaris over SPDY have fine-grained dependency information. That information allows Polaris to issue out-of-lexical-order fetches which
reduce page load time while respecting the page's intended data flow semantics.

In theory, SPDY-enabled web servers could use Scout's dependency graphs to guide server push policies. However, we believe that clients, not servers, are best qualified to make decisions about how a client's network pipe should be used. A server from origin X cannot see the objects being pushed by origin Y, so different origins may unintentionally overload a client's resource-constrained network connection. Furthermore, Scout's dependency graphs do not capture dynamic critical paths, i.e., the set of object fetches which a client should prioritize at the current moment (§7.3). Thus, a well-intentioned server may hurt load time by pushing objects which are not on a dynamic critical path. Polaris avoids this problem using dynamic client-side scheduling.

4.6 Polaris: Related Work

Prior dependency trackers [27, 56, 110, 113, 175] deduce dependencies using lexical relationships between HTML tags. As discussed in Section 4.2, those lexical relationships do not capture fine-grained data flows. As a result, load schedulers which use those dependency graphs are forced to make conservative assumptions to preserve correctness.

WebProphet [89] determines the dependencies between objects by carefully per-
turbining network fetch delays for individual objects; delaying a parent should delay the loads of dependent children. This technique also relies on course-grained lexical dependencies, since the perturbed browser uses those HTML dependencies to determine which objects to load.

Silo [103] uses aggressive inlining of JavaScript and CSS to fetch entire pages in one or two RTTs. However, Silo does not use the CPU and the network in parallel—all content is fetched, and then all content is evaluated. In contrast, Polaris overlaps computation with network fetches.

Compression proxies like Google FlyWheel [1] and Opera Turbo [141] transparently compress objects before transmitting them to clients. For example, FlyWheel re-encodes images into space-saving formats, and minifies JavaScript and CSS. Polaris is complementary to such techniques.

JavaScript module frameworks like RequireJS [26] and ModuleJS [82] allow developers to manually specify dependencies between JavaScript libraries. Once the dependencies are specified, the frameworks ensure that the relevant libraries are loaded in the appropriate order. Keeping manually-specified dependencies up-to-date can be challenging for a large web site. In contrast, Scout automatically tracks fine-grained dependencies between JavaScript files. Scout also tracks dependencies involving HTML, CSS, and images.
Chapter 5

Vesper: Measuring Time-to-Interactivity for Web Pages

5.1 Overview

Users want web pages to load quickly [145, 181, 185]. Thus, a vast array of techniques have been invented to decrease load times. For example, Polaris (Chapter 4) analyzes data dependencies in page loads to reorder requests in a way that minimizes network round trips. Browser caches try to satisfy network requests using local storage. CDNs [49, 137, 174] push servers near clients, so that cache misses can be handled with minimal network latency. Cloud browsers [7, 142, 157, 177] resolve a page’s dependency graph on a proxy that has low-latency links to web servers; this allows a client to download all objects in a page using a single HTTP round-trip to the proxy.

All of these approaches try to reduce page load time. However, an inconvenient truth remains: none of these techniques directly optimize the speed with which a page becomes interactive. Modern web pages have sophisticated, dynamic GUIs that contain both visual and programmatic aspects. For example, many sites provide a search feature via a text input with autocompletion support. From a user’s perspective, such a text input is worthless if the associated HTML tags have not been rendered; however, the text input is also crippled if the JavaScript code that implements autocompletion has not been fetched and evaluated. JavaScript code can also implement animations or other visual effects that do not receive GUI inputs directly, but which are still necessary for a page to be ready for user interaction. As shown in Figure 5-1, pages often contain hundreds of event handlers that drive interactivity.

In this chapter, we propose a new definition for load time that directly captures page interactivity. We define a page to be fully loaded when:

1. the visual content in the initial browser viewport\(^1\) has completely rendered, and
2. for each interactive element in the initial viewport, the browser has fetched and evaluated the JavaScript and DOM state that supports the element’s interactive functionality.

\(^1\)The viewport is the region of a page that the browser is currently displaying. Content in the initial viewport is often called “above-the-fold” content.
Prior definitions for page load time overdetermine or underdetermine one or both of those conditions (§5.2), leading to inaccurate measurements of page interactivity. For example, the traditional definition of page load time, as represented by the JavaScript onload event, captures when all of a page’s HTML, JavaScript, CSS, and images have been fetched and evaluated; however, this definition is overly conservative, since only a subset of that state may be needed to allow a user to interact with the content in the initial viewport. Newer metrics like above-the-fold time [102] and Speed Index [62] measure the time that a page needs to render the initial viewport. However, these metrics do not capture whether the page has loaded critical JavaScript state (e.g., event handlers that respond to GUI interactions, or timers that implement animations).

To accurately measure page interactivity, we must determine when conditions (1) and (2) are satisfied. Determining when condition (1) has been satisfied is relatively straightforward, since rendering progress can be measured using screenshots or the paint events that are emitted by the browser’s debugger interface. However, determining when condition (2) has been satisfied is challenging. How does one precisely enumerate the JavaScript state that supports interactivity? How does one determine when this state is ready? To answer these questions, we introduce a new measurement framework called Vesper. Vesper rewrites a page’s JavaScript and HTML; when the rewritten page loads, the page automatically logs paint events as well as reads and writes to individual JavaScript variables and DOM elements. By analyzing these logs, Vesper generates a progressive load metric, called Ready Index, which quantifies the fraction of the initial viewport that is interactive (i.e., visible and functional) at a given moment. Vesper also outputs a derived metric, called Ready Time, which represents the exact time at which all of the above-the-fold state is interactive.

Using a test corpus of 350 popular sites, we compared our new load metrics to traditional ones. Figure 5-2a provides a concrete example of the results, showing the differences between page load time (PLT), above-the-fold time (AFT), and Ready Time (RT) for the amazon.com homepage when loaded over a 12 Mbits/s link with a 100 ms RTT. AFT underestimates time-to-full-interactivity by 2.56 seconds; PLT overestimates the time-to-full-interactivity by 2.72 seconds. Web developers celebrate the elimination of milliseconds of “load time,” claiming that a slight decrease can
Figure 5-2: Timelines for loading amazon.com, indicating when critical interactive components become fully interactive. Note that Ready Time best captures when the site is interactive; furthermore, optimizing for Ready Time is the best way to decrease the page's time-to-interactivity. The client used a 12 Mbits/s link with a 100 ms RTT (§5.5.1).

result in millions of dollars of extra income for a large site [24, 37, 184]. However, our results suggest that developers may be optimizing for the wrong definition of load time. As shown in Figure 5-3, prior metrics inaccurately forecast time-to-full-interactivity
<table>
<thead>
<tr>
<th>RTT</th>
<th>PLT</th>
<th>RT</th>
<th>AFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 ms</td>
<td>1.5</td>
<td>1.1</td>
<td>0.8</td>
</tr>
<tr>
<td>50 ms</td>
<td>3.4</td>
<td>2.5</td>
<td>1.9</td>
</tr>
<tr>
<td>100 ms</td>
<td>6.1</td>
<td>3.9</td>
<td>2.9</td>
</tr>
<tr>
<td>200 ms</td>
<td>9.2</td>
<td>5.6</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Figure 5-3: Median (95th percentile) load time estimates in units of seconds. Each page in our 350 site corpus was loaded over a 12 Mbits/s link.

under a variety of network conditions, with median inaccuracies of 24%–39%; as shown in our user study (§5.6), users with interactive goals prefer websites that actually prioritize the loading of interactive content.

The differences between load metrics are particularly stark if a page’s dependency graph (§4.1) is deep, or if a page’s clients are stuck behind high-latency links. In these scenarios, the incremental interactivity of a slowly-loading page is important: as the page trickles down the wire, interactive HTML tags should become visible and functional as soon as possible. This allows users to meaningfully engage with the site, even if some content is missing; incremental interactivity also minimizes the time window for race conditions in which user inputs are generated at the same time that JavaScript event handling state is being loaded [144]. To enable developers to build incrementally-interactive pages with low Ready Indices, we extended Polaris (Chapter 4), which allows a page to explicitly schedule the order in which objects are fetched and evaluated. We created a new Polaris scheduler that optimizes for Ready Index; the resulting scheduler improves RI by a median of 29%, and RT by a median of 32%. Figure 5-2c demonstrates the scheduler’s performance on the amazon.com homepage. Importantly, Figure 5-2b shows that optimizing for above-the-fold time does not optimize for time-to-interactivity.

Of course, not all sites have interactive content, and even interactive sites can be loaded by users who only look at the content. In these situations, pages should optimize for the rendering speed of above-the-fold content. Fortunately, our user study shows that pages which optimize for Ready Index will substantially reduce user-perceived rendering delays too (§5.6). If desired, Vesper enables developers to automatically optimize their pages solely for rendering speed instead of Ready Index.

In summary, this chapter has four contributions. First, we define a new load metric called Ready Index which quantifies a page’s interactive status (§5.3). Determining how interactivity evolves over time is challenging. Thus, our second contribution is a tool called Vesper that automates the measurement of Ready Index (§5.4). Our third contribution is a study of Ready Index in 350 real pages. By loading those pages in a variety of network conditions, we explain the page characteristics that lead to faster interactivity times (§5.5). Our fourth contribution is a new Polaris scheduler that optimizes for a page’s Ready Index or pure rendering speed; both optimizations are enabled by Vesper-collected data. User studies demonstrate that pages which optimize for Ready Index provide better support for immediate interactivity (§5.6).
5.2 Existing Web Performance Metrics

In this section, we describe prior attempts to define “page load time.” Each metric tracks a different set of page behaviors; thus, for a given page load, different metrics may provide radically different estimates of the load time.

The Original Definition: The oldest metric is defined with respect to the JavaScript onload event. A browser fires that event when all of the external content in a page’s static HTML file has been fetched and evaluated. All image data must be present and rendered; all JavaScript must be parsed and executed; all style files must be processed and applied to the relevant HTML tags; and so on. The load time for a page is defined as the elapsed time between the navigationStart event and the onload event. In the rest of this chapter, we refer to this load metric as PLT (“page load time”).

PLT was a useful metric in the early days of the web, but modern web pages often dynamically fetch content after the onload event has fired [61, 98]. PLT also penalizes web pages that have large amounts of statically-declared below-the-fold content. Below-the-fold content resides beneath the initial browser viewport, and can only be revealed by user scrolling. PLT requires static below-the-fold content to be fetched and evaluated before a page load is considered done. However, from a user’s perspective, a page can be ready even if its below-the-fold content is initially missing: the interactivity of the initial viewport content is the primary desideratum.

Time to First Paint: Time to First Paint (TTFP) measures when the browser has received enough page data to render the first pixels in the viewport. Time to First Meaningful Paint [151], or TTFMP, measures the time until the biggest layout change, using the intuition that the associated paint event is the one that matters most. TTFP and TTFMP try to capture the earliest time that a human could usefully interact with a page. For a given PLT, a lower TTFP or TTFMP is better. However, decreasing a page’s PLT is not guaranteed to lower the other metrics, and vice versa [1]. For example, when the HTML parser (which generates input for the rendering pipeline) hits a <script> tag, the parser may need to synchronously fetch and evaluate the JavaScript file before continuing the HTML parse [128]. By pushing <script> tags to the end of a page’s HTML, render times may improve; however, careless deferral of JavaScript evaluation may hurt interactivity, since event handlers will be registered later, animation callbacks will start firing later, and so on.

Above-the-fold Time: This metric represents the time that the browser needs to render the final state of all pixels in the initial browser viewport. Like TTFP, above-the-fold time (AFT) is not guaranteed to move in lockstep with PLT. Measuring AFT and TTFP requires a mechanism for tracking on-screen events. WebKit-derived browsers like Chrome and Opera expose paint events via their debugging interfaces. Rendering progress can also be tracked using screenshots [64, 78].
If a web page contains animations, or videos that automatically start playing, a naive measurement of AFT would conclude that the page never fully loaded. Thus, AFT algorithms must distinguish between static pixels that are expected to change a few times at most, and dynamic pixels that are expected to change frequently, even once the page has fully loaded. To differentiate between static and dynamic pixels, AFT algorithms use a threshold number of pixel updates; a pixel which is updated more often than the threshold is considered to be dynamic. AFT is defined as the time that elapses until the last change to a static pixel.

**Speed Index:** AFT fails to capture the progressive nature of the rendering process. Consider two hypothetical pages which have the same AFT, but different rendering behavior: the first page updates the screen incrementally, while the second page displays nothing until the very end of the page load. Most users will prefer the first page, even though both pages have the same AFT.

Speed Index [62] captures this preference by explicitly logging the progressive nature of page rendering. Intuitively speaking, Speed Index tracks the fraction of a page which has not been rendered at any given time. By integrating that function over time, Speed Index can penalize sites that leave large portions of the screen unrendered for long periods of time. More formally, a page’s Speed Index is $\int_0^{\text{end}} \left( 1 - \frac{p(t)}{100} \right) \, dt$, where \text{end} is the AFT time, and \( p(t) \) is the percentage of static pixels at time \( t \) that are set to their final value. A lower Speed Index is better than a higher one.

Strictly speaking, a page’s Speed Index has units of “percentage-of-visual-content-that-is-not-displayed milliseconds.” For brevity, we abuse nomenclature and report Speed Index results in units of just “milliseconds.” However, a Speed Index cannot be directly compared to a metric like AFT that is actually measured in units of time. Also note that TTFP, AFT, and Speed Index do not consider the load status of JavaScript state. As a result, these metrics cannot determine (for example) when a button that has been rendered has actually gone live as result of the associated event handlers being registered.

**User-perceived PLT:** This metric captures when a user believes that a page render has finished [83, 169]. Unlike Speed Index, User-perceived PLT is not defined programmatically; instead, it is defined via user studies which empirically observe when humans think that enough of a page has rendered for the page load to be “finished.” Like Speed Index, User-perceived PLT ignores page functionality (and thus page interactivity). User-perceived PLT also cannot be automatically measured, which prevents developers from easily optimizing for the metric.

**Progressive Metrics for Object Fetches:** ObjectIndex [22] and ByteIndex [22] are progressive variants of PLT. ObjectIndex tracks the rate at which a browser fetches objects like JavaScript files and images. ByteIndex captures how many of a page’s overall bytes have been fetched at a particular moment in time. These metrics have the same limitations as the standard PLT metric—pages with below-the-fold content...
may be unfairly penalized, and pages which fetch objects quickly (but render slowly) will be unfairly rewarded.

**TTI:** Several commercial products claim to measure a page’s time-to-interactivity (TTI) [140, 150]; however, these products do not explicitly state how interactivity is defined or measured. In contrast, Google is currently working on an open standard for defining TTI [63]. The standard’s definition of TTI is still in flux. The current definition expresses interactivity in terms of time-to-first-meaningful-paint, the number of in-flight network requests, and the utilization of the browser’s main thread (which is used to dispatch GUI events, execute JavaScript event handlers, and render content). TTI defines an “interactive window” as a period in which the main thread runs no tasks that require more than 50 ms; in other words, during an interactive window, the browser can respond to user input in at most 50 ms. A page’s TTI is the maximum of:

1. the time when the **DOMContentLoaded** event has fired, and
2. the start time of the first interactive window that has at most two network requests in flight for 5 consecutive seconds.

This definition for load time has several problems. First, it could declare a page to be loaded even if the page has not rendered all of the content in the initial viewport. Second, condition (2) does not consider whether a network request is for above-the-fold, interactive content; a window with many outstanding network requests may represent an interactive page if those network requests are for below-the-fold state. Similarly, this TTI definition makes no explicit reference to the JavaScript state that supports above-the-fold event handlers, and the JavaScript state that does not. User-perceived interactivity requires the former state to be loaded, but not the latter.

**Summary:** Traditional metrics for load time fail to capture important aspects of user-perceived page readiness. PLT, ObjectIndex, and ByteIndex do not explicitly track rendering behavior, and implicitly assumes that all JavaScript state is necessary to make above-the-fold content usable. AFT, Speed Index, User-perceived PLT, and TTFP/TTFMP consider visual content, but are largely oblivious to the status of JavaScript code—the code is important only to the extent that it might update a pixel using DOM methods [123]. However, AFT, Speed Index, User-perceived PLT, and TTFP/TTFMP completely ignore event handlers (and the program state that event handlers manipulate). Consequently, these metrics fail to capture the interactive component of page usability. Google’s TTI also imprecisely captures above-the-fold, interactive state.

### 5.3 Ready Index

In this section, we formally define Ready Index (RI). Like Speed Index, RI is a progressive metric that captures incremental rendering updates. Unlike Speed Index, RI also captures the progressive loading of JavaScript state that supports interactivity.
Defining Functionality: Let $T$ be an upper-bound on the time that a browser needs to load a page’s above-the-fold state, and make that state interactive. This upper-bound does not need to be tight; we use a static value of 30 seconds (§5.5).

Let $E$ be the set of DOM elements that are visible in the viewport at $T$. For each $e \in E$, let $h(e)$ be the set of all event handlers that are attached to $e$ at or before $T$. Let $t_e$ be the earliest time at which, for all handlers $h \in h(e)$, $h$’s JavaScript function has been declared, and all JavaScript state and all DOM state that would be accessed by $h$’s execution has been loaded. Given those definitions, we express the functionality progress of $e$ as

$$F(e, t) = \begin{cases} 0 & t < t_e \\ 1 & t \geq t_e \end{cases}$$

(5.1)

Intuitively speaking, Equation 5.1 states that a DOM node is not functional until all of the necessary event handlers have been attached to the node, and the browser has loaded all of the state that the handlers would touch if executed.

Defining Visibility: An element $e$ may be the target of multiple paint events, e.g., as the browser parses additional HTML and recalculates $e$’s position in the layout. We assume that $e$ is not fully visible until its last paint completes. Let $P(e)$ be the set of all paint events that update $e$, and let $P_t(e) \subseteq P(e)$ be the paint events that have occurred by time $t$. The visibility progress of $e$ is

$$V(e, t) = \frac{|P_t(e)|}{|P(e)|}$$

(5.2)

Similar to how Speed Index computes progressive rendering scores for pixels [62], Equation 5.2 assumes that each paint of $e$ contributes equally to $e$’s visibility score. Note that $0 \leq V(e, t) \leq 1$.

Defining Readiness: Given the preceding definitions for functionality and visibility, we define the readiness of an element $e$ as

$$R(e, t) = \frac{1}{2} F(e, t) + \frac{1}{2} V(e, t)$$

(5.3)

such that the functionality and visibility of $e$ are equally weighed, and $0 \leq R(e, t) \leq 1$. The readiness of the entire page is then defined as

$$R(t) = \sum_{e \in E} A(e) R(e, t)$$

(5.4)

where $A(e)$ is the area (in pixels) that $e$ has at time $T$.

---

2The use of equal weights reflects our assumption that functionality and visibility are equally important. However, future empirical research may suggest better weighting schemes.
Figure 5-4: Vesper’s two-phase approach for measuring RI and RT. Shaded boxes indicate steps that occur during a page load. Clear boxes represent pre- and post-processing steps.

**Putting It All Together:** An element \(e\) is fully ready at time \(t\) if \(R(e, t) = 1\), i.e., if \(e\) is both fully visible and fully functional. A page’s Ready Time (RT) is thus the smallest time at which all of the above-the-fold elements are ready. A page’s Ready Index (RI) is the area above the curve of the readiness progress function. Thus, RI is equal to

\[
RI = \int_0^T 1 - \frac{R(t)}{R(T)} \, dt
\]

(5.5)

5.4 Vesper: Design

Vesper is a tool that allows a web developer to determine the RI and RT for a specific page. Vesper must satisfy three design goals. First, Vesper must produce high coverage, i.e., Vesper must identify all of a page’s interactive, above-the-fold state. Second, Vesper’s instrumentation must have minimal overhead, such that instrumented pages have RI and RT scores that are close to those of unmodified pages. Ideally, Vesper would also be browser-agnostic, i.e., capable of measuring a page’s RI and RT without requiring changes to the underlying browser.

These design goals are in tension. To make Vesper browser-agnostic, Vesper should be implemented by rewriting a page’s JavaScript code and HTML files, not through modification of a browser’s JavaScript engine and rendering pipeline; unfortunately, the most direct way to track interactive state is via heavyweight instrumentation of all reads and writes that a page makes to the JavaScript heap, the DOM, and the rendering bitmap. Vesper resolves the design tension by splitting instrumentation and log analysis across two separate page loads. Each load uses a differently-rewritten version of a page, with the first version using heavyweight instrumentation, and the second version using lightweight instrumentation. As a result, the second page load injects minimal timing distortion into the page’s true RI and RT scores. Figure 5-4 provides an overview of Vesper’s two-phase workflow. We provide more details in the remainder of this section.
5.4.1 Phase 1

The goal of this phase is to identify the subset of DOM nodes and JavaScript state that support above-the-fold interactivity.

**Element Visibility:** For most pages, only a subset of all DOM nodes will have bounding boxes that overlap with the initial viewport. Even if a node is above-the-fold, it may not be visible, e.g., due to CSS styling which hides the node. Vesper injects a JavaScript timer into the page which runs at time $T$. When the timer function executes, it traverses the DOM tree and records which nodes are visible. In the rest of the section, we refer to this timer as the Vesper timer.

**Event Handlers:** Developers make a DOM element interactive by attaching one or more event handlers to that element. For example, a `<button>` element does nothing in response to clicks until JavaScript code registers `onclick` handlers for the element. To detect when such handlers are added, Vesper shims the event registration interfaces [106]. There are two types of registration mechanisms:

- DOM elements define JavaScript-accessible properties and methods that support event handler registration. For example, assigning a function $f$ to a property like `DOMnode.onclick` will make $f$ an event handler for clicks on that DOM node. Invoking `DOMnode.addEventListener("click", f)` has similar semantics. Vesper interposes on registration mechanisms by injecting new JavaScript into a page that modifies the DOM prototypes [106]; the modified prototypes insert logging code into the registration interfaces, such that each registered handler is added to a Vesper-maintained, in-memory list of the page's handlers.

- Event handlers can also be defined via HTML, e.g., `<img src=... onload=handler()/>`. At $T$, the Vesper timer iterates through the page's DOM tree, identifying event handlers that were not registered via a JavaScript-level interface, and adding those handlers to Vesper's list.

The Vesper timer only adds a handler if the handler is attached to a visible DOM element that resides within the initial viewport.

**Event Handler State:** When a handler fires, it issues reads and writes to program state. That state may belong to JavaScript variables, or to DOM state like the contents of a `<b>` tag. As the handler executes, it may invoke other functions, each of which may touch an additional set of state. The aggregate set of state that the call chain may touch is the *functional state* for the handler. Given a DOM element $e$, we define $e$'s functional state as the union of the functional state that belongs to each of $e$'s event handlers.

If $e$ resides within the initial viewport, then $e$ is not functional until two conditions have been satisfied:

1. all of $e$'s event handlers must be registered, and
2. all of $e$'s functional state must be loaded.

At any given moment during the page load, none, either, or both of these conditions may be satisfied. For example, if $e$'s event handlers are defined in a `<script>` tag, but key functional state is defined by downstream HTML or `<script>` tags, then after evaluation of the first `<script>` tag, condition (1) is true, but condition (2) is not.
To identify a page’s functional state, Vesper instruments the HTML and JavaScript in a page, such that, when the instrumented page loads, the page will log all reads and writes to JavaScript variables and DOM state. When the Vesper timer runs, it actively invokes the event handlers that were captured by event registration shimming. As those handlers fire, their call chains touch functional state. By post-processing the page’s logs, and looking for reads and writes that occurred after the Vesper timer began execution, Vesper can identify a page’s functional state. In particular, Vesper can associate each handler with its functional state, and each DOM element with the union of the functional states of its handlers.

To fire the handlers for a specific event type like click, the Vesper timer determines the minimally-sized DOM subtree that contains all handlers for the click event. Vesper then constructs a synthetic click event, and invokes the built-in `DOMContentLoaded` method for each leaf of the subtree. This approach ensures that synthetic events follow the same dispatch path used by real events.

Some event types are logically related to a single, high-level user interaction. For example, when a user clicks a mouse button, her browser generates `mousedown`, `click`, and `mouseup` events, in that order. Vesper is aware of these semantic relationships, and uses them to guide the generation of synthetic events, ensuring a realistic sequence of handler firings.

**Implementation:** To instrument a page, Vesper could modify the browser’s renderer and JavaScript engine to track reads and writes to DOM objects and JavaScript variables. However, our Vesper prototype leverages Scout (Chapter 3) instead. Scout’s browser-agnostic approach is useful because it allows Vesper to compare a page’s Ready Index across different browser types ($5.5.5$).

The instrumentation that tracks element visibility and handler registration adds negligible overhead to the page load process. However, tracking all reads and writes to page state is more costly; experiments in §3.5 measured Scout-induced load time increases between 3%-5%. Thus, trying to calculate RI and RT directly in Phase 1 would lead to inflated estimates. To avoid this problem, we use the outputs of Phase 1 as the inputs to a second phase of instrumentation. This second phase is more lightweight, and directly calculates RI and RT.

### 5.4.2 Phase 2

In Phase 1, Vesper discovers the DOM nodes and JavaScript variables that support above-the-fold interactivity. In Phase 2, Vesper tracks the rendering progress of the above-the-fold DOM elements that were identified in Phase 1. Vesper also tracks the rate at which functional JavaScript state is created. This information is sufficient to derive RI and RT.

#### 5.4.2.1 Measuring Functionality Progress

A DOM element becomes functional when all of its event handlers have been registered, and all of the functional state for those handlers has been created. An element’s
functional state may span both the JavaScript heap and the DOM. Vesper uses
different techniques to detect when the two types of state become ready.

**JavaScript state:** By analyzing Scout logs from Phase 1, Vesper can determine
when the last write to each JavaScript variable occurs. The “last write” is defined as a
source code line and an execution count for that line. The execution count represents
the fact that a source code line can be run multiple times, e.g., if part of a loop body.

At the beginning of Phase 2, Vesper rewrites a page’s original JavaScript code,
injecting a logging statement after each source code line that generates a final write
to functional JavaScript state. The logging statement updates the execution count for
the line, and only outputs a log entry if the final write has been generated.

**DOM state:** An event handler’s functional state may also contain DOM nodes.
For example, a `keypress` handler may assume the existence of a specific DOM node
whose properties will be modified by the handler. At the beginning of Phase 2, Vesper
rewrites a page’s original HTML to output the creation time for each DOM node.
The rewriting is complicated by the fact that, when a browser parses HTML, it does
not trigger a synchronous, JavaScript-visible event upon the creation of a DOM node.
Thus, Vesper rewrites a page’s HTML to include a new `<script>` tag after *every*
original HTML tag. The new `<script>` tag logs two things: the creation of the
preceding DOM node, and the bounding boxes of all DOM nodes which exist at that
moment in the HTML parse. The `<script>` tag then removes itself from the DOM
tree (so that at any point in the HTML parse, non-Vesper code that inspects the DOM
tree will see the original DOM tree which does not contain Vesper’s self-destructing
tags). *DOM snapshots* using self-destructing JavaScript tags are by far the most
expensive part of the Phase 2 instrumentation; however, they only increase page load
times by 1.9% at the median, and 3.9% at the 95th percentile. Thus, we believe that
the overhead is acceptable.

After the initial HTML parse, DOM nodes may be created by asynchronous
event handlers. Vesper logs such creations by interposing on DOM methods like
`DOMNode.appendChild()`. This interpositioning has negligible overhead and ensures
that Vesper has DOM snapshots after the initial HTML parse.

### 5.4.2.2 Measuring Visibility Progress

DOM snapshots allow Vesper to detect when elements are created. However, a
newly-created element will not become visible until some point in the future, because
the construction of the DOM tree is earlier in the rendering pipeline than the paint
engine. Browsers do not expose layout or paint events to JavaScript code. Fortunately,
Vesper can extract those events from the browser’s debugging output [59]. Each
layout or paint message contains the bounding box and timestamp for the activity.
Unfortunately, the message does not identify which DOM nodes were affected by the
paint; thus, Vesper must derive the identities of those nodes.

After the Phase 2 page load is complete, Vesper collates the DOM snapshots and
the layout+paint debugging events, using the following algorithm to determine the
layout and paint events that rendered a specific DOM element *e*:
1. Vesper finds the first DOM snapshot that contains a bounding box for $e$. Let that snapshot have a timestamp of $t_d$. Vesper searches for the layout event that immediately precedes $t_d$ and has a bounding box that contains $e$'s bounding box. Vesper defines that layout event $L_{first}$ to be the one which added $e$ to the layout tree.

2. Vesper then rolls forward through the log of paint and layout events, starting at $L_{first}$, and tracking all paint events to $e$'s bounding box. That bounding box may change during the page load process, but any changes will be captured in the page's DOM snapshots. Thus, Vesper can determine the appropriate bounding box for $e$ at any given time.

As described in Equation 5.2, each paint event contributes equally towards $e$'s visibility score. For example, if $e$ is updated by four different paints, then $e$ is 25% visible after the first one, 50% visible after the second one, and so on.

In summary, the output of the Phase 2 page load is a trace of a page's functionality progress and visibility progress. Using that trace, and Equations 5.4 and 5.5, Vesper determines the page's RT and RI. Note that, for a given version of a page (i.e., for a particular set of HTML, CSS, and JavaScript files), Phase 1 only needs to run once, on the server-side, with Phase 2 running during live page loads on clients in the wild.

5.4.3 Discussion

The PLT metric is natively supported by commodity browsers, meaning that a page can measure its own PLT simply by registering a handler for the onload event. Newer metrics that lack native browser support require 1) browsers to install a special plugin (the SI approach [57]), or 2) page developers to rewrite content (the approach used by our Vesper prototype). Vesper is amenable to implementation via plugins or native support; either option would enable lower instrumentation overhead, possibly allowing Vesper to collapse its two phases into one.

As a practical concern, a rewriting-based implementation of Vesper must deal with the fact that a single page often links to objects from multiple origins. For example, a developer for foo.com will lack control over the bytes in linked objects from bar.com. As described in Section 5.5, our Vesper prototype uses Mahimahi [132], a web replay tool we built to record all of the content in a page; Vesper rewrites the recorded content, and then replays the modified content to a browser that runs on a machine controlled by the foo.com developer. In this manner, as with the browser plugin approach, a developer can measure RI and RT for any page, regardless of whether the developer owns all, some, or none of the page content.

All load metrics are sensitive to nondeterministic page behavior. In the context of Vesper, such behavior may result in a page having different interactive state across different page loads. For example, an event handler that branches on the return value of Math.random() might access five different DOM nodes across five different loads of the page. Even if a page's state is deterministic, Vesper's synthetic event generation (§5.4.1) is not guaranteed to exhaustively explore all possible event handler interleavings—instead, Vesper tests the most likely event sequences based on how a realistic human user would generate GUI events. Vesper could use symbolic
execution [28] to increase path coverage, but we believe that Vesper’s current level of coverage is sufficiently high for two reasons. First, from the empirical perspective, the pages in our large test corpus do not exhibit nondeterminism that results in different functional state across different loads. Second, the Vesper timer does not fire synthetic events until a page is fully loaded; thus, “unexpected” event-level race conditions arising from partially-loaded content [144] should not arise.

5.5 Evaluating Ready Index with Vesper

In this section, we compare RI and RT to three prior metrics for page load time (PLT, AFT, and Speed Index). We do not evaluate Google’s TTI because the metric’s definition is still evolving.

Across a variety of network conditions, we find that PLT overestimates the time that a page requires to become interactive; in contrast, AFT and Speed Index underestimate the time-to-interactivity (§5.5.2 and 5.5.3). These biases persist when browser caches are warm (§5.5.4). The biases also persist when the same page is loaded in different browser types (§5.5.5). Furthermore, the discrepancies between prior metrics and our interactive metrics are large, with median and 95th percentile load time estimates often differing by multiple seconds (Figures 5-3 and 5-6). Thus, Ready Index and Ready Time provide a fundamentally new way of understanding how pages load.

5.5.1 Methodology

We evaluated the various load metrics using a test corpus of 350 pages. The pages were selected from the Alexa US Top 500 list [4]. We filtered out sites that caused errors with Speedline [78], a preexisting tool for capturing SI, which we used for calibration.

To measure PLT, we recorded the time between the JavaScript navigationStart and onload events (§5.2). RT and RI were measured with Vesper; we set $T$ to 30 seconds. We also used Vesper to measure AFT and SI.³ Calibration experiments showed that Vesper’s estimates of SI were within 2.1% of Speedline’s estimates at the median, and within 3.9% at the 95th percentile.

Measuring PLT is non-invasive, since unmodified pages will naturally fire the navigationStart and onload events. Capturing the other metrics requires new instrumentation, like DOM snapshots (§5.4.2.1). To avoid measurement biases due to varying instrumentation overheads, each experimental trial loaded each page five times, and in each of the five loads, we enabled all of Vesper’s Phase 2 instrumentation, such that each load metric could be calculated. Enabling all of the instrumentation increased PLT by 1.9% at the median, and 3.9% at the 95th percentile.

We used Mahimahi [132] to record the content in each test page, and later replay the content via emulated network links. With the exception of the mobile experiments (§5.5.3), all experiments were performed on Amazon EC2 instances running Ubuntu 14.04. Unless otherwise specified, each page load used Google Chrome (v53) with a cold browser cache and remote debugging enabled to track layout and paint events.

³To compute SI, Vesper only considers element visibility, assigning zero weight to functionality.
5.5.2 Cross-metric Comparisons

On computationally-powerful devices like desktops and laptops, network latency (not bandwidth) is the primary determinant of how quickly a page loads [1, 16, 128, 157]. So, our first set of tests used a t2.large EC2 VM with a fixed bandwidth of 12 Mbits/s, but a minimum round-trip latency that was drawn from the set \{25 ms, 50 ms, 100 ms, 200 ms\}. These emulated network conditions were enforced by Mahimahi.

Figure 5-3 summarizes the results for PLT, RT, and AFT. Recall that these metrics are non-progressive, i.e., they express a page’s load time as a single number that represents when the browser has “completely” loaded the page (for some definition of “completely”). As expected, PLT is higher than RT because PLT requires all page state, including below-the-fold state, to be loaded before a page load is finished. Also as expected, AFT is lower than RT, because AFT ignores the load status of JavaScript code that is necessary to make visible elements functional.

The surprising aspect of the results is that the differences between the metrics are so noticeable. As shown in Figures 5-2a and 5-3, the differences are large in terms of percentage (24.0%–64.3%); more importantly, the differences are large in terms of absolute magnitude, equating to hundreds or thousands of milliseconds. For example, with a round-trip latency of 50 ms, RT and PLT differ by roughly 900 ms at the median, and by 1.4 seconds at the 95th percentile. For the same round-trip latency, RT and AFT differ by approximately 600 ms at the median, and by 1.1 seconds at the 95th percentile.

The discrepancies increase as RTTs increase. This observation is important, because cellular and residential networks often have RTTs that exceed 100 ms [5, 77]. For example, in our emulated network with an RTT of 100 ms, RT differed from PLT by

<table>
<thead>
<tr>
<th>RTT</th>
<th>Ready Index</th>
<th>Speed Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 ms</td>
<td>714 (1522)</td>
<td>568 (1027)</td>
</tr>
<tr>
<td>50 ms</td>
<td>1759 (3846)</td>
<td>1325 (3183)</td>
</tr>
<tr>
<td>100 ms</td>
<td>2737 (6174)</td>
<td>2054 (4549)</td>
</tr>
<tr>
<td>200 ms</td>
<td>4252 (9719)</td>
<td>3071 (6913)</td>
</tr>
</tbody>
</table>

Figure 5-6: Median (95th percentile) load time estimates (see Section 5.2 for unit definitions). Results used our 350 page corpus and a 12 Mbits/s link.
2.2 seconds at the median; RT differed from AFT by 1 second at the median. From the perspective of a web developer, the differences between RT and AFT are particularly important. Users frequently assume that a visible element is also functional. However, visibility does not necessarily imply functionality, and interactions with partially-functional elements can lead to race conditions and broken page behavior [144]. In Section 5.6, we describe how developers can create incrementally-interactive pages that minimize the window in which a visual element is not interactive.

Figure 5-5 compares the RT, PLT, and AFT values for each page in our 350 site corpus. Pages are sorted along the x-axis in ascending AFT order. Figure 5-5 vividly demonstrates that PLT is an overly conservative definition for user-perceived notions of page readiness. The spikiness of the RT line also demonstrates that pages with similar AFT values often have very different RT scores. For example, consider an emulated link with a 100 ms round-trip time. Sites 200 (mashable.com) and 201 (overdrive.com) have AFT values of 3099 ms and 3129 ms, respectively. However, the sites have RT values of 4418 ms and 3970 ms, a difference of over 400 ms. In Section 5.5.6, we explain how the relationships between a page’s HTML, CSS, and JavaScript cause divergences in RT and AFT.

Figures 5-6 and 5-7 compare the two progressive metrics. The results mirror those for the non-progressive metrics. A page’s SI is lower than its RI, because SI does not consider the load status of JavaScript code that supports interactivity. Furthermore, pages with similar SIs often have much different RIs.

5.5.3 Mobile Page Loads

Mobile browsers run on devices with limited computational resources. As a result, mobile page loads are typically compute-bound, with less sensitivity to network latency [16, 157]. To explore RI and RT on mobile devices, we USB-tethered a Nexus 5 phone running Android 5.1.1 to a Linux desktop machine that ran Mahimahi. Mahimahi emulated a Verizon LTE cellular link [180] with a 100 ms RTT. The phone used Google Chrome v53 to load pages from a test corpus. The corpus had the same 350 sites from our standard corpus, but used the mobile version of each site if such a version was available. Mobile sites are reformatted to fit within smaller screens, and to contain fewer bytes to avoid expensive fetches over cellular networks.

As shown in Figure 5-8, mobile page loads exhibit the same trends that we observed
on more powerful client devices. For example, the median PLT is 35.2% larger than the median RT; the median RI is 29.7% larger than the median Speed Index. These differences persist even when considering only the mobile-optimized pages in our corpus. For that subset of pages, the median PLT is 27.4% larger than the median RT, and the median RI is 25.3% larger than the median Speed Index.

### 5.5.4 Browser Caching

Our prior experiments used cold browser caches, meaning that, to load a particular site, a browser had to fetch each of the constituent objects over the network. However,
users often visit the same page multiple times; different sites also share objects. Thus, in practice, browsers often have warm caches that allow some object fetches to be satisfied locally.

To determine how warm caches affect page loads, we examined the HTTP caching headers [47] for each object in our corpus. For each object that was marked as cacheable, we rewrote the headers to indicate that the object would be cacheable forever. We then loaded each page in our corpus twice, back to back: the first load populated the cache, and the second one leveraged the pre-warmed cache. Figure 5-9 shows the results for a desktop browser which used a 12 Mbits/s link with a 100 ms RTT.

Figure 5-9: Page loads with warm browser caches. The desktop browser used a 12 Mbits/s link with a 100 ms RTT.
As expected, pages load faster when caches are warm. However, the general trends from Section 5.5.2 still hold. For example, the median PLT is 38.2% larger than the median RT, which is 26.0% larger than the median AFT. Similarly, median RI is 20.4% larger than median Speed Index. The correlations between various metrics also continue to be noisy. For example, SI increases from 1147 ms to 1168 ms between sites 134 (duckduckgo.com) and 135 (nexusmods.com); however, RI decreases from 1601 ms to 1228 ms.

5.5.5 Cross-Browser Comparisons

Different browsers are built in different ways. As shown in Figure 5-10, those architectural variations impact page load times. Figure 5-10 compares Ready Index on Chrome v53 and Opera v42. Chrome and Opera both use the WebKit rendering engine and the V8 JavaScript runtime. However, the browsers’ code is sufficiently different to produce noticeable biases in RI values: Chrome’s RI values are 6.5% lower at the median, and 11.9% lower at the 95th percentile.

To understand the causes for such discrepancies, developers must analyze the steps that a browser takes to load a page. Tools like WProf [175] and the built-in Chrome debugger allow developers to examine coarse-grained interactions between high-level activities like HTML parsing, screen painting, and JavaScript execution. However, Vesper’s logs describe how interactive state loads at the granularity of individual JavaScript variables and DOM nodes. For example, Vesper allows a developer to
associate a dynamically-created text input with the specific code that creates the input and registers event handlers for the input; Vesper also tracks the JavaScript variables that are manipulated by the execution of the event handlers. None of this information is explicitly annotated by developers, nor should it be: for a large, frequently-changing site, humans should focus on the correct implementation of desired features, not the construction of low-level bookkeeping details about data and code dependencies. Thus, automatic extraction of these dependencies is crucial, since, as we demonstrate in Section 5.6, a fine-grained understanding of those dependencies is necessary to minimize a page’s time-to-interactivity.

5.5.6 Case Studies

Figure 5-11 uses two randomly-selected pages to demonstrate how interactivity evolves. Figure 5-11a describes the homepage for Bank of America, whereas Figure 5-11b describes the homepage for WebMD. Using the terminology from Section 5.3, each graph plots the visual progression of the page \( \sum_{e \in E} V(e, t) A(e) \) and the readiness progression of the page \( R(t) \); in the graphs, each data point is normalized to the range \([0.0, 1.0]\). At any given moment, a page’s readiness progression is less than or equal to its visual progression, since visual progression does not consider the status of functional state.

The gaps between the red and blue curves indicate the existence of visible, interactive DOM elements that are not yet functional. If users try to interact with such elements, then at best, nothing will happen; at worst, an incomplete set of event handlers will interact with incomplete JavaScript and DOM state, leading to erroneous page behavior. For example, the Bank of America site contains a text input that supports autocompletion. With RTTs of 100 ms and above, we encountered scenarios in which the input was visible but not functional. In these situations, we manually verified that a human user could type into the text box, have no autosuggestions appear, and then experience the text disappear and reappear with autosuggestions as the page load completed.

Both the red and blue curves contain stalls, i.e., time periods in which no progress is made. For example, both pages exhibit a lengthy stall in their visual progression—for roughly a second, neither page updates the screen. Both pages also contain stretches that lack visual progress or readiness progress. During these windows, a page is not executing any JavaScript code that creates interactive state.

Functionality progression stalls when the \(<script>\) tags supporting functionality have not been fetched, or have been fetched but not evaluated. Visual progression may stall for a variety of reasons. For example, the browser might be blocked on network fetches, waiting on HTML data so that new tags can be parsed and rendered. Browsers also use a single thread for HTML parsing, DOM node rendering, and JavaScript execution; thus, executing a \(<script>\) tag blocks parsing and rendering of downstream HTML. As described in Section 5.6, developers can use automated tools to minimize these stalls and improve a page’s Ready Time and Ready Index.
Figure 5-11: Exploring how visibility and functionality evolve for two different pages. The client had a 12 Mbits/s link with an RTT of 100 ms. Remember that a progressive metric like Ready Index is calculated by examining the area that is above a curve.

5.6 Optimizing for Interactivity

To minimize a page’s Ready Time and Ready Index, browsers must fetch and evaluate objects in a way that prioritizes interactivity. In particular, a browser should:

1. maximize utilization of the client’s network connection;
2. prioritize the fetching and evaluating of HTML files that define above-the-fold DOM elements;
3. prioritize the fetching and evaluating of `<script>` tags that generate interactive, above-the-fold state; and
4. respect the semantic dependencies between a page’s objects.
By maximizing network utilization (Goal (1)), a browser minimizes the number of CPU stalls that occur due to synchronous network fetches; ideally, a browser would fetch each piece of content before that content is desired by a parsing/evaluation engine. Goals (2) and (3) directly follow from the definitions for page readiness in Section 5.3. However, Goal (4) is in tension with the others: fetching and evaluating objects in a way that satisfies Goals (1), (2), and (3) may break page functionality. For example, two JavaScript libraries may have shared state, like a variable that is written by the first library and read by the second. Invalid reads and other problems will arise if a browser evaluates the two libraries "out-of-order" with respect to the lexical order of their <script> tags in the page’s HTML (§4.2). Polaris (Chapter 4) overcomes these “hidden dependencies” by generating a page’s explicit dependency graph with Scout (Chapter 3), and rewriting the page such that it is self-assembling and satisfies Goals (1) and (4).

At any given moment in a page load, the dynamic critical path is the path in the dependency graph that has the largest number of unfetched objects. The default Polaris scheduler prioritizes the fetching of objects along the dynamic critical path. This policy minimizes PLT, but may increase or decrease RT, depending on whether interactive, above-the-fold state is created by objects along the dynamic critical path.

We created a new scheduling policy, called OPT-RI (“optimize RI”), which prioritizes the loading of interactive content. Let $O_{\text{interactive}}$ be the objects (e.g., HTML files, JavaScript files) that Vesper identifies as generating interactive, above-the-fold state. Given $O_{\text{interactive}}$ and the dependency graph from Scout, OPT-RI assigns node weights of zero to nodes that do not reside in $O_{\text{interactive}}$; for a node in $O_{\text{interactive}}$, OPT-RI finds all of the above-the-fold elements that the node affects, and then weights the node by the fraction of the initial viewport area that those elements cover. During the actual page load, the OPT-RI scheduler prioritizes objects along the weighted dynamic critical path.
Table 5-13: Median (95th percentile) load time improvements using our custom Polaris schedulers and the default one (OPT-PLT). Results used our entire 350-page corpus. Loads were performed on a desktop Chrome browser that had a 12 Mbits/s link with an RTT of 100 ms; the performance baseline was a regular (i.e., non-Polaris) page load. The best scheduler for each load metric is highlighted.

<table>
<thead>
<tr>
<th>Weights</th>
<th>PLT</th>
<th>RT</th>
<th>AFT</th>
<th>SI</th>
<th>RI</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPT-PLT</td>
<td>36% (51%)</td>
<td>13% (22%)</td>
<td>-4% (5%)</td>
<td>-7% (4%)</td>
<td>8% (17%)</td>
</tr>
<tr>
<td>OPT-RI</td>
<td>23% (34%)</td>
<td>32% (48%)</td>
<td>15% (26%)</td>
<td>12% (20%)</td>
<td>29% (35%)</td>
</tr>
<tr>
<td>OPT-SI</td>
<td>10% (19%)</td>
<td>18% (31%)</td>
<td>27% (39%)</td>
<td>18% (28%)</td>
<td>14% (23%)</td>
</tr>
</tbody>
</table>

We also defined OPT-SI, which only considers visual progress. Nodes that do not lead to the creation of visible, above-the-fold DOM elements receive a weight of zero. For each remaining node, OPT-SI finds the DOM elements that the node influences, and assigns a node weight that is proportional to the fraction of the viewport that the elements cover. OPT-SI will not prioritize JavaScript files that only define event handler state; however, OPT-SI will prioritize JavaScript files that dynamically create above-the-fold content via DOM methods like `document.appendChild()`. Figure 5-12 provides an example of a real dependency graph, and the nodes that are prioritized by the various schedulers.

Figure 5-13 compares the performance of the schedulers. OPT-RI and OPT-SI reduce all load metrics, but the targeted metrics decrease the most. Thus, sites that want to decrease time-to-interactivity must explicitly target RI and RT, not preexisting metrics like SI and PLT. For example, consider the search button in Figures 5-2b and 5-2c. OPT-RI makes the button interactive 1.5 seconds earlier than OPT-SI. Differences of that magnitude have significant impacts on user satisfaction and site revenue [24, 37, 184].

As shown in Figure 5-13, OPT-RI reduces RI by a median of 29%, and RT by a median of 32%; PLT, AFT, and SI also drop, but not as much (by 23%, 15%, and 12%, respectively). Interestingly, the default Polaris scheduler (OPT-PLT) improves PLT, RT, and RI, but actually hurts AFT and SI by -4% and -7% at the median. The reason is that JavaScript files often form long dependency chains; evaluating one JavaScript file in the chain leads to the fetching and evaluation of additional JavaScript files. These long dependency chains tend to lie along the dynamic critical paths that are preferentially explored by OPT-PLT. By focusing on those chains, OPT-PLT increases the speed at which event handling state is loaded. However, this approach defers the loading of content in short chains. Short chains often contain images, since images (unlike HTML, CSS, and JavaScript) cannot trigger new object fetches. Deferring image loading hurts AFT and SI, though RT and RI improve, and the likelihood of broken user interactions (§5.5.2 and §5.5.6) decreases.

**User Study 1: Do User-perceived Rendering Times Actually Change?** The results from Figure 5-13 programmatically compare OPT-PLT, OPT-SI, and OPT-RI. We now evaluate how the differences between these optimization strategies are perceived by real users. We performed a user study in which 73 people judged the load
times of 15 randomly-selected sites from our corpus, each of which had three versions (one for each optimization strategy). We used a standard methodology for evaluating user-perceived load times [83, 169]. We presented each user with 10 randomly-selected pages that employed a randomly-selected optimization target; we injected a JavaScript `keypress` handler into each page, so that users could press a key to log the time when they believed the page to be fully loaded. In all of the user studies, content was served from Mahimahi on a Macbook Pro, using an emulated 12 Mbits/s link with a 100 ms RTT.

Unsurprisingly, users believed that OPT-PLT resulted in the slowest loads for all 15 pages. However, OPT-SI did not categorically produce the lowest user-perceived rendering times; users thought that OPT-RI was the fastest for 4 pages, and OPT-SI was the fastest for 11. Across the study, median (95th percentile) user-perceived rendering times with OPT-RI were within 4.7% (10.9%) of those with OPT-SI. Furthermore, the performance of OPT-RI and OPT-SI were closer to each other than to that of OPT-PLT. At the median (95th percentile), OPT-RI was 14.3% (25.3%) faster than OPT-PLT, whereas OPT-SI was 17.4% (32.9%) faster.

These results indicate that a page that only wants to decrease rendering delays should optimize for SI. However, optimizing for RI results in comparable decreases in rendering time. Our next user study shows that optimizing for RI also decreases user-perceived time-to-interactivity.

**User Study 2: Does OPT-RI Help Interactive Sites?** Unlike the first user study, our second one asked users to interact with five well-known landing pages: Amazon, Macy’s, Food Network, Zillow, and Walmart. For each site, users completed a site-specific task that normal users would be likely to perform. For example, on the Macy’s page, users were asked to hover over the “shopping bag” icon until the page displayed a pop-up icon that listed the items in the shopping bag. On the Walmart site, users were asked to search for “towels” using the autocompleting text input at the top of the page; they then had to select the autocompleted suggestion. To avoid orientation delays, users were shown all five pages and the location of the relevant interactive elements at the beginning of the study. This setup emulated users who were returning to frequently-visited sites.

The study had 85 users interact with three different versions of each page: a default page load, a load that was optimized with OPT-SI, and one that was optimized with OPT-RI. For each page, users were presented with the three variations in a random order and were unaware of which variant they were seeing. Users were asked to select the variant that enabled them to complete the given task the fastest; if users felt that there was no perceivable difference between the loads, users could report “none.”

As shown in Figure 5-14, OPT-RI was overwhelmingly preferred, with 83% of users believing that OPT-RI led to the fastest time-to-interactivity. For example, on the Macy’s page, OPT-RI made the shopping bag icon fully interactive 1.6 seconds faster than the default page load, and 2.1 seconds faster than the OPT-SI load. Time-to-interactivity differences of these magnitudes are easily perceived by humans. Thus, for pages with interactive, high-priority content, OPT-RI is a valuable tool for reducing time-to-interactivity (as well as the time needed to fully render the page). Optimizing for interactivity is particularly important for web browsing atop mobile
<table>
<thead>
<tr>
<th>Load method</th>
<th>Preference %</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPT-RI</td>
<td>83%</td>
</tr>
<tr>
<td>OPT-SI</td>
<td>4%</td>
</tr>
<tr>
<td>Default load</td>
<td>7%</td>
</tr>
<tr>
<td>None</td>
<td>6%</td>
</tr>
</tbody>
</table>

Figure 5-14: The results of our second user study. OPT-RI leads to human-perceivable reductions in the completion times for interactive tasks.

devices with poor network connectivity. In these scenarios, users often desire to interact with pages as soon as relevant content becomes visible [67].
Chapter 6

Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs

6.1 Overview

Mobile browsing now generates more HTTP traffic than desktop browsing [54]. On a smartphone, 63% of user focus time, and 54% of overall CPU time, involves a web browser [190]; mobile browsing is particularly important in developing nations, where smartphones are often a user’s sole access mechanism for web content [30, 69]. Thus, in mobile settings, page load time (regardless of its definition) is not the only metric of interest—optimizing bandwidth consumption and energy usage are also important. Improving page load time remains important because users are frustrated by pages that take more than a few seconds to load [41, 50, 101, 173]. Reducing bandwidth overhead allows users to browse more pages without violating data plan limits. Reducing energy consumption improves the overall lifetime of the device, because web browsing is a significant drain on battery power [25, 29, 167, 189, 190].

In this chapter, we describe Prophecy, a new system for improving all three aspects of a mobile page load. A Prophecy web server precomputes much of the information that a mobile browser would generate during a traditional page load. In particular, a Prophecy server precomputes the JavaScript state and the DOM state that belongs to a loaded version of a frame. The precomputed JavaScript heap and DOM tree represent graphs of objects; however, one of Prophecy’s key insights is that this state should be transmitted to clients in the form of write logs, not serialized graphs. At a high level, a write log contains one write operation per variable in the frame’s load-time state. By returning write logs for each variable’s final state, instead of returning traditional, unprocessed HTML, CSS, and JavaScript, the browser can elide slow, energy-intensive computations involving JavaScript execution and graphical layout/rendering. Conveniently, Prophecy’s write logs for a frame are smaller than the frame’s original content, and can be fetched in a single HTTP-level RTT. Thus, Prophecy’s precomputation also decreases bandwidth consumption and the number of round trips needed to build a frame.

Earlier attempts at applying precomputation to web sites have suffered from
significant practical limitations (§6.5), in part because these systems used serialized graphs instead of write logs. Serialized graphs hide data flows that write logs capture; analyzing these data flows is necessary to perform many optimizations. For example, Prepack [42] cannot handle DOM state, and is unable to elide computation for some kinds of common JavaScript patterns. Shandian [177] does not support caching for the majority of a page’s content, does not support immediate page interactivity (§6.2.5), and does not work on unmodified commodity browsers; furthermore, Shandian exposes all of a user’s cookies to a single proxy, raising significant privacy concerns. In contrast, Prophecy works on commodity browsers, handles both DOM and JavaScript state, preserves traditional same-origin policies about cookie security, and supports byte-granularity caching (which is better than HTTP’s standard file-level caching scheme). Prophecy can also prioritize the loading of interactive state; this feature is important for sites that load over high-latency links, and would otherwise present users with rendered GUIs that may not actually be functional. Many of Prophecy’s advantages are enabled by having fine-grained, variable-level understanding of how a page load unfolds.

Experiments with a Nexus 6 phone, loading 350 web pages on real WiFi and LTE networks, reveal Prophecy’s significant benefits: median energy usage drops by 36%, median bandwidth consumption decreases by 21%, and median page load time decreases by 53% (2.8 seconds). Prophecy also helps page loads on desktop browsers, reducing median bandwidth usage by 18%, and median page load time by 38% (0.8 seconds). These benefits are $2.2 \times -4.6 \times$ better than those enabled with Polaris (Chapter 4) in mobile settings. Thus, Prophecy represents a significant advance in web optimization.

### 6.2 Prophecy: Design

Figure 7-2 shows the high-level design of Prophecy. Users employ an unmodified browser to fetch and evaluate a Prophecy page. A single page consists of one or more frames (Chapter 2); content providers who wish to accelerate their frame loads must run server-side Prophecy code that handles incoming HTTP requests for the relevant frames. The server-side Prophecy code uses a headless browser\(^1\) to load the requested frame. The frame consists of individual objects like HTML files, JavaScript files, and images; Prophecy rewrites HTML and JavaScript before it is passed to the headless browser, injecting instrumentation which tracks how the frame manipulates the JavaScript heap and the DOM tree.

After the headless browser has loaded the frame, Prophecy uses the resulting log to create a post-processed version of the frame. The post-processed version contains four items:

- a **write log for the JavaScript heap**, containing an ordered set of writes that a client executes to recreate the frame’s final heap state;

\(^1\)A headless browser lacks a GUI, but otherwise performs the normal duties of a browser, parsing and rendering HTML, executing JavaScript, and so on.
Figure 6-1: The Prophecy architecture.

- a write log for the DOM, containing HTML tags with precomputed styles, such that the client can immediately resurrect the DOM with minimal layout and rendering overheads;
- an image prefetch log, describing the images that the browser should fetch in parallel with the construction of the JavaScript heap and the DOM; and
- the Prophecy resurrection library, a small piece of JavaScript code which orchestrates client-side reconstruction of the frame (§6.2.2), optimizing the reconstruction for a particular load metric (§6.2.5).

During a warm cache load (§6.2.3), the three logs are diffs with respect to the client’s cached logs. By applying the diffs and then executing the patched logs, a client fast-forwards its view of the frame to the latest version.

### 6.2.1 Generating a Prophecy Frame

Prophecy enables web acceleration at the granularity of a frame. However, web developers create the content for a particular frame in a Prophecy-agnostic way, using a normal workflow to determine which objects (e.g., HTML, CSS, JavaScript, and images) should belong in a frame. The process of transforming the normal frame into a Prophecy variant is handled automatically by Prophecy. The transformation can happen online (i.e., at the time of an HTTP request for the frame), or offline (i.e., before such a request has arrived). In this section, we describe the transformation process; later, we describe the trade-offs between online and offline transformation (§6.2.4).
After fetching the frame’s HTML, Prophecy’s server-side component loads the frame in a headless browser. As the frame loads, Prophecy tracks the reads and writes that the frame makes to the JavaScript heap and to the DOM. Prophecy’s design is agnostic as to how this tracking is implemented. Our concrete Prophecy prototype uses Scout (Chapter 3) to inject logging instrumentation into the loaded frame, but Prophecy is compatible with in-browser solutions that use a modified JavaScript engine and renderer to log the necessary information. Regardless, once the frame has loaded, Prophecy analyzes the reads and writes to create the three logs which represent the Prophecy version of a frame.

The JavaScript write log: This log, expressed as a series of JavaScript statements, contains a single \( \text{lhs} = \text{rhs} \); statement for each JavaScript variable that was live at the end of the frame load. The set of operations in the write log is a subset of all writes observed in the original log—only the final write to each variable in the original log is preserved. The write log first creates top-level global variables that are attached to the `window` object (see Figure 2-1); then, the log iteratively builds objects at greater depths from the `window` object. The final write log for the JavaScript heap does not create DOM nodes, so any JavaScript object properties that refer to DOM state are initially set to `undefined`.

The write log must pay special attention to functions. In JavaScript, a function definition can be nested within an outer function definition. The inner function becomes a closure, capturing the variable scope of the outer function. To properly handle these functions, Prophecy rewrites functions to explicitly expose their closure scope\(^{104, 108, 177}\). At frame load time on the server, this allows Prophecy’s write tracking to explicitly detect which writes involve a function’s closure state. Later, when a mobile browser needs to recreate a closure function, the replayed write log can simply create the function, then create the scope object, and then write to the scope object’s variables.

The write log for the JavaScript heap does not contain entries for native objects that belong to the DOM tree. However, the write log does contain entries for the other native objects in a frame. For example, the log will contain entries for regular expressions (RegExps) and timestamps (Dates). Generally speaking, the write log creates native objects in the same way that it creates normal objects, i.e., by calling \( \text{lhs} = \text{new ObjClass()} \) and then assigning to the relevant properties via one or more statements of the form \( \text{lhs.prop} = \text{rhs} \). However, Prophecy does not attempt to capture state for in-flight network requests associated with objects like `XMLHttpRequests`; instead, Prophecy waits for such connections to terminate before initiating the frame transformation process.

The DOM write log: Once Prophecy’s server-side component has loaded a frame, Prophecy generates an HTML string representation for the frame using the browser’s predefined `XMLSerializer` interface. Importantly, the HTML string that is returned by `XMLSerializer` does not contain styling information for individual tags; the string merely describes the hierarchical tag structure. To extract the style information,
Prophecy iterates over the DOM tree, and uses `window.getComputedStyle(domNode)` to calculate each node’s style information. Prophecy then augments the frame’s HTML string with explicit style information for each tag. For example, a tag in the augmented HTML string might look like `<div style='border-bottom-color: rgb(255, 0, 0);border-left-color: rgb(255, 0, 0);'>`. Prophecy modifies all CSS-related tags in the augmented HTML string, deleting the bodies of inline `<style>` tags, and setting the `href` attributes in `<link rel='stylesheet'>` tags to point to the empty string (preventing a network fetch). Prophecy also modifies the `src` attribute of `<script>` tags to point to the empty string (since all JavaScript state will be resurrected using the JavaScript write log).

The augmented HTML string is the write log for the DOM, containing precomputed style information for each DOM node. Note that the style data may have been set by CSS rules, or by JavaScript code via the DOM interface. Also, some of the DOM nodes in the write log may have been dynamically created by JavaScript (instead of being statically created by the frame’s original HTML). Prophecy’s server-side component represents the DOM write log as a JavaScript string literal.

**The image prefetch log:** This log is a JavaScript array that contains the URLs for the images in the loaded frame. The associated `<img>` tags may have been statically declared in the frame’s HTML, or dynamically injected via JavaScript. Note that the write log for the DOM tree contains the associated `<img>` tags; however, as we explain in Section 6.2.2, the image prefetch list allows the mobile browser to keep its network pipe busy as the CPU is parsing HTML and evaluating JavaScript.

The Prophecy frame consists of the three logs from above, and a small JavaScript library which uses the logs to resurrect the frame (§6.2.2). Since the three logs are expressed as JavaScript variables, the Prophecy server can just add those variables to the beginning of the resurrection library. So, the Prophecy frame only contains one HTML tag—a single JavaScript tag with inline content.

### 6.2.2 Loading a Prophecy Frame

A mobile browser receives the Prophecy frame as an HTTP response, and starts to execute the resurrection library. The library first issues asynchronous `Image()` requests for the URLs in the image prefetch log. As the browser fetches those images in the background, the resurrection library builds the frame in three phases.

**Phase 1 (DOM Reconstruction):** The resurrection library passes the DOM write log to the browser’s preexisting `DOMParser` interface. `DOMParser` returns a `document` object, which is a special type of DOM node that represents an entire DOM tree. The resurrection library updates the frame’s live DOM tree by splicing in the `<head>` and `<body>` DOM subtrees from the newly created `document`. After these splice operations

---

2 Prophecy uses additional logic to ensure that the extracted style information includes any default tag styles that apply to the DOM node. These default styles are not returned by `getComputedStyle()`.
complete, the entire DOM tree has been updated; note that the browser has avoided
two the traditional computational overheads associated with layout and rendering,
since the resurrection library injected a pre-styled DOM tree which already contains
the side effects of load-time JavaScript calls to the DOM interface. As the browser
receives the asynchronously prefetched image data, the browser injects the pixels into
the live DOM tree as normal, without assistance from the resurrection library; note
that the browser will not “double-fetch” an image if, at DOM reconstruction time, the
browser encounters an `<img>` tag whose prefetch is still in-flight.

**Phase 2 (JavaScript Heap Reconstruction):** Next, the resurrection library
executes the assignments in the write log for the JavaScript heap. Each write operation
is just a regular JavaScript assignment statement in the resurrection library's code.
Thus, the mobile browser naturally recreates the heap as the browser executes the
middle section of the library.

**Phase 3 (Fixing Cross-references):** At this point, the DOM tree and the
JavaScript heap are largely complete. However, DOM objects can refer to JavaScript
heap objects, and vice versa. For example, an application-defined JavaScript object
might have a property that refers to a specific DOM node. As another example, the
event handler for (say) a mouse click is an application-defined JavaScript function
that must be attached to a DOM node via `DOMnode.addEventListener(evtType, func)`. In Phase 3, the resurrection library fixes these dangling references using
information in the JavaScript write log. During the initial logging of reads and
writes in the frame load (§6.2.1), Prophecy assigned a unique id to each JavaScript
object and DOM node that the frame created. Now, at frame reconstruction time
on the mobile browser, the resurrection library uses object ids to determine which
object should be used to resolve each dangling reference. As hinted above, the
library must resolve some dangling references in DOM nodes by calling specific DOM
functions like `addEventListener()`. The library also needs to invoke the relevant
timer registration functions (e.g., `setTimeout(delay, callback)`) so that timers
are properly resurrected. ³

At the end of Phase 3, the frame load is complete, having skipped intermediate
JavaScript computations, as well as intermediate styling and layout computations for
the DOM tree. A final complication remains: what happens if, post-load, the frame
dynamically injects a new DOM node into the DOM tree? Remember that Prophecy’s
write log for the DOM tree contains no inline `<style>` data, nor does it contain `href`
attributes for `<link rel='stylesheet'>` tags (§6.2.1). So, as currently described, a
Prophecy frame will not assign the proper styles to a dynamically created DOM node.

To avoid this problem, the resurrection library contains a string which stores
all of the frame’s original CSS data. The resurrection code also shims [106] DOM
interfaces like `DOMnode.appendChild(c)` which are used to dynamically inject new

³During the instrumented frame load on the server, Prophecy shims timer registration interfaces
to track timer state [106].
DOM content. Upon the invocation of such a method, the Prophecy shim examines the frame’s CSS rules (and the live style characteristics of the DOM tree) to apply the appropriate inline styles to the new DOM node. After applying those styles, Prophecy can safely inject the DOM node into the DOM tree.

### 6.2.3 Caching, Personalization, and Cookies

To enable frame content to be personalized, we extend the approach from Sections 6.2.1 and 6.2.2. At a high level, when a server receives an HTTP request for a frame, the server looks inside the request for a cookie that bears a customization id. If the server does not find such a cookie, then the server assumes that the mobile browser has a cold cache; in this case, the server returns the Prophecy frame as described in Section 6.2.1, placing a cookie in the HTTP response which describes the frame’s customization id. If the server does find a customization id in the HTTP request, then the server assumes that the client possesses cached write logs for the frame. The server computes the write logs for the latest customization version of the frame. The server then calculates the diffs between the latest write logs and the ones that are cached on the phone. Finally, the server returns the diffs to the mobile browser. The mobile browser applies the diffs to the cached write logs, and then recreates the frame as described in Section 6.2.2.

To efficiently track the client-side versions of a frame, the server must store some metadata for each frame:

- The server stores a baseline copy of the three write logs for a frame. Denote those logs baselineJS, baselineHTML, and baselineimages. These logs correspond to a default version of the frame that has not been customized.
- For each version \( v \) of the frame that has been returned to a client, the server stores three diffs, namely, \( \text{diff}(\text{baselineJS}, \text{customization}_{v, \text{JS}}) \), \( \text{diff}(\text{baselineHTML}, \text{customization}_{v, \text{HTML}}) \), and \( \text{diff}(\text{baselineimages}, \text{customization}_{v, \text{images}}) \). The server stores these diffs in a per-frame table, using \( v \) as the key.

“Customization” has a site-specific meaning. For example, many sites return the same version of a frame to all clients during some epoch \( t_{\text{start}} \) to \( t_{\text{end}} \). In this scenario, a new version is generated at the start of a new epoch. In contrast, if a site embeds unique, per-user content into a frame, then a version corresponds to a particular set of write logs that were sent to a particular user.

In the cold cache case, the server generates the appropriate write logs for the latest frame version \( v \), and then diffs the latest logs against the baseline logs. The server stores the diffs in \( \text{diffTable}[v] \), and then returns the latest write logs to the client as in Section 6.2.1, setting the customization id in the HTTP response to \( v \). The client rebuilds the frame as in Section 6.2.2, and then stores the three write logs in DOM storage.

In the warm cache scenario, the server extracts \( v \) from the HTTP request, finds the associated diffs in \( \text{diffTable}[v] \), and then applies the diffs to the baseline versions of the write logs. This allows the server to reconstruct the write logs that the client possesses for the old copy of \( v \). The server then generates the write logs for the latest
incarnation of \( v \), and diffs the latest write logs against the client-side ones. The server updates \( \text{diffTable}[v] \) appropriately, and then returns the diffs to the mobile browser. The mobile browser reads the cached write logs from DOM storage, applies the diffs from the server, and then rebuilds the frame using the latest write logs. Finally, the browser caches the latest write logs in DOM storage.

Note that the mobile phone and the server can get out-of-sync with respect to cache state. For example, the server might reboot or crash, and lose its per-frame \( \text{diffTables} \). The user of the mobile browser could also delete the phone’s DOM storage or cookies. Fortunately, desynchronization is only a performance issue, not a correctness one, because desynchronization can always be handled by falling back to the cold cache protocol. For example, suppose that a client clears its DOM storage, but does not delete its cookie. The server will send diffs, but then the client-side resurrection library will discover that no locally-resident write logs exist. The library will delete the cookie, and then refresh the page by calling \( \text{window.location.reload()} \), initiating a cold cache frame load.

To minimize the storage overhead for \( \text{diffTable} \), each frame’s baseline should share a non-trivial amount of content with the various customized versions of the frame. Choosing good baselines is easy for sites in which, regardless of whether a user is logged in, the bulk of the site content is the same. For frames with large diffs between customized versions, and a large number of versions, servers can minimize \( \text{diffTable} \) overhead by breaking a single frame into multiple frames, such that highly-customized content lives in frames that are always served using the cold cache protocol (and store no server-side information in a \( \text{diffTable} \)). Less-customized frames can enable support for warm Prophecy caches, and communicate with the highly-dynamic frames using \( \text{postMessage}() \).

For frames that enable the warm-cache protocol, the server may have to periodically update the associated baselines, to prevent diffs in \( \text{diffTable} \) from growing too large as the latest frame content diverges from that in the baselines. One simple pruning strategy is to generate a new baseline once the associated diffs get too large in terms of raw bytes, or as a percentage of the baseline object’s size. After updating a baseline, the server must either discard the associated diffs, or recalculate them with respect to the new baseline.

### 6.2.4 Online versus Offline Transformation

Prophecy’s server-side code transforms a frame’s HTML, CSS, JavaScript, and images into three write logs. The transformation process can happen online or offline. In the online scenario, the server receives an HTTP request for a frame, and then loads and post-processes the frame synchronously, generating the associated write logs on-the-fly. In the offline scenario, the server periodically updates the write logs for each frame, so that, when a client requests a frame, the server already possesses the relevant write logs.

Each approach has trade-offs. Offline processing reduces the client-perceived fetch time for the frame, since the instrumented version of the regular frame does not have
Figure 6-2: An example of how Prophecy determines the interactive DOM subtree to build before the rest of the DOM nodes are recreated. The shaded circles represent DOM nodes that are 1) above-the-fold, and/or 2) are manipulated by the event handlers of above-the-fold DOM nodes. The interactive subtree resides above the red line.

to be analyzed in real time. However, offline processing introduces problems of scaling and freshness if each frame has many customized versions, or those versions change frequently. A frame with many customized versions will require the server to generate and store many different sets of write log diffs, some of which may never be used if clients do not issue fetches for the associated frame versions. If versions change frequently, then the server must either frequently regenerate diffs (thereby increasing CPU overheads), or regenerate diffs less often (at the cost of returning stale versions to clients). In contrast, online processing guarantees that clients receive the latest version of a frame. Online processing also avoids wasted storage dedicated to diffs that are never fetched. A single page that contains multiple frames can use the most appropriate transformation policy for each frame.

6.2.5 Optimizing for Different Definitions of Load Time

To fully load a traditional frame, a browser must fetch and evaluate the frame’s HTML, and then fetch and evaluate the external objects that are referenced by that HTML. The standard definition for a frame’s load time requires all of the external objects to be fetched and evaluated; page load time is then defined as the time that it takes to completely load all of a page’s frames (§5.2). Using write logs, Prophecy improves frame-load time (FLT) by eliding unnecessary intermediate computations and inlining all non-image content. However, as described in Section 6.2.2, Prophecy completely renders the DOM before constructing the JavaScript heap and then patching cross-references between the two. Thus, Prophecy gives higher priority to visual content, resulting in improvements to Speed Index (SI) and Above-the-fold Time (AFT) as well.

Both FLT and SI have disadvantages with respect to optimizing for interactivity (§5.2). However, Prophecy can also explicitly target Ready Index (§5.3). Ready Index (RI) declares a frame to be ready when its above-the-fold content is both visible and
interactive. To optimize for RI, Prophecy feeds its server-side log of reads and writes (§6.2.1) to Vesper (§5.4), which can then identify the frame’s interactive state. Given the DOM nodes in a frame’s interactive state, Prophecy finds the minimal HTML subtree, rooted by the top-level <html> tag, which contains all of the interactive DOM nodes. Figure 6-2 shows an example of this interactive DOM subtree. Prophecy then represents a frame using two HTML write logs (one for the interactive subtree, and one for the remaining HTML subtrees), and two JavaScript write logs (one for the state which supports above-the-fold interactive DOM nodes, and another write log for the remaining JavaScript state). Prophecy keeps a single image prefetch log, but places above-the-fold images first in the log. To load a frame on the client browser, Prophecy first renders the above-the-fold DOM nodes, and then builds the JavaScript state which supports interactivity for those DOM nodes. After patching cross-references, the frame is interactive. Prophecy then attaches the below-the-fold DOM nodes, creates the remaining JavaScript state, and patches a final set of cross-references.

By optimizing for RI, Prophecy can minimize the likelihood that attempted user interactions will fail. However, Prophecy cannot eliminate all such problems. For example, if a user issues GUI events before the first set of write logs are applied, the events may race with the browser’s creation of above-the-fold DOM elements and interactive JavaScript state. Such race conditions are present during regular, non-Prophecy page loads [144]; by optimizing for RI, Prophecy reduces the size of the race window, but does not completely eliminate it.

### 6.2.6 Privacy

A frame from origin X may embed content from a different origin Y. For example, X’s frame may embed images or JavaScript from Y. When the mobile browser sends an HTTP request for X’s frame, the browser will only include cookies from X, since the URL in the HTTP request has an origin of X. As Prophecy’s server-side code loads the frame and generates the associated logs (§6.2.1), the server from X will fetch content from Y. However, in the HTTP requests that the server sends to Y, the server will not include any of Y’s cookies that reside on the mobile browser—the server never received those cookies from the client. This policy amounts to a “no third-party cookie” approach. Variants of this policy are already being adopted by some browsers for privacy reasons, since third party cookies enable users to be tracked across different sites [153]. So, in Prophecy, a server from X only sees cookies that belong to X, and a frame load does not send third party cookies to any external origin Y.

### 6.2.7 Discussion

Prophecy is compatible with transport protocols like HTTP/2 [87] and QUIC [17] that pipeline HTTP requests, leverage UDP instead of TCP to transmit data, or otherwise try to optimize a browser’s HTTP-level network utilization. Prophecy is also compatible with proxy-based web accelerators like compression proxies [1, 155] or split-browsers [7, 132, 142]. From the perspective of these technologies, the content in a Prophecy frame is no different than the content in a non-Prophecy frame.
Prophecy is also compatible with HTTP/2’s server-push feature [17]. Server-push allows a web server to proactively send an HTTP object to a browser, pre-warming the browser’s cache so that a subsequent fetch for the object can be satisfied locally. Prophecy-enabled frames use cookies to record the versions of locally-DOM-cached frames (§6.2.3). So, imagine that a web server would like to push frames. When the server receives an HTTP request for frame \( f_i \), the server can inspect the cookies in the request and determine, for some different frame \( f_j \) to push, whether to push a cold-cache or warm-cache version of the frame.

A Prophecy web server does not track any information about a client’s DOM storage (besides diffs for the write logs that reside in that DOM storage). Since the server does not track client-side DOM storage, the final result of a frame load should not depend on the client’s non-write-log DOM storage—this state will not be available to the server-side frame load that is used to generate write logs. To the best of our knowledge, all web accelerators that use server-side load analysis [7, 132, 142, 177] assume empty client-side DOM storage, since mirroring all of that storage would be expensive, and developer best practice is to use DOM storage as a soft-state cache.

### 6.3 Implementation

On the server-side, Prophecy uses a modified version of Scout (Chapter 3) to rewrite frame content and track reads and writes to the JavaScript heap and the DOM tree. Prophecy extends Scout’s JavaScript translator to rewrite closure scopes (so that Prophecy can efficiently resurrect closure functions). Prophecy also extends the translator to log the classes of objects created via the `new` operator (so that Prophecy can determine the appropriate instance objects to create in the write log for the JavaScript heap).

When rewriting a frame’s HTML, Prophecy injects JavaScript source code for a timer that fires in response to the `load` event. This timer serializes the DOM as described in Section 6.2.1, using `XMLSerializer` to generate the basic HTML string, and using Beautiful Soup [148] to parse and edit the string, e.g., to inject precomputed CSS styles, and to extract the image `src` URLs to place in the image prefetch log.

To support client-side caching, Prophecy servers use the `google-diff-patch-match` library [60] to generate diffs. The scaffolding for Prophecy’s server-side logic is implemented as a portable CGI script for Apache.

On the client-side, Prophecy’s resurrection code is 1.3 KB in size. The code uses the `google-diff-patch-match` library [60] to perform diffing, and a modified version of the `CSSUtilities` framework [91] to apply styles to dynamically-created DOM nodes (§6.2.2).

### 6.4 Evaluating Prophecy

We evaluated Prophecy in both mobile and desktop settings. Mobile page loads were performed on a Sony Xperia X (1.8 GHz hexa-core processor) and a Nexus 6
smartphone (2.7 GHz quad core processor); each phone had 3 GB of RAM, and ran Android Nougat v7.1.1 and Chrome v61. Prophecy’s performance was similar on both phones, so we only show results for the Nexus 6 device. Desktop page loads were performed on a Lenovo M91p desktop running GNU Linux 14.04. The desktop machine had 8 processors with 8 GB of RAM, and used Google Chrome v60 to load pages.

To create a reproducible test environment, we used Mahimahi [132] to record the content in the Alexa Top 350 pages [4], and later replay that content to test browsers. For pages which defined both a mobile version and a desktop version, we recorded both. Later, at experiment time, Mahimahi always returned the desktop version of a page to the desktop browser; when possible, Mahimahi returned the mobile version of a page to the mobile browser. At replay time, Mahimahi used HTTP/2 for pages that employed HTTP/2 at recording time. Server-push events that were seen at recording time were applied during replay.

The desktop machine hosted Mahimahi’s replay environment. For experiments that involved desktop browsing, all web traffic was forwarded over emulated Mahimahi networks with link rates in \{12, 25, 50\} Mbits/s and RTTs in \{5, 10, 25\} ms. We observed similar trends across all of these desktop network conditions, so we only present results for the 25 Mbits/s link with RTTs of 10 ms.

The mobile phone was connected to the desktop via both USB tethering and a live wireless connection (Verizon LTE or WiFi) with excellent signal strength. The desktop ran the test driver, initiating mobile page loads by sending commands through the USB connection. HTTP and DNS traffic between the phone and Mahimahi used the LTE or WiFi link. The live LTE connection had RTTs of roughly 75 ms, and the live WiFi connection had RTTs of roughly 15 ms.

In each of our experiments, we considered two versions of Prophecy: an offline version in which Prophecy frames were computed before clients requested them, and an online version in which the write logs were computed on-demand, in the critical path of each HTTP request (§6.2.4). Throughout this section, we refer to the offline version as Prophecy, and the online version as Prophecy-online. We compared the versions to default Chrome page loads, and to page loads that used Polaris [128] (Chapter 4). Polaris improves page load time through parallel use of the CPU and the network, and by prioritizing the fetching of objects along the dynamic critical path in a page’s dependency graph. However, Polaris does not inline content or apply precomputation.

We evaluated each system on several metrics: Page load time (PLT), Ready Index (RI), and Speed Index (SI). PLT was measured by recording the time between the navigationStart and onload events, while RI and SI were computed using Vesper (Chapter 5). In each experiment, we loaded every page in our corpus 5 times for each system listed above, recording the median value for each load metric. Unless otherwise specified, all experiments used cold browser caches and DNS caches. In experiments with a mobile phone, energy savings were recorded by directly connecting the phone’s battery leads to a Monsoon power monitor [112].
6.4.1 Reducing PLT

Figure 6-3 illustrates Prophecy’s ability to reduce PLT for both mobile devices and desktop machines. Prophecy’s benefits are the largest on mobile devices; for example, when using a phone to load a page over an LTE network, Prophecy reduces median PLT by 53%, and 95th percentile PLT by 67%. Prophecy helps mobile devices more for two reasons.

- First, mobile devices suffer from higher CPU overheads for page loads, compared to desktop machines [126, 178]. So, Prophecy’s elision of intermediate computation (including reflows and repaints) is more impactful on mobile devices.
- PLT is much more sensitive to network latency than to network bandwidth [1, 16, 157, 159]. Cellular links typically exhibit higher latencies than wired or WiFi links. Prophecy’s aggressive use of inlining allows clients to fetch all frame content in a single HTTP-level RTT. Such RTT elision unlocks disproportionate benefits in cellular settings.

That being said, Prophecy enables impressive benefits for desktop browsers too—median PLT decreased by 38%, and 95th percentile PLT reduced by 45%.

Polaris elides no computation; in fact, client-side computational costs are slightly higher due to the addition of the JavaScript library which orchestrates object fetches and evaluation. Polaris also inlines no content. So, even though Polaris can keep the client’s network pipe full, clients must fetch the same number of objects as in a normal page load. Since browsers limit the number of parallel HTTP requests that a page can make, Polaris generally cannot overlap all requests, leading to serial HTTP-level RTTs to build a frame. In contrast, Prophecy uses a single HTTP-level RTT to build a frame. As a result of these differences, Polaris provides fewer benefits than Prophecy. For example, on a mobile browser with an LTE connection, Polaris reduces median PLT by 23%, whereas Prophecy reduces median PLT by 53%.

As expected, PLT improvements with Prophecy-online are lower than with Prophecy, since Prophecy-online generates a frame’s write logs on-demand, upon receiving a request for that frame. However, Prophecy-online still reduces median PLT by 49% on the LTE connection.
<table>
<thead>
<tr>
<th>Setting</th>
<th>System</th>
<th>Bandwidth Savings (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile</td>
<td>Prophecy</td>
<td>262 (587)</td>
</tr>
<tr>
<td>Mobile</td>
<td>Polaris</td>
<td>-37 (-5)</td>
</tr>
<tr>
<td>Desktop</td>
<td>Prophecy</td>
<td>336 (695)</td>
</tr>
<tr>
<td>Desktop</td>
<td>Polaris</td>
<td>-41 (-12)</td>
</tr>
</tbody>
</table>

Table 6.1: Median (95th percentile) per-page bandwidth savings with Prophecy and Polaris. The baseline was the bandwidth consumed by a normal page load. The average mobile page in our test corpus was 1519 KB large; the average desktop page was 2388 KB in size.

6.4.2 Reducing Bandwidth

Prophecy's server-side frame transformations have different impacts on the size of JavaScript state, HTML state, and image state:

- A Prophecy frame contains a write log which generates the final, precomputed JavaScript heap for the frame. The JavaScript write log is typically smaller than the frame’s original JavaScript source code; although the write log must recreate the original function declarations, the log can omit intermediate function *invocations* that would incrementally create frame state.

- The HTML write log for a frame consists of an augmented HTML string that contains precomputed, inline styles for the appropriate tags. The HTML write log tends to be *larger* than a frame’s original HTML string, since traditional CSS declarations can often cover multiple tags with a single CSS rule.

- The image prefetch log does not change the size of images. The log is simply a list of image URLs.

Table 6.1 depicts the overall bandwidth savings that Prophecy enables; note that bandwidth savings are identical for Prophecy and Prophecy-online. Table 6.1 shows that Prophecy’s large reductions in JavaScript size outweigh the small increases in HTML size, reducing overall bandwidth requirements by 21% in the mobile setting, and 18% in the desktop setting. In contrast, Polaris *increases* page size by a small amount. This is because a Polaris page consist of a page’s original objects, plus the client-side scheduler stub and scheduler metadata.

6.4.3 Energy savings

Figure 6-4 demonstrates that Prophecy significantly reduces the energy consumed during a mobile page load. Median reductions in per-page energy usage are 36% on an LTE network, and 30% on a WiFi network. For both networks, Prophecy eliminates the same amount of browser computation, the same number of HTTP-level RTTs, and the same amount of HTTP-level transfer bandwidth. However, LTE hardware consumes more energy in the active state than WiFi hardware [157]; thus, reducing network traffic saves more energy on an LTE network than on a WiFi network.

Prophecy provides more energy reductions than Prophecy-online—36% versus 31% for LTE, and 30% versus 23% for WiFi. The reason is that, in Prophecy-online,
Figure 6-4: Percent reduction in per-page energy usage with Prophecy, Prophecy-online, and Polaris, relative to a default page load. Bars show median values, and error bars range from the 25th to the 75th percentiles. Results were collected using a Nexus 6 smartphone.

server-side request handling takes longer to complete. As a result, the client-side phone must keep its network hardware active for a longer period.

Polaris reduces energy usage by 14% on the LTE network, and 10% on the WiFi network. Polaris keeps the client’s network pipe full, decreasing the overall amount of time that a phone must draw down battery power to keep the network hardware on. However, because Polaris elides no computation, Polaris cannot save as much energy as Prophecy.

6.4.4 Reductions in SI and RI

As described in Sections 5.2 and 6.2.5, SI and RI only consider the loading status of above-the-fold state. SI tracks the visual rendering of above-the-fold content, whereas RI considers both visibility and functionality. Our exploration of SI used Prophecy’s default configuration. In contrast, the RI experiments used the version of Prophecy which explicitly optimizes for Ready Index (§6.2.5).

Speed Index: As shown in Figures 6-5a and 6-5b, Prophecy actually reduces SI more than it reduces PLT. For mobile browsing over an LTE network, the median SI reduction is 61%; for desktop browsing over a 25 Mbits/s link with a 10 ms RTT, the reduction is 52%. Recall that, by default, a Prophecy frame reconstructs the entire DOM tree before resurrecting the JavaScript heap (§6.2.1). Prioritizing DOM construction results in better SI scores, since the browser totally dedicates the CPU to rendering pre-computed HTML before replaying the JavaScript write log. As with prior experiments, the synchronous computational overheads of Prophecy-online result in slightly worse performance compared to Prophecy—57% SI reduction versus 61% in
the mobile scenario, and 45% SI reduction versus 52% in the desktop setting. However, the benefits are still significant, and Prophecy-online has several advantages over Prophecy with respect to server-side overheads (§6.2.4).

In the mobile setting, Polaris only reduces SI by a median of 10%. In the desktop setting, Polaris actually increases SI by 2%. The reason is that Polaris’ client-side scheduler is ignorant of which objects correspond to interactive state—Polaris simply tries to load all objects as quickly as possible. Reducing overall PLT is only weakly correlated with reducing SI.

**Ready Index:** Figures 6-5c and 6-5d show that when Prophecy explicitly optimizes for RI (§6.2.5), Prophecy reduces median RI by 43% in a mobile browsing scenario, and 40% in a desktop setting. User studies indicate that, when users load a page with the expectation of interaction, optimizing for RI leads to happier users [131]. Of course, not all sites have interactive content, or a typical engagement pattern that involves immediate user input. These sites can use the standard Prophecy configuration and enjoy faster PLTs and SIs.
Figure 6-6: Breakdown of the performance benefits enabled by individual optimizations. Optimization bars begin with image prefetching, and incrementally add new optimizations until "All content inlined," which represents Prophecy's default configuration.

6.4.5 The Sources of Prophecy’s Benefits

Prophecy optimizes a frame load in several ways:

- Image prefetching: The resurrection library issues asynchronous fetches for images before constructing the DOM tree and the JavaScript heap. The asynchronous fetches keep a client’s network pipe busy as the CPU works on constructing the rest of the frame.
- CSS precomputation: The DOM write log contains precomputed CSS styles for all DOM nodes, including ones that were dynamically injected by JavaScript.
code. Precomputation reduces client-side CPU overheads for styling, layout, and rendering.

- JavaScript write log: By only writing to each JavaScript variable once, a browser avoids wasting time and energy on unnecessary JavaScript computations.
- All content inlined: Prophecy consolidates all of the frame content into a single inlined JavaScript file which stores all of the information that is needed to rebuild the frame. Thus, a browser can fetch the entire frame in one HTTP-level round trip, as opposed to needing multiple RTTs to fetch multiple objects.

To better understand how the individual optimizations affect Prophecy's performance, we loaded each page in our corpus with a subset of the optimizations enabled. Our experiments considered mobile browsing using LTE or WiFi networks; we also tested a desktop browser with a 25 Mbits/s, 10 ms RTT link. In all scenarios, we used a cold cache and measured PLT. In the mobile settings, we also measured energy usage. Note that Prophecy's bandwidth savings are primarily from the JavaScript heap log; image prefetching and CSS precomputation do not have a significant impact on bandwidth usage (§6.4.2).

As shown in Figure 6-6, Prophecy's largest reductions in page load time and energy consumption are enabled by the JavaScript write log optimization. The next most critical optimization is content inlining; saving round trips not only reduces load time, but also reduces the amount of time that a mobile device must actively listen to the network (thereby reducing energy consumption).

6.4.6 Server-side overhead

When a Prophecy-online server receives a request for a frame, the server must load the requested frame and generate the necessary write logs on-demand. If the client has a warm cache, then the server must also calculate write log diffs on-demand. Figure 6-7 depicts the impact that these online calculations have on server response throughput.
throughput. We used the Apache benchmarking tool ab [9] to scale client load and measure response times. The server and ab ran on the same machine, to isolate the computational overheads of Prophecy-online. We evaluated five server-side configurations: a default server which returned a frame's normal top-level HTML; Prophecy_cold_cache and Prophecy_cold_cache_v100000, in which clients had cold caches, and the server had either an empty diffTable or one that had 100,000 54 KB entries; and Prophecy_warm_cache_v100 and Prophecy_warm_cache_v100000, in which clients had warm caches and the server had the indicated number of diffTable entries. For warm cache experiments, we orchestrated ab so that all frame versions were accessed with an equal random likelihood. In all experiments, the baseline frame was the top-level frame in the amazon.com homepage, and the diff was empirically-observed from two snapshots of the frame that were captured a day apart.

As shown in Figure 6-7, the performance differences grow as client load grows. Up to 6,500 concurrent requests, all server variants are within 12.1% of each other, but at 10,000 concurrent requests, the difference between the default server and the warm-cache servers is 31.4%. Performance overheads with Prophecy are mostly due to online write log generation. Also note that the CPU overhead of diffing, not the memory overhead of a diffTable, leads to the degraded response throughput.

6.4.7 Warm Browser Caches

The experiments presented thus far in this section have assumed a cold browser cache. Here, we explore the performance of Prophecy when caches are warm, finding that Prophecy still unlocks significant decreases in page load time, bandwidth consumption, and energy expenditure.

For each page in our test corpus, we used Mahimahi to take several snapshots of the page. Each snapshot used a different time separation from the initial snapshot: no separation (i.e., a back-to-back load), 1 hour, 8 hours, and 24 hours. During an experiment which tested a particular age for a browser cache, we loaded each page twice. After clearing the browser cache, we loaded the page once using the initial snapshot. We then immediately loaded the later version of the page, recording the time of the second, warm-cache page load. Below, we discuss results for a 1 hour separation, but we observed similar trends for the other time separations.

PLT reductions: Figure 6-8 corroborates prior caching studies which found that mobile caching is less effective than desktop caching at reducing PLT [170, 130]. However, Figure 6-8 demonstrates that Prophecy still provides substantial benefits compared to both Polaris and a default page load. For example, Prophecy enables median PLT reductions of 43% in the mobile setting, and 34% in the desktop setting. An important reason for Prophecy’s persistent benefit is that, even in a warm-cache Prophecy frame (§6.2.3), Prophecy elides computation that must be incurred by Polaris and a default page load.

Polaris’ gains drop to 15% in the mobile case, and 9% in the desktop setting. All of Polaris’ benefits derive from the ability to cleverly schedule network fetches, and
overlap those fetches with computation. In a warm cache scenario, a page issues fewer network requests, giving Polaris fewer opportunities for optimization.

6.4.8 Diff Characteristics

Bandwidth savings: Table 6.2 demonstrates that Prophecy reduces per-page bandwidth consumption by 26% (176 KB) for mobile browsing, and 30% (298 KB) for desktop browsing. The raw savings are less than the cold cache scenarios for obvious reasons. However, since Prophecy can cache at byte granularity, not file granularity (§6.2.3), Prophecy downloads fewer network bytes than either Polaris or a default load.
Table 6.2: Median (95th percentile) per-page bandwidth savings with Prophecy and Polaris, using warm browser caches. The baseline was the bandwidth consumed by a default browser with a warm cache. The average warm cache mobile page load in our test corpus consumed 664 KB; the average desktop page used 973 KB.

<table>
<thead>
<tr>
<th>Setting</th>
<th>System</th>
<th>Bandwidth Savings (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile</td>
<td>Prophecy</td>
<td>176 (441)</td>
</tr>
<tr>
<td>Mobile</td>
<td>Polaris</td>
<td>-37 (-5)</td>
</tr>
<tr>
<td>Desktop</td>
<td>Prophecy</td>
<td>298 (571)</td>
</tr>
<tr>
<td>Desktop</td>
<td>Polaris</td>
<td>-41 (-12)</td>
</tr>
</tbody>
</table>

Table 6.3: Update frequencies for the pages and Prophecy frames in our corpus.

<table>
<thead>
<tr>
<th>Update frequency</th>
<th>Pages</th>
<th>Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 1 hour</td>
<td>124</td>
<td>313</td>
</tr>
<tr>
<td>1-2 hours</td>
<td>77</td>
<td>114</td>
</tr>
<tr>
<td>2-4 hours</td>
<td>41</td>
<td>76</td>
</tr>
<tr>
<td>4-8 hours</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>8-24 hours</td>
<td>49</td>
<td>109</td>
</tr>
<tr>
<td>&gt;= 24 hours</td>
<td>41</td>
<td>229</td>
</tr>
</tbody>
</table>

Energy savings: Prophecy’s energy savings decrease in warm cache page loads. The reason is that caching is more effective at reducing energy costs than page load time [29]; having an object cached will always avoid the battery drain associated with a network fetch, but may not decrease PLT much if the cached object is not on the critical path in the page’s dependency graph [29, 128]. Regardless, Prophecy still provides substantial energy savings, reducing median and 95th percentile consumption by 17% and 29% for an LTE network. Prophecy-online’s energy savings are lower than Prophecy (12% and 21%), but are higher than those of Polaris (6% and 12%).

6.4.9 Additional Sites

In addition to the 350 site corpus that we used for our main experiments, we also evaluated Prophecy on two additional sets of sites. First, using a web monkey, we generated a list of 200 additional pages by performing clicks on the pages in our original corpus; we generated 4 clicks per page, and then randomly selected 200 pages from the 1400 page list. These pages represented interior pages for websites, rather than the landing pages which are provided by the Alexa lists. We performed the same PLT experiments as described in Section 6.4.1, loading pages with a mobile phone over an LTE network. The trends were similar to those in our primary corpus. Median speedups with Prophecy increased to 57%, while Prophecy-online and Polaris accelerated PLT by 53% and 26%, respectively.

We also performed experiments with 100 randomly selected pages from the Alexa top 1000 list. The pages were chosen from the latter part of the list, such that no site was a member of our original corpus. For the new set of pages, the median PLT for a
default mobile load was over 2 seconds slower than the median PLT in our original corpus. Nevertheless, the basic trends from our main experiments persisted. Prophecy reduced the median PLT by 51%, whereas Prophecy-online and Polaris decreased PLT by 45% and 20%, respectively.

To understand how large diffs would be in practice, we recorded 6 versions of each page in our corpus: a baseline version (at time $t=0$), and versions recorded at $t$ values of 1 hour, 2 hours, 4 hours, 8 hours, and 24 hours. We then computed Prophecy frames for each version of each page. Finally, we computed diffs for each version of each frame, comparing against the baseline frame from $t=0$. The server’s diff calculations were fast: across all versions of the page, the median computation time was 4.6 ms, and the 95th percentile time was 8.8 ms. The median size for the largest diff across all frame versions was 38 KB; the 95th percentile largest diff size was 81 KB.

As shown in Table 6.3, 35% of the pages in our corpus require diff updates at least once an hour. In contrast, 12% of the pages do not require any diff updates within a single day. Similarly, some frames must be updated frequently, and some rarely change.

As a final exploration of diff behavior, we considered personalized versions of a subset of the pages in our corpus. We selected 20 pages from our corpus and created 2 different user profiles on each page. When possible, the preferences for each profile were set to different values. We then recorded three versions of each page: the default page (with no user logged in), the first user’s page, and the second user’s page. We created Prophec frames for each version of each page, and compared each user’s Prophecy frames to the default frames. The median diff size across all frames was 15 KB, while the maximum diff size was 31 KB. Many diffs were 0 KB, making the average diff size 6 KB.

## 6.5 Prophecy: Related Work

### 6.5.1 Prepack

Prepack [42] is a JavaScript-to-JavaScript compiler. Prepack scans the input JavaScript code for expressions whose results are statically computable; Prepack replaces those expressions with equivalent, shorter sequences that represent the final output of the elided expressions. If JavaScript code contains dynamic interactions with the environment (e.g., via calls to `Date()`), Prepack leaves those interactions in the transformed code, so that they will be performed at runtime.

Prepack does not handle the DOM, or interactions between HTML, CSS, and JavaScript. Prepack is also unaware of important desiderata for web pages, like object cacheability and personalization (§6.2.3), and incremental interactivity (§6.2.5). Thus, Prepack is insufficiently powerful to act as a general-purpose web accelerator. Prepack’s ability to elide intermediate JavaScript computations is shared by Prophecy, but Prophecy’s elision is more aggressive. Prepack uses symbolic execution [28, 33] and abstract interpretation [32] to allow the results of environmental interactions to live in post-processed JavaScript as abstract values; in contrast, Prophecy evaluates all
environmental interactions on the server-side, allowing all of the post-processed data to be concrete. This aggressive elision is well-suited for Prophecy’s goal of minimizing client-side power usage. For example, if environmental interactions occur in a loop, Prophecy only outputs the final results, whereas Prepack often has to output an abstract, finalized-at-runtime computation for each loop iteration.

6.5.2 Shandian

Shandian [177] uses a proxy to accelerate page loads. The proxy uses a modified variant of Chrome to load a requested page and generate two snapshots:

- The load-time snapshot is a serialized version of (1) the page’s DOM nodes and (2) the subset of the page’s CSS rules that are necessary to style the DOM nodes. Importantly, the load-time snapshot does not contain any JavaScript state (although the serialized DOM nodes may contain the effects of DOM calls made by JavaScript).
- The post-load snapshot contains JavaScript state and the page’s full set of CSS rules.

A user employs a custom Shandian browser to load the page. The browser fetches the load-time snapshot, deserializes it, and displays it. Later, the browser asynchronously fetches and evaluates the post-load snapshot.

At the architectural level, the key difference between Prophecy and Shandian is that Prophecy tracks fine-grained reads and writes during a server-side page load. Shandian does not. This design decision has cascading ramifications for performance, deployability, and robustness. For example, Shandian cannot optimize for RI; more generally, Shandian cannot interleave the resurrection of JavaScript code and the DOM tree. The reason is that Shandian lacks an understanding of how the JavaScript heap and the DOM tree interact with each other, so Shandian cannot make interleaved reconstruction safe. The specific lack of write logs for the DOM tree and the JavaScript heap also makes it difficult for Shandian to resurrect state and support caching. JavaScript is a baroque, dynamic language, and the lack of write logs forces Shandian’s resurrection logic to use complex, overly conservative rules about (for example) which JavaScript statements are idempotent and which ones are not. The complicated logic requires in-browser support to get good performance, and makes caching semantics sufficiently hard to get right that Shandian does not try to support caching for load-time state (and Shandian only supports a limited form of caching for post-load state). In contrast, Prophecy’s use of read/write tracking enables straightforward diff-based caching, safe interleaving of DOM construction and JavaScript resurrection, and browser agnosticism (since Prophecy’s write logs are just JavaScript variables). Prophecy also enforces traditional privacy policies for cookies, unlike Shandian.

Shandian’s source code is not publicly available, and there are no public Shandian proxies. So, we could not perform an experimental comparison with Prophecy. Based on the performance numbers in the Shandian paper, we believe that Prophecy’s PLT savings are roughly equivalent to those of Shandian, but Prophecy’s bandwidth savings are roughly 20% better. The Shandian paper did not evaluate energy consumption, but we believe that Prophecy will consume less energy due to a simpler resurrection
algorithm and less network traffic at resurrection time. Prophecy provides these benefits while enabling a constellation of important features (e.g., cacheability, optimization for interactivity) that Shandian does not provide.

Below we provide more technical detail about how Shandian works for the interested reader. We also explain why we believe that Prophecy’s write log approach is advantageous.

Robustness: Shandian’s load-time snapshot is just serialized HTML and CSS. However, Shandian’s post-load snapshot cannot contain the page’s unmodified JavaScript code, since client-side execution of the code would encounter a different DOM environment than what would have been seen in a normal page load. Thus, resurrecting the JavaScript state is challenging. Shandian’s approach is to create a post-load snapshot which contains (1) a serialized graph of JavaScript objects minus their methods, and (2) a set of JavaScript function definitions that Shandian extracted from the page’s original JavaScript code. Splicing this post-load state into the client-side environment requires complex, subtle reasoning about idempotency and ordering. For example, in a JavaScript program, a single function definition can be evaluated multiple times, with each evaluation binding to a different set of closure variables that are chosen using dynamic information. Some of the closure variables may themselves be functions. Thus, Shandian requires careful logic to generate a function evaluation order that results in the desired final state; using the lexical order of function definitions in the original source code is insufficient. Our personal experience writing JavaScript heap serializers [104, 108] has convinced us that serialization-based approaches are fragile and difficult to make correct. Prophecy’s ability to track writes dramatically simplifies matters. With knowledge of the final state of each function, object, and primitive property, Prophecy can apply a straightforward three-pass algorithm to recreate an interconnected DOM tree and JavaScript heap (§6.2.2). Thus, we believe that a write log approach is simpler and more robust than a serialization-based approach.

Liveness: Shandian lacks a fine-grained understanding of interactions between the JavaScript heap and the DOM, so Shandian cannot safely interleave DOM construction and JavaScript evaluation. As a result, Shandian must restore all JavaScript state at once, after the DOM has been constructed. This limitation prevents Shandian from making pages incrementally interactive (§6.2.5). Deferring JavaScript execution has other disadvantages, like timer-based animations not starting until the associated JavaScript code has been fetched and evaluated. In contrast, Prophecy can identify related clusters of DOM nodes and JavaScript state, enabling safe, interleaved construction of a page’s DOM tree and JavaScript heap. Prophecy also returns all of the page state to the client in a single HTTP round trip, unlike Shandian, which requires multiple RTTs.

Deployability: Shandian requires modified client browsers to parse Shandian’s special serialization format for JavaScript state, CSS rules, and DOM state. In
contrast, Prophecy logs are expressed using regular HTML and JavaScript. Thus, Prophecy works on unmodified browsers, improving deployability.

**Caching:** Shandian provides no caching support for the content in the initial snapshot. So, if just a single byte in the initial snapshot changes, the client must download an entirely new snapshot, spending precious energy and network bandwidth. Shandian supports caching for the post-load data, but the content in that snapshot is dependent on the content in the load-time snapshot! Thus, if the load-time snapshot changes, then cached post-load content is invalidated. In contrast, Prophecy provides a straightforward caching scheme that supports byte-level differencing (§6.2.3), maximizing the amount of cached content that can be used to reconstruct new versions of a page. Prophecy’s caching approach is naturally suggested by Prophecy’s use of write logs—these write logs are easily diffed using standard algorithms. In contrast, given Shandian’s complex resurrection approach, it is not immediately clear how Shandian could be extended to support traditional caching semantics.

**CSS:** On the client browser, Shandian evaluates load-time CSS rules twice: once during the initial load, and again during the evaluation of a page’s post-load CSS. As the Shandian paper states, the result is “additional energy consumption and latencies.” We cannot quantify the costs due to lack of access to a Shandian system. Thus, we merely observe that Prophecy’s inlining of CSS styles avoids all client-side CSS parsing for load-time DOM nodes—the associated CSS rules are evaluated zero times on the client. Note that, post-load, a Prophecy page can immediately style dynamically-created DOM nodes (§6.2.2). In contrast, Shandian will either have to wait for post-load CSS styles to be fetched (which may take a long time on a slow mobile link), or style the node immediately, but possibly incorrectly (leading to broken page state).

**Privacy:** In Shandian, a client-side browser ships all cookies, regardless of their origin, to a proxy. This scheme allows a proxy to load arbitrary personalized content on behalf of a user, but risks privacy violations if the proxy is intrinsically malicious, or becomes subverted by an external malicious party. Prophecy only exposes the cookies for origin X to servers from X.

### 6.5.3 Split browsers

In a split-browser system [132, 156, 157], a client fetches the top-level HTML in a page via a remote proxy. The proxy forwards the request to the appropriate web server. Upon receiving the response, the proxy uses a headless browser to load the page; as the proxy parses HTML, executes JavaScript, and discovers external objects in the page, the proxy fetches those objects and then forwards them to the client. Since the proxy has fast, low-latency network paths to origin servers, the time needed to resolve a page’s dependency graph [29, 128] is mostly bound by proxy/origin RTTs (which are small), not the last-mile client/proxy RTTs (which may be large).
Prophecy is compatible with such approaches—a Prophecy frame can be loaded by a split-browser proxy. However, the only external objects that the proxy would discover are images, since a Prophecy frame inlines the (final effects of) external CSS and JavaScript objects. Also note that the goal of a split-browser is to hide the network latency associated with a client’s object fetches; split browsers cannot identify _client-side computations_ that may be elided. Prophecy does find such computations, while also eliminating fetch RTTs via inlining.

### 6.5.4 Mobile web optimizations

Several projects optimize mobile page loads by exploiting knowledge about the specific hardware used by a phone. For example, several systems have explored how a phone that has multiple, heterogeneous cores can use specific cores (and specific processor frequencies) to minimize the energy consumed by page loads [189, 25]. The Chameleon web browser [35] depicts web content using special color schemes that result in less power consumption on a phone’s OLED screen. Prophecy is hardware-agnostic, with a generic goal of reducing the client-side computation and HTTP-level round trips that are needed to load a page. Thus, Prophecy is compatible with the approaches mentioned above.

Klotski [27] is a mobile web optimizer that uses server-push (§6.2.7). When a browser fetches HTML for a particular page, the Klotski web server returns the HTML, and also pushes high-priority objects which are referenced by the page (and will later be requested by the browser). Klotski identifies high-priority objects in an offline phase using a utility function (e.g., that prioritizes above-the-fold content). Prophecy is compatible with server-push, but at the granularity of entire frames, not individual objects, since Prophecy inlines content (§6.2.7). Inlining, combined with final-state patching, allows Prophecy to both lower load time and decrease energy consumption. In contrast, a Klotski page elides no computation. VROOM [149] is similar to Klotski, except that clients prefetch data instead of receiving server pushes; a VROOM server uses link preload headers [68] in returned HTTP responses to hint to clients which objects can be usefully prefetched.

AMP [58] accelerates mobile page loads by requiring pages to be written in a restricted dialect of HTML, CSS, and JavaScript that is faster to load. For example, AMP forces all external _<script>_ content to use the _async_ attribute so that the browser’s HTML parse can continue as the JavaScript code is fetched in the background. AMP forces a page to have at most one CSS file, which must be an inlined _<style>_ tag whose contents are less than 50 KB in size. Prophecy is designed to support arbitrary pages that use arbitrary HTML, CSS, and JavaScript. However, Prophecy can be applied to AMP pages since those pages are just HTML, CSS, and JavaScript.
Chapter 7

Cascade: Using Data Flow Analysis For Speculative “What If?” Debugging of Web Applications

7.1 Overview

Debugging the client-side of a web application is hard. The DOM interface [123], which specifies how JavaScript code interacts with the rest of the browser, is sprawling and constantly accumulating new features [76, 105]. Furthermore, the DOM interface is pervasively asynchronous and event-driven, making it challenging for developers to track causality across event handlers [75, 106, 125, 186]. As a result, JavaScript bugs are endemic, even on popular sites that are maintained by professional developers [138, 139].

Commodity browsers include JavaScript debuggers that support breakpoints and watchpoints. However, fixing bugs is still hard. Breakpoints and watchpoints let developers inspect program state at a moment in time; however, in an event-driven program with extensive network and GUI interactions, bug diagnosis often requires complex temporal reasoning to reconstruct a buggy value’s provenance across multiple asynchronous code paths. This provenance data is not exposed by more advanced tools for replay debugging or program slicing (§7.2).

In this chapter, we introduce Cascade, a new debugger for web applications. Cascade has three features which enable a fundamentally more powerful debugging experience. First, Cascade tracks precise value provenance, i.e., the exact set of reads and writes (and the associated source code lines) that produce each program value. Like a traditional replay debugger [36, 106, 162], Cascade records all of the nondeterministic events from a program’s execution, allowing Cascade to replay a buggy execution with perfect fidelity. Unlike a traditional replay debugger, Cascade also records the deterministic values that are manipulated by reads and writes of page state. Using this extra information at replay time, Cascade enables developers to query fine-grained data flow logs and quickly answer questions like “Did variable \(x\) influence variable \(y\)?” or “Was variable \(z\)’s value affected by a control flow that

97
traversed function \( f() \)?” Cascade’s overhead for logging both nondeterministic and deterministic events is small: for the median web page in our 300 page test corpus, Cascade increases page load time by only 5.5\%, while producing logs that are only 45.4 KB in size.

Cascade’s second unique feature is support for speculative bug fix analysis. At replay time, Cascade allows a developer to pause the application being debugged, edit the code or data of the application, and then resume the replay. Post-edit, Cascade replays the remaining nondeterministic events in the log, using carefully-defined semantics (§7.3.3) to determine how those events should be replayed in the context of the edited program execution. Once the altered execution has finished replaying, Cascade identifies the control flows and data flows which differ in the edited and original executions. These analyses help developers to determine whether a hypothesized bug fix would have helped the original program execution. Speculative edit-and-replay is unsound, in the sense that a post-edit program can misbehave in arbitrary ways, e.g., by attempting to read an undefined variable. However, even without Cascade, the process of testing bug fixes is unsound. A developer typically lacks a priori knowledge about whether a hypothesized fix will work. The developer implements the hypothesized fix, and then runs tests and tries to determine whether the fix actually worked; even if all of the tests pass, there is no guarantee that the fix is completely correct, since the tests may miss corner cases. However, Cascade provides the developer with an important new weapon: the ability to compare the data flows and the control flows in the original execution and the ostensibly bug-free execution. As we demonstrate through case studies (§7.5.2), the ability to diff program executions is a powerful debugging tool.

Cascade’s third novel feature is to support wide-area debugging for applications whose server-side components use single-threaded, event-driven architectures like Node [136], Redis [147], or NGINX [133]. For these components, the event loop interface provides a narrow, semantically-well-defined abstraction layer at which to log and replay the components. Thus, Cascade can use vector clocks and a small assortment of additional tricks (§7.3.4) to track wide-area causality. Cascade provides two levels of support for server-side components:

- **Node components execute JavaScript code.** Thus, Cascade can apply its client-side framework to the server-side, and track variable-level data flows and control flows between multiple browsers and multiple server-side Node instances.
- **Cascade treats an event-driven (but non-JavaScript) component like Redis as a black box.** Cascade logs and replays the component at the level of the component’s externally-visible event interface, tracking data flows emanating from, and terminating at, server-side events.

Cascade supports speculative bug fix analysis for data stores and JavaScript state on either side of the wide area. For example, a developer can edit the value that server-side code receives from a Redis database, and then explore how the edited value impacts the remainder of the replaying application’s execution.

In summary, our contribution is the first distributed replay debugger that provides fine-grained data flow analysis and speculative bug fix analysis. Cascade’s logging infrastructure is efficient enough to run in production (§7.5.1). At replay time, the
Figure 7-1: EtherCalc [163] is a web-based, collaborative spreadsheet. Multiple users can simultaneously issue edits to the same spreadsheet, with a Node server broadcasting edits to all users, and storing the spreadsheet data in Redis. Bug #314 in EtherCalc’s issue tracker involves a GUI-based edit from client 1 that is not reflected to client 2’s DOM. Ideally, a debugging framework could efficiently answer two questions. First, how is the relevant DOM state and JavaScript heap state from client 1 being transmitted through the server-side components to the DOM and JavaScript heap of client 2? Second, given recorded state from a buggy execution run, as well as a hypothesized bug fix that modifies code and/or data on clients or servers, would the hypothesized fix remove the problematic behavior in the recorded execution? In Section 7.5.2.2, we show how Cascade can answer these questions.

reconstruction of data flows is fast (§7.5.2); these data flows, in combination with Cascade’s well-defined semantics for applying nondeterministic events from one replay to another (§7.3.3), provide the first general-purpose environment for speculative edit-and-continue debugging. In Section 7.5.2, we demonstrate that Cascade is useful for diagnosing complex bugs in real web applications.

7.2 Background

Cascade’s design was motivated by our previous experiences with debugging complex web applications. Having used the debuggers in commodity browsers [46, 65, 116], and having built several state-of-the-art debuggers ourselves [104, 107, 172], we found that no prior tool supported fine-grained analysis of the control flows and data flows that span clients and servers. We also found that no prior tool supported speculative bug fix analysis. In this section, we provide more detail about why prior debugging techniques are insufficient to realize the vision of Figure 7-1.
7.2.1 Traditional debuggers

Standard debuggers are built around the abstractions of breakpoints [46, 65, 116]. A breakpoint allows a debugger to specify a source code line which, if reached during program execution, will cause a program to transfer control to the debugger. Once a breakpoint is hit, stepping allows the developer to move the program’s execution forward by a single statement or function call per step; after each step, the developer inspects the program’s state to try to diagnose the bug. Debuggers like Visual Studio [109] and Eclipse [38] also allow developers to edit some types of program values at a breakpoint, and then resume the program’s live execution. Breakpoints are undoubtedly useful, but they force a human developer to guess which source code locations are buggy.

Some debuggers support watchpoints, which pause an application when a specific memory location is read or written. Watchpoints eliminate the need for a developer to guess when and where a particular buggy assignment will occur. However, watchpoints do not capture temporal data flows throughout a program. So, developers still have to manually reconstruct reverse temporal flows to determine how the value in a buggy write was generated. Our case studies (§7.5.2) demonstrate that automated construction of value provenance eliminates human-driven guess work about how program state is created.

7.2.2 Deterministic replay

Traditional debuggers pause and inspect the state of live programs. In contrast, replay debuggers [6, 36, 43, 52, 53, 85, 90, 107, 162] first log the nondeterministic events in a live execution run, and then replay the program in a controlled environment, using the log to carefully recreate the original order and content of nondeterministic events. Replaying the nondeterministic events is sufficient to induce the remaining, deterministic program behavior, so there is no need to log the values that are manipulated by deterministic reads and writes.

The ability to reliably recreate a buggy execution makes it easier to test fault hypotheses. Replay debugging is particularly useful for studying heisenbugs that rarely occur and involve specific event orderings. Some replay debuggers support backwards-stepping [43, 52, 90, 162], such that a developer can set a breakpoint or a watchpoint, and then move execution forwards or backwards in time. However, even backwards-stepping debuggers force human developers to manually track value provenance. Thus, root cause analysis is still hard.

7.2.3 Program slicing

A program slice is a subset of program statements that may have influenced the values that are accessed by a specific line of source code [168]. The tuple \(<\text{sourceCodeLine}, \text{variablesOfInterest}>\) is called the slicing criterion. Given a slicing criterion, a static slice is derived purely from analysis of source code [18, 21, 44, 71, 134]; in contrast, a dynamic slice assumes a set of concrete values (e.g., at the slicing criterion) to narrow the set of potentially relevant program statements [2, 3, 70, 86, 168].
Slicing algorithms lack a complete, concrete log of the reads and writes made during a real execution; thus, slicing algorithms are often imprecise, particularly for complex programs. Imprecision hurts the use of slices for bug diagnosis, since developers must consider source code lines that may not be causally related to the bug. Imprecision compounds itself if slices are used to reconstruct wide-area execution behavior. In contrast, Cascade provides guaranteed-precise, provenance-annotated execution traces (§7.3.2). Similar to an instruction trace, a Cascade trace provides a temporal log of the source code statements that a program executed; however, the traces also describe the values that the executed statements manipulated, and the provenance of those values.

7.2.4 Data Provenance

Provenance-aware file systems [124, 152] allow users to determine which input files were read by a process during the production of output files. Cascade deals with the provenance of application state at the granularity of individual program variables that reside on clients and servers. Thus, Cascade tracks how storage data spreads throughout an application, but does so at the level of fine-grained, variable-level flows.

Provenance-aware network platforms let operators discover the route that a packet took [188], or the reason why network switches have certain NDlog rules [182, 183, 187]. Cascade is agnostic about network-level configuration state, but is compatible with systems that track it.

7.2.5 Speculative edit-and-continue

Dora [171] is a single-machine replay debugger that records the OS-level interactions that belong to a group of processes. Dora allows for limited types of edits to occur during replay. If an edit causes a replay to diverge, Dora explores multiple execution paths that are rooted at this initial divergence. Dora executes each post-divergence path on a live machine, recording the subsequent (and nondeterministic) OS-level interactions. Like Cascade, Dora defines policies for handling new calls to timekeeping functions or socket interfaces. After Dora has explored several potential futures of the divergent replay, Dora identifies the most plausible divergent execution using a metric akin to string edit distance, comparing the system calls of each explored path to those of the original execution.

Dora’s speculative power is highly restricted by two factors. First, Dora’s vantage point is at the OS layer. In contrast, Cascade’s vantage point is within the managed runtime of a JavaScript engine, or at the event loop interface of a single-threaded program like Redis. This difference is fundamental, and represents a key insight of Cascade: by introspecting on program execution at a higher level of abstraction, Cascade can handle a wider variety of speculative edits, because the side effects of an edit can be reasoned about with respect to a constrained set of events, instead of the much wider and messier POSIX interface. For example, Dora cannot handle edits which modify thread scheduling, e.g., to cause fewer threads to run, because Dora cannot enumerate and model the ensuing avalanche of side effects upon low-level
POSIX state like pthread locks and shared memory pages. In contrast, Cascade can handle a schedule-altering edit that changes the number of client-side frames (the JavaScript equivalent of processes). Cascade can tractably reason about such changes because frames cannot share raw memory, are internally single-threaded, and only communicate via pass-by-value postMessage events. Thus, the only way that a newly created frame can impact another frame is via the generation of new postMessage events. By leveraging these kinds of JavaScript-specific semantics, Cascade can handle post-edit divergences in a more principled manner (§7.3.3) than Dora, without appealing to edit-distance heuristics that are not directly tied to reasonable application behaviors.

Dora’s second restriction is that it does not track individual reads and writes to raw memory, because doing so would be too expensive for non-trivial programs [127]. One consequence is that Dora cannot provide variable-level value provenance; another consequence is that Dora may incorrectly replay post-edit memory accesses if the edit changes which memory page contains an object. In contrast, Cascade introspects at the JavaScript level, allowing Cascade to efficiently track all reads and writes to application-visible state. This difference is fundamental. Logging all reads and writes enables wide-area causality tracking, and is critical for explaining divergences between a logged program run and a speculatively-edited replay (§7.3.3).

7.3 Cascade: Design

Figure 7-2 provides an overview of Cascade’s architecture. A web application consists of multiple clients and servers. Clients are assumed to be standard web browsers which execute JavaScript. Both server-side and client-side components are assumed to be single-threaded and event-driven. Each component records its nondeterministic events; if a component uses a JavaScript engine, then the component also records its deterministic reads and writes to JavaScript state and the DOM (§7.3.2). Distributed causality between hosts, e.g., via HTTP requests, is tracked using vector clocks (§7.3.4). At debug time, Cascade uses the global event log to replay each client or server in isolation, or together as a single logical application (§7.3.3 and §7.3.4).

7.3.1 Overview of JavaScript

Understanding the JavaScript execution model is crucial for understanding how Cascade supports speculative edit-and-continue. In this section, we will briefly review some of the properties of JavaScript that are particularly relevant to Cascade’s design. Thus, here we will focus on how debugging works for JavaScript-only components; later, in Section 7.3.4, we explain how Cascade interacts with black-box components. Execution environment: JavaScript exposes a single threaded, event-driven programming interface. A JavaScript file defines initialization code that runs once, at the time that the file is parsed by the JavaScript engine. The initialization code registers event handlers that the JavaScript engine will fire in response to GUI interactions, timer expirations, network activity, and so on. Once a browser has evaluated all of
the JavaScript files in a page’s HTML, the subsequent execution of the page is driven solely by the reception of asynchronous events.

An event handler often calls other functions. Thus, firing a handler can initiate a call chain that is rooted by the handler. A program can register multiple handlers for a single event type. Thus, the call chain for an event is the union of the call chains for the associated event handlers. In the rest of this chapter, the unadorned term “call chain” refers to the aggregate call chain for a particular event.

**Sources of Nondeterminism:** In a JavaScript program, the primary source of nondeterminism is the order in which events arrive (and the content of those events) [106]. JavaScript code may also invoke a small number of nondeterministic functions. For example, `Math.random()` returns a random number. `Date()` returns the current time using a millisecond granularity.

By default, a JavaScript program consists of a single event loop. However, a web page can incorporate multiple frames [117] or web workers [72]; each one represents a new event loop that runs in parallel with the others [115]. Concurrent execution contexts can only interact with each other via the asynchronous, pass-by-value `postMessage()` interface [119]. The browser delivers those messages by firing an event in the recipient’s execution context. Thus, from the perspective of the recipient, handling message nondeterminism is no different than handling other event-driven nondeterminism like GUI interactions.

**Externalizing Output:** A JavaScript program can externalize three types of output:
• The DOM interface [123] lets a program update the visual content that users see. The DOM interface defines methods for dynamically manipulating a page’s HTML structure, e.g., by adding new HTML tags, or by changing the CSS styles of preexisting tags.

• A JavaScript program can also write to local storage. Cookies [14] can store up to 4 KB of data, whereas IndexedDB [118] and the localStorage interface [120] can hold MBs of information.

• To send network data, a program uses the XMLHttpRequest [122] and WebSocket [121] interfaces. XMLHttpRequest is an older interface which only supports request/response interactions. WebSocket supports full-duplex streams.

A JavaScript program can also generate audio content by embedding sound files or videos. In this dissertation, we ignore such multimedia objects, since we focus on the debugging of pure HTML, CSS, and JavaScript state.

7.3.2 Analyzing Value Provenance

To track data flows, Cascade first logs nondeterministic and deterministic events. After reconstructing the data flows, Cascade uses them to support flow queries, and express state divergences caused by speculative edit-and-continue.

Logging Nondeterminism: Prior work has explored various ways to deterministically replay client-side JavaScript code. For example, nondeterministic events can be logged and replayed by the browser itself, using a modified JavaScript engine [13]. Alternatively, a JavaScript library, included by the page of interest, can use JavaScript reflection to interpose on sources of nondeterminism at both logging and replay time [106]. Our Cascade prototype uses JavaScript-level interpositioning (§7.4), but Cascade’s design makes no deep assumptions about how nondeterminism is logged or replayed.

A Cascade log has an entry for each nondeterministic event; each log entry contains event-specific data that is sufficient for recreating that event. For example, the log entry for a timer firing contains a reference to the timer callback. The log entry for a call to Math.random() contains the return value of the function. The log entry for a mouse click stores which mouse button was clicked, the x and y coordinates for the click, and so on.

At the beginning of logging, Cascade takes a snapshot of the client’s local storage (e.g., cookies). Cascade also registers its own handlers for GUI events like mouse clicks. So, if the logged application only installs handlers for (say) keypress, but not keydown or keyup, Cascade will still log when the latter two kinds of events occur. This information is useful for handling speculative edits which add new GUI handlers (§7.3.3).

Logging Deterministic Reads and Writes: In JavaScript, each object is essentially a mutable dictionary, with string keys (i.e., property names) mapping to property values. The global namespace is also reified via the special window object, such that references to a global variable x are implicitly translated to window.x. Our concrete Cascade prototype uses Scout (Chapter 3) to log reads and writes to the JavaScript
Reconstructing Data Flows: Using the log of deterministic reads and writes, Cascade can reconstruct the provenance of all JavaScript variable values at any moment in a program’s execution. Given a slicing criterion which mentions variable $x$ at time $t$, Cascade finds the prior write for which $x$ was the left-hand side. For the variables on the right-hand side, Cascade finds the prior write which assigned to those variables. Cascade continues this recursive process until reaching the beginning of the program; the traced path represents the provenance for the slicing criterion. Note that the path may be a tree, not a line, because a single assignment may involve multiple right-hand sides (e.g., $x = y + z$). The path may also cross the event handlers that belong to multiple high-level events like key presses or the arrival of network data.

Cascade’s log associates each deterministic read or write with a source code line. Thus, Cascade can also generate source-code-level execution traces which provide a serial history of each source code line that a program ran. The core visualization tool that Cascade provides to developers is an execution trace that is overlaid with provenance information: each variable mentioned in each source code line is associated with the prior source code line which generated the variable’s value. Figure 7-7 depicts an example of such a trace, and Section 7.5.5 describes some of the pruning techniques which improve the comprehensibility of traces. For now, we merely explain a developer-guided pruning approach that is simple, and important in practice: targeted dynamic tracing. A targeted dynamic trace allows a developer to drill down on the executed source code lines (and associated data flows) that affected a specific variable. As the developer explores the initial trace, the developer can add or remove additional target variables, expanding or shrinking the targeted trace. Our case studies (§7.5.2) show that targeted dynamic traces are fast to generate, and provide helpful diagnostic information.

Cascade uses Scout to also track data flows between the JavaScript heap and the DOM. For example, the DOM tree is a data structure which mirrors a page’s dynamic HTML tree; each HTML tag has an associated DOM node that is exposed to JavaScript code. Cascade understands the semantics of DOM methods like `Node.appendChild(newChild)`. Thus, Cascade can track how JavaScript values flow to DOM nodes, and how DOM values are assigned to JavaScript variables.

Cascade’s logs capture a variety of additional behavior. For example, Cascade explicitly tracks aliasing relationships, as shown in Figure 7-3. Cascade’s logs also contain useful information about branches; in particular, for each executed branch, Cascade records the associated source code line, and the values consumed by the branch test. This information allows Cascade to apply classical algorithms for building dynamic control flow dependencies [86]. Cascade easily handles the special case of execution flows that span `try/catch` blocks, since Cascade records both the exception-throwing line, and the catching line.

7.3.3 Speculative Edit-and-Continue Debugging

Speculative edit-and-continue debugging has five phases:

- logging the events in a baseline execution run;
Figure 7-3: To capture aliasing relationships, Cascade distinguishes between an underlying object and its multiple names. Writes to an aliased object create horizontal arrows in data flow diagrams, since time flows downward and the aliases are updated simultaneously.

- replaying the execution up to a specified point;
- changing the program’s state in some way;
- resuming execution, with nondeterminism from the original run “influencing” the post-edit execution; and finally,
- comparing the behavior of the original and altered runs to understand the effects of the speculative fix.

Below, we define the semantics for executing post-edit code under the guidance of a log whose nondeterministic values may not cleanly apply to the post-edit execution. The nondeterministic input vectors for a JavaScript program are well-known and (compared to POSIX) very small in number [106]; however, Cascade must address edits that potentially mutate each type of nondeterminism.

**Inside the call chain that contains the edit:** Once we have replayed execution to the edit point and modified the necessary state, we resume the call chain’s execution. Post-edit, the chain may explore different branches than were visited in the original run. Thus, the chain may issue fewer or additional calls to nondeterministic functions like `Date()`.

- If the post-edit code makes fewer calls to a nondeterministic function \( f \), we simply extract return values for \( f \) from the log, replaying the same nondeterminism that the original run experienced. Once the call chain finishes, and we must replay the next event’s call chain, we replay \( f \)’s values from the log, starting with the value that was first seen by the original execution of the call chain for the new event. For example, suppose that, during the original program execution, two events fired; the first call chain consumed random numbers \( r_0 \ldots r_4 \), and the second chain consumed \( r_5 \ldots r_9 \). Suppose that the first call chain is edited, such that it only makes two calls to `Math.random()`. When the second call chain executes in the post-edit run, `Math.random()` will return \( r_5 \), then \( r_6 \), and so on, since these are the random numbers that the second call chain saw during its original run.
- If the post-edit code generates more calls to a nondeterministic function than seen at logging time, we use a function-specific extrapolation technique to generate
additional values once the call chain has exhausted the values that are associated with it in the log. For `Math.random()`, we simply generate new random numbers. For time-related functions like `Date()`, we return monotonically increasing time values that are smaller than the next logged time value. Once the call chain finishes and we trigger the call chain for a new event, we return to using the log to provide values for nondeterministic functions.

Post-edit code may also generate new externalized output. For example, an edited value may be written to local storage, or sent over the network via the query string of an `XMLHttpRequest`. Post-edit code may also modify event handler state in ways that cause fewer or additional events to fire in the future. For example, post-edit code may register timers that were never created in the original run; post-edit code may also `deregister` timers that fired in the original run. Post-edit code may also generate entirely new network requests, or register/deregister handlers for GUI events. Below, we discuss how to incorporate these changes into the post-edit universe.

**After the call chain which contains the edit has finished execution:** At this point, the replay framework has completed execution of the call chain. The framework can now manipulate program state before releasing the next event and invoking the appropriate event handlers.

Due to the edit, the current execution context may possess different event handlers than what the program had at the equivalent moment in the original execution run. The replay framework must integrate any changes into the log of nondeterminism; some post-edit events in the log must be marked as “do not replay,” and some new events must be added to the log:

- **If the edit resulted in the deletion of a timer, we mark all of the timer’s subsequent events as “do not replay.”** If the edit created a new timer, we inject new timer events into the log, using the logged wall-clock time of preexisting events to determine where the new timer events should go, relative to the preexisting events.
- **If the edit deleted a DOM handler, and the edited program has no remaining DOM handlers for a particular event type, we label all post-edit instances of that event as “do not replay.”** For example, if the deletion of a `keypress` handler leaves the program with no `keypress` handlers at all, then we suppress future dispatches of logged `keypress` events (because such events cannot trigger any call chains). If an edit registers a new DOM handler, then no special action is required from the replay framework—when the framework dispatches a relevant event, the framework will invoke the new handler as usual. Remember that Cascade records all GUI events at logging time, even if the application has not registered its own handlers for those events (§7.3.2). Thus, at replay time, Cascade can invoke new handlers for a particular event at the appropriate moment.
- **If an edit closes XMLHttpRequests or WebSockets, the replay framework cancels future events that involve those network connections.** If the edit creates a new, unlogged network request, then the replay framework must inject new network events into the log. If the server-side responder is also being replayed, then
Cascade inserts a new request into the server-side log; the request represents a speculative server-side edit. When the response is generated, Cascade buffers it, and uses a model of network latency to determine where to inject the response into the client-side log (§7.3.4). If Cascade does not control the server-side responder, Cascade can terminate replay; alternatively, Cascade can issue the request to the live (but uncontrolled) responder, and then insert the response into a downstream position in the client log, using the observed network latency of the live fetch to determine where to place the response.

- The post-edit code may issue new reads or writes to local storage. The replay framework does nothing special to handle synchronous accesses to cookies or DOM storage—the framework simply passes those IOs to the underlying storage. For asynchronous accesses to IndexedDB, the replay framework must inject new IO events into the log, using a model for the expected latency of those events.

Note that the replay framework never injects new GUI events into the post-edit universe. For example, the framework will never inject new mouse clicks or key presses. Nothing prevents the framework from doing so, but, lacking a reasonable model for how user intent would change in the post-edit world, the framework is content to merely replay the GUI events from the original program run.

Once the replay framework has patched the log, the framework extracts the next high-level event from the log, and initiates the relevant call chain. The event may or may not have been seen in the original program run.

**Inside the call chain for a new event which did not occur during the original execution:** The replay framework uses extrapolation to generate return values for nondeterministic functions like Math.random(). When the call chain terminates, we add and remove top-level events as described above.

**Inside the call chain for an event which did occur in the original execution:** We use the log to replay return values for nondeterministic functions; if the call chain’s nondeterministic values are exhausted before the call chain finishes, we use extrapolation to generate additional values. When the call chain finishes, we add and remove events from the log as described above.

Figure 7-4 shows an end-to-end example of replaying events after an edit has been made. Once an altered replay finishes, developers can compare the data flows of the original and altered executions, looking for evidence that the hypothesized bug fix actually succeeded. If a particular value has a linear provenance chain in both executions, Cascade uses string diffing algorithms like Damerau-Levenshtein edit distance [12] to quickly identify the reads and writes that diverge in the two provenance chains. Similar edit distance metrics exist for trees [20]. Cascade’s logs contain enough information to reconstruct execution traces at the granularity of individual source code lines (see Figure 7-7); thus, developers can use differential slicing [81] to align divergent executions with respect to shared and non-shared lines of executed source code.
A snippet of the program's original execution, showing two GUI events (each of which triggers two top-level event handlers), a network event which indicates the reception of data from a remote server, and a timer which fires at a wall clock time of 4 seconds after the program started execution.

(b) During the replay process, the developer edits function y(). As a result, the XHR event is never replayed; additionally, the timer fires early, and receives a different value from Date().

Figure 7-4: An example of how an edit changes the replay process. Beneath each event, we depict the associated call chains. Red indicates functions whose behavior is altered by the edit. Grey indicates events from the original execution which do not occur in the post-edit universe.

7.3.4 Debugging Across the Wide Area

Node: Node [136] is a server-side implementation of the JavaScript runtime. Like a browser-based JavaScript engine, Node exposes a single-threaded, event-driven interface. A Node application runs headless, i.e., without a GUI, but otherwise has access to timers, nondeterministic functions like Date(), and asynchronous IO channels like network sockets. Cascade interposes on these nondeterministic inputs using the
same techniques that it leverages on the client-side. For example, event sources like sockets are subclasses of Node’s EventEmitters class; by interposing on handler registration methods like EventEmitters.on(), Cascade can wrap event handlers in the necessary logging code, or control how those handlers are executed at replay time.

To track causality between a client and a Node server, Cascade uses vector clocks [45, 96] to establish a partial ordering over the distributed events. At logging time, when a client issues an XMLHttpRequest, Cascade transparently adds a new cookie value which contains the client’s clock. On the server-side, Cascade transparently modifies the HTTP request handler to extract the client clock and update the server’s clock appropriately. When the server generates the HTTP response, Cascade uses a Set-Cookie header to transmit the server’s updated clock to the client; the client extracts the cookie and updates the local clock. The client browser automatically persists the cookie on local storage, as the browser would do for any other type of cookie.

In JavaScript, a program can associate a single top-level event with multiple handlers. At logging time, a client or server updates the local clock at the beginning of each event dispatch, before handlers run. The use of browser cookies to store client clocks allows a client to detect when passively-fetched content triggers server-side JavaScript execution. For example, suppose that client-side JavaScript code injects a new <link> tag into the page using the innerHTML DOM method; such a tag might represent a new style sheet. Client-side JavaScript will not have an opportunity to inspect the HTTP response headers for the <link>. However, when the next JavaScript-visible event fires, the first handler for that event can inspect the cookie that was set by the <link> fetch, extract the server’s vector clock, and then update the local vector clock appropriately.

At replay time, Cascade collates the client logs and the server logs, using logical clocks to generate a total ordering over all events. Cascade then replays events from the total ordering; at each step, Cascade moves either a client or a server one event further in the global log.

Note that each host’s log contains sufficient information to replay the host in isolation—the log contains all of the external nondeterministic stimuli that affected the host, as well as internal nondeterminism like GUI events or the values returned by clock reads. So, if a host communicates with multiple parties, but only some of them run Cascade, then the host can be replayed by itself, or in concert with some or all of the Cascade-enabled hosts. However, Cascade must be vigilant for speculative edits that generate new, unlogged requests to entities that are not participating in the replay (§7.3.3).

**Black-box components:** A client-side browser and a server-side Node engine both run single-threaded, event-driven JavaScript code. In contrast, server-side components like Redis and NGINX are single-threaded and event-driven, but are written in C, C++,

---

1 A subtlety is that, if a server is concurrently handling multiple requests for a particular client, the server must ensure that the client receives a sequence of responses whose vector clocks have strictly increasing numbers in the server’s slot. This policy is necessary because the client-side browser uses a “last-write-wins” policy for cookies.
or another non-JavaScript language. Cascade treats each such component as a black box, logging incoming requests and outgoing responses using a proxy. For example, our Cascade prototype intercepts HTTP traffic that is exchanged with a Redis server, using Redis-specific rules to extract `get(k)` and `put(k, v)` commands, and serialize the order in which commands are sent to Redis. Cascade assumes that each event handler inside a black box is deterministic, such that replaying a serialized stream of requests will result in 1) the same internal state for the component, and 2) the same responses being returned. These assumptions are reasonable for server-side components like Redis that act as fairly simple front-ends to storage; however, these assumptions may not hold for server-side components that are written in arbitrarily-expressive, non-JavaScript languages like C++ or Go.

Cascade uses vector clocks to establish causality between black-box components and JavaScript-based components. However, Cascade does not log the reads and writes that black-box components make to internal state. Thus, data flows involving black-box state originate and terminate in the high-level requests and responses that black-box components exchange with external parties. For example, Cascade can track a JavaScript value on a client to the server-side Redis `get()` responses that influenced that value; however, Cascade cannot peer inside Redis to see why those responses were completed in a particular way. Fortunately, developers can use component-appropriate tools like `gdb` to examine the internal state associated with a replayed black-box component.

At the beginning of logging, Cascade takes a snapshot of a black-box component’s initial state using native mechanisms (e.g., Redis' built-in snapshot facility). At the beginning of replay, Cascade uses the snapshot to initialize the component.

**Speculative wide-area edits:** Cascade allows a developer to pause the wide-area application, edit client-side or server-side JavaScript, DOM, or storage state, and then resume execution. In general, Cascade uses the techniques from Section 7.3.3 to handle divergence on both sides. However, wide-area debugging introduces some new divergence scenarios.

Define a requestor as a component that generates a request, and the responder as the component that responds to the request. Browsers always act as requestors, with server-side components acting as the corresponding responders; however, as server-side components talk to each other, they may act as requestors or responders at different points in time.

An edit may cause a responder to return a different response to a particular request, where “a particular request” is defined as a request that is made at a specific vector clock time. Cascade can detect such divergence because, at replay time, Cascade interposes on the methods that the responder uses to return data; Cascade compares the replay-time value of the response to the logged value that was generated during the original execution. If the values are different, Cascade rewrites the appropriate

---

2 At replay time, individual responses may be emitted in a different order than the logging-time one, due to nondeterministic replay-time access delays to storage media like SSDs. However, Cascade’s distributed replay driver buffers the responses, and ensures that response data is delivered to clients in the logging-time order.
requestor-side log entries, propagating the new data. Later, replaying those events will naturally inject the new data into the requestor-side execution state. Subsequent requestor-side divergence is then handled using the approaches from Section 7.3.3.

If an edit induces a requestor to send a modified request to a responder, Cascade rewrites the appropriate responder-side log entry. When the replay logic applies the log entry to the responder, the replay logic buffers the response, and then replays the response on the requestor at the moment indicated in the log; note that the response may contain altered content with respect to the original, logged version of the response.

An edit can induce a requestor to generate a completely new request at a vector clock time that was not associated with a request during the original execution. In this scenario, Cascade's requestor-side replay driver does not allow the request to hit the real network. Instead, the responder-side driver injects a fake request into the responder-side log, and then resumes the replay process. Eventually, the replaying responder will handle the new event, and generate a response. The replay driver will buffer the response, and use a network model to determine when to replay the response on the requestor.

An edit may cause a requestor to not generate a logged request. In this case, Cascade does not replay the associated downstream events on the responder or the requestor. For example, if a browser does not issue a logged XMLHttpRequest to a Node server, then Cascade will not replay the Node-side HTTP request event, or the downstream browser-side events corresponding to the reception of the HTTP response.

**Additional concerns:** In Section 7.5.4, we discuss two subtleties of replay. The first involves pages that include content from multiple origins. The second involves the proper ordering of JavaScript load events.

### 7.3.5 Limitations

Cascade logs and replays at the level of the JavaScript runtime or a high-level event interface. However, a program fault may occur because of a problem at a lower layer, e.g., a nondeterministic OS bug. Cascade is not guaranteed to recreate such errors at replay time. However, by operating at the JavaScript interface, Cascade can analyze many application-layer bugs that are impossible for lower-level debugging frameworks to capture (§7.2.2).

Testing bug fixes is inherently unsound, even without Cascade (§7.1). However, replay for an edited execution occasionally leads to confusing results; in our experience, these incidents usually involve the replaying of GUI events. For example, if an edit changes the visual locations of DOM nodes, then replaying (say) a mouse click to a particular \((x, y)\) coordinate may result in unexpected event handlers firing. Cascade's ability to diff the data flows and control flows of a logged execution and an edited one can identify the reason for divergence, but developers must be diligent about checking the diffs when exploring counterintuitive behaviors in a post-edit replay.
7.4 Implementation

To log deterministic reads and writes to the JavaScript heap and the DOM, Cascade uses a modified version of Scout. The default version of Scout (described in Chapter 3) rewrites JavaScript and HTML, injecting instrumentation that runs during each read or write to JavaScript or DOM state. Cascade extends Scout so that it logs nondeterministic JavaScript events like mouse clicks and timer firings. At replay time, Cascade reconstructs data flows using the low-level Scout traces. Cascade defines a default set of data flow manipulations, like targeted dynamic traces (§7.3.2). However, Cascade stores raw data flow logs in a simple JSON format, and defines a plugin model which allows developers to create their own queries. To display data flow graphs, Cascade uses the NEATO visualization library [66].

At replay time, Cascade injects a custom JavaScript library into the application code that runs on a browser or a Node instance. The library acts as a replay driver, dispatching high-level events from the log as requested by the human developer who is managing the debugging workflow. The event dispatch process is similar to that of prior replay frameworks like Mugshot [106] or Jardis [13], although Cascade dispatches events across multiple hosts during wide-area replay (§7.3.4). Black-box components like Redis are logged and replayed using a component-specific replay proxy (§7.3.4).

To support speculative edit-and-continue, Cascade must be able to modify the code or data belonging to a paused JavaScript execution context. One implementation option would be to change the C++ code inside a JavaScript engine to expose mutation hooks for internal state. Our Cascade prototype uses a different approach—it executes the replaying JavaScript code atop MetaES [15], a JavaScript interpreter that is written in JavaScript. This approach allows Cascade to be used with arbitrary client browsers or Node implementations, since Cascade can mutate application state without assistance from the underlying JavaScript engine. A developer expresses a pause point as a 2-tuple consisting of a source code line and a trigger condition, e.g., “the i-th iteration of the enclosing loop.” Cascade will pause the MetaES interpreter at the appropriate moment. The developer can inspect the program state, devise an edit, and then express that edit to Cascade in the form of a JavaScript statement for Cascade to evaluate. After evaluating the edit, Cascade resumes the program’s execution, using the semantics described in Section 7.3.3 to handle any post-edit divergences from the logged behavior in the original execution.

7.5 Evaluation

In this section, we demonstrate two things. First, whole-program data flow analysis, and speculative edit-and-continue, are both computationally practical. Second, these primitives are useful for diagnosing real bugs in complex programs.
Intuition might suggest that logging all deterministic and nondeterministic events would be prohibitively expensive. However, Figure 7-5 shows that the median number of reads and writes that occur during a page load are 13,275 and 6,328, respectively. That number is perhaps surprisingly low, given the fact that an average web page includes 401 KB of JavaScript source code [179]. However, diagnosing bugs is still tricky: a graph with thousands of nodes is small enough to be efficiently analyzed by a computer, but big enough to be difficult for a human to understand. For example, across the Alexa Top 300 pages [4]:

- the median number of object aliases was 4, and the 95th percentile was 18;
- the median number of writes per variable was 8, with a 95th percentile of 210;
- the median number of unique source code lines that wrote a variable was 5, with a 95th percentile of 22;
- when considering the final value for each variable, the median length of the value’s provenance chain (§7.3.2) was 16, with a 95th percentile of 131.

These statistics are for the JavaScript code which executes during a page load. After the load completes, additional JavaScript executes in response to GUI interactions, the firing of timers, and so on. Executing post-load JavaScript results in more reads and writes for Cascade to track, but the volume is low compared to the activity that is generated by the initial page load. For example, on the wsj.com website, hovering over a menu item at the top of the page will trigger several event handlers for mouse activity. However, firing these handlers only generates 486 reads and 107 writes. For comparison, the initial page load generates 33,844 reads and 16,121 writes.
Figure 7-6: Overhead of tracking all reads and writes in a page. The overhead is expressed as the amount by which Cascade’s instrumentation slows down page load time (PLT). Results are for the entire 300 page corpus, and are similar for faster last-mile bandwidths.

Figure 7-6 shows that Cascade's logging activity has minimal impact on overall page load time. Thus, Cascade's client-side instrumentation is fast enough to add to real, customer-facing pages. The logs for those pages will grow slowly: across our 300 page corpus, the median (gzipped) log size after a page load was 45.4 KB, with a 95th percentile size of 113.2 KB. Given such a log, Cascade required a median of 7.8 seconds to generate a full data flow graph; the 95th percentile time was 32.3 seconds. Note that graph generation can be performed in the background during the logging run. Thus, much or all of the cost of graph generation can be paid before a human developer begins the debugging process.

Section 7.5.3 describes server-side logging costs. For now, we merely note that, like the client-side overheads, the server-side overheads are low.

7.5.2 Debugging Case Studies

In this section, we describe our experiences with using Cascade to debug web applications that we did not create, and whose code we had no previous familiarity with. The case studies described in this section also cover scenarios that involve wide-area, speculative edit-and-continue.

7.5.2.1 Mailpile

Mailpile [94] is an email client which uses a browser-based GUI. The GUI, shown in the top part of Figure 7-7, is a sophisticated piece of software that contains 121 HTML files, and 70 JavaScript files that span 15,329 lines of code. As shown in Figure 7-8, Mailpile is one of the most complicated sites in our test corpus, having a large number of reads, writes, and live objects in the JavaScript heap and in the DOM.

Mailpile has a public bug database, so we decided to use Cascade to diagnose some unresolved bugs. For example, Bug #1771 is triggered when a user tries to archive the messages in an email thread. The thread should disappear from Mailpile’s display of active, non-archived threads; instead, the GUI does not change, and Mailpile throws a JavaScript exception (“Uncaught TypeError: Cannot read property ‘split’ of undefined”). We recreated this bug, recording a session in which the user composes and sends two new emails, reads several preexisting emails, archives an email thread (generating an exception), and then composes a new email.

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>RTT</th>
<th>Median PLT Slowdown</th>
<th>95th Percentile PLT Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 Mbits/s</td>
<td>200 ms</td>
<td>2.7%</td>
<td>5.3%</td>
</tr>
<tr>
<td>12 Mbits/s</td>
<td>100 ms</td>
<td>3.8%</td>
<td>6.6%</td>
</tr>
<tr>
<td>12 Mbits/s</td>
<td>50 ms</td>
<td>5.5%</td>
<td>8.9%</td>
</tr>
</tbody>
</table>
Figure 7-7: A buggy Mailpile session. The bottom part of the diagram shows targeted dynamic traces for several DOM nodes. The i-th executed source code line is prefixed with lineNumInSrcFile(i). Black arrows represent data flows between executed source code lines: data written by the source code at the base of an arrow is read by source code at the head of the arrow. The blue and red text was manually added to highlight specific parts of the debugging narrative from Section 7.5.2; Cascade generates the information in the rest of the diagram.

<table>
<thead>
<tr>
<th>Mailpile</th>
<th>EtherCalc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total writes</td>
<td>24,202</td>
</tr>
<tr>
<td>Total reads</td>
<td>67,335</td>
</tr>
<tr>
<td>JavaScript heap objects</td>
<td>2,822</td>
</tr>
<tr>
<td>DOM nodes</td>
<td>619</td>
</tr>
<tr>
<td>Wall-clock time to bug at logging time</td>
<td>12.8 secs</td>
</tr>
<tr>
<td>Wall-clock time to bug at replay time</td>
<td>3.1 secs</td>
</tr>
</tbody>
</table>

Figure 7-8: Summary statistics for the Mailpile and EtherCalc case studies. Note that, during replay, Cascade can skip user think time, so Cascade can replay a buggy execution faster than it originally occurred.
The exception was thrown by a loop inside of Mailpile's event handler for clicks on the “archive thread” button. The loop iterated through DOM nodes whose class name started with .pile-message-. For each such node, the loop tried to access the .tids property of each node. In particular, the failing line of code tried to perform $(\cdot pile-message-\cdot + i).data(\cdot tids\cdot).split(\cdot , \cdot)$. Finding the source code location of this problematic line was easy with both Cascade and the browser’s standard JavaScript debugger. However, using Cascade, we could quickly explore the temporal evolution of program state, try speculative bug fixes, and then observe how post-edit replays unfolded.

The first debugging task was to learn which DOM nodes did have .tids properties, and how those properties were assigned. To discover this information, we replayed the program to just before the exception, and searched backwards through the dynamic trace of the loop to find DOM nodes which didn’t trigger exceptions. For each of those nodes, we then traced backwards through the targeted dynamic trace for that node (§7.3.2); computing and rendering each targeted dynamic trace took 2 seconds on average. The bottom part of Figure 7-7 shows a simplified dynamic trace for the DOM node which triggered an exception (the content <div>) and a DOM node which did not trigger an exception (the message <tr>). The targeted traces quickly allowed us to determine that .tids properties were set by the Mailpile event handler for clicking upon the “send mail” button; each .tids value was a random string created by the generate_tid() function.

Having identified the proximate cause of the exception, our next task was to devise a bug fix. Hypothesizing that 1) all instances of the .pile-message-* class should have .tids properties, and 2) the failure to assign such a property was a bug, we edited the mail-sending event handler to forcibly assign .tids to every DOM node that was associated with the message-to-send. We performed this edit at the beginning of replay, before Cascade had dispatched any events. We then started the replay. The edit successfully removed the exception and allowed Mailpile to refresh the GUI. Replay proceeded, with the composition of the new email succeeding. However, throughout the post-edit replay, the GUI displayed improperly-large counts for the number of messages in each thread. Dynamic traces for state touched by the counting code revealed that every DOM node with a .tids property was counted as a message.

The failure of this fix suggested that the problem lay not in the mail-sending function, but in the thread-archiving function—perhaps the thread-archiving function had a mistaken belief (not shared by the rest of the code) that all instances of the .pile-message-* class should have .tids properties. We tried another, more trivial fix in which we injected a try/catch block around the exception-generating statement in the thread-archiving function, such that the catch block issued a continue statement which resumed the function’s iteration through the messages in the thread. This fix resulted in Mailpile successfully clearing the GUI of the archived message. The GUI’s message counts were correct, and replay continued past the (no-longer-thrown) exception to the composition of a new email.

By differencing the data flow graphs of the original execution and the speculative execution with the second fix, we verified that the second fix only led to state changes in archived messages—the rest of the application was unaffected. However, this
observation did not prove that the fix was “correct” in a deep sense. Indeed, the fix was
distressingly shallow; the exception-throwing line of code implied a mismatch between
the expectations of the developer and the actuality of the program with respect to
which DOM nodes had .tids fields. However, we had not identified the precise
mismatch. To further analyze the problem, we examined the targeted dynamic traces
for all of the DOM nodes which visually represented messages in the GUI. We saw that
Mailpile represented each email thread as an HTML table, assigning each table element
(i.e., each <td> and <tr> element) a class with a prefix of .pile-message-. However,
the mail-sending function only assigned .tids properties to <td> elements, which
represented the subject line in an email thread. The thread-archiving function was the
only part of the dynamic trace which (mistakenly) assumed that all .pile-message-*
DOM nodes would have a .tids property (hence the exception thrown when trying
to access a method on a non-existent property).

Diagnosing this problem would certainly be possible with a traditional debugger.
However, Cascade’s speculative edits and targeted dynamic traces made it easy for us
to explore a buggy execution and test hypotheses without deep, a priori knowledge of
the code. At the time of this chapter’s writing, Bug #1771 had been unresolved for
over six months. We have reported our findings to the Mailpile development team,
and we are awaiting their response.

### 7.5.2.2 EtherCalc

As shown in Figure 7-1, EtherCalc is a collaborative spreadsheet application. A single
document can be simultaneously viewed and edited by multiple users, with a Node
server disseminating updates across browsers, and storing the canonical spreadsheet
state in a Redis database. EtherCalc is the largest application that we examined,
consisting of 36 HTML files, 899 JavaScript files, and 232,662 total lines of code.
Figure 7-8 provides additional statistics about the application.

Bug #314 involves a broken propagation of auto-fill operations between two
browsers. In an auto-fill operation, a user enters data (e.g., “1,2,3”) into a few
exemplar cells; the user then highlights the cells, and drags the bottom edge of the
highlighted region downward, causing the spreadsheet to guess the pattern in the
exemplar cells and automatically apply the pattern (e.g., “4,5,6”) to subsequent cells.
In Bug #314, auto-fill operations that are generated on one browser are not properly
delivered to other browsers. In the example above, the first browser would correctly
display “1,2,3,4,5,6”, but the second browser would display “1,1,1,1,1,1”. We recreated
this problem, recording a buggy session that involved two browsers, a Node server,
and a Redis server. In the buggy session, a user on the first browser employed auto-fill
functionality on two different batches of cells. In both cases, the auto-fills were
improperly reflected in the GUI of the second browser.

This bug could have been caused by faulty behavior in one, some, or all of the
four components (i.e., the two browsers, the Node server, and the Redis server). We
started the debugging process by looking for data flows between the first browser
and the Node server, hypothesizing that the first browser was sending an incorrect
update to the Node server, and the server was then propagating the faulty update to
the second browser. We temporally pruned the data flow graph to start immediately before the user selected the highlighted region to auto-fill. We found the relevant data flow between the first browser and the Node server; by examining the associated execution trace, we found that the Node server was indeed receiving an improper operation (\texttt{"extend('1,1,1,1,1,1")"}) to distribute to other clients. Looking at the client-side of the execution trace, we saw that the browser generated the operation to send by calling a function named \texttt{ExecuteSheetCommand()}. This function accessed a global variable called \texttt{range} that indicated the start cell and end cell for the initial pattern (e.g., “1,2,3”) to be used as the basis for the autofill. The start cell and end cell were undefined, even though an auto-fill operation had already occurred on the browser! Tracking backwards through the targeted dynamic trace for the range object, we saw that the \texttt{range} object was reset at the end of an earlier call to \texttt{ExecuteSheetCommand()}. This earlier call appeared to be the source of the bug. EtherCalc uses operation replaying to distribute an edit to other browsers; the initiating browser calls \texttt{ExecuteSheetCommand()} once to apply an operation to the local EtherCalc state, and then calls \texttt{ExecuteSheetCommand()} again (but with slightly different parameters) to build an operation that is not applied locally, but instead is sent to the Node server, stored in Redis, and then distributed to other clients. The first call to \texttt{ExecuteSheetCommand()} appeared to be incorrectly resetting the \texttt{range} object to represent an empty range, such that the second call did not find the exemplar cells for the auto-fill operation, and therefore applied a default auto-fill, leading to the “1,1,1,1,1,1” displayed by the second browser.

An obvious potential fix was to modify \texttt{ExecuteSheetCommand()} so that the first call in a pair of invocations did not clear the \texttt{range} object. We made this edit and then performed a speculative replay. The edit initially appeared successful—the Node server received the auto-fill operation, stored it on Redis, and then sent it to the second browser, who correctly applied the operation. However, our recorded session contained two auto-fill operations involving two distinct sets of cells; our hypothesized fix prevented the second auto-fill operation from appearing in the GUI of the first browser or the second browser. Looking at the distributed data flow, we saw that the first browser was not generating a second auto-fill locally, or sending a second auto-fill operation to the Node server.

Further investigation of \texttt{ExecuteSheetCommand()}’s code revealed that the first call in a pair of invocations expects the \texttt{range} object to be set to a default value—otherwise, \texttt{ExecuteSheetCommand()} terminates without updating the local spreadsheet, or sending an update to remote clients via the Node server. We tried a different bug fix in which the second call to \texttt{ExecuteSheetCommand()} clears the \texttt{range} object at the \texttt{end} of \texttt{ExecuteSheetCommand()}’s execution. The speculative replay for this fix led to no problems—both auto-fill operations were properly displayed on both browsers.

Of course, the successful speculative replay was not a proof of the fix’s correctness; the successful replay was essentially the successful passing of a unit test involving a particular usage scenario. However, this case study demonstrates how replay debugging, wide-area data flow tracking, and speculative edits work in concert to ease the cognitive overhead of understanding large code bases.
7.5.2.3 An nba.com Article

The DOM interface is complex, and a rich source of bugs [138]. Cascade tracks data flows through both the JavaScript heap and the DOM, so Cascade can diagnose faults that involve buggy DOM interactions. For example, the author from the first case study noticed that an article on the nba.com site was throwing an exception: “TypeError: Cannot set property ‘innerHTML’ of null”. The .innerHTML property is defined by DOM nodes; writing to the property dynamically overwrites a node’s set of child HTML tags. For some reason, the nba.com page had a null reference that should have pointed to a DOM node.

Using Cascade, we were able to quickly determine the provenance of this erroneous value, without possessing deep, a priori understanding of the code. We determined that the error arose from confusion between the page’s HTML and two JavaScript libraries that were written by different parties. The page’s statically-defined HTML file contained a <div> tag with an id of “Popular Topics.” An inline script (written by the developers at nba.com) used the .getElementById() and .firstChild() DOM methods to generate a reference to the “Popular Topics” tag. The script then added new DOM nodes to the tag, and changed the tag’s id to be “Latest News.” Later in the page’s statically-declared HTML, there was an external script from a third party. The goal of this script was to add new headlines from third-party-affiliated sources. This external script used .getElementById(“Popular Topics”) to ostensibly assign a DOM node reference to a variable named z. The script then tried to invoke a method on z, unaware that the DOM call had returned null because the page no longer contained a tag with an id of “Popular Topics.”

This gory example demonstrates the importance of threading data flows through the JavaScript heap and the DOM. Using Cascade’s data flow analysis, we traced the provenance of the null value, and determined that the page expected to find a DOM node with an id of “Popular Topics.” We then searched the full-program data flow graph for the creation of a DOM node with that id. Once that object was found, we rolled forward in the data flow graph and watched the evolution of the object, seeing it receive new children and then have its id reset to “Latest News.” Diagnosing this kind of fault is extremely tedious with standard breakpoints or watchpoints—developers must repeatedly guess where to place breakpoints, or which state to watch, and then re-execute the program, hoping that the bug reoccurs, and that the new breakpoints or watchpoints were correctly positioned to catch the error.

7.5.2.4 Bleacher Report

bleacherreport.com is a popular site for sports news. During our implementation of Cascade, an author noticed that the front page for the site was throwing a JavaScript exception that was output to the browser’s error console. The specific error message was “TypeError: item.get is not a function”; the error was generated because the page expected item to be an object, but item had somehow been set to the boolean value false.

Using Cascade’s data flow analysis, we were able to quickly determine the prove-
nance of this erroneous value, without possessing deep, a priori understanding of the code. We determined that bleacherreport.com used Marionette [10] to manage the layout of “views.” Each view represented a visual element that was associated with a type, a parent view, and child views. The page defined view types like “game” and “Tweet.” The site created a variable called View1 that was intended to only contain views of type “game.” However, a non-“game” view was mistakenly added. Later, when the site iterated over View1’s children and found the unexpected view, the site overwrote that view, assigning false to the view’s position in View1’s list of children. Hundreds of executed statements later, the site iterated through View1’s children, calling .get(“type”) on each one, and eventually throwing a TypeError.

This error spanned 3 different JavaScript files; some were provided by Marionette, and others were written by Bleacher Report developers. Using a value provenance graph for the erroneous false value, we could easily trace the execution sequence which ultimately led to the exception.

7.5.2.5 Wide-area debugging

jQuery [165] is a popular client-side library that provides a high-level interface to low-level DOM methods. jQuery has a public bug-tracking system. To test Cascade’s support for wide-area debugging, we recreated a subtle bug that affected jQuery v1.9.1 [166]. Prior to that version, invoking jQuery’s .data() method on a non-existent DOM node would return undefined. In v1.9.1, such a call would return null. This behavioral change could modify the data that clients send to servers via HTTP POST—jQuery’s .post() function sends no data for undefined values, but sends empty strings for null values. As described by the bug report, server-side parsing code which expected pre-1.9.1 behavior might fail, or return unexpected results to clients.

The original bug report did not provide the full source code of the broken server. Thus, we created a simple photo gallery application which emulated problematic server-side behavior. The application used jQuery to display pictures and communicate with the back-end server. A user could “like” or “dislike” a photo by clicking buttons next to the photo. The page maintained two global counters which tracked the total number of photos that the user liked or disliked. The page displayed the counter values in two <div> tags. Each <div> tag was created at the time of the first relevant vote; for example, if the user had clicked “like” on five photos, but disliked no photos, the page would have a <div> tag for “likes,” but no <div> tag for “dislikes.” The page used jQuery’s .data() and .post() methods to send the user’s votes to a Node server. Upon receiving such an HTTP POST, the server would parse the request, extract the votes, and then respond with an aggregate count of likes and dislikes across all users of the application. The client would then update its GUI to inform the user of the aggregate vote statistics.

The server used the built-in parseInt() function to extract client votes from a tokenized HTTP POST query string. In pre-1.9.1 jQuery, the query string from the example above would be “likes=5”. In v1.9.1 jQuery, the string would be “likes=5&dislikes=null”. The latter string would cause the server to invoke parseInt() on an empty string.
The resulting NaN value would be sent back to the client, who would display “NaN dislikes” to a now-confused user.

Using Cascade’s data flow analysis, a developer can quickly identify the root cause of the problem. For example, suppose that the developer has two variants of the client-side code, one of which uses jQuery v1.9.1, and another which uses a pre-1.9.1 version. The developer can load each variant of the page, and in each variant, click “like” on the same photo five times. The developer can then use Cascade to diff the two data flow graphs. The graphs will track wide-area data flows, so the developer can trace the “NaN dislikes” value back to the server invocation of parseInt() on an empty string. Tracing backwards from the parseInt() call, the developer will discover the divergent HTTP POST requests that the server received from the two clients. The developer can then follow the data flow back to the client-side, and determine the root cause of the differing query strings (namely, the differing return values of jQuery’s .data() method).

Once the root cause is found, the developer can use speculative edit-and-continue to test different bug fixes. As a concrete example, we replayed the v1.9.1 version of the application up to the server’s event handler for the HTTP POST request. We edited the server’s code to check for empty strings before invoking parseInt(); if an empty string is found, the server avoids calling parseInt(), and directly assigns 0 to the variable used to create the relevant part of the HTTP response. Once the edit was complete, Cascade rewrote downstream log entries to ensure that the client would replay the edited response data. After resuming the replay, we verified that the client GUI in the altered universe displayed “0 dislikes” instead of “NaN dislikes.”
7.5.2.6 CPU-bound code

Most JavaScript applications are IO-bound, waiting for inputs from the user or the network [146]. As described in the Bleacher Report and jQuery case studies, data flow graphs help to clarify the movement of information across asynchronous IO events. However, data flows graphs can also explain the behavior of CPU-intensive code. As a concrete example, consider SHA256. Like many hash functions, SHA256 use multiple iterations to consume parts of an input, scramble that input, and feed the scrambled result into the beginning of the next iteration. The reference specification of SHA256 has a branch statement which controls the particular block of input which is processed during the current round:

```java
if (i < 16){
    //use i-th block
} else{
    //use the (i-th & 15) block
}
```

However, as a developer translates the reference specification into a real implementation, transcription errors may arise. For example, what if the developer accidentally uses the branch test `i < 15` instead of `i < 16`? That simple change will ripple throughout the data flow graph, leading to the hash function emitting a radically different output. Cascade’s data flow graphs can provide succinct analyses of those ripple effects. Figure 7-9 shows the graph diff for two different implementations of SHA256, one of which uses the correct branch test, and one of which uses the mistranscribed test; the correct implementation came from the Stanford JavaScript Cryptographic Library [158]. Cascade shows how the initially small differences of intermediate values have increasingly larger downstream effects.

7.5.3 Server-side Overheads

Cascade’s logging approach for a Node server is similar to Cascade’s logging approach for a client-side browser (§7.3.4). However, a Node server that handles many clients will produce log entries more quickly than a client browser which loads a single page and then intermittently handles user input. To examine Cascade overheads on Node, we wrote a simple Node web server. For each request, the server returned the dynamic string “Hello world at ” + (new Date()).getTime(). For each request, Cascade had to log the incoming HTTP request, a few dozen reads and writes inside the server’s request handler, the timestamp returned by `Date()`, and the outgoing HTTP response. This toy server was a pessimistic test of Cascade’s overheads, since real server code has a higher ratio of executed source code lines per nondeterministic value logged.

We used the Apache benchmarking tool `ab` [9] to generate HTTP requests. We placed the Node server and `ab` on the same machine, to emphasize Cascade’s computational overheads. As shown in Figure 7-10, we varied the number of concurrent client requests from 25 to 10,000, measuring response throughput for a normal version of the server, and a Cascade-enabled variant. The throughputs of the two servers were 123
Figure 7-10: Response throughput for two versions of a Node server. Each data point represents the throughput across 100,000 requests.

within 3% of each other. CPU utilization was also similar for the two servers.

The growth of Cascade’s compressed log was 258 Kbps (equivalent to 32.3 KB per second). Note that black box components like Redis have slower log growth—for these components, Cascade logs incoming requests and responses, but not deterministic reads or writes to internal black-box state.

7.5.4 Replay Subtleties

A single web page can embed content from multiple origins. Cookies are isolated using the same-origin policy [119], so a page from origin foo.com cannot access cookies that are set by (say) <img> fetches to bar.com. Thus, JavaScript code in the enclosing foo.com page cannot read bar.com’s latest vector clock. An in-browser implementation of Cascade can easily avoid this problem by allowing cross-origin cookie accesses when the browser is running in debug mode. A JavaScript-level implementation of Cascade must force all remote servers to reside in the same origin. This is often infeasible for the production version of a complex page, but possible for a testing version in which all page content is recorded using Mahimahi [132] or Fiddler [164], and then served from a single proxy that rewrites URLs to point to the proxy’s origin.

A JavaScript-level implementation of Cascade must also be careful to replay load events properly. These events cannot be synthetically generated or deferred by JavaScript code, since JavaScript code has no ability to force the network stack to release bytes at controlled intervals. So, to properly replay the load event for a passively fetched object like an <img>, Cascade must ensure that, from a client’s perspective, the <img> (and its vector-clock-containing cookie) arrive at a time that respects the vector clocks in the client-side log events. Practically speaking, this
means that the server-side replay driver must coordinate with the client-side driver, and only release the last byte of a passively-fetched object when the wide-area replay has reached the appropriate point [106].

### 7.5.5 Pruning Data Flow Graphs

As shown in Figure 7-7, Cascade's primary visualization depicts a dynamic execution trace. Each column expresses the source code lines that wrote or read a particular JavaScript variable or DOM node; arrows that are rooted in a particular source code line \( X \) indicate which temporally-subsequent source code lines read data that was written by \( X \). For a single web page load, the visualization for a full dynamic trace is typically thousands of columns wide (because the distributed application has many variables) and tens of thousands of rows tall (because the distributed program executes many source code lines). Thus, pruning strategies are important for maintaining the readability of dynamic traces. Cascade exposes several pruning techniques:

- The most important one is the ability for developers to specify which subset of variables should be analyzed by Cascade. Starting from zero variables, a developer iteratively adds new variables (or deletes old ones) as new diagnostic hypotheses are generated. This pruning approach underlies the notion of a targeted dynamic trace (§7.3.2, §7.5.2); as a side effect, this approach prunes the number of rows in the visualization, since most variables are only updated a few dozen times at most.
- Cascade also allows developers to vertically prune a graph, by specifying a particular time period for which Cascade should display data flows for the targeted variables. Such temporal bracketing prunes columns as a side effect, since Cascade will ignore variables that send or receive values to/from a target variable if those source/sink operations are outside the bracketed time window.
- A developer can prune by the origin of JavaScript code, such that Cascade only visualizes trace data for variables that were written by code from a particular hostname. This pruning strategy is useful for assigning blame if an application contains multiple server-side Node instances, and/or multiple client-side browsers which load JavaScript libraries from multiple origins.
- Cascade allows developers to selectively disable trace output for all JavaScript heap variables, or for all DOM state. This pruning technique is useful when a bug is hypothesized to involve only JavaScript state, or only DOM state. This technique is also useful when a bug involves both types of state, but, e.g., the developer already understands how the JavaScript state evolved during the logged execution, and desires to focus on how the DOM state changed.
- For a particular variable, a developer can enable or disable the visualization of control flow dependencies. The developers can also globally enable or disable these visualizations.
- Finally, Cascade supports the visual diffing of traces. This is useful to understand divergences between a logged execution, and a speculatively-edited replay.

In our experience, these techniques make the comprehension of trace data tractable for a human developer.
Chapter 8

Conclusion

To conclude this dissertation, we outline several avenues of future work motivated by the results we presented. We then summarize the key takeaways from this dissertation and discuss the broader implications.

8.1 Future Work

We believe that the capture and analysis of fine-grained data flows will have even broader implications for web applications beyond those presented in this dissertation. We discuss several potential directions below.

Enforcing Security and Privacy Policies: Web applications increasingly must enforce security and privacy policies to ensure that certain requirements are met. For example, browsers enforce the Same Origin Policy [119] to control how different client-side state in a web page load is accessed. Specifically, this policy mandates that state in a page load is isolated at a frame level; content in one frame cannot access JavaScript heap or DOM state belonging to another frame (Chapter 2). Enforcing this policy is increasingly challenging as web applications support more features and include content from increasing numbers of domains.

Many existing approaches to enforce client-side security policies require browser modifications to track data flows (e.g., through explicit labeling [84]). In contrast, we believe that web pages can use knowledge of fine-grained data flows to automatically enforce those policies. The reason is that data flows inherently capture how and where (potentially private) state is accessed, and by whom. Thus, we believe that data flows can be used to enforce security policies at runtime. For example, web pages could be automatically instrumented to track data flows. This information can then be used to either flag or prevent disallowed accesses to secure state. This approach could also be ported to enforce server-side access policies (discussed below).

Automatic Task Offloading: Mobile applications commonly offload compute-intensive techniques like big data analytics and machine learning algorithms to remote servers. Though job completion times are lower on these servers than on mobile devices, offloading tasks can add additional network latency to applications. As a result, applications must perform a balancing act to provide rich functionality while
respecting stringent user latency demands. This entails identifying compute-intensive tasks to offload, and then making intrusive changes to the application code base to robustly support offloading while minimizing perceived delay.

We believe that fine-grained data tracking can help automate this process. Prior approaches either require developers to manually partition applications [48, 111], or to run multiple instances of applications on distributed servers for coarse-grained method offloading [34]. In contrast, by reasoning about low-level execution, we can automatically partition applications into local and remote components that optimize for certain performance metrics (e.g., user-perceived latency and resource utilization) and can be run using generic computing servers (e.g., serverless compute platforms [8, 111]). For example, applications could be divided into “code blocks” by analyzing how the execution of individual source code lines affects CPU utilization and data flows. Code blocks could then be distributed to closely match the performance requirements specified by developers; for instance, code blocks which share data often should be run in the same location. Data flow analysis can also be used to patch applications after offloading such that no program values are missing for a code block prior to execution.

**Debugging and Monitoring Microservices** Many web applications like Uber and Netflix have adopted the “microservice” architecture, in which backends are composed of hundreds or thousands of small, loosely coupled services [97, 74]. This approach enables rapid development and testing of individual services. However, the proliferation of cross-service interactions creates significant problems for debugging and performance optimization.

Much like we demonstrated for web pages, fine-grained data flow tracking could be used to simplify these tasks in microservice-based applications. Existing tracing frameworks for distributed systems log coarse-grained events such as incoming client requests, particular system call invocations, or the execution of manually-specified lines of code [92, 93, 154]. In contrast, we propose that a distributed application log all reads and writes to all program variables. By tracking this information, we can better understand the execution behavior of each microservice, and recreate a “global” view of the overall distributed system. Cascade (Chapter 7) tracked server-side data flows with web applications, but assumed few server-side components. Microservice environments raise new scalability and heterogeneity questions with respect to the capture and processing of fine-grained data flows. How can we quickly process data flows to support real-time monitoring? Can we prune logging or storage to improve efficiency, without sacrificing insights? How can we track provenance across services that use diverse data schemas and storage layers?

### 8.2 Takeaways and Broader Impact

The main contribution of this dissertation is the introduction of fine-grained data flows as a new mechanism for understanding and optimizing increasingly complex web applications. We presented a practical system called Scout that can efficiently track the precise data flows in a web page load. We then described four concrete that leverage Scout to improve various aspects of web applications:
• Polaris (Chapter 4) dynamically reorders client requests in a page load to minimize network round trips without violating data flow dependencies, reducing page load times by 34% (1.3 seconds).

• Prophecy (Chapter 6) generates a snapshot of a mobile page’s post-load state, which clients can directly process to elide intermediate computations that are normally discarded. Prophecy results in reductions of 21% in bandwidth consumption, 36% in energy usage, and 53% (2.8 seconds) in page load times. By using data flow graphs to guide the construction of snapshots, Prophecy provides numerous advantages over prior snapshotting techniques, including support for object caching and incremental interactivity.

• Vesper (Chapter 5) is the first system that measures how quickly a page becomes interactive. Vesper determines a page’s interactive state, which is not explicitly annotated, by finding and executing event handlers, and analyzing the corresponding data flows. User studies show that users prefer pages that optimize for interactivity. However, existing metrics inaccurately measure the time until a page is interactive by 39%. Optimizing for these metrics forgoes 32% of the benefits of targeting interactivity directly.

• Cascade (Chapter 7) is the first replay debugger that makes fine-grained data flows explicit and queryable, enabling provenance tracking across multiple clients and servers. Cascade also supports speculative bug fix analysis, where a developer can replay a program to a certain point, change code in the program, and then resume replay to evaluate if the hypothesized bug fix would have helped the original execution.

There are two main takeaways highlighted by these systems. First, capturing fine-grained data flow information at the managed runtime and event-loop level is efficient enough to run in production. Second, the low-level system information provided by fine-grained data flows enables more aggressive optimizations and debugging tools.

Moving forward, we hope that the results in this dissertation motivate the capture and analysis of fine-grained data flows in distributed systems. In particular, we hope to have illustrated the tradeoff spectrum between the insights provided by data flows and the tracing overheads associated with data flow collection. Of course, capturing all low-level data flow information is not practical in certain systems. However, through careful selection of tracing abstractions, we believe that data flows can be efficiently collected in many instances, providing clarity into the complex execution of distributed systems. Such clarity can alleviate the need to build optimizations, measurement systems, and debugging tools that rely on heuristics.
Bibliography


[12] G. V. Bard. Spelling-error tolerant, order-independent pass-phrases via the
damerau-levenshtein string-edit distance metric. In Proceedings of the Fifth
Australasian Symposium on ACSW Frontiers, ACSW. Australian Computer


org/html/rfc6265, April 2011.


PFDGMi, April 8, 2010.


Web Server Design. World Wide Web Conference on Computer Networks :


November 2007.

[22] E. Bocchi, L. De Cicco, and D. Rossi. Measuring the Quality of Experience of
Web Users. In Proceedings of the 2016 Workshop on QoE-based Analysis and

[23] A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the Eye of the Beholder:
Meeting Users’ Requirements for Internet Quality of Service. CHI, The Hague,
The Netherlands, 2000. ACM.


143


