UNIVERSITY OF CALIFORNIA

Los Angeles

Bridging the Performance Gap Between Mobile Applications and Mobile Web pages

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Shaghayegh Mardani

2022

ABSTRACT OF THE DISSERTATION

Bridging the Performance Gap Between Mobile Applications and Mobile Web pages

by

Shaghayegh Mardani
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2022
Professor Ravi Arun Netravali, Co-Chair
Professor Todd D. Millstein, Co-Chair

Despite the rapid increase in mobile web traffic, page loads still fall short of user performance expectations. Numerous solutions have attempted to optimize the web performance, however, these state-of-the-art techniques are either ineffective or impractical in real-world settings due to complexity or deployment challenges. Inspired by mobile apps which provide faster user-perceived performance, we target two significant bottlenecks in the page load process; First, clients have to suffer multiple round trips and server processing delays to fetch the page's main HTML; during this time, a browser cannot display any visual content which frustrates users. Next, these pages include large amounts of JavaScript code in order to offer users a dynamic experience. These scripts often make pages slow to load, partly due to a fundamental inefficiency in how browsers process JavaScript content which fails to leverage the multiple CPU cores that are readily available even on low-end phones. We pursue a programmatic approach that works on legacy web pages and unmodified browsers. We built two fully-automatic systems that are complementary to each other and each optimizes one of the above contributors to slow page loads: Fawkes and Horcrux.

Fawkes leverages our measurement study finding that 75% of HTML content remains unchanged across page loads spread 1 week apart. With Fawkes, web servers extract static, cacheable HTML templates (e.g., layout templates) for their pages offline. Upon client request, the static template is sent back to the client immediately and is rendered while dynamic content (e.g., news headlines) is generated which expresses the updates required to transform those rendered templates into the latest page versions. Fawkes reduces the startup delays incurred during the fetch of page's HTML and improves interactivity metrics such as Speed Index and Time-to-first-paint by 46% and 64% at the median in warm cache settings; results are 24% and 62% in cold cache settings.

Our second system, Horcrux addresses the client-side computation overheads through offline analysis of all the JavaScript code on the server-side to conservatively identify the page state across all loads of the page. Horcrux's JavaScript scheduler then uses this information to judiciously parallelize JavaScript execution on the client-side while ensuring correctness, accounting for the non-determinism intrinsic to web page loads, and the constraints placed by the browser's API for parallelism. Horcrux reduces median browser computation delays by 31-44% and page load times by 18-37%.

The dissertation of Shaghayegh Mardani is approved.

University of California, Los Angeles

2022

To my grandfathers, in loving memory . . .

Babajoon Reza (1935 – 2018)

Babajoon Abdol-karim (1935 – 2022)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor Ravi Netravali to whom I am grateful beyond what words can describe. In 2018, when I was thinking of shifting the direction of my research, I could not have chosen a better advisor. He believed in me even when I was doubtful of my academic strengths. Being Ravi's student allowed me to do research in an environment where I could be passionate about the research problem at hand, explore new ideas, ask the simplest questions, learn how to take charge of my research, and most important of all, feel confident as a researcher. I am grateful to him for his unwavering support, through deadlines and dead ends, for his patience and understanding while I was going through tough parts of my life, and for his drive to push me to be a better researcher and person. He taught me to "stay positive, even though it is hard when you are in the weeds" and to "hyper-plan for my future". I have been extremely fortunate to have him as my advisor, collaborator, mentor, and friend.

I would like to extend my sincere thanks to my committee members, Todd Millstein, Harry Xu, and Mani Srivastava for serving on my Ph.D. committee and for their valuable insights and questions. Todd has been a great mentor, reliable support system, and someone to go to when I faced challenges during my Ph.D. Todd always made himself available for help and advice to students. Our discussions and conversations throughout the years have helped me to gain an improved perspective on research problems and what it means to be a researcher in general.

I also wish to show my appreciation for my collaborators, Harsha V. Madhyastha, Ayush Goel, Ronny Ko, and Mayank Singh for their help and support. In particular, I am thankful to Harsha for being a reliable mentor and for receiving his insights, perspective, and feedback on technical, writing, and presentation aspects while we were working on Horcrux.

During the summer of 2021, I was fortunate to have the opportunity to do an internship with Edgecast (previously known as Verizon Media) research team. Many thanks to my internship mentor Marcel Flores, my manager, Harkeerat Bedi, and the rest of the research team, Anant Shah, Kyriakos Zarifis, and Orpaz Goldstein for such a wonderful internship experience. I truly enjoyed working with such amazing researchers (as well as hanging out with them), being part of their team and their discussions, and facing industrial-scale research problems. I must thank Marcel for his mentorship and his ability to explain challenges and difficult concepts intuitively. I am also grateful to Bedi for his constant support and for offering his wisdom while I was deciding my next steps.

I would like to especially thank Joseph Brown, graduate student affairs officer, who has shown nothing but support and kindness to me during the past 4 years. Joseph has been the first person for me to go to whenever I was facing an issue with anything related to UCLA. His positive attitude and understanding have helped me through many occasions.

I would like to express my deepest gratitude to my family and friends for their unrelenting support and everlasting love during the ups and downs of my Ph.D. journey. It has been a challenging road that has made me a stronger person in the end. They have cheered me on during difficult times, cried with me, put up with my complaints, and celebrated my victories. Even though I have been far away from home and most of my friends are living in different parts of the world, they have never failed to give me their love and support from thousands of miles away virtually.

I am most grateful to my mother, Sepideh, for her never-ending encouragement and her smiles that always brings brightness and light to my days, to my father, Mahmood, for giving me strength and always believing in me, and to my sister, Shabnam, for being my role model in never giving up. I want to thank Aziz, Sara, Atrin, and Mohammad Reza for their constant optimism and the joy they bring into my life.

A huge thanks to my friend and old roommate, Homa Esfahanizadeh, for her everyday support and comfort. I will always cherish our memories together for the four and half years that we were roommates, the times that both of us stayed up late into the night working, went and sat in cafes to work, or the countless times that we walked around the neighborhood and talked about everything.

I would also like to thank Rashin Darvish who made me laugh whenever we talked and always cheered me on to keep going.

I want to show my appreciation to Nasim Rajabi, who stayed with me during my most difficult times and listened to me with infinite patience, and never failed to offer me comfort and/or solutions when I needed them.

Other huge thanks to Nazanin Farahpour, who taught me everything I needed to know when I first came to the US, who always understood my academic concerns, and who has been a constant supporter and when needed a fair critic.

I would like to express my sincerest thanks to Aishwarya Sivaraman who this journey was not possible without her presence and support. She accompanied me on coffee walks while we shared challenges, encouragement, and concerns. I am also thankful to Murali Ramanujam who has been a great friend to me, as well as a collaborator. Aishwarya and Murali gave me a second home throughout my time at UCLA.

I am thankful to Alexandria Davis, my roommate during the past two years, for being the unexpected blessing in my life and for supporting and encouraging me with a lot of talks and bubble teas while I was making the hardest decisions of my life to this point. I am grateful to Julius Young and Trisden Shaw for cheering me on and encouraging me to take further steps out of my comfort zone. The three of them have been a great source of comfort and have given me so many pleasant memories during the ups and downs of the last year of my Ph.D. and for that, I am always grateful.

Many thanks to Shadi Torabi for always supporting me and listening to me, to Soheil Nazar Shahsavani, Farzaneh Khoshnevisan, Neda Shafiee, Siamak Rajabi, and Aida Amini for their support, pleasant conversations, and always caring for me. I want to thank Cameron Danesh, Pooya Vahidi, Aysan Rangchian, Ghazal Ershadi, Farhad Shahmohammadi, Ehsan Qasemi, Hanieh Hashemi, Ehsan Abbasi, and Ehsan Emamjomeh-Zadeh for making some of my best memories during the past 7 years with their friendship.

VITA

2015 – 2021	Research/Teaching Assistant, Computer Science Department, UCLA.
Summer 2021	Research Intern, Edgecast Research Team, Yahoo!
2010 – 2015	B.S. (Computer Engineering), University of Tehran.

PUBLICATIONS

Mardani, S., Goel, A., Ko, R., Madhyastha, H. V., & Netravali, R. (2021). Horcrux: Automatic JavaScript Parallelism for Resource-Efficient Web Computation. In 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21).

Mardani, S., Singh, M., & Netravali, R. (2020). Fawkes: Faster mobile page loads via app-inspired static templating. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20) (pp. 879-894).

Gulzar, M. A., Mardani, S., Musuvathi, M., & Kim, M. (2019, August). White-box testing of big data analytics with complex user-defined functions. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 290-301).

Gulzar, M. A., Han, X., Interlandi, M., Mardani, S., Tetali, S. D., Millstein, T., & Kim, M. (2016). Interactive debugging for big data analytics. In 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16).

CHAPTER 1

Introduction

Mobile web browsing has rapidly grown in popularity, generating more traffic than its desktop counterpart [Eng19, Eth16, Pet19]. Over the past decade the global web traffic has increasingly shifted toward mobile. As shown in Figure 1.1, starting from 2017, mobile web traffic accounts for over half of global web traffic. Along with the ongoing shift toward mobile, user have grown to expect faster and smoother experience from surfing the web on both desktop and mobile platforms [Gooc]. Numerous studies have shown that, although users show various tolerance levels, 57% of users tend to abandon web pages that take longer than 3 seconds to load [BBK00, BKB00, GHM04]. Providing smoother user experience and ensuring user satisfaction is of great importance to online business services as it directly affects their revenue. For instance, taking one second longer to load a page results in 11% lower page views and 16% decrease in customer satisfaction. In dollar terms, if an online business earns $100,000 a day, such one second delay could bring about a $2.5 million revenue loss in a year [Man18].

However, despite accounting for increasing share of global web traffic, mobile browsing in the wild continues to operate far slower then what users can endure, with page loads often taking upwards of 10 seconds [An18, RNU17], even on a state-of-the-art phone and LTE cellular network. The current sub-optimal state of mobile web performance negatively impacts not only user experience, but also the revenue of content providers [EK19], as users are more likely to abandon pages that are slow to load. Consequently, there has been a great number of research done with the goal to improve web page loads both on desktop and mobile platforms.

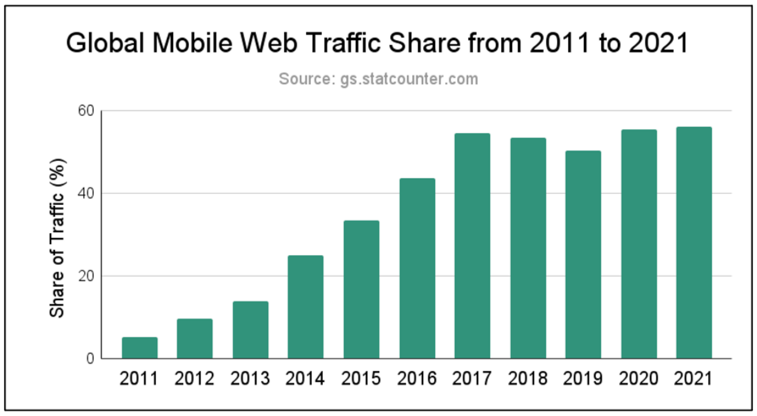


Figure 1.1: Mobile web traffic share over the past decade

1.1 Background and Motivation

Many systems have been developed by both industry and academia to accelerate page loads. Prior approaches have focused on pushing content to clients ahead of time [RNU17, WBK14, ZWH18, EGJ15], compressing data between clients and servers [ABC15, Vol12, SMK15], intelligent dependency-aware request scheduling [BWW15, NGM16], offloading tasks to proxy servers [NSM19, SPG14, BTS17, Ama18], and rewriting pages for the mobile setting (either by automatically serving post-processed objects to clients [WKW16, NM18a, Ope18a], or by manually modifying pages to follow mobile-focused guidelines [Gooa, JBW19]). Yet despite these efforts, mobile page loads continue to fall short of user expectations in practice, due to two reasons: 1) not being effective enough in the complex real world settings, 2) deployment challenges that come at the expense of manual effort and page functionality.

Given the importance of user satisfaction, many businesses offer their services on both mobile web platform as well as mobile applications. User studies show that users perceive mobile apps performance faster than mobile web pages [Tal]. One potential argument is abandoning mobile web in favor of mobile apps. Mobile apps offer a host of advantages over mobile web pages, including allowing access to specific device hardware such as Bluetooth and sensors, storing user preferences on the mobile device, and yet the most prominent, better perceived performance. However, even given all the advantages that mobile apps bring, we cannot completely discard web pages, since web pages offer a distinct set of advantages of their own. Choosing mobile web pages, allows use of services without requiring users to manually install a mobile app on their device or putting an effort to keep the apps updated. Also, mobile web pages provide seamless transition across different domains within one page. Considering the inherit benefits of mobile web pages, we pursue a contrary approach: instead of attempting to *replacing* the web pages with mobile apps, we seek to draw inspiration from mobile apps to *improve mobile web performance*. In particular, this work studies how mobile apps deliver better performance and what are the key contributors to this perceived performance gap between the two platforms, and how to bring the mobile apps benefits to mobile web such that users can exploit the benefits of mobile web pages without paying the cost of slower performance.

1.2 Thesis Approach

The key goal across this work is to improve performance for legacy web pages *pragmatically* without requiring developer intervention to manually rewrite pages or modify web browsers. To achieve this goal, our general approach is to extract web page semantics, obtain runtime information such resource usage, and then use this information to automatically rewrite pages that provide optimized performance.

Figure 1.2 describes my thesis general approach. On the server-side, we programmatically extract insights and constraints from the page and if needed the back-end. These insights and constrains express what the developer indented the page to look like as well as what is deemed as correct final page state and behavior. We use different program analysis techniques to obtain the needed insights and constrains and ensure the correctness of our approach for any given legacy web page. Also, our approach provides a rewritten version of the page without requiring developer effort. The rewritten page includes our custom JavaScript optimizer library that correctly enforces the desired optimizations on the given page during client page load with an unmodified browser, achieving another part of our goal which is avoiding browser modifications.

During the client page load, we use JavaScript to pass in the collected insights and constrains to

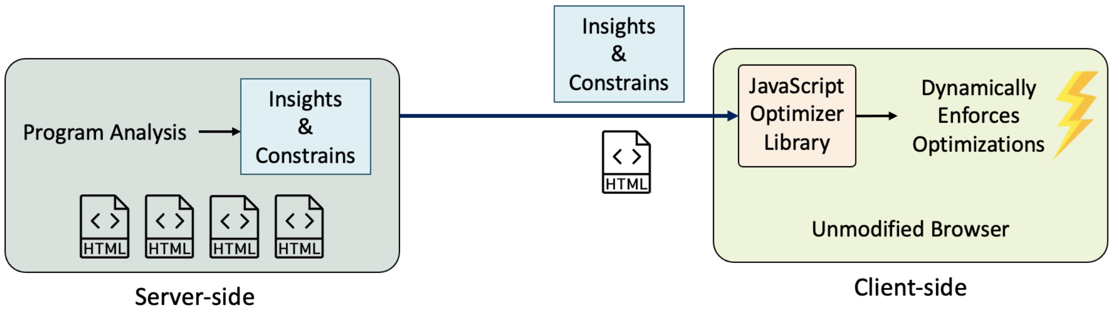


Figure 1.2: Mobile web traffic share over the past decade

the client-side with an unmodified browser along with the rewritten page. Our customer JavaScript library leverages the the insights and constrains generated by the server-side in addition to runtime resource usage information to drive the page load in a manner that makes the best use of resources, such as available CPU cores. Collectively these components provide faster page loads compared to the original legacy page load.

Specifically, I propose two automatic systems that tackle web performance bottlenecks from two complementary aspects: startup delays, and client-side computations:

1. *Fawkes* (§2) targets the slow startup delays in mobile web and provides faster page loads through separating static and dynamic contents of the page, inspired by how mobile apps use static templates to load content on the screen as soon as the app is launched. Fawkes takes a similar approach by sending static contents of the page to the client upon receiving the request, while dynamic parts of the page are generated, hence enabling the client browser to load static contents to the screen well before the back-end server is finished processing the request and has sent the generated dynamic parts back to the client.

2. *Horcrux* (§3) focuses on the reducing the amount of computation that browsers must perform to load a page via parallelizing JavaScript computation. Again inspired by how mobile apps take advantage of concurrent execution on multiple threads, Horcrux offloads JavaScript executions of a page to different threads whenever the opportunity arises, effectively reducing the amount of time required for client-side computations.

There are numerous challenges shared between the two systems that achieve our goals of providing faster page loads for legacy web pages:

1. Ensuring correctness: In order to either reduce startup delays or client-side computations, we require a comprehensive understanding of page load process including how JavaScript code in the page can interact with the heap and DOM state during the client page load time given dynamic runtime information and possible non-determinism in the page. As mentioned we use different program analysis techniques to extract constrains and insights to ensure correctness. However, these techniques are expensive and cannot be afforded to be performed during client page load. As a result, we perform a conservative program analysis during an offline phase which enables us to take into account all possible scenarios during page load runtime.
2. Legacy web compatibility and limitations: To make sure our solutions are applicable to legacy web pages, we have to consider web semantics and the limitations of web APIs used as part of our solutions. Fawkes and Horcrux are designed to take into account different HTML semantics and how it interacts with DOM APIs, as well as limitations of web worker APIs.
3. Optimizer Overheads: While our custom JavaScript library enforces optimization to the rewritten page, it also incurs an overhead during the client page load time. To achieve faster page load times, we employ heuristic and other smart tactics gained via experimental insights to minimize such overheads.

1.3 Improving Startup Delays

Our first key observation is that, while existing optimizations are effective at reducing network fetch delays and client-side computation costs during page loads, *they all ignore a large and fundamental bottleneck in the page load process: the download of a page's top-level HTML file*. To fetch a page's top-level HTML, a browser often incurs multiple network round trips for connection setup (e.g., DNS lookups, TCP and TLS handshakes), server processing delays to generate and serve content, and transmission time. These tasks can sum to delays of hundreds of milliseconds, particularly on high-latency mobile links.[1] Only after receiving and parsing a page's HTML object can the browser discover subsequent objects to fetch and evaluate, make use of previously cached objects, or render any content to the blank screen. Thus, from a client's perspective, the entire page load process is blocked on downloading the page's top-level HTML object. This is true even in warm cache scenarios, since HTML objects are most often marked as uncacheable [NM18b] (§2.1).

Eliminating these early-stage inefficiencies would be fruitful for two reasons. First, overall load times would reduce since client-side computation and rendering tasks for cached content could begin earlier and be overlapped with network and server-side processing delays for new content; the CPU is essentially idle as top-level HTML files are fetched in traditional page loads. Second, and more importantly, browsers could immediately display static content, rather than showing only a blank screen as the HTML is fetched (Figure 1.3). This is critical as numerous web user studies and recent performance metrics highlight user emphasis on content becoming visible quickly and progressively [NNM18, Goo12, KRB17, NB19, VBN16].

To enable these benefits, we draw inspiration from mobile apps which, despite sharing many components with the mobile web (e.g., client devices, networks, content), are able to deliver lower startup delays (Figure 1.3). Apps reduce startup times by aggressively separating static content from dynamic content. At the start of executing a task (akin to loading a page), an app will issue a request for dynamic content *in parallel* with rendering locally cached content like structural templates, images, and banners. Once downloaded, dynamic content is used to patch the already-displayed static content.

Like apps, web pages already cache significant amounts of content across loads: 63% and 93% of the objects and bytes are cacheable on the median page. Yet startup times in warm cache page loads remain high due to download delays of top-level HTML files (§2.1). But why are HTML

[1]These delays persist even for HTML objects served from CDNs since last-mile mobile link latencies still must be incurred.

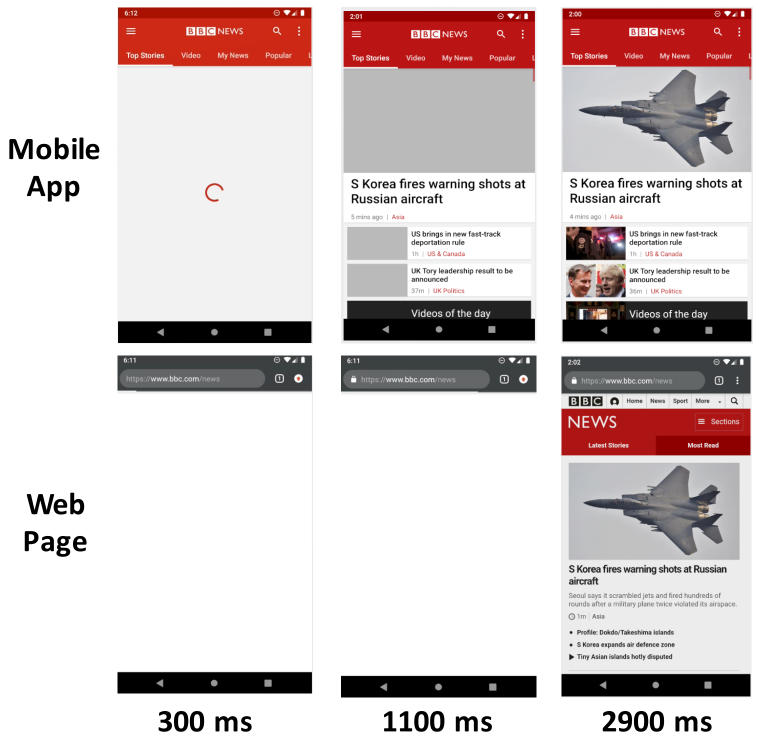

Mobile App
Web Page
300 ms
1100 ms
2900 ms
S Korea fires warning shots at Russian aircraft
US brings in new fast-track deportation rule
UK Tory leadership result to be announced
Videos of the day
NEWS
Latest Stories
Most Read

Figure 1.3: Comparing the mobile app and mobile web browser loading processes for BBC News over an LTE cellular network.

objects marked as uncacheable? The reason is that they typically bundle static content defining basic page structure with dynamic content (e.g., news headlines or search results). HTML which embeds dynamic content must be uncacheable so clients see the most recent page. Thus, at first glance, it appears that addressing this challenge with app-like templating would require a rethink of how web pages are written. However, our measurement study (§2.1) reveals that web pages are already highly amenable to such an approach given the large structural and content similarities for HTML objects across loads of a page. For instance, 75% of HTML tags on the median top 500 page have fixed attributes and positions across 1 week, and could thus be separated into static templates.

We present **Fawkes**, a new web acceleration system that modifies the early steps in the page load process to mirror that of mobile apps. Fawkes optimizes page loads in a two-step process. In the first phase, which is performed offline, web servers *automatically* produce static, cacheable HTML templates, which capture all content that remains unchanged across versions of a page's top-level HTML. The second phase occurs during a client page load; servers generate dynamic patches, which express the updates (i.e., DOM transformations) required to convert template page state into the latest version of a page. During cold cache page loads, browsers download precomputed templates while dynamic patches are being produced, and can quickly begin rendering template content and fetching referenced external objects as patches are pushed. In warm cache settings, browsers can immediately render/evaluate templates and referenced cached objects while asynchronously downloading the dynamic patch needed to generate the final page.

Realizing this approach with legacy pages and unmodified browsers requires Fawkes to solve multiple challenges:

- On the server-side, generating templates is difficult: traditional tree comparison algorithms [ZS89, Kle98, DMR07, PA11, PA15] do not consider invariants involving a page's JavaScript and DOM state, but templates execute to completion prior to patches being applied and thus must be *internally consistent*. For example, removing an attribute on an HTML tag can trigger runtime errors if downstream JavaScript code accesses that attribute; an acceptable template must keep or omit both of these components. In addition, graph algorithms are far too slow to be used for online patch generation. Instead, Fawkes uses an empirically-motivated heuristic which trades off patch generation time for patch optimality (i.e., number of operations; note that the final page is unchanged). Our insight is that tags largely remain on the same depth level of the tree as HTML files evolve over time. This enables Fawkes to use a breadth-first-search variant which generates patches 2 orders of magnitude faster (in 20 ms) with comparable content.
- On the client-side, each static template embeds a special JavaScript library which Fawkes uses to asynchronously download dynamic patches and apply the listed updates. The primary challenge is in ensuring *view invariance* for JavaScript code that is inserted via an update: that code must see the same JavaScript heap and DOM state as it would have seen during a normal page load. For example, consider an update which adds a `<script>` tag to the top of the HTML template. If that script executes a DOM method that reads DOM state, the return value may include DOM

nodes pertaining to downstream tags in the template—this state is already loaded in a Fawkes page load, but not in a default one, and may trigger execution errors. To provide view invariance, Fawkes uses novel shims around DOM methods which prune the DOM state returned by native methods based on knowledge of page structure and the position of the script calling the method.

We evaluated Fawkes using more than 500 real pages, live wireless networks (cellular and WiFi), and two smartphone models. Our experiments reveal that Fawkes significantly accelerates warm cache mobile page loads compared to default browsers: median benefits are 64% for Time-to-first-paint (TTFP), 46% for Speed Index (SI), 26% for Time-to-interactive (TTI), and 22% for page load time (PLT). Despite targeting warm cache settings, Fawkes speeds up cold cache loads by 62%, 24%, 20%, and 17% on the same metrics. Fawkes also outperforms Vroom [RNU17] and WatchTower [NSM19], two recent mobile web accelerators, by 69%-73% and 10%-24% on warm cache TTFP and SI. Importantly, Fawkes is complementary to these approaches; Fawkes with Vroom achieves Fawkes's TTFP and SI benefits, while exceeding Vroom's PLT improvements by 22%. Source code and experimental data for Fawkes are available at https://github.com/fawkes-nsdi20.

1.4 Reducing Client-side Computations

Another key contributor to slow page loads on mobile devices is the computation that browsers must perform to load a page [WBK13, NB16, NM18a, CZH20], most of which is accounted for by the execution of JavaScript code (§3.1). Numerous solutions have attempted to reduce the amount of necessary client-side computation, either by requiring developers to manually rewrite pages [Gooa] or by having clients offload page load computations to more powerful servers [NSW15, NSM19, Ope18a, Ama18, NM18a, Fac19a]. However, solutions of the former class come at the expense of manual effort and page functionality, while those in the latter class are largely unviable in practice (§3.6). Offloading to proxies [NSW15, NSM19, Ope18a] is infeasible in today's HTTPS-by-default web, while systems [NM18a] in which origin servers return post-processed pages that elide computations risk compromising correctness since servers lack visibility into client-side state (e.g., localStorage) that can affect control flow in a page load.

Instead of attempting to *reduce* the amount of computation that web clients must perform, we seek to execute the necessary computation on client devices *more efficiently*. In particular, we observe that there exists a fundamental inefficiency in the computation model that browsers employ (§3.1). To simplify page development, JavaScript execution is single-threaded [NM19, NM18a], and worse, JavaScript and rendering tasks are forced to share a single "main" thread per frame in a page [Goob]. Consequently, browsers are unable to take advantage of the growing number of CPU cores available on popular phones in both developed and emerging regions [DVB18a, DVB18b]. This inefficiency will only worsen as, due to energy constraints, increased core counts have become the main source of compute resource improvements on phones [Phi, GYC18].

A natural solution to this inefficiency is to parallelize JavaScript computations across a device's available cores. Browsers have included support for pages to spin up parallel JavaScript computation threads in the form of Web Workers [MDN20, web17] for over 8 years now. Yet, only a handful of the top 1,000 sites use Workers on their landing pages, largely due to the challenges of writing efficient, concurrent code [AXC20, HPJ16]. These challenges manifest in two ways for the web.

- Determining *which* JavaScript executions on a page frame can be safely parallelized requires a precise understanding of the page state accessed by every script, due to the language's lack of synchronization mechanisms (e.g., locks). Placing the onus of this task on web developers [MTK12, par20] is impractical, while reliance on browsers to speculatively make parallelism decisions [CFM13, MHS11] is inefficient (§**??**).
- *How* to efficiently execute scripts in parallel is also not straightforward due to the restrictions that browsers impose on Workers. In particular, they cannot access a page's JavaScript heap or DOM tree, and coordinating with the main thread, which has these privileges, adds overheads.

Our goal is to automatically parallelize JavaScript computations for *legacy pages* on *unmodified browsers*, thereby addressing the cognitive and operational overheads involved in explicitly making parallelism decisions.

Our solution, **Horcrux**, achieves these goals by employing a judicious split between clients and servers, hence preserving HTTPS' end-to-end content integrity and privacy guarantees [NSM19]. Servers perform the heavy lifting of finding parallelism opportunities and embed that information in their pages. Clients then run a JavaScript scheduler that efficiently manages parallelism using runtime information that servers lack, i.e., number of available threads, control flows taken in the current load. Three primary insights guide our design of Horcrux.

First, we ensure that any introduced parallelism preserves the final page state that developers expected when they wrote the page. For this, Horcrux forces computations that exhibit state dependencies (e.g., a read-write dependency on a global variable) to run serially in an order that matches the legacy load, while allowing other computations to run in parallel. Key to enabling this is Horcrux's offline use of concolic execution [KML21, GKS05, SMA05] on servers to explore all possible control flows on a page and identify all state that each JavaScript function *might* access, irrespective of how client-side nondeterminism could affect any particular load.

Second, Horcrux minimizes client-side coordination overheads by carefully partitioning responsibilities between the main browser thread and the Web Workers it spawns. Horcrux reserves the main thread for coordinating Worker offloads, managing global page state, and running DOM computations (which Workers cannot); all other computations are offloaded. This yields two benefits. First, scripts typically interleave computations that can and cannot be offloaded; Horcrux maximally parallelizes the former while carefully mediating the latter. Second, by keeping the main thread largely idle, Horcrux quickly adapts the parallelization schedule to the runtimes of JavaScript computations, offloading the next computation as soon as a Worker becomes available.

Third, the granularity at which Horcrux parallelizes JavaScript execution is crucial with respect to overheads and potential parallelism.

A natural solution would be to offload the invocations of JavaScript functions, which account for 94% of JavaScript source code. However, the sheer number of invocations in a typical page load makes this too costly. Instead, we observe that functions are typically invoked hierarchically (i.e., nested functions), with significant state sharing within a hierarchy, but less across them. Therefore, Horcrux offloads at the granularity of root function invocations, or the root of each hierarchy along with its nested constituents. Compared to per-function offloading, this requires 4× fewer offloads while achieving 73% of the potential speedup.

We evaluated Horcrux using over 650 diverse pages, live mobile networks (LTE and WiFi), and three phones, that collectively represent browsing scenarios in both developed and emerging regions. Our experiments across these conditions reveal that Horcrux reduces median browser computation delays by 31-44% (0.9-1.5 secs), which translates to page load time and Speed Index speedups of 18-29% and 24-37%, respectively. Further, Horcrux's median benefits are 1.3-2.1× larger than prior compute-focused web accelerators, and 1.4-2.1× more than (complementary) network optimizations.

Taken together, our results highlight that, despite being written for a serial browser computation model, existing pages are surprisingly ripe with parallelization opportunities. Horcrux shows how such opportunities can be exploited under the hood, without having developers manually rewrite their pages. Source code and datasets for Horcrux are available at https://github.com/ShaghayeghMrdn/horcrux-osdi21.

CHAPTER 2

Fawkes: Faster Mobile Page Loads via App-Inspired Static Templating

2.1 Motivation

In order to improve mobile web startup delays, first we need to understand the discrepancies between mobile apps and web pages, so we begin with a range of measurements that illustrate these discrepancies (§2.1.1), and the amenability of web pages to app-like templating (§2.1.2). Results used the LTE setup described in §2.4.1.

2.1.1 Mobile Apps vs. Mobile Web

We compare the load process of mobile apps and web pages by analyzing equivalent tasks across 10 web services; services were selected by randomly choosing web pages from the Alexa US top 100 list [Ale18], and discarding those without a corresponding mobile app. Our corpus includes news, recommendation platforms, search engines, and social media applications. For each service, we equate loading a homepage with a mobile browser to loading the home screen with the mobile app. When applicable, we also compare equivalent searches on both platforms. We load each task in the mobile app and website back to back, and for each task, we log the time until the first paint to the screen and collect screenshots for three events: the first time either platform displays content to the user (Time-to-first paint, or TTFP), an intermediate checkpoint with additional displayed content, and the time when both platforms reach their final visual state. Mobile app screenshots and paint events are captured via the Apowersoft Recorder [APO19] and Android Systrace [Goo19f] tools, respectively. Since apps have content cached during installation, for fair comparison, we consider

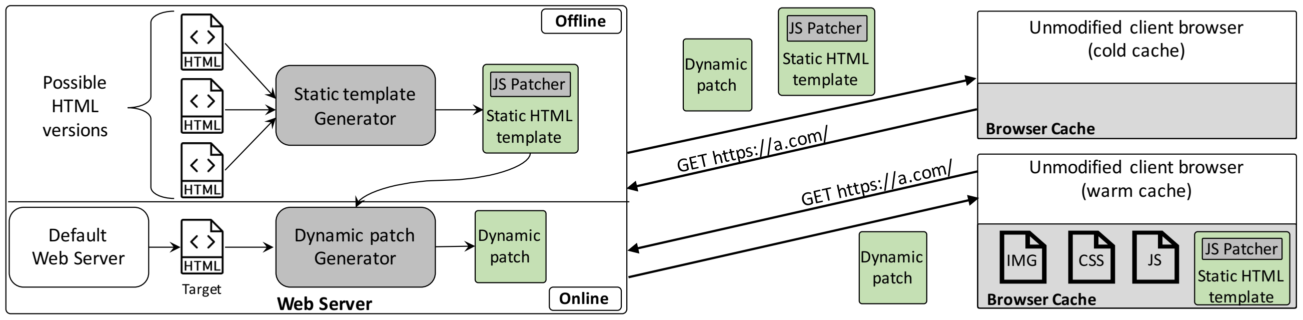


Figure 2.1: Overview of cold and warm cache page loads with Fawkes. Servers return static, cacheable HTML templates, as well as uncacheable dynamic patch files that list the updates required to convert those templates into the latest page. Updates are performed dynamically using the Fawkes JavaScript patcher library that is embedded in the templates.

warm cache page loads (back to back). Certain mobile apps operate by displaying a logo for several seconds during startup, prior to displaying the home screen. We do not consider such apps here.

Startup delays are far lower with apps than web pages. Across the corpus, our experiments reveal that TTFP values are between 3.5×–5.2× lower with mobile apps than mobile web pages. Figure 1.3 provides a representative example of loading the home page for BBC News. As shown, despite the high network latencies and potential server processing delays, the mobile app is able to quickly display static content that establishes the overall app layout and logos in under 300 ms. We verified that the reason for this is that the app quickly pulls this content from its local cache while asynchronously fetching dynamic news headlines. In contrast, the BBC web page remains blank as the browser establishes a connection to the backend and downloads the top-level HTML for the page. Only upon receiving the HTML object can the browser begin rendering any static or dynamic content to the screen–this does not begin until 1200 ms, 4× longer than the app.

Problem: uncacheable HTML limits caching benefits for the web. The above discrepancies between mobile apps and web pages are indicative of a fundamental difference in the startup tasks on the two platforms. Web HTML objects are used to set the context for the remainder of a page load, establishing render and JavaScript engine processes, creating a DOM tree (i.e., the browser's

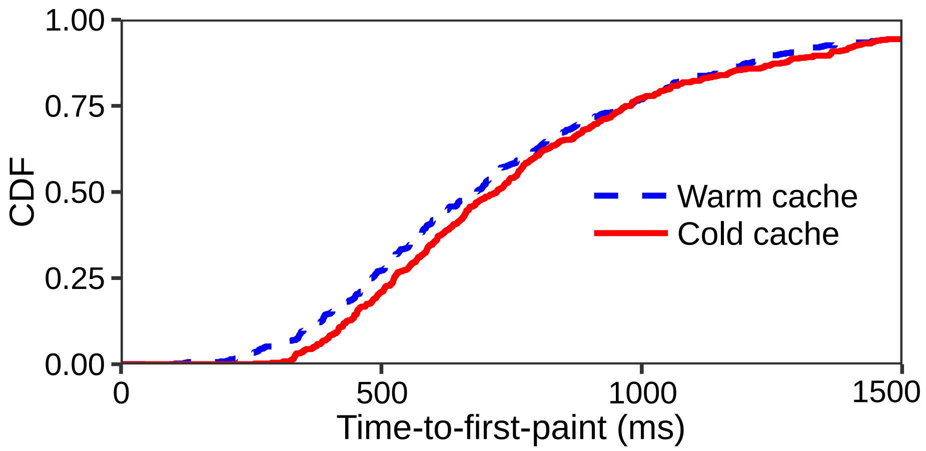


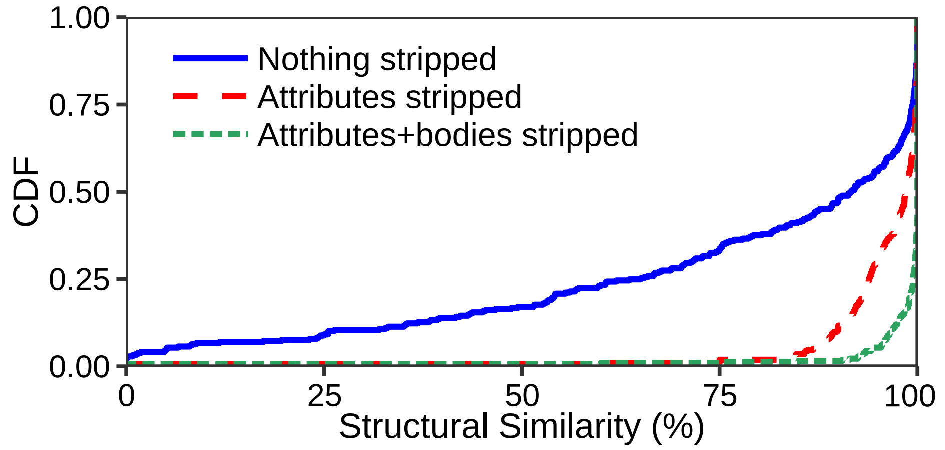
Figure 2.2: Caching has minimal impact on time-to-first-paint since browsers cannot render cached content until they download the top-level HTML (typically uncacheable).

programmatic representation of the page's HTML) and JavaScript heap, and so on. However, most HTML objects are marked as uncacheable. For example, 72% are uncacheable across back to back loads of the top 500 pages; this number jumps to 85% for loads separated by 5 minutes. As a result, browsers are unable to make use of other objects marked as cacheable (e.g., images, CSS) until they download an HTML object; for reference, 53% and 93% of objects and bytes are cacheable on the median page. Figure 2.2 illustrates this point: median TTFP values are only 5.3% lower in warm cache scenarios than during cold cache page loads despite so many objects being cached.

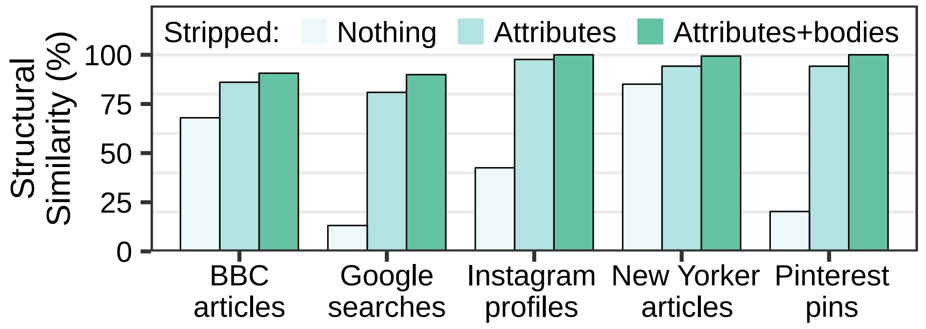
2.1.2 Templating Opportunities for the Web

Motivated by the startup discrepancies between mobile app and web page loads described above, we investigated how amenable web pages are to app optimizations. Our analysis focuses on the feasibility of extracting static templates from HTML objects that can be cached across page loads. We consider two different sets of sites: the Alexa top 500 landing pages [Ale18], and a smaller set which includes 10 pairs of different pages of the same type (e.g., different news articles or search results). More dataset details are provided in §2.4.

We loaded each page (or pair of pages in the smaller corpus) multiple times to mimic different warm cache scenarios: back to back, 12 hours apart, 24 hours apart, and 1 week apart. In each



(a) **Top 500 pages loaded 12 hours apart.**



(b) **Different pages of the same type.**

Figure 2.3: Structural similarity for HTML files over time. Similarity is defined as the percentage of shared tag sequences (including tag attributes, bodies, and types).

setting, we compared the resulting top-level HTML objects to determine structural similarities. We identify each tag as a tuple consisting of its tag type (e.g., `<div>`), HTML attributes (e.g., `class`), and body (e.g., inline script code). Since static templates can be patched during page loads, we also consider tuple versions with all tag attributes stripped, and with both tag attributes and bodies stripped. Additionally, since HTML can be modeled as a tree where ordering matters, for each tag T, we generate a sequence of tags by following parent tags up from T to the root node. We then define structural similarity as the fraction of sequences that remain identical across the HTML versions.

Opportunity: HTML structure and content is largely unchanged over time. Figure 2.3 shows that HTML objects exhibit high structural similarity. For example, for the median top 500 page, 92% of HTML tags remain identical across loads separated by 12 hours; these numbers jump to 98% and 100% when attributes are stripped alone or with bodies. These trends persist for different pages of the same type. For instance, two different Instagram profile pages exhibit structural similarity of 98% when only attributes are stripped. The trends also persist for other time windows. For example, median similarities in the 1-week setting are 75% and 95% with nothing and attributes stripped, respectively.

Key Takeaways:

- Mobile apps exhibit a desirable startup process compared to mobile web pages because apps explicitly separate static and dynamic content, and immediately render cached static content while dynamic content is fetched. Web pages, on the other hand, remain blocked (blank screen) on downloading uncacheable HTML objects, despite most other objects being cacheable.
- Mobile web pages are amenable to app-like templating of static content since HTML objects (typically uncacheable) have large structural similarities over long time periods.

2.2 DESIGN

Figure 2.1 shows the high-level design of Fawkes. Clients use unmodified browsers to load pages as normal. On the server side, websites must run Fawkes to handle incoming client HTTP(S) requests. The server-side Fawkes code performs two primary tasks. For a given page, Fawkes statically analyzes possible variants of the unmodified top-level HTML objects for the page and extracts a single **static HTML template** which maximally captures shared HTML content across versions. The generation of the static HTML template is performed offline, i.e., not during a client page load. Then, when a user loads the page, Fawkes compares the static HTML template to the target HTML, or the one that the default web server would have served without Fawkes, and generates a **dynamic patch**, which is a JSON file with an ordered list of DOM updates required to convert the template page into the target one.

The static HTML template includes an inline **JavaScript "patcher" library** that asynchronously downloads the dynamic patch file, and upon receiving it, dynamically applies the listed updates. During cold cache loads, Fawkes's server first returns the cacheable, static HTML template, and then streams the dynamic patch with HTTP/2 server push soon after; the template is sent earlier since it is precomputed, and this allows the browser to quickly start rendering template content and fetching referenced external resources. During warm cache page loads, the browser immediately begins to evaluate and render the cached template and other cached objects that it references as the patch downloads.

2.2.1 Server-Side Operation

In order to generate static HTML templates, Fawkes's server-side component leverages state-of-the-art tree matching algorithms [PA11, PA15]. The goal of these algorithms is to determine the minimum distance between two (or more) tree structures; recall that HTML files are structured as trees (§2.1). In particular, these algorithms take as input a set of trees whose nodes are assigned labels. The algorithms then compute a set of operations that, if applied, would efficiently transform the first input tree into the second. Operations typically comprise three primary types: *delete* operations remove a node and connect its children to its parent, *insert* operations add a node to a specific position in the tree, and *rename* operations do not change node positions but instead only alter a node's label. Algorithm execution works much like string edit distance techniques, using dynamic programming and assigning each operation a cost of 1.

Altering tree matching algorithms: Fawkes must alter existing tree matching algorithms in several ways to ensure that they are compatible with HTML and web semantics.

First, existing algorithms require each tree node to be labeled with an individual string. However, HTML tags can include state beyond simple tag type (e.g., `<div>` or `<link>`), each of which could be shared across versions of a page. Properties include attributes (e.g., `class`) that control the tag's behavior with respect to CSS styling rules and interactions with JavaScript code, and bodies such as inline JavaScript code or text to print.

Failing to consider attributes during HTML comparison can result in either broken pages if attributes are incorrectly treated as equivalent, or suboptimal templates if shared attributes are not maximally preserved, i.e., any attribute discrepancy would require omitting a tag. Thus, Fawkes's tree comparison algorithm labels each HTML tag with a (type, [attributes], body) three-tuple.

Second, Fawkes opts to not support *rename* operations, and instead only supports new *merge* operations. Unlike rename operations that can entirely change a node's label to deem it equivalent to a node in the other tree, merge operations can only alter a tag's attributes or body to claim such equivalence. Importantly, merge operations do not allow tag types to be modified. The reasoning behind this decision is that different tags impose different semantic restrictions on HTML structure. For instance, an `<img>` tag is self-closing, and cannot contain children tags, while `<div>` tags can have arbitrary children structures. Rewriting a `<div>` tag to an `<img>` tag would thus trigger cascading effects on existing children tags, leading to smaller templates.

Generating static HTML templates: Fawkes uses the above tree matching framework to generate static HTML templates from a set of HTML files. We describe how to get this input set in §2.2.3, and for simplicity, describe the approach assuming two input HTML files. To start, Fawkes runs its tree matching algorithm to generate a set of operations which, if applied, would convert *HTML1* to *HTML2*. Fawkes then iterates through HTML1 and selectively applies certain updates to only keep content that is shared across the inputs. Delete operations are directly applied to HTML1 as they represent content which is not shared across versions and thus should not be part of the static HTML template. Similarly, insert operations are ignored as they represent content that must be added to reach HTML2 and is thus not shared. Finally, Fawkes strips all content (tag attributes and bodies) referenced by merge operations as these highlight discrepancies between HTML versions.

While applying these operations and generating static HTML templates, Fawkes must be careful to preserve page semantics and not violate inherent dependencies between page state. In particular, Fawkes must ensure that static HTML templates are *internally consistent* and do not trigger JavaScript execution errors when parsed; this is important as templates are parsed to completion prior to any patch updates being applied. The key challenge is that altering an HTML tag's attributes or body can have downstream effects due to the shared state between JavaScript code and the DOM tree [NGM16]. For example, a downstream `<script>` tag may access an upstream `<p>` tag's attribute. Deleting that `<p>` tag's attribute can thus trigger execution errors when the browser reaches the `<script>` tag. Similarly, different `<script>` tags can share state on the JavaScript heap. As a simple example, an upstream tag may define a variable which the downstream tag accesses. Thus, cutting the upstream tag's body can trigger downstream execution errors.

Existing tree matching algorithms are unaware of such dependencies and are agnostic to the HTML execution environment. Thus, Fawkes applies a post-processing step to ensure that such dependencies are not violated in the static HTML template. Fawkes essentially iterates through the static HTML template, and upon detecting an altered tag, cuts downstream `<script>` tags. Fawkes could leverage techniques like Scout [NGM16] to more precisely characterize the dependencies between tags and JavaScript code in an effort to preserve more state in static HTML templates. However, accurately capturing such fine-grained dependencies would require web servers to also execute HTML content and load pages. Our empirical results motivate that templates derived from static tree analysis sufficiently keep the browser occupied with render and fetch tasks as dynamic patches are fetched, obviating the need for dynamic processing.

Generating dynamic patches: Fawkes servers must generate dynamic patches that list updates which, if applied, would convert the page state produced by a template into its desired final form. The inputs for patch generation are the static HTML template and the target HTML which is the file that a default web server would serve during the current page load.

The tree comparison algorithm described above can produce the desired set of transformation updates that a patch must contain. However, such algorithms are far too slow for patch generation which, unlike template generation, must be performed online, during client page loads. Thus, Fawkes uses a tree comparison heuristic which trades off patch generation time for optimality in terms of number of operations in the patch. The key insight is empirically motivated: we observe that tags most often remain on the same depth level of a tree as HTML files evolve over time. Our analysis of HTML files for 600 pages over a week revealed that, at the median and 95th percentile, 0% and 1% of tags in target HTML files were at a different depth than they were a week earlier.

This property favors a breadth-first-search approach over a depth-first-search one, and implies that we need not consider new positions for a tag outside of its current level (as traditional algorithms would). So, for each level in the target HTML file, Fawkes's algorithm works as follows:

1. Create hash maps for both the target and template HTML files that list all of the nodes for a given tag type in the order they appear on that level (from left to right).
2. Iterate through the template's level from left to right and handle each node in one of two ways. If the node's tag type exists in the same level of the target, match this node to the closest node of the same type with the same parent, and remove that node's entry from the target's hash map; if no node has a matching parent, match to the closest node of the same type. Record a *merge* operation by comparing the attributes and bodies for the matched nodes. Else, if the tag type does not exist in the same level of the target, delete this node in the template and record a *delete* operation.
3. Once we reach the end of the template's level, apply *move* operations to order all matched nodes in the template in the same way as they appear in the target; objects which remain in the same position do not require any operation. Note that *move* operations (which simply change the position of a node) are not supported by traditional tree diffing algorithms, and are only enabled by our heuristic's "look ahead" hash maps. Also, note that moves made at this level are immediately reflected in lower levels of the tree as children are reordered.
4. Finally, from left to right, insert any remaining nodes listed in the target's hash map to the appropriate position. Record *insert* operations for these additions.

The key limitation of this heuristic is with respect to nodes moving across levels in the tree. Traditional algorithms can identify such cases, while Fawkes's approach would automatically require a delete at the original level and an insert at the new level. However, as noted above, such transformations are rare. In addition, matching nodes to their closest counterparts with the same parent could be suboptimal: an inserted node in the target can create cascading suboptimal rename and move operations for nodes of that type. Despite such potential inefficiencies, correctness of the final page load is unaffected. We compare this heuristic to standard tree comparison algorithms and other heuristics in §2.4.5.

Each update in a dynamic patch must identify a node to which the update should be applied. Fawkes identifies DOM nodes by their child paths from the root of the HTML tree. For example, a child path of [1,3,2] represents an HTML tag that can be reached by traversing the first child of the root HTML tag, the third child of that tag, and then the second child of that tag. Child path ids are easy to compare and can be computed purely based on HTML tree structure.

2.2.2 Client-Side Operation

To load a page, a mobile browser first loads the static HTML template, whose initial tag is the Fawkes patcher JavaScript library. The patcher begins by issuing an asynchronous XMLHttpRequest (XHR) request for the page's latest dynamic patch. The patcher defines a callback function on the XHR request which will be executed upon receiving the dynamic patch JSON file to apply updates. The patcher then defines the DOM shims required by the callback to apply the dynamic patch updates (described below). Finally, the patcher removes its HTML `<script>` tag from the page to prevent violating downstream state dependencies and to ensure that the final page's DOM tree is unmodified. We note that the state defined by the patcher persists on the JavaScript heap despite its tag being removed from the page.

Applying updates: Upon downloading the dynamic patch, the patcher's callback function iterates over the listed updates and applies them in order until completion. To apply a given update, the patcher first obtains a reference to the affected DOM node (i.e., the one listed in the update) by walking the DOM tree based on the listed child path. The patcher then uses native DOM methods to apply the update.

For insert operations, the patcher first creates a new DOM node using document.createElement(), sets the appropriate attributes with Element.setAttribute(), and then adds it to the appropriate position in the DOM tree by calling document.insertBefore(). Adding nodes to the DOM tree can

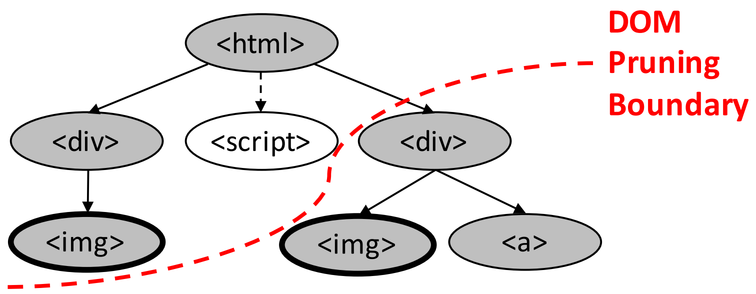


Figure 2.4: Update challenge 1: provide view invariance to JavaScript code by hiding downstream DOM state. Shaded nodes are part of the static HTML template. The `<script>` tag is inserted via Fawkes's patcher and calls a DOM method to find `<img>` tags. The native DOM method would return the two nodes outlined in bold, even though the rightmost one would not be returned in the default page load; Fawkes's DOM shims prune the rightmost node from the return value.

have cascading effects with respect to rendering and layout tasks (both of which are expensive). To mitigate these overheads, Fawkes intelligently looks ahead in the update list to determine if subsequent updates reference the node being added by the current insert update [NM18a, Fac19b]. In these cases, Fawkes constructs a DOM subtree on the JavaScript heap prior to applying the entire subtree to the actual DOM. Fawkes uses similar techniques to handle merge and delete operations.

Handling DOM discrepancies: There are two main challenges with applying updates, both of which relate to JavaScript execution and its interaction with the DOM tree. Fawkes handles both using a novel set of shims (or wrappers) around DOM methods, which are the vehicles with which JavaScript can access or modify the DOM tree.

- The first issue is with providing *view invariance* for JavaScript code inserted via an update: that code must see the same JavaScript heap and DOM state as it would have seen during a normal page load. This is challenging since updates are not applied until after a page's static HTML template is entirely parsed. For example, consider Figure 2.4 where an inserted `<script>` tag invokes a DOM method to read `<img>` tags in the page.

  The return value for this method would include an `<img>` tag that is downstream in the page's HTML; this divergence from the default page load could trigger JavaScript execution errors

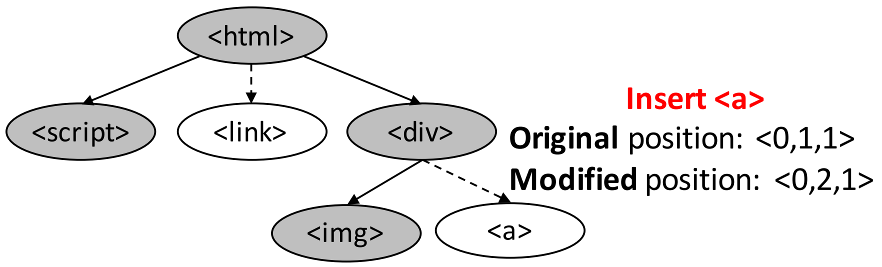


Figure 2.5: Update challenge 2: revise update positional information to reflect JavaScript execution. Shaded nodes are part of the static HTML template. Upon execution, the `<script>` tag inserts an adjacent `<link>` tag into the page. Later, when the patcher tries to apply an update to insert an `<a>`, the listed child path id has gone out of date and must be updated.

or alter page semantics. To ensure view invariance, Fawkes shims all DOM methods which return a DOM node or a list of DOM nodes; examples include document.getElementById() and document.getElementsbyTagName(). Each shim calls the native method and prunes the result prior to returning it to the client. Pruning is done by identifying the position in the DOM tree of the script invoking the DOM method, and then removing DOM nodes in the result which are below that position in the DOM tree. Fawkes's shims skip pruning for callback functions (e.g., timers) and provide *view plausibility* since the page makes no guarantee on what DOM state those asynchronous events can encounter. We note that it is not possible for inserted scripts to see less DOM state than it would in a default page load because updates are ordered with respect to HTML positions.

- The second issue is that JavaScript code can alter the DOM tree in ways that affect the child path ids for subsequent updates. The reason is that the child path ids listed in the dynamic patch are based on the static HTML, which does not consider JavaScript execution, but are applied to the dynamic DOM tree which JavaScript code can manipulate (Figure 2.5).

  To handle this, Fawkes shims DOM methods that affect DOM structure, either by adding, removing, or relocating nodes; example methods include document.appendChild() and document.insertBefore().

  Each shim calls the native method, logs its effect on DOM structure (e.g., the child path of an added node), and then returns the value. When the patcher attempts to apply an update, it first checks this log, and modifies the child paths in the remaining updates based on the listed DOM changes. We note that JavaScript can also invalidate a listed update, e,g,. by replacing a `<div>` tag with an `<img>` tag in the same position. Fawkes's shims detect these alterations, and the patcher discards such updates since JavaScript takes priority over HTML for final page structure.

2.2.3 Identifying HTML Objects to Consider

Fawkes's server-side static template generation inherently relies on having a set of representative HTML files from which to extract a template. Here we discuss several approaches for websites to generate this input set for each of their pages; Fawkes is agnostic to the specific approach that a site uses for this. We note that the input set need not be comprehensive and cover all possible HTML versions for a page since patches will include all necessary updates to reach the target page.

However, considering a comprehensive set of HTML objects can reduce the number of updates required at runtime, leading to improved performance.

Option 1: empirical analysis: One approach is for web servers to log the HTML objects that they would serve to clients over time without Fawkes. Fawkes can then periodically recompute a static HTML template based on the latest served HTML files to account for structural modifications that developers make to the page. An advantage of this approach is that the static HTML templates will inherently be based off of the popular pages that are actually served to clients. For instance, if a very rare state configuration would alter the structure of a page, most page loads would benefit from *not* considering this version in template generation.

Option 2: leveraging web frameworks: An alternative approach is to leverage the model-view-controller architecture that many popular web frameworks (e.g., Django, Ruby on Rails, and Express) use. In these systems, incoming requests are mapped to a controller function which generates a response by executing application logic code that combines application data and premade HTML blocks. Note that these blocks are small, spanning only a few tags each, and are significantly augmented with HTML tags generated by application logic–this precludes us from using these blocks as our templates. To leverage this structure, we can perform standard static program analysis [SWS16, LAG14] on application code (particularly the controller for the URL under consideration) to determine the possible HTML block combinations and dynamically produced HTML code that could result for a page.

Option 3: hybrid approach: A final approach is to perform static program analysis on the application backend source code to determine what inputs affect HTML structure, e.g., Cookie values, database state, time of day, etc. Fawkes can then simply probe the backend with different input values to generate a range of potential HTML objects that could be returned to clients; static HTML template generation would then work in the same way as Option 1.

Case studies: Our evaluation (§2.4) primarily focuses on Option 1. However, to validate the feasibility of the remaining options, we analyzed the source code of two real open source web applications: Reddit [red] and ShareLatex [sha]. Both applications follow the MVC model described above, with Reddit using the Python Pylons framework [Pyl19] and ShareLatex using NodeJS's Express [Nod19]. For both applications, we wrote custom static analyzers which profile the controller for the sites' landing pages. The output of the profiler is an intermediate template that intertwines HTML code with Python (or JavaScript) logic that, when executed, reads in application variables and outputs a fully formed HTML file. Following branch conditions and unrolling loop bodies in the intermediate template revealed that a ShareLatex project page has 16 possible HTML structures, while Reddit can have over 150. We note that, for this analysis, any tag insertion/deletion or change in tag composition (e.g., an attribute value) is treated as a new page structure. Consequently, despite the large number of potential HTML structures, both pages are highly amenable to large static templates.

2.2.4 Subtleties

Handling different template versions: Since Fawkes clients cache static HTML templates, and Fawkes servers can decide to generate new templates based on page modifications or popularity changes, it is possible that different clients have different template versions cached. One option is to have clients check for updates prior to evaluating cached templates using the If-Modified-Since HTTP header, but this would eliminate most of Fawkes's warm cache benefits as a browser would have to incur multiple round trips before rendering any content for the user. Instead, to handle these differences, static templates include a hash of the template contents as a variable in the inline patcher code. The patcher includes this information in its XHR request for a dynamic patch file; no browser modifications are required.

In order to make use of hash information in client requests, Fawkes servers must maintain a mapping of hashes to past static HTML template files which covers the max duration over which the templates are cacheable, i.e., if the templates for a URL are set to be cacheable for 1 day, the Fawkes server must store an entry for each template version served over the past day. Importantly, we expect these storage overheads to be low as our results highlight that templates remain largely unchanged on the order of weeks, and across personalized versions of pages for different users (§2.1 and §2.4).

Updating cached templates: Fawkes can use the hash-based approach described above to ensure benefits despite variations in cached templates. However, over time, Fawkes servers may wish to update cached templates to reflect significant changes in page structure that may deem past versions poor in terms of performance. For this, Fawkes servers simply send updated templates along with dynamic patches served with HTTP/2 server push. Because the pushed templates will remain cacheable for longer durations than the currently cached versions, default browsers will automatically replace the cached template for subsequent loads.

Static templates across URLs: In scenarios where static templates are generated for individual URLs by considering their possible HTML variants, templates can be cached directly under the page's URL. However, as we discuss in §2.1 and §2.4, Fawkes's template caching approach can provide significant benefits across different URLs of the same page type, e.g., different search result pages or news articles. To support such scenarios efficiently, browsers must slightly alter their caching approach to allow objects to be cacheable across multiple URLs. Websites can specify a regular expression that precisely covers the URLs for which the template applies, and browsers would use the same cached template for any load which matches that regular expression.

2.3 IMPLEMENTATION

On the server-side, Fawkes's template and dynamic patch generation code are written in 1912 and 462 lines of Python and C++ code, respectively. Both components are implemented as standalone modules for seamless integration with existing web servers and content management platforms [Dru19, Wor19]. Module inputs are a set of HTML files, and outputs are full formed HTML and JSON files that can be directly shipped to client browsers. For template generation, Fawkes extended the APTED tree comparison tool [PA15, PA18]. HTML parsing and modification are done using Beautiful Soup [Ric19].

On the client-side, Fawkes's JavaScript patcher library consumes 3 KB when compressed with Brotli [Goo19b]. The patcher is written entirely using native DOM and JavaScript methods, and is thus compatible with unmodified web browsers. We note that the DOM shims are shared across pages, and thus could be cached as a separate object from each page's static template to reduce bandwidth costs.

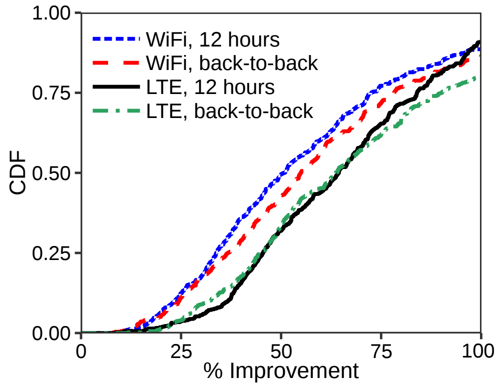
2.4 EVALUATION

2.4.1 Methodology

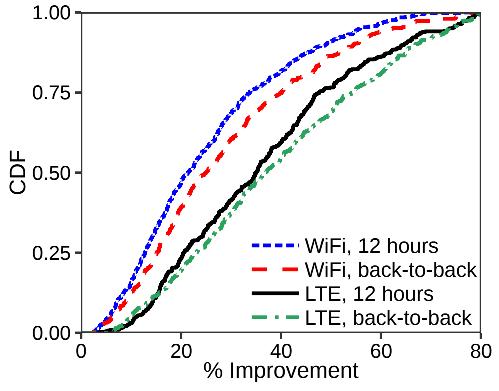
We evaluated Fawkes using two phones, a Nexus 6 (Android Nougat; 2.7 GHz quad core processor; 3 GB RAM) and a Galaxy Note 8 (Android Oreo, 2.4 Ghz octa core; 6 GB RAM). Fawkes performed similarly across the two devices, so we only report results for the Nexus 6. Unless otherwise noted, page loads were run with Google Chrome (v75).

Our experiments consider two different sets of pages:

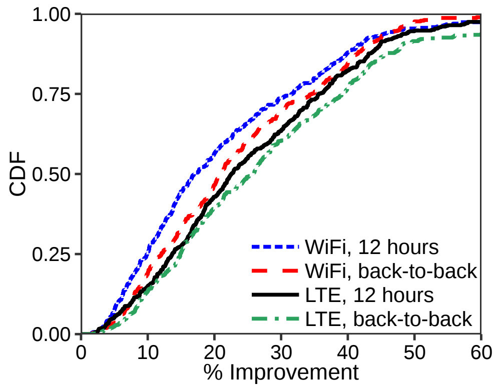
- Alexa top 500 US landing pages [Ale18]. We augment this list with 100 interior pages that were randomly chosen from a pool of 1000 pages generated by a monkey crawler [sel19] that clicked



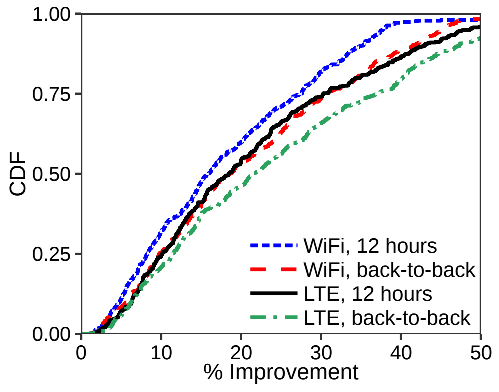
(a) Time-to-first-paint (TTFP)



(b) Speed Index



(c) Time-to-interactive (TTI)



(d) Page load time (PLT)

Figure 2.6: Distributions of warm cache (back to back and 12 hour) per-page improvements with Fawkes vs. a default browser (i.e., using each page's default HTML) for 600 pages.

links on each site's landing page.

- a smaller set of 20 pages that includes pairs of different pages of the same type. Starting from the Alexa top 50 list, we identified page types that have many versions, and manually generated pairs for each one, e.g., two Google search results and two public Twitter profile pages.

In order to create a reproducible test environment and because Fawkes involves page modifications, our evaluation uses the Mahimahi web record-and-replay tool [NSW15]. We recorded versions of each page in our corpus at multiple times to mimic different warm cache scenarios: back to back page loads, and page loads separated by 12 and 24 hours. Mobile-optimized (including AMP [Gooa]) pages were used when available. To replay pages, we hosted the Mahimahi replay environment on a desktop machine. Mobile phones were connected to the desktop machine via both USB tethering and live wireless networks (Verizon LTE and WiFi) with strong signal strength. The desktop initiated page loads on the mobile device using Lighthouse [Goo19c], and all control traffic for this was sent over the USB connection. All web and DNS traffic were sent over the live wireless networks into Mahimahi's replay environment. We modified Mahimahi to faithfully replay the use of HTTP/2 (including server push decisions) and server processing delays observed during recording.

To compute per-object server processing delays, we first recorded the RTT to each origin in a page as the median time between the TCP SYN and SYN/ACK packets across all connections with that origin. We then defined the server processing delay for an object as its TTFB minus 1 RTT (for the transmission of the HTTP request and initial response bytes); when applicable, we also subtracted out connection setup delays (1 or 2 RTTs depending on whether the resource was downloaded via HTTP or HTTPS). Lastly, we modified Mahimahi's `replayserver` to wait for the corresponding server processing delay before shipping back any object.

In accordance with §2.2, in all experiments, Fawkes's templates are generated a priori (i.e., offline). We apply server processing delays for a given template as the median delay observed for objects marked as cacheable in the default load of the page; these objects likely represent premade content. Note that this strategy ensures that templates experience the observed server-side delays that do not relate to content generation, e.g., delays due to high server load. Unless otherwise noted, templates are generated using the first and current versions of a page (i.e., a version at time 0, and the version in the back to back load, 12 hours later, etc.); we present results for other template generation strategies in §2.4.5. Dynamic patches are generated online by Mahimahi's web servers. Server processing times for patches include both the observed server processing time for the page's original HTML file, as well as the time taken to generate a patch.

We evaluate Fawkes on multiple web performance metrics. Page load time was measured as the time between the `navigationStart` and `onload` JavaScript events. We also consider three state-of-the-art metrics which better relate to user-perceived performance: 1) Speed Index (SI), which represents the time needed to fully render the pixels in the intial view of the page, 2) Time-to-first-contentful-paint (TTFP), which measures the time until the first DOM content is rendered to the screen, and 3) Time-to-interactive (TTI), which measures how quickly a page becomes interactive with rendered content, an idle network, and the ability to immediately support user inputs. All three metrics were measured using pwmetrics [Iri19]. In all experiments, we load each page three times with each system under test, rotating amongst them in round robin fashion; we report numbers per system based on the load with the median page load time.

Correctness and limitations: To ensure a faithful evaluation, we analyzed the pages in our 600-page corpus to identify and exclude those that experience replay errors due to either Mahimahi's (22 pages) or Fawkes's limitations (17 pages).

To ensure a faithful evaluation of Horcrux, we analyzed the pages in our corpora to identify and exclude pages that experience replay errors due to either Mahimahi or Horcrux. We excluded 22 pages due to Mahimahi replay errors, most of which were the result of SSL errors for pages that leverage the Server Name Indication (SNI) feature in SSL/TLS certificates (which Mahimahi does not support), and missing resources that Mahimahi's URL matching heuristic was unable to remedy.

On the remaining pages, correctness with Horcrux was evaluated by forcing determinism upon JavaScript execution (e.g., using fixed return values for calls to `Math.Random()`) [MEH10, NM19], and comparing loads with and without Horcrux in three ways: 1) a pixel-by-pixel analysis of the final page (using pwmetrics' screenshots and visual analysis tools [Iri19]), 2) the number of registered JavaScript event handlers (logged using shims on the `addEventListener` mechanism and by iterating over the DOM tree after the `onload` event fired [NNM18]), and 3) the browser console errors printed during the page load. We excluded the 17 pages that differed on any of these three properties from our evaluation. Further investigation revealed two key reasons for such discrepancies, which are limitations with Horcrux:

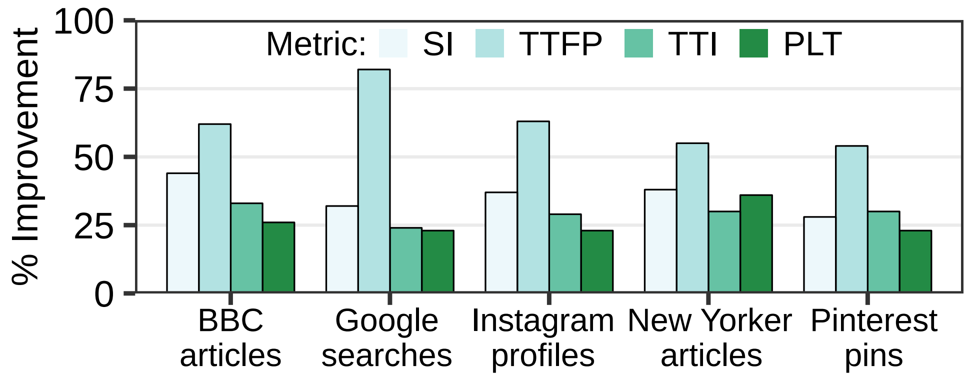


Figure 2.7: Warm cache speedups for sites in our smaller corpus.

- Although Horcrux cuts downstream JavaScript in templates after the first tag removal or alteration, it does not remove CSS. The reason is that CSS rules can significantly affect the styling of template content, bringing it closer to the final page version. However, CSS and JavaScript code can share state in the form of DOM node attributes. As a result, downstream CSS files in a page's template can modify DOM attribute state that patched (upstream) JavaScript code can subsequently access—this can alter page execution and lead to errors.
- JavaScript code can dynamically rewrite downstream HTML using the `document.write()` interface. However, Horcrux's patches are based on a page's static HTML, which does not reflect JavaScript execution. Thus, because our current implementation of Horcrux does not use shims for `document.write()`, it is possible for JavaScript code in the template to (correctly) rewrite downstream HTML content, that is (incorrectly) resurrected by the Horcrux patcher.

2.4.2 Improving User-Perceived Web Performance

Warm Cache: Figure 2.6 illustrates Fawkes's ability to improve performance for our 600-page corpus, compared to a default browser, across a variety of web performance metrics and warm cache settings; we omit results for the 24 hour setting due to space constraints, but note that the trends were the same. Benefits with Fawkes are most pronounced on the metrics that evaluate visual loading progress, SI and TTFP. For example, in the 12 hour warm cache setting, median SI improvements are 38% and 22% on the LTE and WiFi networks, respectively. Improvements jump

Property	back to back	12 hours	24 hours
Static template size (KB)	102 (601)	77 (343)	73 (358)
Dynamic patch size (KB)	6 (249)	44 (491)	52 (460)

Table 2.1: Analysis of Fawkes's templates and dynamic patches across warm cache scenarios with different time windows. Results list median (95th percentile) values for each property.

to 67% and 51% for TTFP; these benefits directly characterize Fawkes's immediate rendering of static HTML templates, compared to the lengthy blank screen in a default page load. Table 2.3 lists the raw time savings pertaining to these improvements.

Despite targeting quick visual feedback, Fawkes's results are also significant for more general web performance metrics like TTI and PLT: median improvements in the 24 hour scenario are 20% and 17% in the LTE setting. The reason is that in warm cache settings, Fawkes enables browsers to utilize network and CPU resources that go idle in standard page loads as HTML objects are being loaded. Browsers can immediately perform required rendering and processing tasks (which are non-negligible on mobile devices [NM18a, NB16, WBK13, RNU17]) of both template content and referenced cached objects; at the same time, browsers can issue requests for any referenced uncacheable objects to make use of the idle network.

Across all metrics, Fawkes's benefits are higher in LTE settings than on WiFi networks. The reason is that network latencies are higher on LTE networks: in our setup, last mile (access link) RTT values were consistently around 82 ms for LTE and 17 ms for WiFi. Higher round trip times increase the time that default page loads are blocked on fetching top-level HTML objects while Fawkes parses its templates.

As expected, benefits were consistently higher in the back-to-back warm cache setting than when page loads were separated by 12 or 24 hours. This is because HTML objects undergo fewer changes across back-to-back loads, leading to larger templates and fewer updates (Table 2.1). Larger templates result in immediate feedback that more closely resembles the final page, as well as increased opportunities to utilize idle CPU and network resources. Note that patch sizes include page content

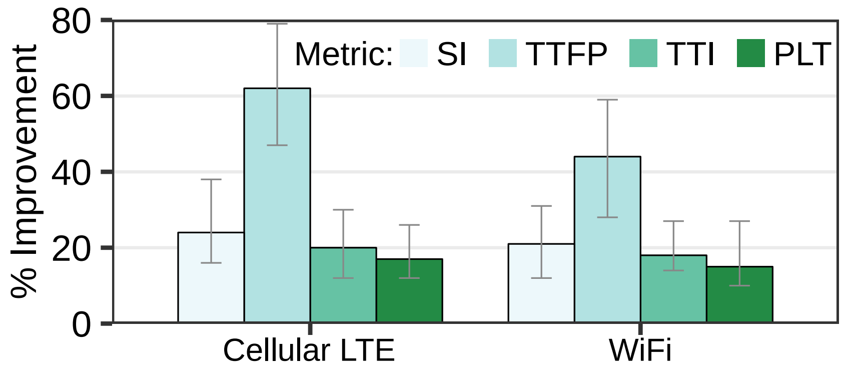
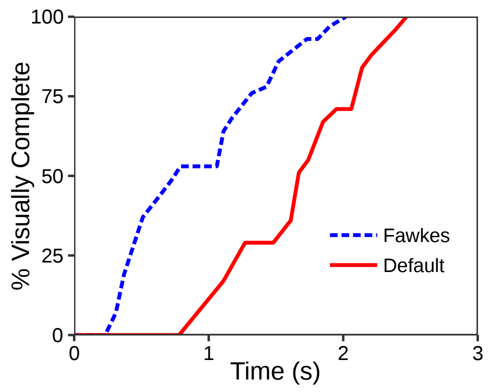


Figure 2.8: Cold cache speedups with Fawkes versus a default browser on our 600-page corpus. Bars represent medians, and error bars span from the 25th to 75th percentile.

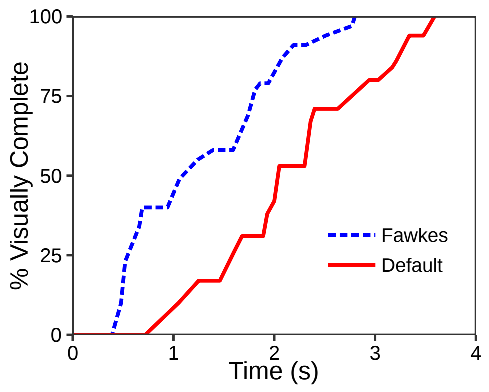
(e.g., inline scripts) to be inserted.

Figure 2.7 illustrates similar warm cache benefits (over the LTE network) for representative sites in our smaller corpus. Templates are made by consideringdifferent versions of the same page type. We note that TTFP benefits were highest for Google search pages because those pages incur the highest server processing times (for result generation).

Cold Cache: Although Fawkes's template-based approach primarily targets warm cache settings, benefits are significant in cold cache scenarios (Figure 2.8). For example, median SI and TTFP improvements were 24% and 62% for the LTE network. These results consider templates generated using HTML files generated 24 hours apart. The reason for these benefits mirror those in warm cache settings, but with smaller savings. Since static HTML templates are served faster than dynamic patches, browsers still have a window to perform template rendering and compute tasks with Fawkes while the default page load is blocked. Browsers largely use this time to quickly fetch referenced uncached objects, making better use of the idle network. Like in warm cache settings, SI and TTFP benefits drop to 21% and 44% on the WiFi network due to the decreased network round trip times.



(a) Warm cache.



(b) Cold cache.

Figure 2.9: Visual progress with and without Fawkes for the Yahoo homepage. Warm cache loads were 12 hours apart.

2.4.3 Understanding Fawkes's Benefits

Case study: To better understand Fawkes's performance, we analyzed the visual progress of page loads both with and without Fawkes. Visual progress tracks the fraction of the browser viewport (i.e., the part of the page that is visible without scrolling) that has been rendered to its final form.

Figure 2.9 shows warm and cold cache results for a representative site in our corpus, the Yahoo homepage. In the warm cache scenario, Fawkes is able to make an immediate jump (53% in 790 ms) in visual progress by parsing and rendering a large part of the static HTML template, as well as referenced static objects which are also in the browser cache. In contrast, the default browser is blocked on the multiple network round trips and server processing delays required to fetch the page's top-level HTML object; visual progress does not increase until 1110 ms into the load. The initial render (29% in 1270 ms) is also much smaller than with Fawkes because the default HTML parse gets quickly blocked on fetching an uncacheable JavaScript file—rendering is blocked until this file is fetched and evaluated. Fawkes also has to fetch this file, but this occurs via an applied update, at which point Fawkes has already reached 53% visual completeness. We note that evaluation of this script (and thus blocked rendering) is 27 ms worse with Fawkes due to overheads from DOM shims. However, these overheads are overshadowed by the large early lead in visual

Algorithm	# of operations	Execution time (ms)
Fawkes's heuristic	2 (667)	20 (59)
Insert-first heuristic	2 (3013)	30 (70)
Delete-first heuristic	2 (3065)	30 (70)
State-of-the-art tree diffing algorithm (§2.2)	17 (136)	2717 (19702)

Table 2.2: Fawkes's dynamic patch generation heuristic yields a desirable tradeoff between patch generation time and patch optimality, compared to other heuristics and a state-of-the-art tree diffing algorithm (which Fawkes uses for template generation). Results list median (95th percentile) values.

progress that Fawkes achieves.

In the cold cache setting, both Fawkes and the default browser incur network delays to fetch an HTML object for the page. However, this delay is lower for Fawkes as the static template is pre-generated. From this point, the page loads are largely similar to the warm cache setting: Fawkes makes a larger immediate jump in visual progress (40% in 690 ms vs. 17% in 1250 ms) as the default browser gets quickly blocked on fetching an external script, while Fawkes does so only after the template is parsed. From there, both page loads progressively render content, but Fawkes never relinquishes its lead. We note that, though it is not visible in the graphs, Fawkes issues requests for non-blocking external objects (e.g., images) that are listed in the template earlier.

Template content: Fawkes's early template parsing enables browsers to 1) process referenced cached objects sooner in warm cache loads, and 2) quickly issue requests for referenced external objects in cold cache loads. To understand how often these optimizations are applied, we analyzed the static URLs listed in Fawkes's templates; we considered templates generated using loads 12 hours apart. We found that, on the median page, templates referenced 46% of the page's objects, of which 72% were cacheable.

Patch generation: As described in §**??**, Fawkes opts to run a tree comparison heuristic rather than a state-of-the-art tree diffing algorithm. Fawkes's heuristic is designed to trade off patch generation time for optimality (in terms of number of operations). To evaluate Fawkes's heuristic on

System	SI	TTFP	TTI	PLT
Default	2.9 (3.9)	0.5 (0.5)	3.6 (4.4)	4.0 (5.2)
Fawkes	1.8 (2.9)	0.2 (0.3)	2.8 (3.5)	3.3 (4.2)
Vroom	2.4 (3.4)	0.6 (0.5)	2.9 (3.3)	3.2 (3.8)
WatchTower	2.0 (2.5)	0.6 (0.6)	2.6 (3.0)	2.8 (3.6)
Fawkes + Vroom	1.8 (2.9)	0.2 (0.3)	2.6 (3.14)	2.5 (3.4)

Table 2.3: Median warm (cold) cache raw times for our 600-page corpus on an LTE cellular network. All results are in seconds, and warm cache loads are spread by 12 hours.

this tradeoff, we compare it to the tree diffing algorithm Fawkes uses for template generation, and two additional heuristics: 'insert-first' and 'delete-first' breadth-first-search approaches where discrepancies discovered when comparing a level in the template and target are handled by first inserting the missing node or deleting the mismatched node, respectively, and then accounting for any remaining deltas (Table 2.2). As shown, Fawkes's heuristic runs 2 orders of magnitude faster than existing tree diffing algorithms. Median operations are lower with Fawkes's heuristic due to its *move* operation. 95th percentile operations are 5$\times$ worse with Fawkes's heuristic due to the inefficiences described in §3.3.1, but we note that this large gap is only present in 8% of pages.

Importantly, across all warm cache page loads, Fawkes completes heuristic execution and shipping patches to clients *before* client-side template processing completes; shipping patches before this does not improve performance a template parsing must conclude prior to patch application.

2.4.4 Comparison with Vroom and WatchTower

We compared Fawkes with two recent mobile web optimization systems, Vroom [RNU17] and WatchTower [NSM19]. With Vroom, web servers user HTTP/2 server push to proactively send static resources that they own to clients ahead of future requests. In addition, Vroom servers send HTTP preload headers [W3C18] to let clients quickly download resources that they will soon need from other domains. In contrast, WatchTower accelerates page loads by selectively using proxy servers based on page structural properties and network conditions. When enabled, a proxy loads a page locally using a headless browser and fast network links, and streams individual resources back to the client for processing. Our evaluation considers WatchTower's HTTPS-sharding mode, where each HTTPS origin runs their own proxy to preserve HTTPS security. Proxies were run on EC2 in California where the WatchTower paper reported the highest speedups.

Table 2.3 compares Fawkes with Vroom and WatchTower for both cold and warm cache loads of our 600 page corpus over an LTE network; trends were similar on the WiFi network. As shown, Fawkes is able to significantly improve performance on the interactivity-focused metrics compared to these systems. For example, median warm cache Speed Index values were 24% and 10% lower with Fawkes than with Vroom and WatchTower, respectively. Fawkes's TTFP benefits over these systems were 69%-73% since acceleration techniques with WatchTower and Vroom only take affect *after* incurring multiple network round trips and server processing delays to download HTML objects.

Our results also show that Vroom and WatchTower are more effective than Fawkes at reducing PLT; median benefits are 3.3% and 16.6%, respectively. The reason is that both Vroom and WatchTower can mask network round trips required to fetch external objects throughout the page load, including those triggered by non-HTML objects. Fawkes, on the other hand, focuses on early parts of a page load–indeed, targeting startup bottlenecks is what differentiates Fawkes from prior acceleration techniques. Importantly, we note that Fawkes's early-stage optimizations are largely complementary to prior techniques.To validate this, we reran the experiment above using a combination of Fawkes and Vroom. Vroom's server hints on the top-level HTML were sent along with Fawkes's dynamic patches. As shown in Table 2.3, this combination outperforms any tested system in isolation.

2.4.5 Additional Results

Stale HTML templates: Our warm cache evaluation considered static templates that were generated using the HTML object at time 0 and the one for the current time (e.g., 12 hours). Although this is possible using the techniques presented in §2.2.3 to generate representative HTML files for a page, it is not the sole practical deployment scenario. An alternative approach would be to generate static HTML templates with only back-to-back loads at a time 0, and use this for future warm cache loads. To evaluate the impact of such stale templates, we loaded the pages in our corpus using both stale (i.e., generated at time 0) and up-to-date (generated using HTML files at time 0 and the current time) templates. We considered staleness of 12 and 24 hours, and observed minimal performance degradations. For example, on the LTE network, median SI values dropped by only 4.2% and 6.8% for the 12 and 24 hour scenarios; TTFP values were unchanged.

Personalized pages: We selected 20 sites from our 600 page corpus that supported user accounts. For each site, we made two user accounts, selecting different preferences when possible, e.g., order results based on time or popularity. We then generated static templates from the HTML objects that each user account fetched. Finally, we loaded one of the user's pages 12 hours later with a warm cache, and compared performance to that of a default browser. Fawkes was able to reduce SI by 27% and 18% on the LTE and WiFi networks, respectively. It is important to note that these trends may not hold for all personalization strategies. For example, pages like Facebook can display structurally diverse content over time and across users. However, our results illustrate that many pages do remain structurally similar across users.

Energy savings and other browsers: Fawkes reduces (per-page) energy usage by 7-18%, and its speedups persist across other browsers (e.g., Firefox).

Energy consumption: To examine the impact that Horcrux has on energy usage, we connected a Nexus 6 phone to a Monsoon power monitor [Inc18] and loaded our 600 page corpus. During cold cache loads, Horcrux's speedups reduce median per-page energy usage by 11% and 7% compared to a default browser on the LTE and WiFi networks, respectively. Benefits jump to 18% and 11% in warm cache settings (12 hours apart). In both cases, benefits are higher on LTE due to the higher network latencies and the fact that LTE radios consume more energy than WiFi hardware when active [SPG14].

Additional browsers: Since Horcrux does not require any browser modifications, we also evaluated Horcrux with Firefox (v68) using our 600 page corpus and the same experimental setup from §**??**.

Benefits in the 12 hour warm cache setting were quite comparable, despite Firefox using a different rendering engine than Chrome. Horcrux reduced median SI by 21% and 34% on the WiFi and LTE networks.

2.5 Related Work

Server push systems: Numerous systems, including Vroom (§2.4.4), aim to accelerate mobile page loads by leveraging HTTP/2's server push feature [BPT18], where servers proactively push resources to clients in anticipation of future requests [RNU17, EGJ15, WBK14]. Fawkes is largely complementary to server push systems as these approaches reduce fetch times for resources loaded after the top-level HTML. In contrast, Fawkes speeds up page startup times.

Proxy and backend accelerators: Compression proxies [ABC15, SMK15, Vol12, Ope18b] compress objects in-flight between clients and servers, while remote dependency resolution proxies [SPG14, NSM19, NSW15, SJN17] perform object fetches on behalf of clients. Fawkes is orthogonal to these approaches, and can mask the network indirection and computation overheads associated with proxying. In addition, Fawkes preserves the end-to-end nature of the web, avoiding the security challenges of proxying.

More recently, Prophecy [NM18a], Shandian [WKW16], and Opera Mini [Ope18a] return post-processed versions of objects to reduce client-side computation and bandwidth costs. All three systems must incur the same network round trips and (more) server processing delays that default page loads to download top-level HTML objects–only then do their acceleration techniques help. These delays are exactly what Fawkes aims to alleviate. We also note that Fawkes's patcher and shims tackle a fundamentally different challenge than those in Prophecy and Shandian: Fawkes must execute JavaScript code in an environment with fast-forwarded DOM state.

Dependency-aware scheduling: Certain systems have improved the scheduling of network requests based on inherent dependencies in page content. Klotski [BWW15] analyzes pages offline to identify high-priority objects in terms of user utility, and uses knowledge of network bandwidth to stream them to clients before they are needed. Polaris [NGM16] uses a client-side request scheduler that reorders requests to minimize the number of effective round trips in a page load without violating state dependencies. However, both systems are unable to process or render content prior to an HTML download. Thus, these systems can work side by side with Fawkes.

JavaScript UI frameworks: Libraries like Vue.js [You19], AngularJS [Goo19a], and React [Fac19b] efficiently update client-side page state during page loads. A key feature across these frameworks is the use of a virtual DOM, where JavaScript-based DOM updates are first performed on a lightweight DOM representation, and aggregate results (rather than intermediate layout and render events) are applied to the actual DOM. Using such efficient update strategies, these frameworks support client-side page rendering, whereby a page's top-level HTML embeds only a single JavaScript library that is responsible for downloading and rendering downstream page content. While these frameworks focus on efficiently updating content during a page load and require developers to rewrite pages, Fawkes operates on unmodified pages and aims to quickly display content shared across page loads. Further, unlike the client-side page rendering approach, Fawkes's static templates embed both the JavaScript patcher library *and* all of a page's static HTML content. This, in turn, ensures that Fawkes can render static content while fetching downstream (dynamic) content.

Accelerating HTML loading: Google's SDCH [BLM] allows web servers to specify cacheable components of HTML files; on subsequent loads, servers need only send new components or deltas to cached ones, thereby saving bandwidth. Unlike Fawkes, SDCH does not allow browsers to render cached HTML components *until the entire HTML is constructed*. Thus, SDCH does not face the view invariance challenges that Fawkes's patcher does, and SDCH is unable to reduce web startup times by rendering cached HTML content quickly for users. Other industry efforts have focused on dividing pages into modular components called "pagelets", which can be generated and processed independently and in parallel [Bro17, Jia10]. Pagelets share Fawkes's goal of improving resource utilization to more quickly display content to users. However, unlike Fawkes, individual pagelets do not include a mechanism for automatically separating static and dynamic HTML, and instead use a single response that is shipped only after the pagelet's dynamic content is generated.

Mobile-optimized pages: Certain sites cater to mobile settings by serving pages that involve less client-side computation, fewer bytes, and fewer network fetches. For example, Google AMP [Gooa, JBW19] is a recent mobile web standard that requires developers to rewrite pages using restricted forms of HTML, JavaScript, and CSS. Unlike AMP, Fawkes accelerates legacy pages without needing developer effort. Further, Fawkes provides complementary benefits and can lower AMP startup delays: Fawkes's TTFP and SI reductions were 58% and 27% for the 23 AMP pages in our corpus.

Prefetching: Prefetching systems predict user browsing behavior and optimistically download objects prior to user page loads [PM96, LRS12, WLZ12]. Unfortunately, such systems have witnessed minimal adoption due to challenges in predicting what pages a user will load and when; inaccurate page and timing predictions can waste device resources or result in stale page content [RAP13]. By rendering static templates as soon as a user navigates to a page, Fawkes is able to achieve comparable TTFP reductions without the issues of prefetching.

Progressive Web Apps (PWAs): Google recently proposed PWAs [Goo19d], applications that are written using standard web languages (e.g., HTML, JavaScript), can be loaded by a standard web browser, but are installed as an application on a user device. PWAs use service workers [Goo19e] which employ aggressive caching and custom update logic to run offline and support push notifications from servers. Fawkes shares the idea of improving use of web caching and app-like update logic. However, in contrast to PWAs which require developer effort for creation (and potentially maintenance), Fawkes transparently applies app-like templating to legacy pages.

2.6 CONCLUSION

Inspired by the mobile app startup process, we presented Fawkes, a mobile web acceleration system that generates cacheable, static HTML templates that can be immediately rendered to quickly display content to users as page updates are fetched. Fawkes represents a shift in the web acceleration space, by focusing on leveraging underutilized resources at the *beginning* of page loads. We find that Fawkes brings median warm cache reductions of 46%, 64%, 26%, and 22% for SI, TTFP, TTI, and PLT, and outperforms state-of-the-art server push and proxy-based acceleration systems by 10%-24% and 69%-73% on SI and TTFP.

CHAPTER 3

Horcrux: Automatic JavaScript Parallelism for Resource-Efficient Web Computation

3.1 Motivation and Background

Numerous studies have reported that client-side (browser) computation is a significant contributor to poor mobile web performance [WBK13, NB16, RNU17, NM18a]. We reproduce this finding below (§3.1.1), present measurements to elucidate why such delays are so pronounced (§3.1.1), and trace the origins for this poor performance to the computation model used by browsers today (§3.1.2). Our experimental setup (§3.5.1) covers web workloads in both developed and emerging markets by considering popular pages in the US and Pakistan and loading those pages on common phones in each region. Pages in the emerging region generally involve less JavaScript code, but are loaded on phones with fewer compute resources.

3.1.1 Web Computation Delays

Computation delays are significant. To quantify the computation delays in page loads, we replayed each page locally, without any network emulation, i.e., all object fetches took ≈ 0 ms. As shown in Figure 3.1, even without network delays, popular pages in developed and emerging markets have median load times of 2.7 and 3.8 seconds, respectively. Worse, 48% and 63% of pages require more than the 3 second load times that users tolerate [3se18]. These intolerable delays persist with metrics evaluating page visibility (i.e., Speed Index), with 39% and 52% of pages in the developed and emerging regions taking more than 3 seconds to fully render.

JavaScript execution is the main culprit. To break down these high computation delays, we an-

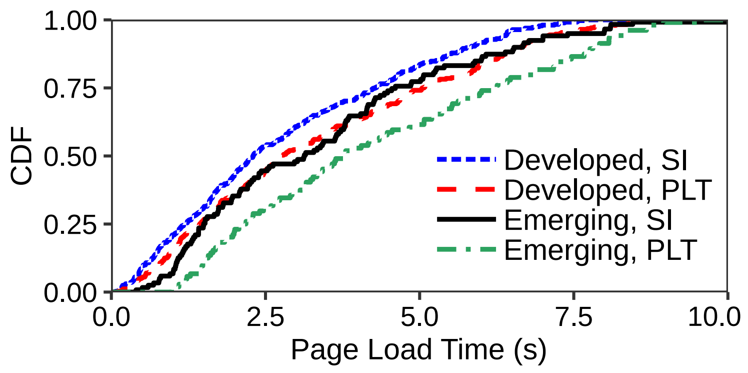


Figure 3.1: Load times often exceed user tolerance levels (3 seconds) even when all network delays are removed.

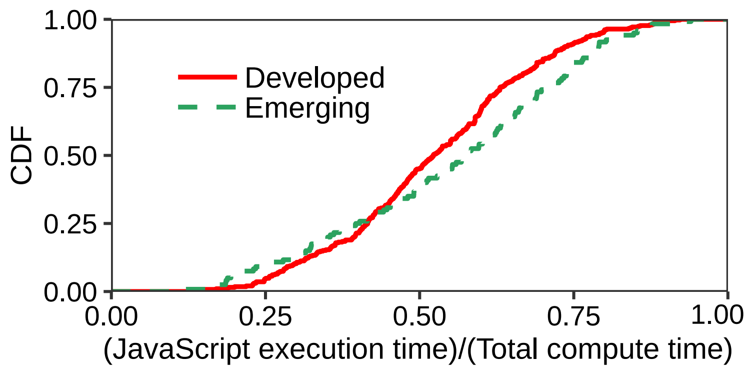


Figure 3.2: JavaScript's role in browser computation delays.

alyzed data from the browser's in-built profiler which lists the time spent performing various browser tasks including JavaScript execution, HTML parsing, rendering, and so on. Figure 3.2 illustrates our finding that JavaScript computation is the primary contributor, accounting for 52% and 58% of overall computation time for the median page in the two settings.

Browsers make poor use of CPU cores. Computation resources on mobile phones have globally increased in recent years, with improvements in both CPU clock speeds and total CPU cores. However, due to the energy constraints on mobile devices, increased core counts have been (and likely will continue to be) the primary source of improvements [Phi, GYC18]. For example, since their inception in 2016, Google's Pixel smartphones (our developed region phone) have improved

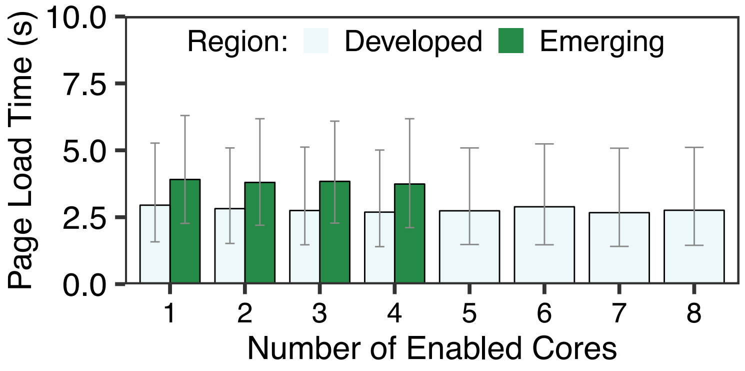


Figure 3.3: Additional CPU cores have minimal impact on load times. The developed and emerging region phones have 8 and 4 cores. Bars list medians, with error bars for 25-75th percentiles.

clock speeds from 1.88 GHz to 2.15 GHz, while doubling the number of CPU cores (from 4 to 8). Similarly, the popular Redmi A series in India and Pakistan [red20] (our emerging market phone) observed the same doubling in CPU cores (2 to 4) during that time period, while seeing only a modest clock speed improvement from 1.4 GHz to 1.75 GHz.

Unfortunately, although browsers can automatically benefit from clock speed improvements, we find that they fail to leverage available cores. To illustrate this, we iteratively disabled CPU cores on the phones in each setting and observed the impact on page load times. As shown in Figure 3.3, additional CPU cores yield minimal load time improvements, e.g., going from 1 to 8 cores on the Pixel 3 resulted in only a 8% speedup for the median page.

3.1.2 Browser Computation Model

To determine the origin of these computation inefficiencies, we must consider the computation model that browsers use today. Our discussion will be based on the Chromium framework [Goob], which powers the Chrome, Brave, Opera, and Edge browsers that account for 70% of the global market share [chr20, bro20]. Figure 3.4 depicts Chromium's multi-process architecture. We focus on the *renderer process* which houses the Rendering and JavaScript engines, and thus embeds the core functionality for parsing and rendering pages.

The Rendering engine parses HTML code, issues fetches for referenced files (e.g., CSS, JavaScript,

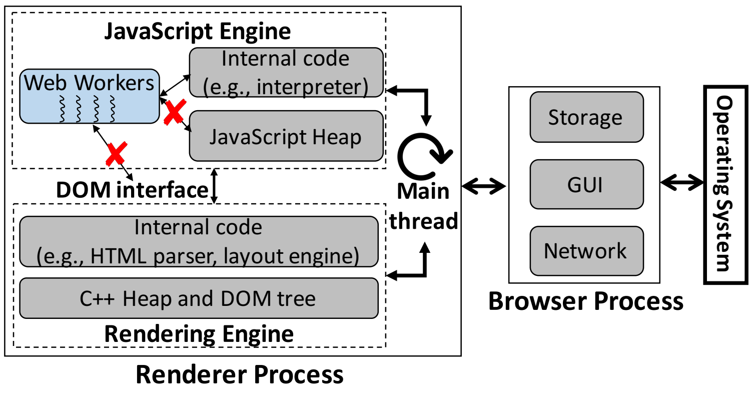


Figure 3.4: Computation model for Chromium browsers.

images), applies CSS styles, and renders content to the screen. During the HTML parse, the rendering engine builds a native representation of the HTML tree called the *DOM tree*, which contains a node per HTML tag. As the DOM tree is updated, the rendering engine recomputes a layout tree specifying on-screen positions for page content, and issues the corresponding paint updates to the browser process.

The JavaScript engine is responsible for parsing and interpreting JavaScript code specified in HTML `<script>` tags, either as inline code or referenced external files. During the page load, the JavaScript engine maintains a managed heap which stores both custom, page-defined JavaScript state and native JavaScript objects (e.g., `Dates` and `RegExps`). JavaScript code can initiate network fetches via the browser process (e.g., using `XMLHttpRequests`), and can also access the rendering engine's DOM tree (to update the UI) using the DOM interface. The DOM interface provides native methods for adding/removing nodes and altering node attributes; DOM nodes accessed via these methods are represented as native objects on the JavaScript heap.

The problem: single-threaded execution. JavaScript execution is single-threaded and non-preemptive [NM19, NM18a]. Worse, within a renderer process, all tasks across the two engines are coordinated to run on a single thread, called the *main thread*.[1] This lack of parallelism largely explains the poor use of

[1] Some Chromium implementations move final-stage rendering tasks to raster/composite threads that create bitmaps of tiles to paint to the screen.

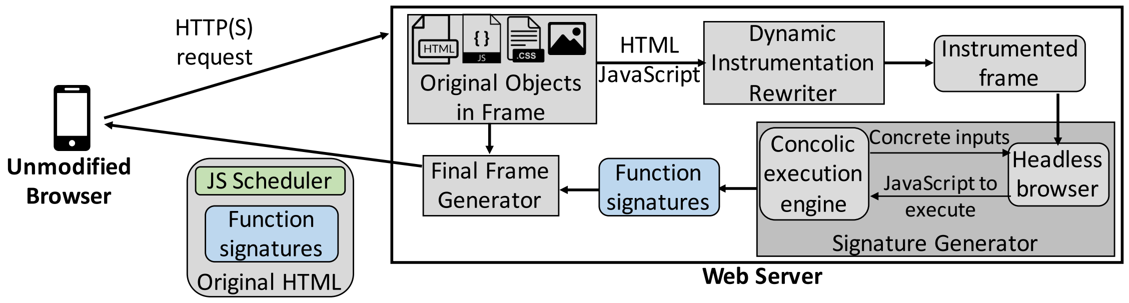


Figure 3.5: Overview of the generation and fetch of each frame's top-level HTML file with Horcrux. Offline, servers collect a frame's files (both local and third-party), generate comprehensive state access signatures for each JavaScript function, and embed that information in their pages. Parallelism decisions are made online by unmodified client browsers using the Horcrux JavaScript scheduler, which manages computation offloads to Web Workers and maintains the page's JavaScript heap and DOM tree.

CPU cores in §3.1.1. A primary reason for this suboptimal computation model is that the JavaScript language and DOM data structure (shared between the two engines) lack synchronization mechanisms (e.g., locks) to enable safe concurrency. Adding thread safety is feasible, but browsers have continually opted for a serial-access model to simplify page development. Browsers do create a separate renderer process per cross-domain `iframe` in a page (as per the Same-origin content sharing policy [sop20]). However, for the median page in the Alexa top 10,000, the top-level frame accounts for 100% of JavaScript execution delays.

In summary, despite benefits regarding simplified page development, the single-threaded execution model that browsers impose results in significant underutilization of mobile device CPU cores, inflated computation delays, and degraded page load times. We expect this negative interaction to persist (and worsen) moving forward given the steady and unrelenting increase in the number of JavaScript bytes included in mobile web pages, e.g., there has been a 680% increase over the last 10 years [HTT20].

# of Cores	% Speedup in Total JavaScript Runtime
2 cores	54%
4 cores	79%
8 cores	88%

Table 3.1: Potential parallelism speedups with varying numbers of cores. Results list median speedups in total time to run all JavaScript computations per page in the developed region.

3.2 Overview

Given the results in §3.1, a natural solution to alleviate client-side computation delays in mobile page loads is to *parallelize* JavaScript execution across a device's available CPU cores. However, not all workloads are amenable to parallel execution. In particular, we face the restriction that any introduced parallelism should preserve the page load behavior (and the final page state) that developers expected when writing their legacy pages—we call this property *safety*.

3.2.1 Potential Benefits

To estimate the potential benefits of parallelism with legacy pages, we analyzed the JavaScript code for each page in our corpus; in this section, we focus on page loads representative of those in developed regions, and we show later in §3.5 that similar benefits are achievable for page loads in emerging markets. Since JavaScript functions account for 94% of the JavaScript source code on the median page, our analysis operates at the granularity of functions, i.e., when splitting computations on a page across CPU cores, all code within a function runs sequentially on the same core. For completeness, we convert all code outside of functions into anonymous functions. For each function, we recorded both its runtime in a single load, as well as all accesses that it made to page state (in the JavaScript heap or DOM tree, as described below) in that load; §3.3.1.1 details the data collection process.

Using these logs, we estimated an *upper bound* on parallelism benefits by maximally packing function invocations to available cores and recording the resulting end-to-end computation times.

To ensure safety (defined above), our analysis respects two constraints: 1) functions can run in parallel if they access disjoint subsets of page state or only read the same state, and 2) functions that exhibit state dependencies (i.e., access the same state and at least one writes to that state) execute in an order matching that in the legacy page load.

As shown in Table 3.1, legacy pages are highly amenable to safely reaping parallelism speedups. For example, distributing computation across 4 cores could bring a 75% reduction in the total time required to complete all JavaScript computations on the median page. These resources are now common in both developed and emerging markets [DVB18a].

3.2.2 Goals and Approach

To realize these benefits in practice, we seek an approach that minimizes the bar to adoption. As a result, requiring developers to rewrite pages [MTK12] is a non-starter, given the complexities involved in manually reasoning about the impact of concurrently running portions of the JavaScript code on a page. Moreover, approaches that only require changes to browsers would either have to speculatively parallelize code [CFM13, MHS11] or perform client-side analysis of JavaScript code (akin to the analysis that informed our estimated benefits above). As we show in §3.5.3, the overheads imposed by either strategy make them untenable, especially on energy-constrained phones.

Therefore, we pursue an approach which can safely parallelize JavaScript execution on *legacy* web pages with *unmodified* browsers. As shown in Figure **??**, our solution, Horcrux, only necessitates server-side changes that do not require developer participation to rewrite pages. Web servers perform the *expensive* task of tracking the state accessed (in the JavaScript heap or DOM tree) by every JavaScript function in a page frame, and include this information in that frame in the form of per-function *signatures*. Servers also embed a JavaScript (JS) scheduler library in the frames they serve, which enables unmodified client browsers to perform the *cheap* task of managing parallelism using function signatures obtained from servers. Dynamically determining the parallelization schedule at the client helps Horcrux account for information only available at runtime, e.g.,

Delay type	0.5 KB	1 KB	100 KB	1 MB
Startup	128 ms	155 ms	237 ms	317 ms
Value I/O	0 ms	1 ms	1 ms	7 ms

Table 3.2: Web Worker overheads for different sizes of state transfers, i.e., source code size for startup delays and JavaScript object size for I/O delays.

the number of available threads and the control flows in the current load.

3.2.3 Challenges

The key building block in Horcrux is the widespread support in browsers for the Web Workers API [MDN20], which allows the JavaScript engine to employ additional computation threads (Figure 3.4), as specified by a page's source code. Using Web Workers to parallelize JavaScript execution, however, presents numerous challenges that complicate achieving the idealistic parallelism benefits from above.

1. Ensuring correctness. Determining what JavaScript code can be *safely* offloaded requires a comprehensive understanding of how that code will access JavaScript heap and DOM state *in the current page load*. This, in turn, depends on the traversed control flows, which can vary due to both client-side (e.g., `Math.Random`) and server-side (e.g., cookies) sources of nondeterminism in JavaScript execution [MEH10]. Missed state accesses can lead to dependency violations and broken pages.
2. Constrained API. Web Workers impose restricted computation models in two ways. First, due to the lack of synchronization mechanisms in JavaScript, Workers cannot access the JavaScript heap, and instead can only operate on values explicitly passed in by the browser's main thread (via `postMessages`). Thus, offloading computations to a Worker requires knowledge of precisely what state is required for those computations. Workers must then communicate computation results back to the main thread, which applies the results to the heap. Second, regardless of the state passed in, Workers cannot perform any DOM com-

putations, including invocations of native DOM methods or operations on live DOM nodes referenced in the heap. We note that Workers can spawn and manage other Workers, but still must rely on the main thread for access to any global state.

3. Offloading costs. Lastly, operating Web Workers is not free. Instead, as shown in Table 3.2, spinning up a Web Worker can take over 100 ms, even for small amounts of source code being passed in. Pass-by-value I/O adds an additional several milliseconds, depending on the size of the transferred state. Moreover, JavaScript execution is non-preemptive; so, Workers that finish their tasks may go idle for long durations if the thread responsible for assigning them more tasks is busy.

We posit that, it is for these reasons that only a handful of the Alexa top 1,000 pages use Workers, despite support by commodity browsers for over 8 years. We next describe how Horcrux overcomes these issues to automatically parallelize JavaScript execution for legacy pages.

3.3 Design

In designing Horcrux, we primarily need to answer two questions: 1) how to determine which JavaScript functions on a page can be executed in parallel without compromising correctness?, and 2) how to realize parallel execution at low overhead despite constraints placed by the browser's API? We present our solutions to these issues by separately describing server-side and client-side operation in Horcrux. Table 3.3 summarizes the main techniques underlying our design.

3.3.1 Server-side Operation

The goal of Horcrux's server-side component is to annotate page frames with per-function signatures that list the state that each function *might* access. Operating at a frame level, rather than at the granularity of entire pages, is in accordance with the browser's content sharing model [sop20, NM18a]. As in prior web optimizations that involve page alterations [NGM16, NM18a, MSN20], Horcrux assumes access to a frame's source files. These files can be quickly collected either using

Goal	Techniques	Section
Ensure correctness	For each function, use concolic execution to identify union of the state it accesses across all control flows	§3.3.1.1
	Adapt offloading schedule during a page load to account for control flow in current load	§3.3.2.2
Account for API restrictions	Main browser thread centrally manages global page state, coordinates offloads, and performs unoffloadable (DOM) computations	§3.3.2.1, §3.3.2.3
	Intercept any Web Worker's DOM tree accesses and relay to the main thread	§3.3.2.3
	Use function signatures to determine what heap values to pass-by-value from main thread to workers and back	§3.3.2.3
Minimize overheads	Offline server-side generation of per-function signatures	§3.3.1.1
	Offload at the granularity of root functions	§3.3.1.2
	Dynamically determine offloading schedule based on function runtimes in current page load	§3.3.2.1

Table 3.3: Overview of the main insights that Horcrux uses to address the challenges outlined in §**??**.

a headless browser[2] or via integration into content management systems (for local files) [Dru19, Wor19]. Source file collection and signature generation is retriggered based on hooks that many content management systems fire any time a (local) file-altering change is pushed [Dru19, Wor19, KRN21, MSN20], e.g., for A/B testing; we discuss third-party content changes and personalization in §3.3.2.2.

3.3.1.1 Generating signatures

Since web servers cannot precisely predict the control flows that will arise in any particular page load (e.g., due to client-side nondeterminism), each function's signature must conservatively list *all possible* state accesses for that function. For this reason, we cannot directly apply recent web dependency tracking tools [NGM16, NM18a] that rely purely on dynamic analysis to track data

[2]A headless browser performs all of the tasks that a normal browser performs during a page load except those that involve a GUI.

flows *in a given page load*. At the same time, pure static analysis approaches are ill-suited for JavaScript's dynamic typing, use of blackbox browser APIs, and event-driven/asynchronous execution [PLR16, MLF13]. For example, static analysis of variable name resolution is complicated by JavaScript statements that push objects to the front of the scope resolution chain (`with(obj)`{}), or dynamically generate code (`eval()`). Similar issues arise from JavaScript's extensive use of variable aliasing for DOM objects, and the fact that property names are routinely accessed via dynamically-generated strings instead of static ones.

Thus, we turn to concolic execution [GKS05, SMA05, KML21], a variant of symbolic execution that executes programs concretely (rather than symbolically) while ensuring complete coverage of all control paths. A concolic execution engine loads a program with a concrete set of input values and observes its execution; §**??** describes the inputs we consider, including browser state (e.g., cookies, screen size) and nondeterministic functions. Each input value and program-generated value is also given a symbolic expression constrained only by its type. For example, an input integer `a` may get a concrete value of 10 and an expression of $0 \leq \texttt{a} \leq 2^{32} - 1$; symbolic expressions for a given variable are inherited by others via assignment statements. At each branch condition, the execution follows the appropriate path based on the current program state. In addition, the engine restricts the symbolic expressions for the values that influenced the chosen path according to the branching predicate. Once the program completes, the engine performs a backwards scan through the executed code, selects branching decisions to invert, and inverts the relevant symbolic expressions; an SMT solver [DB08] then generates concrete input values that satisfy the new constraints. This process repeats until all paths are explored.

Note that, for efficiency, many recent symbolic execution tools opt for a form of concolic execution, rather than a purely symoblic approach [CS13]. More specifically, concolic execution engines consult the expensive constraint solver only at the end of each path (rather than at intermediate branches), and eliminate the need to accurately model each input source to a program (and the ensuing traversal of paths that arise due to modeling errors).

To generate function signatures, in addition to the default output of a concolic execution engine –

a list of potential control paths, with a concrete set of inputs to force each one – we must also log the state accessed by each path. To do this, prior to concolic execution, Horcrux instruments the JavaScript source code to log all accesses to state in both the JavaScript heap and DOM tree; our instrumentation matches recent dynamic analysis tools [NGM16, NNM18, NM18a], but with the following differences based on our parallelism use case.

- First, we care not just about the state that remains at the end of the page load [NM18a], but also any state accessible by multiple functions during a page load. Hence, in addition to global heap objects, Horcrux tracks all accesses to closure state: non-global state that is defined by a function *X* and is accessible by all nested functions that execute in *X*'s enclosed scope (anytime during the page load) [Mic12].
- Since signatures will ultimately be used for pass-by-value offloading to Workers, only the finest granularity of accesses are logged. For instance, if object `a`'s "foo" property is read, Horcrux would log a read to `a.foo`, not `a`.
- For the DOM tree, Horcrux adopts a coarser approach than prior work. Instead of logging reads and writes to individual nodes in the DOM tree, Horcrux only logs whether a function accesses any live DOM nodes, either via DOM methods or references on the heap, and if so, whether they are reads or writes. Tracking at the coarse granularity of accesses to the entire DOM tree is conservative with respect to parallelism. However, finer-grained tracking is not beneficial because, as we explain later, our design has the browser's main thread serialize all DOM operations.

3.3.1.2 Signature granularity

Ideally, to limit client-side bookkeeping overheads, signatures should match the granularity at which computation is offloaded. However, determining the appropriate offloading granularity is challenging. On the one hand, fine-grained offloading reduces the chance that offloadable computations access shared state, thereby improving the potential parallelism and use of available Workers. On the other hand, finer granularities imply increased coordination overheads.

To address this tradeoff, Horcrux generates signatures (and offloads computation) at the granularity of *root function* invocations, i.e., invocations made directly from the global JavaScript scope. The signature for each root function invocation includes the state accessed not only by that function, but also by any nested functions that are invoked in the call chain until the global scope is reached again.

Root function signatures are desirable for two reasons. First, they leverage our finding that functions already account for 94% of JavaScript code on the median page (§3.2) and thus provide a natural granularity for offloading; as in §3.2, Horcrux servers wrap all JavaScript code outside of any function into anonymous functions. Second, and more importantly, root functions impose far smaller offloading overheads compared to finer-grained function-level offloading, while enabling comparable parallelism benefits: the number of offloads drops by $4\times$, while the median potential benefits remain within 27% of those in Table 3.1. The reason is that there often exists significant state sharing within the invocations for a given root function (and its nested components), but less so across root functions, enabling parallelism.

3.3.2 Client-side Operation

Even with function signatures, a server cannot precompute a parallel execution schedule because the precise control flow, and hence, the set of functions executed and their runtimes, will vary across loads. Instead, Horcrux employs a client-side JavaScript computation scheduling library that unmodified browsers can run to dynamically make parallelism decisions based on signatures and the aforementioned runtime information. The key challenges are in efficiently ensuring correctness while offloading to multiple Workers, and handling the fact that signatures may be missing for certain functions. We next discuss how Horcrux addresses these challenges.

3.3.2.1 Dynamic scheduling

To load a page frame, any unmodified browser first downloads the top-level HTML, whose initial tag is an inline `<script>` housing Horcrux's scheduler library and all root function signatures. The scheduler runs on the browser's main thread and begins by asynchronously creating a pool of uninitialized Workers. This helps hide the primary overhead of spawning Workers amongst unavoidable delays for parsing initial HTML tags and fetching files they reference.

The scheduler then operates entirely in event-driven mode, whereby it waits for incoming postmessages specifying computations to perform or those that have completed, and makes subsequent offloading decisions. Importantly, to keep the main thread as idle as possible, the scheduler offloads *all* computations that Web Workers can support, and is primarily responsible for managing Workers and maintaining the page's global JavaScript heap and DOM state. This helps adapt the parallelization schedule within any page load to the runtimes of every root function in that load. The reason is that the main thread will be available to quickly assign a new function invocation (if one exists) to any worker that completes executing the function previously assigned to it.

Once the scheduler is defined, the browser operates normally, recursively fetching and evaluating referenced HTML, JavaScript, CSS, and image files. However, all JavaScript function invocations are modified to pass through the scheduler for offloading. More specifically, each root function is rewritten such that, upon invocation, the function sends a post message to the scheduler specifying its original source code and that of any nested functions. Special care is taken for asynchronous functions (e.g., timers) whose invocations are regulated by the browser's internal event queue which the scheduler does not have access to. To ensure visibility to such functions, the downloaded page includes shims around registration mechanisms for asynchronous functions, e.g., `setTimeout()`. Each shim modifies registered functions to send messages to the scheduler upon invocation.

Each time a root function is invoked, the scheduler uses its signature to determine whether or not it can be immediately offloaded. If not, the function is stored in an in-memory queue of *ordered*, to-be-invoked functions along with its signature. Functions are not offloadable if there are no available Workers, or if they might access state that is being modified by an already-offloaded or queued function. Note that functions that may access the DOM can be offloaded in parallel; we will discuss how to ensure safety in these cases shortly.

Regardless of the decision for a given invocation, the browser continues its execution. At first glance, it may appear that continuation after a queued invocation may generate errors since the queued function could alter the set of downstream invocations. However, recall that Horcrux offloads at the granularity of root functions—any nested invocations are already offloaded, and the ordering of root functions is mostly predefined by the page's source code. There are two exceptions. First, a function can alter downstream source code using `document.write()`; to handle this, the scheduler synchronously offloads such functions, thereby blocking downstream execution. Second, a root function can register an asynchronous function with a 0-ms timer—such functions are intended to run immediately after the current invocation. For this, the root signature includes state accesses for the 0-ms timeout functions they define. Once the root function is discovered, the scheduler adds a placeholder for the timeout function to its queue, thereby blocking downstream invocations that share state with the timeout function.

3.3.2.2 Handling Missing Signatures

We have assumed so far that the HTML file of every page frame includes accurate signatures for all JavaScript functions executed in that frame. This may not always hold.

- *Stale signatures.* A frame can include JavaScript content from multiple origins, and to preserve HTTPS content integrity and security [NSM19], Horcrux has each origin serve its own files directly to clients. A third-party origin may update a script without explicitly informing the top-level origin to regenerate signatures. We expect this to be rare for two reasons. First, JavaScript files often have long cache lives (median of 1 day in our corpus), indicating infrequent changes. Second, scripts in a frame can share state [sop20]. Thus, even today, if a third-party origin significantly alters a script it serves, this should be communicated to other origins to avoid unexpected or broken behavior.
- *Dynamically-generated or personalized scripts.* JavaScript files may be created or personalized in response to user requests [MSN20], e.g., based on Cookies. Unfortunately, generating signatures during client page loads would be far too slow. To handle this, for dynamically-generated or personalized first-party content, Horcrux could perform concolic execution on server-side content generation logic to determine the execution paths for all variants of a given response

(§3.4). Third-party content of this type may result in missing signatures since the top-level origin does not have access to a user's third-party Cookies (or personalized content). However, many browsers preclude third-party Cookies in frames to prevent the tracking of users across sites [Coo20].

- *Timeouts of concolic execution.* Given infinite time, concolic execution is guaranteed to explore all possible JavaScript execution paths in a page [GKS05, SMA05]. However, the process may timeout, either during the execution of a given path (if the SMT solver cannot invert a branch condition), or less likely, due to a time bound placed on overall signature generation. Regardless of the reason, the effect is a potentially missing or incomplete signature. Such timeouts did not arise (i.e., concolic execution completed) for any pages in our experiments (§3.5.1). However, in the event that concolic execution does not complete, Horcrux could detect such timeouts *prior* to serving content to clients, and could thus address the corresponding missing signatures (described below) or revert to a normal page load.

Horcrux accounts for missing or inaccurate signatures in two ways. First, any underexplored function X is assigned a signature of $\star$, indicating that X may access all page state. This overconstrains the client's load, but ensures correctness: the client will execute X serially, and will also serialize downstream functions since X might alter the state they access. Second, signatures are keyed by a hash of the function's source code. Invocations without matching signatures are assigned signatures of $\star$ by Horcrux's client-side scheduler.

3.3.2.3 Function Offloading and Execution

Lastly, we discuss the mechanics of how every function invocation that is offloaded by the Horcrux scheduler is executed in a Web Worker. Figure 3.6 illustrates this process.

For each offloadable function, the scheduler uses its signature to generate a JSON package listing the information that the Worker will require for execution, i.e., the source code (including nested functions) and the current values for the function's read state. The source code is modified such that, upon completion, values in the write state are gathered and sent to the main thread (as exe-

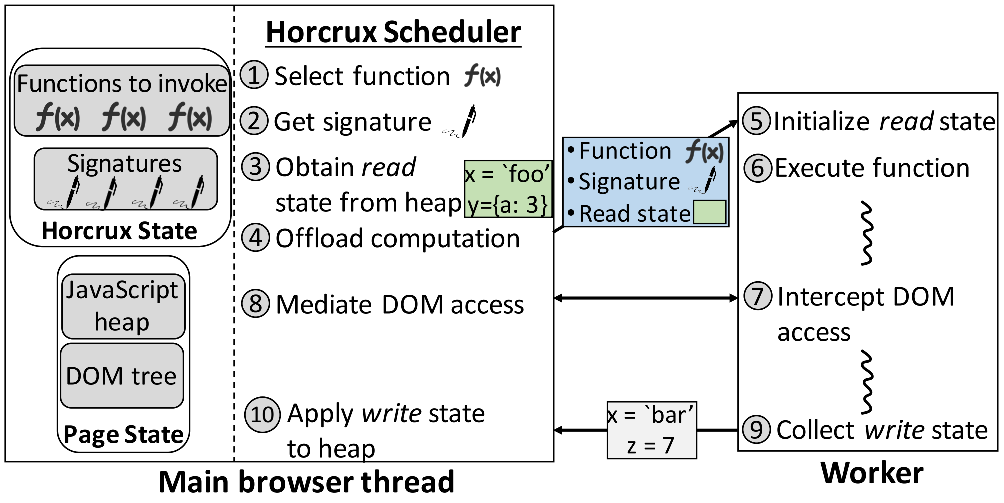


Figure 3.6: A single function offload with Horcrux.

cution results). In addition, closure values in the read state are embedded into the corresponding function's source code. Upon reception, the post message handler inside the Worker sets up the read state in the Worker's global scope via assignment statements, and runs the code using the browser's `Function` constructor.

For the most part, functions execute normally, with JavaScript heap accesses hitting in the Worker's global scope, or for nested functions, in the scope of their parents–recall that Horcrux offloads entire root functions so nesting relationships are preserved. However, the key difference is with respect to DOM accesses: workers cannot call native DOM methods or operate on live DOM nodes referenced in the heap (§3.2.2). To handle this, all DOM computations are mediated by the scheduler and are serially applied to the live DOM tree. To intercept DOM computations, Horcrux includes shims around all DOM methods in each Worker environment; returned DOM nodes are replaced with proxy objects to interpose on direct accesses. Each intercepted DOM access blocks execution in the Worker, and is sent to the scheduler where it is queued; blocking is enforced using JavaScript generator functions [Mic14].

The scheduler grants readers-writer locks to each Worker that may need to access the DOM tree (as per their signatures). Locks are granted in the order that the scheduler receives function invocations; note that this may not match the order in which functions are offloaded, but it preserves the relative ordering of DOM updates seen in a normal page load. As a concrete example, consider a function `a` that reads from the DOM, and a later function `b` that writes to the DOM. `b` may attempt to access the DOM first (e.g., if it is offloaded earlier or its DOM access occurs early in the function), but the scheduler will block it and wait to grant the lock to `a` first. Workers release DOM locks at the end of their execution. In essence, root functions that only read from the DOM tree can run in parallel, although their constituent DOM accesses are serialized on the browser's main thread. Root functions that write to the DOM are run serially with respect to other DOM-accessing root functions (to match the relative ordering of DOM updates in an unmodified load); for context, only 7.4% of the root functions on the median page in our corpus involve DOM writes, mitigating the effects of such serialization. Importantly, locking is done at the granularity of entire root invocations because the scheduler does not definitively know whether a given function will access the DOM in the current load (and if so, how many times); signatures only list that a DOM access *may* occur.

Once a function completes execution, the Web Worker sends its computation results (i.e., its write state) to the scheduler. The Worker then clears any state in its scope in order to be ready for the next offload. Upon receiving computation results, the scheduler applies the writes to the global JavaScript heap; recall that DOM writes have already been made. One subtlety here is with respect to scope, and closure state in particular. The scheduler can access and apply computation results to the global scope's heap. However, root functions can also modify shared closure variables which are not accessible from the global scope (§3.3.1). For such writes, the scheduler maintains a global hash map listing the latest closure values. This map is updated as Workers complete computations, and is also queried to obtain read values when offloading; note that correctness is ensured because all offloads and computation results pass through the scheduler.

Finally, once the scheduler applies computation results to the JavaScript heap, it scans through its queue of ordered, to-be-executed functions and offloads the next one that can safely run. Given the serialization of DOM computations described above, if a function that writes to the DOM is queued, the scheduler prioritizes queued functions that do not access the DOM (and thus won't incur locking delays).

3.3.3 Discussion

Key to Horcrux's operation is the decision to maximally offload and parallelize JavaScript computations at the granularity of root functions. Recall that this decision was motivated by our analysis of the JavaScript computation on the pages in our experimental corpus (§3.2.1 and §3.3.2), which revealed that root function offloading favorably balances client-side overheads (e.g., pass-by-value I/O, main thread responsiveness) with the achievable speedups from parallelization.

However, these decisions may not deliver the largest speedups for certain pages. For example, root function-level offloading might be too restrictive and forego significant parallelism benefits, e.g., if a root function includes two nested functions that access entirely disjoint state but both involve significant runtime. Similarly, the root functions for certain pages could each embed only a single nested function, thereby inflating offloading costs relative to parallelization speedups, and potentially harming overall performance.

Although we did not observe these behaviors for any of the pages in our experiments (§3.5), we note that developers could perform analyses similar to the one presented in §3.2.1 to determine whether automatic parallelization of JavaScript code is desirable for (i.e., can speed up) their pages, and if so, what the best offloading granularity is. Importantly, these analyses do not require further instrumentation of web pages, and instead can directly leverage Horcrux's signatures, the per-function runtimes reported by in-built browser profilers, and the relatively stable offloading costs reported in Table 3.2.

3.4 IMPLEMENTATION

Horcrux instruments JavaScript source code to generate signatures and prepare frames using Beautiful Soup [Ric19], Esprima [Hid], and Estraverse [Suz]. To employ concolic execution, we use a modified version of Oblique [KML21], which runs atop a headless version of Google Chrome (v85) and the ExpoSE JavaScript concolic execution engine [LMK17]. Our Oblique implementation considers inputs specified by HTTP headers (e.g., Cookie, User-Agent, Origin, Host), the device (e.g., screen coordinates), and built-in browser APIs including nondeterministic functions [MEH10] (e.g., `Math.Random`) and DOM methods. Input values suggested by the SMT solver are fed into the page load via either 1) rewritten HTTP headers, or 2) shims for browser APIs.

We grant Oblique a maximum of 10 mins to consider a given execution path, and 45 mins to explore all paths for a given page; we find that these time values are sufficient to ensure that concolic execution completes for all of the pages in our experimental corpora (§**??**). Signatures from each load are sent to a dedicated analysis server for aggregation. Since our current implementation operates directly on downloaded page source code and not live web backends (§3.5.1), Horcrux eschews Oblique's ability to perform concolic execution on server-side application logic. In total, Horcrux's implementation involves 5.6k LOC in addition to Oblique, including 4.5k for dynamic tracing (both static instrumentation and runtime tracking) and 1.1k for client-side scheduling.

Overheads. On the client-side, Horcrux inflates page sizes by 13 KB at the median (when using Brotli compression [Goo19b]). The scheduler accounts for 3 KB of that.

3.5 Evaluation

We empirically evaluate Horcrux across a wide range of real pages, live mobile networks, and phones from both developed and emerging markets. Our key findings are:

- Horcrux reduces median browser computation delays by 31-44% (0.9-1.5 secs), which translates to page load time and Speed Index improvements of 18-29% and 24-37%. Improvements grow with warm browser caches (§3.5.2).
- Horcrux delivers larger speedups than prior web optimizations that 1) reduce required computations (by 1.7-2.1$\times$), 2) speculatively parallelize computations (by 1.3-1.6$\times$), and 3) mask network round trips (by 1.4-2.1$\times$); Horcrux is complementary to network optimizations and running them together lowers load times by 31-45% (§3.5.3).

- Although the median page has 12 possible execution paths, Horcrux's reliance on conservative signatures (for correctness) only foregoes 7-10% of speedups compared to using signatures that target a specific load (§3.5.4).
- Horcrux is highly amenable to partial deployment: benefits are within 2% of total adoption when only a page's top-level origin runs Horcrux. Benefits persist for personalized pages and desktop settings (§3.5.5).

3.5.1 Methodology

We evaluated Horcrux in two different scenarios:

- **Developed regions.** We consider 700 US pages, randomly selected from and equally distributed amongst the following sources: popular landing pages from the Alexa [Ale18] and Tranco [LGT] top 1000 lists, popular interior pages from the Hispar 100,000 list [ACF20], and less popular pages (landing and interior) from the 0.5 million-site DMOZ directory [dmo]. Thus, our corpus involves diversity in terms of both page popularity and location within a website (i.e., landing vs. interior). From this set, we report results for the 582 pages that our current implementation could generate accurate signatures for. More specifically, we removed pages for the following reasons: 1) inaccurate signatures due to unsupported language features, which led to premature JavaScript termination (92) or rendering defects (22), and 2) unsupported features with Oblique (4). For all of the remaining pages, Horcrux's concolic execution and overall signature generation completes, the total JavaScript runtime with Horcrux falls within a standard deviation of that in default loads, and the final rendered page is unchanged. Our experiments consider two powerful phones, a Pixel 3 (Android Pie; 2.0 GHz octa-core processor; 4 GB RAM) and a Galaxy Note 8 (Android Oreo; 2.4 GHz octa-core; 6 GB RAM). For space, we only present results for the Pixel 3, but note comparable results with the Galaxy Note 8.
- **Emerging regions.** Web experiences in emerging regions often comprise different page compositions and devices than those considered above [DVB18a, ABC15, AHQ16, ZCP14]. To mimic such scenarios, we focus on a single emerging region: Pakistan. We consider a corpus of 100

landing and interior pages (50 each) selected from the Alexa Top 500 sites in Pakistan. Our evaluation uses the Redmi 6A phone (Android Oreo; 2.0 GHz quad-core processor; 2 GB RAM) that is popular in the region [red20]. As per the same correctness checks as above, we report numbers on 91 pages.

Unless otherwise noted, page loads were run with Google Chrome for Android (v85). Mobile-optimized (including AMP [Gooa]) pages were always used when available.

To create a reproducible test environment, and because Horcrux involves page rewriting, we use the Mahimahi web record-and-replay tool [NSW15]. Emerging regions pages were recorded using a VPN to mimic a client in Pakistan. As described in §3.3, Horcrux's signature generation and page rewriting were performed offline. To replay pages, we hosted the Mahimahi replay environment on a desktop machine. Our phones were connected to the desktop via USB tethering and live Verizon LTE and WiFi networks with strong signal strength; LTE speeds for emerging regions experiments were throttled to Pakistan's 7 Mbps average [Ope20]. We used Lighthouse [Goo19c] to initiate page loads via the USB connection, and all page load traffic traversed the wireless networks.

We evaluated Horcrux on multiple web performance metrics: 1) *Total Computation Time* (TCT), or the critical path of time spent parsing/executing source files and rendering the page, 2) *Page Load Time* (PLT) measured as the time between the `navigationStart` and `onload` JavaScript events, and 3) *Speed Index* (SI) which captures the time required to progressively render the pixels in the initial viewport to their final form. TCT and PLT are measured using the browser profiler, while SI was reported by Lighthouse. In all experiments, we load each page three times with each system under test, rotating amongst them in a round robin; we report numbers per system from the load with the median TCT.

3.5.2 Page Load Speedups

Cold cache. Figure 3.7 illustrates Horcrux's ability to reduce browser computation delays compared to default page loads. TCT reductions were 41% (1.0 sec) and 44% (1.5 sec) for the median page in the developed and emerging region's WiFi settings, respectively. Improvements were

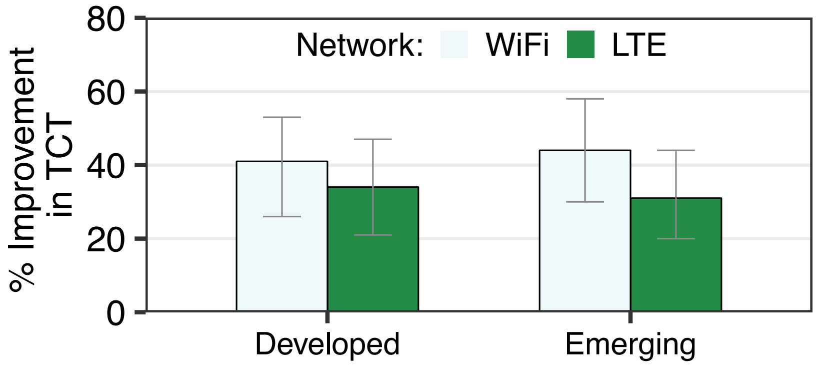
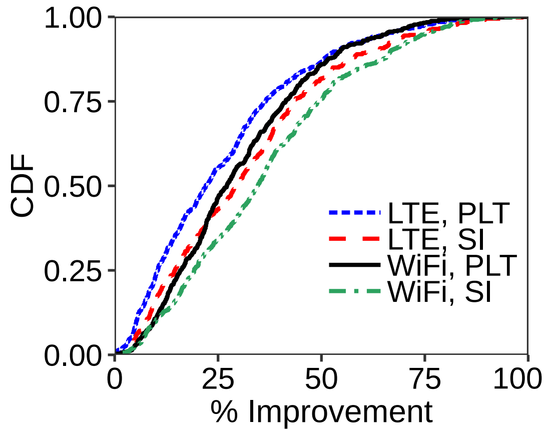
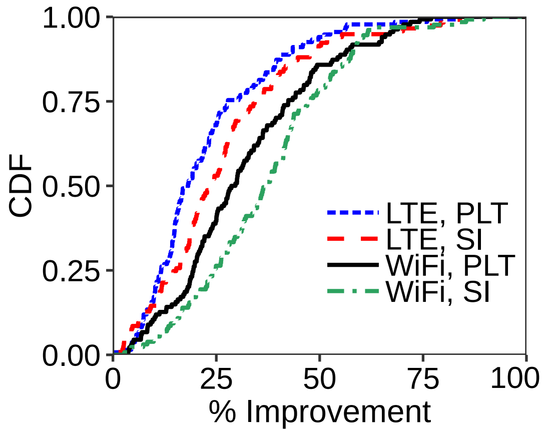


Figure 3.7: Cold cache TCT improvements over default page loads. Bars list medians, with error bars for 25-75th percentiles.



(a) Developed region.



(b) Emerging region.

Figure 3.8: Distributions of cold cache per-page improvements with Horcrux vs. default page loads.

34% and 31% with LTE. Figure 3.8 shows how these computation speedups translate into faster end-to-end (i.e., including network delays) loads. For example, on WiFi, median improvements in the developed region setting were 27% for PLT and 35% for SI. Despite the lower CPU clock speeds, these numbers only marginally increase to 29% and 37% in the emerging region. Further improvements were hindered primarily by the lower number of available cores (and thus ability to parallelize). Benefits with Horcrux on LTE were comparable, but consistently lower than with WiFi. For example, in the developed region, PLT and SI speedups were 22% and 29%. The reason is that network delays (which Horcrux does not improve) account for larger fractions of end-to-end

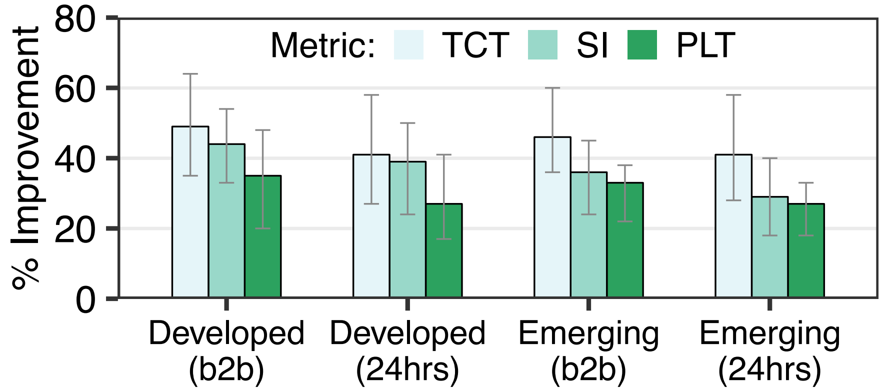


Figure 3.9: Warm cache speedups over default page loads on LTE. Bars list medians, with error bars for 25-75th percentiles.

load times on LTE.

Warm cache. Figure 3.9 shows Horcrux's speedups in different browser caching scenarios. We consider back-to-back page loads, as well as those separated by 12 and 24 hours. As shown, Horcrux's improvements grow as browsers house more objects in their caches. For example, in the back-to-back scenario, PLT and SI improvements in the developed region's LTE setting were 35% and 44%; for context, these improvements were 22% and 29% with cold caches. Improvements drop to 27% and 39% in the 24-hour warm cache scenario. The reason is that more cache hits lead to lower network delays and computation dominating end-to-end performance. In addition, cache hits enable browsers to begin processing JavaScript files earlier. This, in turn, provides Horcrux's scheduler with more invocation options at any time, thereby increasing the amount of potential parallelism.

3.5.3 Comparison to State-of-the-Art

Network optimizations. We first considered Vroom [RNU17], a system in which web servers intelligently use HTTP/2's server push and preload features to aid clients in discovering (and downloading) required files ahead of time. Thus, Vroom is primarily a network-focused optimization. However, key to Vroom's benefits is the improved CPU utilization that results from eliminating blocking network fetches.

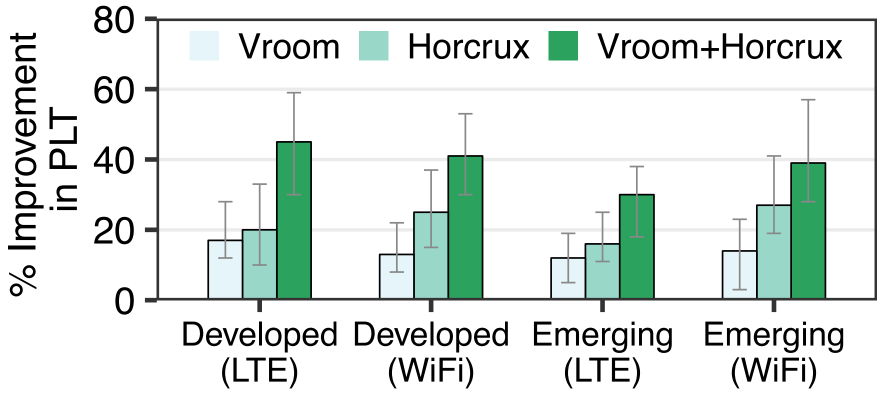


Figure 3.10: Horcrux vs. Vroom [RNU17] over LTE networks. Bars list medians, with error bars for 25-75th percentiles.

As shown in Figure 3.10, Horcrux delivers larger speedups than Vroom. For example, in the developed region, median PLT speedups with Horcrux are 2.1× and 1.3× higher than Vroom's on WiFi and LTE, respectively. In the LTE setting, Vroom delivers larger PLT speedups for 9% of pages. The reason is that network delays play a larger role in end-to-end load times for these pages, either due to less computation or more required network fetches. This drops to 1% and 3% when we move to the developed region's WiFi network or the emerging market's LTE network; in both cases, compute becomes more of a determinant of overall delays. Importantly, Figure 3.10 also confirms that Horcrux and Vroom are largely complementary to one another, with the combined systems outperforming each in isolation.

Reducing required computations. Prepack [Fac19a] is a server-side system that reduces the amount of JavaScript computation that clients must perform to load pages. To do this, Prepack performs static analysis on a page's JavaScript code, identifies expressions whose results are statically computable, and replaces those expressions with equivalent but simpler versions that remove intermediate computations. Importantly, computations involving client-side or nondeterministic state are unmodified; this helps Prepack preserve page behavior and correctness, unlike other computation reduction systems (§3.6). As shown in Table 3.4, Horcrux is more effective at reducing computation delays than Prepack: median TCTs are 26% and 24% lower with Horcrux in the developed and emerging regions, respectively.

System	**Developed**	**Emerging**
Horcrux	1.63 (1.98)	2.15 (2.37)
Prepack [Fac19a]	2.19 (2.47)	2.82 (3.36)
Speculative parallelization	2.01 (2.28)	2.50 (3.07)

Table 3.4: Comparing Horcrux with prior compute optimizations. Results are for WiFi networks and list median (75th) percentile TCTs in seconds.

Speculative parallelism. Prior efforts to increase parallelism in page loads (§3.6) primarily rely on speculative decisions about what can run in parallel, and runtime checks to detect (and revert from) dependency violations. Although these systems do not target all JavaScript execution, we considered a baseline that employs a similar parallelism strategy for JavaScript computation. Our baseline opportunistically parallelizes all root function invocations, and uses JavaScript proxy objects to track state accesses in each Worker. Any parallelized computations that share state are discarded, and the corresponding functions are rerun serially on the main browser thread. As shown in Table 3.4, Horcrux delivers superior median TCT values that are 14-19% lower across the two regions. The reason is twofold. First, proxy-based tracking to ensure correctness adds 10% overhead to JavaScript runtimes. Second, any speculation errors result in serial execution on the main thread and wasted computation (and thus, more overall computation). Using the setup in §3.5.5, we observe that this wasted computation inflates mobile device energy consumption by 9% for the median page on WiFi.

3.5.4 Understanding Horcrux's Benefits

Dissecting Horcrux's speedups. We analyzed Horcrux's behavior (and improvements) along three different axes. We focus on the developed region, but note that the trends hold for the emerging region setting. First, as expected, Horcrux's improvements are larger for pages that require more computation to load. For instance, with WiFi, median PLT improvements with Horcrux were 35% for pages with more than 3 seconds of computation time, as compared to 23% for pages that did not meet that criteria. This divide carries over to different page types as well: improvements were

15% higher for interior pages than landing pages. The reason is that interior pages often involve more computation [ACF20, NM18a].

Second, within each load, we investigated the degree of parallelism that Horcrux achieves for JavaScript computation. For the median page, when loaded over WiFi, Horcrux reaches a maximum of 6 concurrent Web Workers; this drops to 4 on LTE due to the aforementioned network delays limiting the scheduler's parallelism options.

Third, in addition to JavaScript parallelization, Horcrux reduces TCT by freeing the browser's main thread for rendering tasks. To understand the contribution of each source to Horcrux's speedups, we analyzed the browser's computation profiler. Overall, we find that both sources provide substantial benefits. For instance, on WiFi, Horcrux shrinks effective JavaScript computation times by a median of 42% (557 ms), and decreases end-to-end rendering delays by 36% (465 ms).

Server-side overheads. Signature generation took 33 minutes for the median page, and involved two primary overheads: the median page involved exploring 12 different execution paths via concolic execution, and our dynamic instrumentation (incurred in each load) inflated load times by 44%. These non-negligible delays are why Horcrux performs comprehensive signature generation offline, on servers. To understand how often servers have to incur these overheads, we recorded a random set of 50 pages from our emerging region's corpus every 12 hours for 1 week. The median page's signatures remained unchanged for the entire duration, in part due to Horcrux's coarse-grained DOM tracking which is unaffected by changes to HTML state (e.g., headlines).

Cost of conservative signatures. Horcrux relies on conservative signatures that list the state accesses across all possible control flows. While this ensures correctness, it may over-constrain a client load that traverses only a subset of those control flows. To understand the impact of this conservative strategy, we compared Horcrux with a variant that generates signatures for the precise control flows traversed in the target client load. Surprisingly, we observe that Horcrux's conservative behavior results in only mild performance degradations: improvements drop by 10% and 7% for PLT and SI, respectively, for the developed region WiFi setting. The reason is that conservative signatures

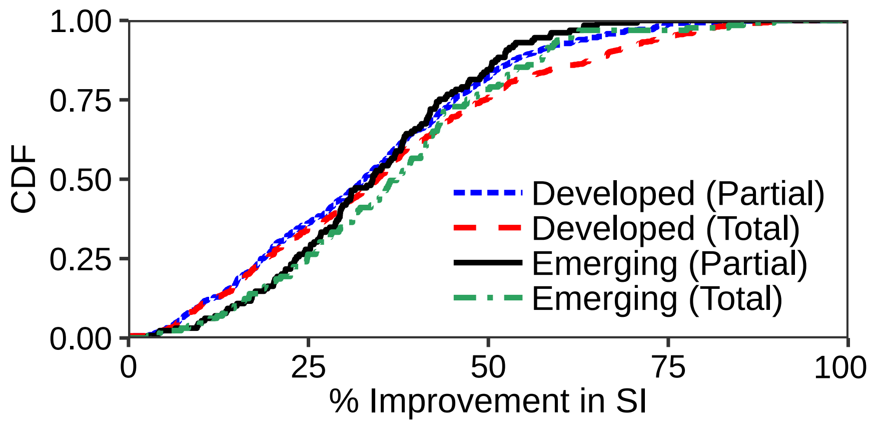


Figure 3.11: Evaluating Horcrux when only a page's top-level origin participates. Results are for WiFi networks.

typically either add only a few extra state accesses to a given root function, or many that are only accessed for short durations (i.e., within a root function)—neither significantly restricts parallelism.

3.5.5 Additional Results

Partial deployment. Our results thus far assumed that each frame in a page adopts Horcrux, i.e., embeds Horcrux's scheduler and signatures in the HTML. Figure 3.11 shows Horcrux's benefits when only the top-level origin for the page participates—this represents the simplest deployment scenario as the top-level origin is directly incentivized to accelerate loads of its pages. In this scenario, all JavaScript code in third-party-owned frames runs serially; JavaScript in the main frame can still be safely parallelized as browsers prevent cross-frame state sharing [sop20]. As shown, most of Horcrux's benefits persist, despite the lack of adoption by third-party frames. For example, in the developed region's WiFi setting, median SI benefits are within 2% of those with total adoption. The reason is that most JavaScript runtime (100% on the median page) resides in the page's main frame.

Personalized pages. To evaluate Horcrux in settings where pages dynamically generate or personalize their content, we selected 20 pages from our developed region's corpus that supported user accounts. For each page, we created two user accounts, selecting different preferences when possible, e.g., order results based on time or popularity. For every file that does not appear in both loads, or whose content is different across the page versions, we assign its constituent functions signatures of $\star$ (§3.3.2.2). Overall, we observe that such personalization has minimal impact on Horcrux's speedups: in the WiFi setting, Horcrux's median load time benefits drop by only 4%. The reason is that only 6% of computation delays are accounted for by personalized scripts.

Energy savings. We connected our Pixel 3 phone to a Monsoon power monitor [Inc18] and loaded the pages in our developed region corpus. With cold caches, Horcrux's speedups drop median per-page energy usage by 12% and 15% on WiFi and LTE. Savings are primarily from accelerating end-to-end computation (and load times), which results in lower active durations for WiFi or LTE radios.

Desktop page loads. Horcrux's acceleration techniques can also speed up desktop page loads. To evaluate this, we recorded desktop versions for the pages in our developed region corpus, and loaded them using a Dell G5 desktop and a wired network connection. We find that Horcrux reduces median TCT by 39% (0.52 secs). These speedups translate to PLT and SI improvements of 25% and 31%, respectively. At first glance, these improvements may appear surprising given the faster CPU clock speeds that desktops possess. However, desktops also possess more cores and load pages with more JavaScript computation [HTT20], enabling more parallelism.

3.6 Related Work

Parallelization efforts. ParaScript [MHS11] and others [NKH16] leverage new runtimes and compiler information to speculatively parallelize iterations for hot loops in long-running JavaScript code (not page loads, where compilation overheads are too costly). In contrast, Horcrux operates with unmodified browsers, targets parallelism for general JavaScript code beyond loop iterations, and sidesteps the significant overheads of speculation errors and runtime checks (§3.5.3) by using conservative signatures. Zoomm [CFM13] and Adrenaline [MTK12] leave JavaScript execution unchanged, and instead parallelize tasks such as CSS rule parsing. These systems are orthogonal to Horcrux, which focuses entirely on JavaScript parallelization. Lastly, several libraries [par20, thr20] aid developers in writing parallel JavaScript code by abstracting inter-worker messaging. However, developers are responsible for identifying and enforcing (safe) parallelism decisions—Horcrux automates these tasks for legacy pages.

Reducing web computation overheads. Prior measurement studies have analyzed the performance of mobile web browsers [WBK13, NB16, DVB18a, Osm18]. Like us (§3.1), they find that browser computations are a primary contributor to high page load times. In response to these studies, three separate lines of work have aimed to alleviate browser computation delays. First, certain sites have manually developed mobile-optimized versions of their pages using restricted forms of HTML, JavaScript, and CSS, e.g., according to the Google AMP standard [Gooa, JBW19]. In contrast, Horcrux accelerates legacy pages without developer effort. Further, we find that Horcrux is able to accelerate the loading of AMP pages, which constitute 27% of our corpora.

Second, some systems [Ama18, WKW16, Ope18a, CZH20] offload computation tasks to well-provisioned proxy servers, which return computation *results* that are fast to apply. Though effective, such systems pose significant scalability challenges to support large numbers of mobile clients [SJN17]. Worse, by relying on (often third-party) proxy servers, these systems violate HTTPS' end-to-end security guarantees [NSM19]; clients must trust proxies to preserve the integrity of their HTTPS objects, and also must share private Cookies to accelerate personalized page content. In contrast, Horcrux is HTTPS-compliant.

Third, systems like Prophecy [NM18a] enable servers to return post-processed page files that elide intermediate computations. However, content alterations with these systems may break page functionality [ABC15], particularly for pages that adapt execution based on client-side state that servers are unaware of, e.g., localStorage. In contrast, Horcrux does not alter the set of computations required to load a page, and instead aims to execute those computations more efficiently.

Network optimizations for the web. Systems such as Alohamora [KRN21], Vroom [RNU17], and others [EGJ15, WBK14] leverage HTTP/2's server push and preload features to proactively serve files to clients in anticipation of future requests (thereby hiding download delays). Fawkes [MSN20] develops static HTML templates that can be rendered while dynamic data is fetched. Polaris [NGM16] and Klotski [BWW15] reorder network requests to minimize the number of effective round trips while respecting inter-object dependencies. Cloud browsers [SPG14, NSM19, NSW15] shift network round trips to wired proxy server links. Content delivery networks [NSS10, FFM04] serve popular objects from proxy servers that are geographically close to clients, while compression proxies [ABC15, SMK15, Ope18b] selectively compress objects in-flight between servers and clients. Lastly, a handful of systems prefetch content according to predicted user browsing behavior [PM96, LRS12, WLZ12]. As shown in §3.5.3, these efforts are complementary to Horcrux, which reduces browser computation delays by parallelizing JavaScript execution. Further, recall that computation delays often exceed user tolerance levels on their own (§3.1).

Concolic execution for web optimization. Like Horcrux, Oblique [KML21] uses concolic execution to accelerate web page loads. Indeed, Horcrux's server-side component builds atop Oblique's JavaScript concolic execution engine by adding dynamic instrumentation to capture per-function signatures (§3.4). However, despite this similarity, Oblique and Horcrux target different delays in the page load process: Oblique enables third-party servers to securely prefetch URLs that a client will need during a page load (hiding the associated network fetch delays), while Horcrux parallelizes the JavaScript execution required to load a page (reducing the associated computation delays). Consequently, as with other network-focused optimizations (§3.5.3), Oblique can run alongside Horcrux to provide complementary benefits.

3.7 Conclusion

Horcrux automatically parallelizes JavaScript computations in legacy pages to enable unmodified browsers to leverage the multiple CPU cores available on commodity phones. To account for the non-determinism in page loads and the constraints of the browser's API for parallelism, Horcrux employs a judicious split between clients and servers. Servers perform concolic execution of JavaScript code to conservatively identify parallelism opportunities based on potential state accesses, while clients use those insights along with runtime information to efficiently manage parallelism. Across browsing scenarios in developed and emerging regions, Horcrux reduced median browser computation delays and load times by 31-44% and 18-37%.

[3se18] “Find out how you stack up to new industry benchmarks for mobile page speed).” https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/, 2018.

[ABC15] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. “Flywheel: Google’s Data Compression Proxy for the Mobile Web.” NSDI ’15. USENIX, 2015.

[ACF20] Waqar Aqeel, Balakrishnan Chandrasekaran, Anja Feldmann, and Bruce Maggs. “On Landing and Internal Web Pages.” In *Proceedings of the 2020 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC. ACM, 2020.

[AHQ16] Sohaib Ahmad, Abdul Lateef Haamid, Zafar Ayyub Qazi, Zhenyu Zhou, Theophilus Benson, and Ihsan Ayyub Qazi. “A View from the Other Side: Understanding Mobile Phone Characteristics in the Developing World.” In *Proceedings of the 2016 Internet Measurement Conference*, IMC ’16, p. 319–325. Association for Computing Machinery, 2016.

[Ale18] Alexa. “Top Sites in the United States.” http://www.alexa.com/topsites/countries/US, 2018.

[Ama18] Amazon. “Silk Web Browser.” https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html, 2018.

[An18] Daniel An. “Find out how you stack up to new industry benchmarks for mobile page speed.” https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/, 2018.

[APO19] APOWERSOFT. “Apowersoft Screen Recorder.” https://play.google.com/store/apps/details?id=com.apowersoft.screenrecord&hl=en_US, 2019.

[AXC20] Sotiris Apostolakis, Ziyang Xu, Greg Chan, Simone Campanoni, and David I. August. “Perspective: A Sensible Approach to Speculative Automatic Parallelization.” In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for*

Programming Languages and Operating Systems, ASPLOS '20, p. 351–367. Association for Computing Machinery, 2020.

[BBK00] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. "Integrating User-perceived Quality into Web Server Design." World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Networking, 2000.

[BKB00] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. "Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service." CHI, The Hague, The Netherlands, 2000. ACM.

[BLM] Jon Butler, Wei-Hsin Lee, Bryan McQuade, and Kenneth Mixter.

[BPT18] M Belshe, R Peon, and M Thomson. "HTTP/2.0 Draft Specifications." https://http2.github.io/, 2018.

[Bro17] Arnaud Brousseau. "Generating Web Pages in Parallel with Pagelets, the Building Blocks of Yelp.com." https://engineeringblog.yelp.com/2017/07/generating-web-pages-in-parallel-with-pagelets.html, 2017.

[bro20] "Browser Market Share Worldwide." https://gs.statcounter.com/browser-market-share, 2020.

[BTS17] Debopam Bhattacherjee, Muhammad Tirmazi, and Ankit Singla. "A Cloud-based Content Gathering Network." In *Proceedings of HotCloud*, 2017.

[BWW15] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha Madhyastha, and Vyas Sekar. "Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices." In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2015.

[CFM13] Calin Cascaval, Seth Fowler, Pablo Montesinos-Ortego, Wayne Piekarski, Mehrdad Reshadi, Behnam Robatmili, Michael Weber, and Vrajesh Bhavsar. "ZOOMM: A parallel web browser engine for multicore mobile devices." In *PPoPP*, 2013.

[chr20] "The 10 Best Chromium Browser Alternatives Better Than Chrome." https://www.makeuseof.com/tag/alternative-chromium-browsers/, 2020.

[Coo20] Cookiebot. “Google ending third-party cookies in Chrome.” https://www.cookiebot.com/en/google-third-party-cookies/, 2020.

[CS13] Cristian Cadar and Koushik Sen. “Symbolic Execution for Software Testing: Three Decades Later.” *Commun. ACM*, **56**(2):82–90, February 2013.

[CZH20] Moumena Chaqfeh, Yasir Zaki, Jacinta Hu, and Lakshmi Subramanian. “JSCleaner: De-Cluttering Mobile Webpages Through JavaScript Cleanup.” In *Proceedings of The Web Conference 2020*, WWW ’20. ACM, 2020.

[DB08] Leonardo De Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver.” In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS’08/ETAPS’08, p. 337–340. Springer-Verlag, 2008.

[dmo] “Directory of the web.” https://dmoz-odp.org/.

[DMR07] Erik D. Demaine, Shay Mozes, Benjamin Rossman, and Oren Weimann. “An Optimal Decomposition Algorithm for Tree Edit Distance.” In *Proceedings of the 34th International Conference on Automata, Languages and Programming*, ICALP’07, pp. 146–157, Berlin, Heidelberg, 2007. Springer-Verlag.

[Dru19] Drupal. “Drupal - Open Source CMS.” https://www.drupal.org/, 2019.

[DVB18a] Mallesham Dasari, Santiago Vargas, Arani Bhattacharya, Aruna Balasubramanian, Samir R. Das, and Michael Ferdman. “Impact of Device Performance on Mobile Internet QoE.” In *Proceedings of the Internet Measurement Conference 2018*, IMC ’18, New York, NY, USA, 2018. ACM.

[DVB18b] Mallesham Dasari, Santiago Vargas, Arani Bhattacharya, Aruna Balasubramanian, Samir R. Das, and Michael Ferdman. “Impact of Device Performance on Mobile Internet QoE.” In *IMC*, 2018.

[EGJ15] Jeffrey Erman, Vijay Gopalakrishnan, Rittwik Jana, and Kadangode K. Ramakrishnan. “Towards a SPDY’Ier Mobile Web?” *IEEE/ACM Trans. Netw.*, **23**(6):2010–2023, December 2015.

[EK19] Tammy Everts and Tim Kadlec. “WPO stats.” https://wpostats.com/, 2019.

[Eng19] Eric Enge. “MOBILE VS. DESKTOP USAGE IN 2019.” https://www.perficientdigital.com/insights/our-research/mobile-vs-desktop-usage-study, 2019.

[Eth16] Darrell Etherington. “Mobile internet use passes desktop for the first time, study finds.” https://techcrunch.com/2016/11/01/mobile-internet-use-passes-desktop-for-the-first-time-study-finds/, 2016.

[Fac19a] Facebook. “Prepack.” https://github.com/facebook/prepack, 2019.

[Fac19b] Facebook. “React: A JavaScript library for building user interfaces.” https://reactjs.org/, 2019.

[FFM04] Michael J. Freedman, Eric Freudenthal, and David Mazières. “Democratizing Content Publication with Coral.” NSDI. USENIX, 2004.

[GHM04] Dennis F. Galletta, Raymond Henry, Scott McCoy, and Peter Polak. “Web Site Delays: How Tolerant are Users?” Journal of the Association for Information Systems, 2004.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: Directed Automated Random Testing.” In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’05, p. 213–223. ACM, 2005.

[Gooa] Google. “Accelerated Mobile Pages Project – AMP.” https://www.ampproject.org/.

[Goob] Google. “Chromium.” https://www.chromium.org/Home.

[Gooc] Google. “Why Performance Matters?” https://developers.google.com/web/fundamentals/performance/why-performance-matters.

[Goo12] Google. “Speed Index - WebPagetest Documentation.” https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index, 2012.

[Goo19a] Google. “AngularJS: Superheroic JavaScript MVW Framework.” https://angularjs.org/, 2019.

[Goo19b] Google. “Brotli compression format.” https://github.com/google/brotli, 2019.

[Goo19c] Google. “Lighthouse.” https://developers.google.com/web/tools/lighthouse/, 2019.

[Goo19d] Google. “Progressive Web Apps.” https://developers.google.com/web/progressive-web-apps/, 2019.

[Goo19e] Google. “Service Workers: an Introduction.” https://developers.google.com/web/fundamentals/primers/service-workers/, 2019.

[Goo19f] Google Android. “Understanding Systrace.” https://source.android.com/devices/tech/debug/systrace, 2019.

[GYC18] Y. Geng, Y. Yang, and G. Cao. “Energy-Efficient Computation Offloading for Multicore-Based Mobile Devices.” In *IEEE Conference on Computer Communications*, INFOCOM, pp. 46–54, 2018.

[Hid] Ariya Hidayat. “Esprima.” http://esprima.org.

[HPJ16] Jialu Huang, Prakash Prabhu, Thomas B. Jablin, Soumyadeep Ghosh, Sotiris Apostolakis, Jae W. Lee, and David I. August. “Speculatively Exploiting Cross-Invocation Parallelism.” In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT ’16, p. 207–221, New York, NY, USA, 2016. Association for Computing Machinery.

[HTT20] HTTP Archive. “State of Javascript.” https://httparchive.org/reports/state-of-javascript, 2020.

[Inc18] Monsoon Solutions Inc. “Power Monitor Software.” http://msoon.github.io/powermonitor/, 2018.

[Iri19] Paul Irish. “pwmetrics: Progressive web metrics.” https://github.com/paulirish/pwmetrics, 2019.

[JBW19] Byungjin Jun, Fabian E. Bustamante, Sung Yoon Whang, and Zachary S. Bischof. “AMP up your Mobile Web Experience: Characterizing the Impact of Google’s Accelerated Mobile Project.” In *Proceedings of the 25th Annual International Conference on Mobile Computing and Networking*, MobiCom. ACM, 2019.

[Jia10] Changhao Jiang. “BigPipe: Pipelining web pages for high performance.” https://www.facebook.com/notes/facebook-engineering/bigpipe-pipelining-web-pages-for-high-performance/389414033919/, 2010.

[Kle98] Philip N. Klein. "Computing the Edit-Distance Between Unrooted Ordered Trees." In *Proceedings of the 6th Annual European Symposium on Algorithms*, ESA '98, pp. 91–102, London, UK, UK, 1998. Springer-Verlag.

[KML21] Ronny Ko, James Mickens, Blake Loring, and Ravi Netravali. "Oblique: Accelerating Page Loads Using Symbolic Execution." In *Proceedings of the 18th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2021. USENIX Association.

[KRB17] Conor Kelton, Jihoon Ryoo, Aruna Balasubramanian, and Samir R. Das. "Improving User Perceived Page Load Time Using Gaze." In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pp. 545–559. USENIX Association, 2017.

[KRN21] Nikhil Kansal, Murali Ramanujam, and Ravi Netravali. "Alohamora: Reviving HTTP/2 Push and Preload by Adapting Policies On the Fly." In *Proceedings of the 18th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2021. USENIX Association.

[LAG14] Guodong Li, Esben Andreasen, and Indradeep Ghosh. "SymJS: Automatic Symbolic Testing of JavaScript Web Applications." In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pp. 449–459, New York, NY, USA, 2014. ACM.

[LGT] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and title = Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation series = NDSS year = 2019 Joosen, Wouter.

[LMK17] Blake Loring, Duncan Mitchell, and Johannes Kinder. "ExpoSE: Practical Symbolic Execution of Standalone JavaScript." In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, SPIN 2017. ACM, 2017.

[LRS12] Dimitrios Lymberopoulos, Oriana Riva, Karin Strauss, Akshay Mittal, and Alexandros Ntoulas. "PocketWeb: Instant Web Browsing for Mobile Devices." In *Proceedings of* *the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII. ACM, 2012.

[Man18] Nitish Mantri. "How Websites Slow loading can Eat Your Revenue." https://dexecure.com/blog/business-web-performance-slow-website-eats-up-your-revenue/, 2018.

[MDN20] MDN. "Web Workers API." https://developer.mozilla.org/en-US/docs/Web/API/Worker, 2020.

[MEH10] James Mickens, Jeremy Elson, and Jon Howell. "Mugshot: Deterministic Capture and Replay for Javascript Applications." In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, p. 11. USENIX Association, 2010.

[MHS11] Mojtaba Mehrara, Po-Chun Hsu, Mehrzad Samadi, and Scott Mahlke. "Dynamic Parallelization of JavaScript Applications Using an Ultra-lightweight Speculation Mechanism." In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA, 2011.

[Mic12] James Mickens. "Rivet: Browser-Agnostic Remote Debugging for Web Applications." In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, USA, 2012. USENIX Association.

[Mic14] James Mickens. "Pivot: Fast, Synchronous Mashup Isolation Using Generator Chains." SP '14, p. 261–275. IEEE Computer Society, 2014.

[MLF13] Magnus Madsen, Benjamin Livshits, and Michael Fanning. "Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries." In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, p. 499–509. Association for Computing Machinery, 2013.

[MSN20] Shaghayegh Mardani, Mayank Singh, and Ravi Netravali. "Fawkes: Faster Mobile Page Loads via App-Inspired Static Templating." In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2020. USENIX Association.

[MTK12] Haohui Mai, Shuo Tang, Samuel T. King, Calin Cascaval, and Pablo Montesinos. "A Case for Parallelizing Web Pages." In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism*, HotPar. USENIX Association, 2012.

[NB16] Javad Nejati and Aruna Balasubramanian. "An In-depth Study of Mobile Browser Performance." In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pp. 1305–1315. International World Wide Web Conferences Steering Committee, 2016.

[NB19] James Newman and Fabian E Bustamante. "The Value of First Impressions: The Impact of Ad-Blocking on Web QoE"." In David Choffnes and Marinho Barcellos, editors, *Passive and Active Measurement - 20th International Conference, PAM 2019, Proceedings*, pp. 273–285, Germany, 1 2019. Springer Verlag.

[NGM16] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. "Polaris: Faster Page Loads Using Fine-grained Dependency Tracking." In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX, 2016.

[NKH16] Yeoul Na, Seon Wook Kim, and Youngsun Han. "JavaScript Parallelizing Compiler for Exploiting Parallelism from Data-Parallel HTML5 Applications." *ACM Trans. Archit. Code Optim.*, **12**(4):64:1–64:25, January 2016.

[NM18a] Ravi Netravali and James Mickens. "Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs." In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2018. USENIX Association.

[NM18b] Ravi Netravali and James Mickens. "Remote-Control Caching: Proxy-based URL Rewriting to Decrease Mobile Browsing Bandwidth." In *Proceedings of the 19th International Workshop on Mobile Computing Systems and Applications*, HotMobile '18. ACM, 2018.

[NM19] Ravi Netravali and James Mickens. "Reverb: Speculative Debugging for Web Applications.” In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC ’19, p. 428–440, New York, NY, USA, 2019. Association for Computing Machinery.

[NNM18] Ravi Netravali, Vikram Nathan, James Mickens, and Hari Balakrishnan. “Vesper: Measuring Time-to-Interactivity for Web Pages.” In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX, 2018.

[Nod19] NodeJS. “Express: Fast, unopinionated, minimalist web framework for Node.js.” https://expressjs.com/, 2019.

[NSM19] Ravi Netravali, Anirudh Sivaraman, James Mickens, and Hari Balakrishnan. “WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution.” In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys ’19, pp. 430–443. ACM, 2019.

[NSS10] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. “The Akamai Network: A Platform for High-performance Internet Applications.” *SIGOPS Oper. Syst. Rev.*, **44**(3):2–19, August 2010.

[NSW15] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. “Mahimahi: Accurate Record-and-Replay for HTTP.” Proceedings of ATC ’15. USENIX, 2015.

[Ope18a] Opera. “Opera Mini.” http://www.opera.com/mobile/mini, 2018.

[Ope18b] Opera. “Opera Turbo.” http://www.opera.com/turbo, 2018.

[Ope20] OpenSignal. “Pakistan: Mobile Network Experience Report, February 2020.” https://www.opensignal.com/reports/2020/02/pakistan/mobile-network-experience, 2020.

[Osm18] Addy Osmani. “The Cost of JavaScript.” https://medium.com/@addyosmani/the-cost-of-javascript-in-2018-7d8950fbb5d4, 2018.

[PA11] Mateusz Pawlik and Nikolaus Augsten. “RTED: A Robust Algorithm for the Tree Edit Distance.” *Proc. VLDB Endow.*, **5**(4):334–345, December 2011.

[PA15] Mateusz Pawlik and Nikolaus Augsten. “Efficient Computation of the Tree Edit Distance.” *ACM Trans. Database Syst.*, **40**(1):3:1–3:40, March 2015.

[PA18] Mateusz Pawlik and Nikolaus Augsten. "APTED algorithm for the Tree Edit Distance." https://github.com/DatabaseGroup/apted, 2018.

[par20] "Parallel.js." https://github.com/parallel-js/parallel.js, 2020.

[Pet19] Christo Petrov. "52 Mobile vs. Desktop Usage Statistics For 2019 [Mobile's Overtaking!]." https://techjury.net/stats-about/mobile-vs-desktop-usage/, 2019.

[Phi] Gavin Phillips. "Smartphones vs. Desktops: Why Is My Phone Slower Than My PC?" https://www.makeuseof.com/tag/smartphone-desktop-processor-differences/.

[PLR16] Joonyoung Park, Inho Lim, and Sukyoung Ryu. "Battles with False Positives in Static Analysis of JavaScript Web Applications in the Wild." In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, p. 61–70. ACM, 2016.

[PM96] Venkata N. Padmanabhan and Jeffrey C. Mogul. "Using Predictive Prefetching to Improve World Wide Web Latency." *SIGCOMM Comput. Commun. Rev.*, **26**(3):22–36, July 1996.

[Pyl19] Pylons Project. "Pylons Project." https://pylonsproject.org/, 2019.

[RAP13] Lenin Ravindranath, Sharad Agarwal, Jitendra Padhye, and Christopher Riederer. "Give in to Procrastination and Stop Prefetching." In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII. ACM, 2013.

[red] "Reddit." https://github.com/reddit-archive/reddit.

[red20] "10 Most Popular Phones in India in 2020 – Xiaomi and Samsung Rules." https://candytech.in/most-popular-phones-in-india/, 2020.

[Ric19] Leonard Richardson. "Beautiful Soup." https://www.crummy.com/software/BeautifulSoup/bs4/doc/, 2019.

[RNU17] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V. Madhyastha. "Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution." In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM. ACM, 2017.

[sel19] "SeleniumHQ Browser Automation." https://selenium.dev/, 2019.

[sha] "Overleaf: A web-based collaborative LaTeX editor." https://github.com/overleaf/overleaf.

[SJN17] Ahiwan Sivakumar, Chuan Jiang, Seong Nam, P.N. Shankaranarayanan, Vijay Gopalakrishnan, Sanjay Rao, Subhabrata Sen, Mithuna Thottethodi, and T.N. Vijaykumar. "Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks." In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, Mobicom. ACM, 2017.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A Concolic Unit Testing Engine for C." In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, p. 263–272. ACM, 2005.

[SMK15] Shailendra Singh, Harsha V. Madhyastha, Srikanth V. Krishnamurthy, and Ramesh Govindan. "FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers." In *Proceedings of the 21st Annual Conference on Mobile Computing and Networking*, MobiCom. ACM, 2015.

[sop20] "Same-origin Policy." https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy, 2020.

[SPG14] Ashiwan Sivakumar, Shankaranarayanan Puzhavakath Narayanan, Vijay Gopalakrishnan, Seungjoon Lee, Sanjay Rao, and Subhabrata Sen. "PARCEL: Proxy Assisted BRowsing in Cellular Networks for Energy and Latency Reduction." In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pp. 325–336, New York, NY, USA, 2014. ACM.

[Suz] Yusuke Suzuki. "Estraverse." https://github.com/estools/estraverse.

[SWS16] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. "SoK: (State of) The Art of War: Offensive Techniques in Binary

Analysis." In *IEEE Symposium on Security and Privacy*, 2016.

[Tal] Steve Tally. "Students prefer apps to the Web when using smartphones." https://www.purdue.edu/newsroom/releases/2012/Q3/students-prefer-apps-to-the-web-when-using-smartphones.html.

[thr20] "Threads.js." https://github.com/andywer/threads.js, 2020.

[VBN16] Matteo Varvello, Jeremy Blackburn, David Naylor, and Konstantina Papagiannaki. "EYEORG: A Platform For Crowdsourcing Web Quality Of Experience Measurements." In *Proceedings of the 12th Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16. ACM, 2016.

[Vol12] Joseph Volpe. "Nokia Xpress brings cloud-based compression to the Lumia line." Engadget. https://www.engadget.com/2012/10/03/nokia-xpress-brings-cloud-based-compression-to-the-lumia-line/, October 3, 2012.

[W3C18] W3C. "Preload." Editor's Draft. https://w3c.github.io/preload/, January 9, 2018.

[WBK13] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. "Demystifying Page Load Performance with WProf." In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2013.

[WBK14] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. "How Speedy is SPDY?" In *Proceedings of NSDI*, NSDI'14, pp. 387–399, Berkeley, CA, USA, 2014. USENIX Association.

[web17] "Web Workers." https://w3c.github.io/workers/, 2017.

[WKW16] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. "Speeding Up Web Page Loads with Shandian." In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2016.

[WLZ12] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. "How Far Can Client-only Solutions Go for Mobile Browser Speed?" In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12. ACM, 2012.

[Wor19] WordPress. “Blog Tool, Publishing Platform, and CMS – WordPress.” https://wordpress.org/, 2019.

[You19] Evan You. “Vue.js: The Progressive JavaScript Framework.” https://vuejs.org/, 2019.

[ZCP14] Yasir Zaki, Jay Chen, Thomas Pötsch, Talal Ahmad, and Lakshminarayanan Subramanian. “Dissecting Web Latency in Ghana.” In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC ’14, p. 241–248. Association for Computing Machinery, 2014.

[ZS89] K. Zhang and D. Shasha. “Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems.” *SIAM J. Comput.*, **18**(6):1245–1262, December 1989.

[ZWH18] Torsten Zimmermann, Benedikt Wolters, Oliver Hohlfeld, and Klaus Wehrle. “Is the Web ready for HTTP/2 Server Push?” In *Proceedings of the 14th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT. ACM, 2018.