Overcoming Barriers to Information Exchange on the Web

by

Ayush Goel

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2023

Doctoral Committee:

Adjunct Associate Professor Harsha V. Madhyastha, Co-Chair
Professor Atul Prakash, Co-Chair
Assistant Professor Xun Huan
Associate Professor Ryan Huang
Assistant Professor Ravi Netravali

Ayush Goel
goelayu@umich.edu
ORCID iD: 0000-0002-2343-670X

To my family and friends.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the guidance of my amazing mentors. First and foremost, I would like to thank my advisor Harsha V. Madhyastha, who is an excellent researcher and an even better mentor. I would not have been able to persevere through the hardships that come with a PhD if it were not for his guidance. He relentlessly supported me and taught me how to conduct principled research. I could not have asked for a better PhD advisor.

Ravi Netravali, my secondary advisor, significantly helped to shape my thesis by providing critical feedback during our meetings. He would always push the boundaries of what I perceived was feasible, which drastically improved the quality of my research.

My brilliant lab mates from office 4929 were always available to answer my technical and non-technical queries, which were, admittedly, numerous. Muhammed, Chris, Vaspol, Jingyuan, Yuan, Andrew, David and Joseph were a constant source of inspiration. Muhammed has been particularly helpful, be it debugging my R plotting scripts or navigating the various requirements of the CSE PhD degree. Jingyuan has rescued me multiple times right before paper deadlines by helping me run critical experiments. Also, the CSE graduate staff, specifically Steve, Jamie, Ashley and Jasmin made sure that my finances, travel logistics and any I.T. requirements were immediately met, allowing me to focus on my research. I would also like to thank my dissertation committee Atul Prakash, Ryan Huang and Xun Huan for their valuable feedback on this thesis.

I would not have made it this far in a foreign country without the love and support of my friends. Abhinav and Andrea were a big support system throughout school. Some of my most cherished memories in grad school were made during our countless trips, weekend hangouts, movie nights, and facetime calls. Also, my close friend Shubham from my undergraduate in Delhi, was always available to provide support from Chicago. His residing a mere 4 hour drive away from Ann Arbor meant that I have visited Chicago more than any other city in the country. Madhav, Richa, and Shardul made sure that I never missed home too much, by celebrating Indian festivals, hosting Bollywood movie nights, and getting Indian food together. Austin, ZZ, and Elizabeth made weekends so much more exciting with 4+ hour game nights, downtown excursions that ended at 4am or getting spicy chicken wings at Bdubs that left us bed ridden the following day.

An even longer list of friends have made my time at grad school so much more enjoyable: Stanley, Nihal, Megan, Micaela, Hanna, Petra, Adarsh, David, Dinku, Suha, Wilka, Rishi and many others. Thanks to the various communities I was a part that connected me to a wide variety of people: friends from Baker, the coop I lived in for 5 years with 20+ grad students, fellow dancers from my latin dance studio, board members of GRIN, the international student organization I was a part of for 4 years, and my beloved book club.

I would like to thank Kaitlin. She has seen it all – the stressful days leading up to deadlines, the lows that follow paper rejections, the doubts I've had in my abilities to do research and most importantly the sheer joy of having your paper accepted at a top conference. She has supported me during the difficult times even though she might be having a difficult time herself, and has been the first person to celebrate with for whenever the occasion called for it.

Lastly, I would like to thank my family. My parents, Sham and Prachi, are the reason behind every single accomplishment of my life. Their constant love and support has given me the unwavering strength to pursue every goal I've had and tackle any challenges that I've faced along the way. My baby sister, Tanya, took care of my parents and all the other responsibilities back at home, while navigating a very intense and competitive career, just so that I could focus on my studies and not worry about home. I am so very proud of all that she has achieved, and as a result inspiring me while I was on this grad school journey. My cousin Aru have been a constant source of joy. Our weekly video calls ensured that I was up to date on every family gossip from back home.

Finally, I would like to thank God for believing in me, and allowing me to embark on this journey. It was an extremely exciting, albeit challenging adventure and I can't wait to explore what's next in store for me.

PREFACE

Previously Published Material

Chapter 4 revises previous publications [111, 149]:

Ayush Goel, Vaspol Ruamviboonsuk, Ravi Netravali, Harsha V. Madhyastha. Rethinking Client-Side Caching for the Mobile Web. ***In HotMobile, Virtual, February 2021.***

Shaghaya Mardani, Ayush Goel, Harsha V. Madhyastha, Ravi Netravali. Horcrux: Automatic JavaScript Parallelism for Resource-Efficient Web Computation. ***In OSDI, Virtual, July 2021.***

Chapter 5 revises previous publication [113]

Ayush Goel, Jingyuan Zhu, Ravi Netravali, Harsha V. Madhyastha. Sprinter: Speeding up High-Fidelity Crawling of the Modern Web. ***In NSDI, Santa Clara, April 2024.***

Chapter 6 revises previous publication [112]

Ayush Goel, Jingyuan Zhu, Ravi Netravali, Harsha V. Madhyastha. Jawa: Web Archival in the Era of JavaScript. ***In OSDI, Carlsbad, July 2022.***

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

ABSTRACT

We are increasingly relying on the internet and specifically the world wide web (WWW) to exchange information and access services. Despite its ubiquitous use, there are two key barriers to accessing information that is shared on the web: 1) Many web pages suffer from poor performance with respect to both end-user loading latency and crawling throughput as observed by large-scale web crawlers. 2) Many web pages cease to exist over time causing a significant fraction of published information to no longer be available.

My dissertation addresses these issues by employing fine-grained data-flow and control-flow analysis of web computations, specifically JavaScript execution. Using this analysis, I am able to extract and modify JavaScript runtime behavior during web page loads and leverage this ability to build a number of web systems. First, I propose a client-side computation caching system that stores results of JavaScript (JS) execution to reduce compute delays and improve web page load times. I show that up to 85% of JavaScript runtime can be skipped by using such a computation cache. Second, I demonstrate that legacy JavaScript code has untapped potential for parallelization across multiple cores of modern smartphones to improve page load times. I show that 88% speedup in JS execution can be achieved by parallelizing execution on 8 cores of a given mobile device. Third, I built Sprinter, a distributed web crawler that crawls the web at 5 times the rate of traditional browser-based crawlers while preserving perfect fidelity. Sprinter accomplishes this by carefully selecting a subset of pages on any site to be crawled which it crawls using a browser, and caches the corresponding compute. It then performs browser-less crawling of the remaining pages on that site using those cached computations. Finally, I built Jawa, a web archival crawler that reduces the storage overhead of web archives by 41% while eliminating all fidelity issues. Jawa accomplishes this by exploiting the differences between live and archived pages, and accurately identifying and patching the sources of non-determinism that impair JavaScript execution on archived pages.

CHAPTER 1

Introduction

The unique ability of the human species to efficiently communicate and preserve knowledge has led to its significant advancement. For most of our existence, such discourse occured either through word of mouth or written text. In the modern era, however, much of this communication occurs on the internet. The internet has helped overcome the space-time barriers to information exchange by exponentially reducing the delay incurred by alternate means of communication.

A key component of the internet that enables such communications is the world wide web (WWW). A vast majority of the information and services that we consume today are exchanged over the web. Common examples include online shopping, e-banking and the exchange of published information in the form of research articles and digital journals. As of today, there are over 2B websites hosting around 6B webpages, and around 4.5B people have interacted with these pages to date [41]. Each day this number grows and our reliance on the web is only expected to increase with time.

1.1 Problems with the Modern Web

Despite decades of work to improve the web, it continues to be impeded by two major issues.

- **Poor Performance.** This manifests itself in two distinct ways. First, poor latency for web page loads, as observed by individual users loading pages on mobile devices such as smartphones. Despite accounting for more than half of global web traffic [97, 100, 182], mobile web performance in the wild continues to operate at a far slower pace than users are willing to tolerate [78, 82, 108]. Not only does this result in poor quality of experience for users and high web page abandonment rates [117], but also millions lost in revenues for major web service providers [101].

The second performance issue is poor crawling throughput as observed by large-scale web crawlers. These crawlers power a large number of modern services such as search engines and web archives and they enable web researchers to study the web. Further, recent smart-assistants and generative AI tools rely on data from the web to train various kinds of machine learning models. To keep up with the web's rate of the change in terms of the content on existing pages as well as additions of new pages, it is critical that web crawlers crawl pages at an extremely high throughput.

- **High Ephemerality**. The modern web notoriously suffers from link rot i.e., a page that existed at a particular URL ceases to exist at some point in the future. For example, millions of external links included in Wikipedia articles no longer work [36, 119]. The net result is that the work that authors of web pages put into identifying which external pages they should link to is going to waste over time. In turn, users end up missing out on the carefully selected context and pointers to related information/services that web publishers intended to provide them.

  To make external links on web pages resilient to link rot, the state-of-the-art solution is to capture a snapshot of every linked page when a link is created and to serve this page snapshot to users who may choose to visit this link in the future. For example, when a web publisher wishes to link to a page, they can ask web archives to crawl the page and store a copy. The publisher can then link to this stored page snapshot on the archive, which remains unmodified over time. There are around 150 web archives in the US alone, with Internet Archive being on of the biggest archives, storing 600B web pages, and other digital information like software, audio and video files [30].

  Despite such massive web archiving initiatives, most archives suffer from two fundamental issues. First, due to the consistently growing size of web pages, web archives must spend large amounts of money on their storage costs. Since most web archives are non-profit institutions, this increasing cost restricts the portion of the web they can archive. Resulting poor web coverage means that for a large number of pages on the live web, not one single archival copy exists. Second, a large number of pages when loaded from the archives suffer from poor fidelity. The final rendered page is often missing a number of critical resources such as images, or the content is improperly laid out due to various kinds of runtime errors.

1.2 Thesis Statement and Contributions

In this dissertation, I propose a number of optimizations that mitigate both the above issues of the web. In doing so, this dissertation supports the following **thesis statement** – *It is practical to speed up web page loads both for individual users and web crawlers and enable efficient web archiving without any loss in fidelity by leveraging runtime behaviors of web computations which can be accurately and efficiently extracted with the help of fine-grained program analysis.*

With this ability to understand and modify runtime behavior during web page loads in various contexts, I have built systems that reduce the page load times of web pages as perceived by end-users, improve crawling throughput of web crawlers and reduce storage overhead of web archives while simultaneously improving fidelity of archived page loads. The key contributions can be summarized as follows:

1. **Improve web page load times by reducing computation delays**. Over the last few decades, web pages have consistently increased in size. As a result, not only is there an increase in network utilization to download more page bytes, but also an increase in the amount of client-side computations performed while loading these pages. I studied both the cause of this delay and its net impact on the end-to-end latency of web page loads. Specifically, I found that despite eliminating all network delays, client-side computation alone degrades quality of experience beyond what users are willing to tolerate. Moreover, JavaScript execution is the key contributor to this client-side computation delay.

   To reduce the JavaScript execution overhead while preserving correctness I propose two separate optimizations. First, JavaScript memoization, where I envision augmenting the client browser with a computation cache that stores results of JavaScript executions. This cache can be used to identify and skip repeated executions across loads of the same page. This client-side cache is compatible with legacy web pages as it is maintained entirely on the client-side, requiring no support from web servers. My analysis across roughly 230 pages reveals that, even on a modern smartphone, such an approach could reduce client-side computation by a median of 49% on pages which are most in need of such optimizations.

   Second, parallelize JavaScript execution across different cores available on modern smartphones. As a first step towards this vision, I identify which functions can be safely parallelized depending on the state they access. I discovered that modern web pages are highly amenable to safely reaping parallelism speedups. For example, offloading computations across 8 cores of a flagship phone can speedup JavaScript execution

time by 88%. Multiple cores on smartphones are now common in both developed and emerging markets [**?**]

Investigating the potential benefits of both optimizations required building a custom JavaScript analysis framework. This framework takes JavaScript code as input and outputs 1) a modified JavaScript code and 2) a runtime library. The instrumented code enabled me to extract the relevant runtime behaviors in order to perform the above analysis. I describe this analysis engine in more detail in chapter 3.

2. **High-throughput high-fidelity crawling of the Modern Web.** Increasing client-side computations have a detrimental impact on the crawling throughput of web crawlers as well. Increasing JavaScript bytes on pages implies that the rendering process is increasingly relying on client-side execution of JavaScript in order to discover resources and add visual and functional components to the page. As a result, traditional static crawlers, devoid of any JavaScript interpretation capabilities, miss out on critical resources when crawling pages. Conversely, replacing static crawlers with browser-based crawlers results in exponentially higher compute overheads which yields a much lower crawling throughput in exchange for the accurate discovery of all resources required to render any page.

   To resolve this performance-fidelity trade-off, I built Sprinter, a new hybrid web crawler that efficiently combines browser-based and browserless (static) crawling to get the best of both. The key to Sprinter's design is my observation that crawling workloads typically include many pages from every site that is crawled and, unlike in traditional user-facing page loads, there is significant potential to reuse client-side computations across pages. Taking advantage of this property, Sprinter crawls a small, carefully chosen, subset of pages on each site using a browser, and then efficiently identifies and exploits opportunities to reuse the browser's computations on other pages. Sprinter was able to crawl a corpus of 50,000 pages 5x faster than browser-based crawling, while still closely matching a browser in the set of resources fetched.

3. **Reducing storage overhead and eliminating fidelity issues of web archives.** JavaScript on web pages does not only negatively impact its performance. It turns out that it is detrimental to web archival as well. Increasing JavaScript bytes per page implies an increasing storage cost of archiving each page. More importantly, the non-determinism introduced by JavaScript often causes archived pages to load with poor fidelity, i.e., runtime errors and missing resources. To address these problems, I built Jawa, a new design for web archives that significantly reduces the storage necessary to save modern web pages while also improving the fidelity with which archived pages

are served. Key to enabling Jawa's use at scale are my observations on a) the forms of non-determinism that impair the execution of JavaScript on archived pages, and b) the ways in which JavaScript's execution fundamentally differs between live web pages and their archived copies. On a corpus of 1 million archived pages, Jawa reduces overall storage needs by 41%, when compared to the techniques currently used by the Internet Archive.

1.3 Dissertation Plan

This dissertation is organized as follows: In chapter 2, I describe the page loading process of modern web pages and the different components involved. I explore the loading process in three different contexts: user-facing page loads, large-scale crawling by web crawlers, and storage and retrieval of web pages by web archives. In chapter 3, I describe the design and implementation of a custom JavaScript analysis engine that will be used in the remainder of the dissertation. In chapter 4, I describe the two optimizations to reduce computation delays for end-user page loads: client-side memoization and parallelization of JavaScript execution on mobile devices. In chapter 5, I describe the design of Sprinter, a web crawler which crawls pages at 5 times the rate of traditional browser-based crawlers while ensuring near perfect fidelity of crawls. In chapter 6, I describe the design of Jawa, a web crawler that significantly reduces the storage cost of web archiving while eliminating any JavaScript induces fidelity issues. I conclude the dissertation and discuss future directions in chapter 7.

CHAPTER 2

Background and Related Work

2.1 Background

I begin by describing the web page loading process in three different execution contexts. First, user-facing page loads. Second, web crawlers performing large-scale crawls of the web. Third, storing and loading archived web pages. For each of the context, I describe the performance metrics that are of relevance.

2.1.1 User-facing page loads

The most common form of access to information on the web is individual users loading pages using different devices such as smartphones, tablets, laptops etc. To visit any particular page, a user enters the page URL inside a web browser. This browser is responsible for fetching and rendering the page on the client device. Some of the most commonly used browsers are Google Chrome, Firefox and Microsoft Edge.

HTML Tree. The first step in loading a web page is to download the HTML file corresponding to the page URL. The HTML file is written in a tree-like syntax [26], which is parsed by the browser to create a document object model (DOM) tree. There is a 1-1 mapping between the HTML tree and the DOM tree. While parsing the HTML tree, the browser might encounter URLs to various embedded resources. Each of these resources are fetched over the network and processed according to their type.

JavaScript. Using `<script>` tags HTML files can include JavaScript code. JavaScript is a managed language which is supported by all modern web browsers, and used by page developers to create more dynamic and customized web pages. By default, JavaScript tags block the HTML parser. If the JavaScript code is not inline, then the browser must first fetch the resource file, incurring network latency. It then offloads control to the JavaScript

execution engine embedded inside the browser to parse, compile and execute JavaScript code. This code can access the HTML tree, while also initiating more resource fetches.

To sidestep the network latency incurred while fetching JavaScript resources, modern HTML syntax allows for special tags such as `async` or `defer` which allow for fetching JavaScript resources asynchronously.

Cascading stylesheets. A web page may use cascading stylesheets (CSS) to define the visual presentation of the HTML tree. These files are contained within the `<style>` tags. Similar to JavaScript code, these style files can also be embedded inline or contained inside separate stand-alone files. CSS files also follow a tree-like syntax where a style for a given node automatically gets applied to all the children nodes (unless otherwise specified) resulting in the "cascading" aspect of these stylesheets.

Other embedded resources. Other common media types on web pages include images, fonts, videos, and JSON-based text files. The browser fetches these resources asynchronously, so as to not negatively impact the page loading performance. Apart from these, the latest HTML syntax allows for over 1000 media types [35].

Frames. A single web page can consist of multiple frames. The very first HTML file fetched when a page is visited represents the top-level frame for that page. The top-level frame can embed nested frames using the `<iframe>` tag, where each frame is represented by a unique HTML file. Frames are used for isolating content from different domains, that might be rendered on a given page. Such isolations are critical for various security reasons [199].

Page load timeline. Once the browser completes the construction of the DOM tree (by fully parsing the top-level HTML file and all embedded resources), it fires a `domContentLoaded` event. While creating the DOM tree, the browser also creates a style tree from the CSS style rules to create a CSSOM tree (CSS object model tree). Once both the trees are created, the browser combines them to create a hybrid tree where every DOM node is annotated with the corresponding style. Using this hybrid tree and the screen dimensions of the client device, the browser creates a render tree, wherein it computes the location (x-/y-coordinates) of every visual element of the tree. Note that not all nodes of the tree are visual elements, for e.g, `<li>` is a visual DOM node, whereas `<head>` is not. The final step is to paint this render tree on the screen. Upon completion of this paint task, the browser fires the `onLoad` event indicating that the entire page load process has been completed.

Performance and correctness metrics. For end-user page loads, the key performance metric is the latency of page loads, i.e., *how long* does it take for the web page to complete loading? However, there is no fixed definition of what constitutes a complete page load. There exists a number of metrics that can be used to estimate when a page has finished loading. Some of the most commonly used metrics are page load time (PLT), speed index,

above-the-fold time, largest contentful paint etc. Each metric is determined by the firing of different events on the page. For example, PLT is measured as the time between the `onload` event and the start of the page load.

The correctness of the web page is defined with respect to two components: visual and functional. A correct visual component implies that the web page visually looks identical to what is expected by the web developer. Since modern web pages can be customized to different users, this correctness accounts for all the possible expected variations in the visual component. The functional component accounts for all the interactions on the page. A web page loads correctly from a function point of view if all the interactions on the page work as expected.

2.1.2 Web crawling

To make the most of the enormous trove of information available on the web, all of us today rely upon a range of efforts. Web search engines help users find pages relevant to their needs. Data from the web serves as input to smart assistants such as Siri and Alexa, and is used to train generative AI models that can answer user questions. Web archives store repeated snapshots of web pages to document changes over time and to preserve the content of deleted pages. Researchers continually study the web to help improve its performance and security.

All of the above is enabled with the help of scalable web crawlers. Web page loads initiated by web crawlers can be starkly different from the page loads initiated by individual users. How a web crawler crawls any given page is dictated by the information that the crawler wants to extract. For example, if a crawler simply wants to download all of the resources on a given page, then it would crawl the page by first downloading the page's HTML, and then recursively fetching all embedded links to images, CSS stylesheets, scripts, etc. In some cases, simply downloading the page's HTML suffices. Often times, crawlers want to interact with the page, for example trigger some buttons, menu icons on the page or take a screenshot of the final rendered page. In either case, the crawler has to load the page using a web browser or a tool that emulates a browser [57] in order to capture that information.

Similar to end-user page loads, there is no single event that indicates when crawling a given page has completed. If the crawler wants to capture a visual screenshot of the entire page, then the page is fully crawled only after the `onLoad` event is fired and the page has finished rendering. However, if the crawler wants to capture post page load interactions, then the crawl finishes once the crawler is done triggering the relevant interactions on the page, which in turn needs to wait for the page to finish loading.

Performance and correctness metrics. Large-scale web crawlers do not care about the

latency of individual page loads, as they are often crawling multiple pages in parallel [80, 118, 122, 148, 194]. Instead, the key performance metric for crawlers is the overall throughput which is defined as the number of pages crawled per second.

Correctness of the crawling task is defined with respect to the output. For example, if the crawler only cares about taking screenshots of the final rendered page, then that screenshot needs to be identical to what one would get while crawling the page using a browser.

2.1.3 Web archival

URLs are brittle pointers to information on the web. Over time, a page may cease to exist at the URL where it was originally available [138, 200] or the content available at that URL might change due to the page being modified [172, 105].

Therefore, web archives play a key role in the web ecosystem, enabling users to lookup the content that existed at any particular URL at various times in the past. Web archives are used for a wide variety of use cases, such as web-data analytics, genealogical analysis, and even as legal evidence [129]. To support these uses, a number of organizations—cultural heritage institutions, national libraries, and public museums—operate web archives to ensure long-term preservation of content on the web. A recent survey estimates that there are 119 web archives in the United States alone [104].

The largest and most popular of these archives, Internet Archive (IA), has archived over 600B web pages to date, storing data in excess of 100 petabytes [31]. It repeatedly crawls web pages over time and saves many snapshots of every page. For every page snapshot, IA first downloads all resources (e.g., HTMLs, CSS stylesheets, JavaScripts, images) on the page). It stores these resources after rewriting all URL references to point to the copy hosted by the archive. When a user wants to later view any stored snapshot of a page, the user's browser loads the snapshot from IA in the same manner as it would load any page on the live web. Other archives [44, 39] only store the screen capture of the final rendered page as an image or PDF file.

Performance and correctness metric. Web archives care about a number of performance metrics. Since these archives rely on web crawlers to capture and store web pages, high crawling throughput is critical to efficiently keep up with the growth rate of the web. Post crawling, it is important that the storage cost incurred to preserve these pages is low. This helps to store a larger number of pages and to achieve a better coverage of the web. Also, most of these archives are run by non-profit institutions and therefore a lower storage cost allows them to stay afloat for longer. Finally, spending all of these resources on capturing the web is of little use if this information cannot be accessed accurately. Archived pages

are expected to mimic the live page in most visual and functional aspects. This property is referred to as the fidelity of archived pages and one of the primary concerns of web archives.

2.2 Related Work

2.2.1 Faster web page loads

Prior work on faster web page loads either optimizes for lower compute overhead, network delay, or both.

Reducing web computation overheads. Prior measurement studies have analyzed the performance of mobile web browsers [207, 160**?** , 177]. They have found browser computations to be a primary contributor to high page load times. In response to these studies, three separate lines of work have been done to mitigate these 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 [115, 132]. This requires web developers to rewrite a given web page respecting this new specification.

Second, some systems [71, 210, 175, 89] 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 [196]. Worse, by relying on (often third-party) proxy servers, these systems violate HTTPS' end-to-end security guarantees [170]; clients must trust proxies to preserve the integrity of their HTTPS objects, and also must share private Cookies to accelerate personalized page content.

Third, systems like Prophecy [165] enable servers to return post-processed page files that elide intermediate computations. However, content alterations with these systems may break page functionality [65], particularly for pages that adapt execution based on client-side state that servers are unaware of, e.g., localStorage.

Parallelization efforts. ParaScript [154] and others [159] 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). Zoomm [87] and Adrenaline [147] leave JavaScript execution unchanged, and instead parallelize tasks such as CSS rule parsing. Lastly, several libraries [61, 64] aid developers in writing parallel JavaScript code by abstracting inter-worker messaging. However, developers are responsible for identifying and enforcing (safe) parallelism decisions.

Network optimizations for the web. Systems such as Alohamora [133], Vroom [187],

and others [99, 208] 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 [150] develops static HTML templates that can be rendered while dynamic data is fetched. Polaris [164] and Klotski [86] reorder network requests to minimize the number of effective round trips while respecting inter-object dependencies. Cloud browsers [197, 170, 163] shift network round trips to wired proxy server links. Content delivery networks [173, 106] serve popular objects from proxy servers that are geographically close to clients, while compression proxies [65, 195, 176] selectively compress objects in-flight between servers and clients. Lastly, a handful of systems prefetch content according to predicted user browsing behavior [179, 146, 213].

Concolic execution for web optimization. Oblique is a third-party web accelerator which enables secure outsourcing of page analysis. Oblique symbolically executes the client-side of a page load, generating a prefetch list of symbolic URLs. Each symbolic URL describes a URL that a client browser should fetch, given user-specific values for cookies, the User-Agent string, and other sensitive variables. Those sensitive values are never revealed to Oblique's analysis server. Instead, during a real page load, the user's browser concretizes URLs by reading sensitive local state; the browser can then prefetch the associated objects. Experiments involving real sites demonstrate that Oblique preserves TLS integrity while providing faster page loads than state-of-the-art accelerators. For popular sites, Oblique is also financially cheaper in terms of VM costs.

Compute memoization. Memoization is widely used across different kinds of application. Prior work has leveraged memoization techniques to reduce compile-time latency [201, 130] , improve runtime performance [111, 204], minimize scheduling overheads [92], and enable faster auditing of web applications [135].

2.2.2 Web crawling

Scalable web crawling. The engineering issues associated with web crawling are well studied [122, 216, 66, 139, 79, 80]. Some of these crawlers [139, 80, 79] are able to achieve a crawling throughput of upwards of 1000 pages per server. Lee et al [139] designed IRLbot which was able to crawl 1800 pages per second on a single server, crawling over 6 billion pages in 41 days. UbiCrawler [79] is a scalable distributed crawler that is platform-agnostic and supports features such as graceful degradation in the presence of faults. BUbiNG [80] claims to be the first open-source crawling software which scales linearly with the number of servers achieving throughput of 1000 pages per second per server.

All of these crawlers focus on efficient URL discovery and how to minimize the number of

spam pages crawled. Also all of these crawlers only download the HTML file for every page URL.

Web crawling algorithms A large amount of prior work has explored the best algorithm to discover new URLs to crawl. The earliest work used breadth first [184] and depth first [94] search to discover new URLs. More advanced algorithms used backlinks-based context graphs to estimate the likelyhood of a page leading to a relevant page, even if the source page itself is not relevant. [155] One crawler explored using lexical knowledge, specifically hierarchical topic classifier to select links for crawling [88].

Incremental crawling A large amount of prior work [96, 148, 90, 203] has focused on incremental web crawling, i.e., how to efficiently recrawl pages. These crawlers aim increase the overall freshness of the crawled dataset, i.e., reduce the likelyhood of crawling the same page twice when there is little to no change in the page content between the two crawls. These crawlers used various different models to measure the change in page content over time, and often rely on a heuristic based threshold to identify changes that qualify as significant enough change in the page content.

Resource bottlenecks of large-scale distributed systems. Prior work has studied the bottlenecks in scaling various distributed data processing workloads such as sorting [186], data analytics [178], and distributed deep learning [206, 192, 68]. These efforts first identify the hardware resource (CPU, GPU, network, or disk) that constrains overall performance, and then propose solutions to optimize the utilization of that resource.

2.2.3 Web archiving

Impact of JavaScript on web crawlers. Prior work has shown that it is important for web crawlers to execute JavaScript when crawling pages, both in the context of web archives [83, 84, 85] and web search engines [1], else many important resources on a page will often go uncrawled. Our work highlights that, due to the non-deterministic execution of JavaScripts, archived pages often have poor fidelity even when pages are crawled using a browser which executes all scripts on every page.

Beyond executing JavaScripts while crawling a page, systems like Conifer [18] also save all resources on the page that are fetched while the user is interacting with the page. However, such systems are designed for private web archival, i.e., a user saves a page and its constituent resources for the user's own personal use later. If users load a page archived by a different user using a different device/browser, they will face the same fidelity issues seen on the Internet Archive.

Coverage of web archives. Many measurement studies [67, 70] have demonstrated

that web archives are far from comprehensive in archiving all pages on the web. Prior work [151, 134] has attempted to address the incompleteness caused due to large portions of the web not being openly available (e.g., behind paywalls) and requiring user logins (e.g., social media). In contrast, I seek to enable web archives to improve their coverage by reducing the costs associated with archiving any corpus of pages; thereby, for the same budget, a web archive can crawl and save more pages.

Supporting bulk processing of archives. Web archives are often used by researchers to perform large scale analyses of historical information. Xinyue et al. [211] demonstrate the performance penalties of the WARC format for such batch processing workloads, and many systems [144, 5, 125] have been developed to enable programmatic analysis of large corpuses without needing to access each individual resource on every page.

Archives as data source. A large amount of prior work has studied the efficacy of web archives with respect to their utility in specific tasks. For example, Gomes et al [114] studied how useful web archives is for humanities scholars, specifically historical researchers. Their work explores a number of use cases that illustrate how web-archived information can support future historical research and discuss a number of pre-existing tools that can facilitate this research. Other work [124] has studied a more general scholarly use of web archives. The authors have defined key characteristics of scholarly use of digital sources and translated those into a set of key requirements for web archives. In conclusion, they show how current web archives fail to meet the evolving scholarly requirements, necessitating new web archival methods.

JavaScript record and replay systems. A number of prior systems [72, 158, 190] enable users to record and replay JavaScript execution, both in the context of browsers [72] and independent JavaScript programs [190]. These record and replay tools are critical for debugging JavaScript based errors. Therefore, to ensure high fidelity replay, all of these systems identify and patch all sources of non-determinism to match the recorded version.

Code reachable through event handlers. JavaScript testing tools automate the process of testing by dynamically constructing test cases to achieve maximum code coverage. A key part of this process is identifying all code that can be potentially executed by event handlers. Doing so requires heavyweight symbolic execution analysis [136], or exhaustively going through all possible orders and inputs [74].

Program analysis on the web. JavaScript on the web has been notorious for various kinds of security, privacy and performance issues. A large body of prior work focuses on addressing such issues by relying on sophisticated program analysis techniques [205, 214]. Such techniques, however, incur a high computation cost. This is why, in solutions for optimizing web performance [136, 164, 149] which use computationally expensive JavaScript

analysis techniques, web servers perform such analysis in the background to mitigate the impact of their overheads. For archival systems, even if crawled JavaScript resources are processed offline, the cost for computationally heavyweight processing is not sustainable.

Dead code elimination on the web. One way to reduce the storage cost of web pages is to eliminate dead code (i.e., code that is never reachable) from resources such as JavaScript and CSS. Tools [102, 52] which do so using static analysis are widely used.

CHAPTER 3

JavaScript Analysis Engine

To overcome the various performance and archiving issues of the modern web, I turned to a fine-grained understanding of the client-side computations that are incurred during web page loads. For example, to reduce client-side computation delays by eliminating redundant computations, one needs to identify which computations act on the same input and produce the same output. Modern browsers such as Chrome and Firefox, provide a very high-level information about client-side computations incurred during the page loading process. For example, Chrome provides a breakdown of the total client-side computation with respect to the various computation tasks such as parsing, compiling, JavaScript execution, painting and rendering. If one wants to dig deeper, Chrome's JavaScript profiler can provide runtime information about the various JavaScript functions executed during the page load. However, both pieces of this information are available only after the page has finished loading. There is no web API that would allow a web developer to inspect different properties of the runtime *during the page load itself.*

To overcome these shortcomings of the modern browsers and their managed runtime APIs, I built a custom JavaScript analysis engine. This engine enables me to extract various different properties about JavaScript execution during and after page load, for example, what variables are declared in the global JavaScript scope, and their values at the end of the page load. In the rest of this chapter, I describe the main purpose of such an engine, and our implementation methodology.

3.1 Objective

The high-level objective of the analysis engine is to identify and extract data-flow and control-flow information of JavaScript executions performed during web page loads. Data-flow information consists of all the program state that was accessed, specifically the variables read

or written, the corresponding values at the time of the access, and a reference (or the underlying address) for that variable. Control-flow information, on the other hand, accounts for which parts of the program were executed, and the order of execution. For example, which functions were executed, and which code branches were taken within those functions. Both control-flow and data-flow information can be tracked at various different granularities. For example, a finer granularity would involve tracking data-flow and control-flow at an instruction level, i.e., which state was accessed by any given instruction, and what other instructions preceded and succeeded the execution of the given instruction. A coarser granularity can track the same information at a JavaScript file level.

With the help of such information about JavaScript runtimes, I was able to identify various optimization opportunities. For example, having access to the state read (inputs) and written (outputs) by every JavaScript function on the page, I was able to identify the potential benefits of computation memoization. When the same JavaScript function is invoked multiple times with the same inputs, one can store the corresponding outputs, so that for each future invocation, the execution of the function can be skipped by simply reusing the previously generated outputs. Skipping JavaScript execution in this manner will reduce the total computation while preserving all correctness properties of the execution, i.e., the execution proceeds in a manner identical to how it would without any memoization.

3.2 Implementation

3.2.1 Methodology

To extract runtime properties of JavaScript execution within a browser, I have to inject instrumentation code since there is no browser API that already provides such a functionality. Instrumentations can be applied at two different places. First, since most browsers' source code is openly available, one could modify the browser code to perform such tracking. For example, in the case of Chrome, V8, which is Chrome's JavaScript interpreter, could be modified to track different runtime behaviors. Second, instrumentation code could be injected directly into the JavaScript code itself to perform application level tracking. This is made feasible by the flexibility of the JavaScript language which supports prototype patching and overriding in-built APIs with the help of dynamic shims [121].

Both in-browser and in-language instrumentations have been widely used by prior work [164, 165, 169] to extract different kinds of runtime behaviors about the browsers in general. Both approaches provide different trade-offs. In-browser instrumentation is preferable from a stealth perspective as the runtime code has no means of detecting the instrumented

code. In-browser code is also less likely to inflate the execution time of the instrumented code since it is written in a lower-level language (C or Rust) which has better performance than dynamic languages such as JavaScript. Conversely, in-language code is easier to integrate and modify over time. This is because modern browser code, albeit open sourced, is extremely complex with millions of files and 1000s of dependencies. Even small changes in the code can cause compilation or correctness issues. For example, it took me multiple days to identify the exact portions of the Chrome's source code that had to be modified in order to increase the size of the client-side local storage [2]. For similar reasons, I chose to instrument the language code instead of the browser code.

3.2.2 Analysis framework

The JavaScript analysis framework consists of two parts, a static and a dynamic analyzer. The static analyzer inspects the JavaScript code, identifies *where* and *what* instrumentation code to inject. This analysis is done prior to the JavaScript file being compiled and executed by the browser. It can happen either online, i.e., during the page load process itself by intercepting the JavaScript code enroute to the browser using a man-in-the-middle proxy. Or it can happen offline, either on the web server side, where the server has instrumented the file prior to the client's request for it, or on the client-side where the file might be locally stored in the browser's cache.

The dynamic analyzer is essentially a runtime library that is executed alongside the instrumented code to extract and store the various different runtime properties during the page load process. This library is also injected in the page source code itself (usually at the beginning of the HTML file).

3.2.2.1 Static analysis

The static analysis is written as a NodeJS framework. NodeJS offers support in the form of a large number of third-party libraries that make analyzing and rewriting JavaScript code much easier. I used the Esprima library [123] to parse JavaScript code and create an abstract syntax tree (AST). This AST is used to identify points in the code where relevant code can be injected, for example, function boundaries, function calls, branch statements and more.

The AST is also traversed to identify scopes of the different JavaScript variables used. JavaScript supports multiple variable scopes. Global scope contains all the variables defined outside any function, whereas local scope contains all the variables defined inside a function. JavaScript runtime also supports a closure scope, which is scope inside a nested function, which persists even after the immediately enclosing function has finished execution. For

file.js

```
function foo(){
   a = b.c;
}
```

file-inst.js

```
function foo(){
   b = newProxy(b,fn);
   a = newProxy(a,fn);
   a = b.c; // b = {c:4}
}

fn = function(){
    read: function(obj){log(obj,"read")},
    write: function(obj){log(obj,"write")}
}

log = [["b.c", "read","4"],["a","write","4"]]
```

Figure 3.1: **Input and output of the JavaScript analysis engine**

example, if function B is nested inside function A, then the local scope inside function B can be accessed by the code inside function A even after function B is done executing. My static analyzer also maintains a list of all inbuilt JavaScript APIs so as to not mis-classify them as global variables. For example, the API *Array* is used to create array objects. From the perspective of the AST scope analysis, *Array* can be like any other global variable because it is not declared inside any scope. To handle such cases, my static analyzer exhaustively enumerates all inbuilt JavaScript APIs.

Once all the variables in a given JavaScript code are grouped in their respective scopes, the analysis framework inserts code around their accesses. I use JavaScript proxy objects to track reads and writes to any given variable. JS proxies also allow me to define what value is returned when the underlying object is accessed. Using this property, I can easily implement aliasing. For example, $a = b.c$ is rewritten as $b = newProxy(b, fn); a = newProxy(a, fn); a = b.c$. a and b are replace with their proxy counterparts. Moving forward, anytime a or b are accessed, my custom function fn gets invoked which logs their corresponding values. Also, fn wraps the return object in a proxy object itself, so that all future accesses of the returned object are tracked. Therefore, when $b.c$ is read, the returned value is wrapped in a proxy object that is assigned to a. Any future access to $b.c$ using a as the alias will be automatically tracked. Figure 3.1 contains a listing showing how my static analyzer rewrites a given JavaScript function.

3.2.2.2 Dynamic analysis

JavaScript is a dynamically typed language and therefore static analysis can only assist so much in extracting runtime properties. Certain JavaScript APIs such as *eval* or *document.write* can cause the runtime to compile and execute new JavaScript code. For example, `eval(a)` will cause the JavaScript runtime to evaluate (i.e., compile and execute) the string represented by the variable a. Identifying the value of a just with static analysis is infeasible.

To remedy this, I inject a small runtime library that interacts with the rest of the JavaScript code at execution time to accurately extract different runtime behaviors. In the rest of this section, I describe some of the key characteristics of my dynamic analyzer.

Reference management. Being a dynamically typed language, JavaScript doesn't provide support for pointers. Therefore, there is no way to get the underlying address of any given variable. This can cause issues with accurately identify data-flow information for JavaScript execution. Let's understand this with the help of an example. In JavaScript, all non-primitive (non String, Numeric, Boolean data types) objects are assigned by reference. So when variable a, which refers to an array ($a = [0, 1, 2, 3]$), is assigned to another variable b using $b = a$, then the JavaScript runtime creates a shallow copy, i.e., both a and b refer to the same object, where b is an alias to a. Modifying b using something like $b.push(4)$, will cause a to have the new value $[0, 1, 2, 3, 4]$ as well. Now, consider two functions A and B which write to the variables a and b respectively. Since the JavaScript runtime doesn't provide any means to get the address of the underlying object (with the help of a pointer), my analysis would log two separate variables being written by the two functions. This can have negative ramifications for certain kinds of analysis, for example, dependency analysis where one needs to identify functions with write-write dependencies, since A and B would be falsely annotated to not have any dependencies.

To overcome this limitation, my dynamic analyzer maintains a custom address for each object allocated on the JavaScript heap. In order to do this, it maintains a key-value map, where the keys represent the objects and the values respect a unique address for that object. I simply assign these addresses using a monotonically increasing sequence number, starting at 0. The keys to this map are unique, i.e., two variables pointing to the same underlying object will refer to the same value in this map. So in the above example, when variable a is accessed the first time, it gets added to the key-value map, and gets assigned a new address. Later when variable b is accessed, upon performing a lookup, my analyzer identifies that this object was already assigned an address, and therefore returns the same address as that of a. This helps identifying when same objects are accessed using different aliases.

Serialization Any information extracted during the JavaScript execution needs to be migrated outside of the browser context and stored on disk for future post-hoc analysis. To do this, all the tracked information needs to be serialized, i.e., converted to a string format (referred to as object pickling in some languages). JavaScript provides an in-built API specifically for this purpose `JSON.stringify` which converts any given object into a JSON compatible string. This string can be later parsed using `JSON.parse` to create a JSON object from a given string.

Turns out that this JavaScript's serializer lacks a number of key features necessary to implement efficient and accurate serialization for the purpose of fine-grained analysis. It is unable to serialize a number of JavaScript objects such as classes, functions, specific data types such as Date, RegEx, SharedArrays etc. When any of these objects are passed to the `JSON.stringify` function, it returns "{}" an empty object as the output. It also throws an error while trying to serialize circular objects, i.e., objects that contain a reference to itself. Moreover, it doesn't preserve any reference information during serialization, i.e., all object references are destroyed during serialization and parsing. These are only some of the most notable limitations of the default serializer.

To tackle these limitations, I built a custom serializer on top of the default serializer. This serializer, which I refer to as the *omniSerializer*, is capable of serializing most JavaScript objects such as functions, classes and other in-built data types. For example, to serialize a RegExp object types, I simply extract the underlying regular expression. So when serializing a variable a, such that $a = newRegExp(".*")$, the serialized output looks something like "{type: RegExp, source: '.*'}". It also embeds reference information during serialization, and as a result can directly handle circular objects. I also built a custom parser, similar to `JSON.parse` so as to be able parse any object stringified using my custom serializer.

Additional properties. Thanks to the flexibility offered by the JavaScript runtime in modern browsers, apart from *extracting* runtime behavior, I were also able to *control and modify* it. The browser API specifications [152] allow for modification of most of the APIs, barring a specific few, for e.g, `window.document` or `document.location`. This means that my dynamic analyzer can overwrite most of the in-built APIs to control their behavior. For example, I can overwrite the `Date` API to always return the same date value, regardless of when it was invoked, so as to eliminate the resultant non-determinism. Similar implementations can be used to modify client-characteristic values such as user-agent, width and height of the client device etc.

Using a runtime library, I can also trigger various kinds of interaction with the page. For example, identifying all the existing event handlers for the corresponding events (user interactions) implemented by the page, and triggering them automatically. Scrolling web

pages to discover below-the-fold content, and navigating to different parts of the page or to a different website by interacting with different components of the page.

These JavaScript analysis capabilities enabled by the analysis engine play a critical role in all of the optimizations I describe in the remainder of this dissertation.

CHAPTER 4

Making Page Loads Faster By Reducing Compute Delays

This chapter focuses on improving web page performance from the perspective of individual users loading pages on mobile device. I begin by studying the impact of client-side computations on the end to end page load times. I discover that JavaScript execution disproportionately contributes to the total client-side computations. I then propose two separate techniques to reduce this JavaScript overhead – execution memoization and execution parallelization across multiple cores of mobile devices. As a first step towards both these optimizations, in this dissertation, I estimate the potential for such techniques and the upper-bound in execution time speedups that can be achieved by them.

4.1 Rethinking Client-side Caching for the Mobile Web

Recent years have witnessed significant growth in the amount of web traffic generated through mobile browsing [42]. Unfortunately, mobile web performance has not kept up with this rapid rise in popularity. Mobile page loads in the wild are often much slower than what users can tolerate [107], with many pages requiring more than 7 seconds to fully render [23].

A key contributor to slow mobile page loads is client-side computation—in particular, JavaScript execution—as seen in my measurements (§4.2.1) and in prior studies [161, 187]. Given the importance of fast page loads for both user satisfaction [117] and content provider revenue [3], much effort has been expended to alleviate this bottleneck by reducing the work that mobile devices must do to load pages. However, despite their promising results, existing solutions have (fundamental) practical drawbacks that have hindered adoption.

- Offloading computation tasks in page loads to well-provisioned proxy servers, which ship back computation *results* for clients to apply locally [71, 175, 210], poses numerous security and scalability challenges. Clients must trust proxies to preserve the integrity of HTTPS objects, and they must share (potentially private) HTTP Cookies with proxies in order to support personalization. In addition, proxies require non-trivial amounts of resources to support large numbers of mobile clients [196].
- Having origin web servers return post-processed versions of their pages which elide intermediate computations [165] results in fragile content alternations that may break page functionality [65]. For example, pages may adapt execution based on client-side state (e.g., localStorage); servers are inherently unaware of this state while generating post-processed pages, and thus risk violating page correctness. Moreover, like proxy-based solutions, this approach places undue burden on web servers to generate optimized versions for the large number of pages they serve (including versions personalized to individual users).

We argue that the key to easing deployability is to shift the focus to solutions which only require client-side changes. Doing so sidesteps the security, privacy, and correctness concerns discussed above. Moreover, only a handful of browsers need to be updated for most users to benefit [9].

As a first step towards this vision, in this paper, I ask: *how much web computation can be eliminated by a purely client-side solution?* To answer this question, I propose a rethink of the functionality of client browser caches. While client-side caching has been a staple optimization in page loads for decades, browsers have used their caches only to eliminate *network fetches*; recent caching proposals for improved hit rates share the same focus [167, 209]. In contrast, I propose that browser caches be extended to enable *reuse of computations* from prior page loads.

The idea of computation reuse, commonly known as computation memoization, dates all the way back to late 1960s [156] when the idea of a function *"remembering"* results corresponding to any set of specific inputs was first introduced. Memoization has found wide applicability in language compilers [171, 201, 202] as well as other domains such as image search [128], image rendering [81] and data center computing [120, 77, 140]. In this paper, I study the potential of such an approach in the context of web page loads and make contributions along the following dimensions:

1. **Granularity.** The granularity at which computation is cached can have a significant impact on potential benefits, and must be amenable to the fact that cache entries may be from page loads performed several minutes or even hours ago. For example, we find that 11% of JavaScript code on the landing page of the median Alexa top 500 site changes each hour, thereby precluding computation reuse at a page level [165, 175].

Instead, we propose finer-grained computation caching at the granularity of JavaScript functions. Our proposal is rooted in my finding that 96% of JavaScript code is housed inside JavaScript functions on the median page.

2. **Efficacy.** To measure the potential benefits of my proposal, I developed an automated JavaScript tracing tool that dynamically tracks all accesses to page state made by each function invocation in a page load; this information is required to determine the reusability of computation from prior page loads. I experiment with a state-of-the-art phone (Google Pixel 2) and over 230 pages (landing pages of top Alexa sites and random sites from DMOZ [20]). For the subset of these pages which require clients to perform over 3 seconds of computation ("high-compute pages"), I estimate that client-side reuse of JavaScript executions can eliminate 49% of client-side computation on the median page.
3. **Practicality.** Finally, I sketch the design of a browser-based system that performs computation caching. I outline the practical challenges of such a system, which largely revolve around the high state tracking and cache management overheads. To alleviate those overheads without sacrificing substantial reuse opportunities, my key finding is that 80% of total JavaScript execution time is accounted for by 27% of functions on the median high-compute page. This allows for the system to target only a small fraction of functions while reaping most of the potential computation caching benefits.

4.1.1 Motivation

I begin by presenting a range of measurements to illustrate the large (negative) impact that client-side computation has on overall page load times (PLTs). Our experiments use a modern smartphone (Google Pixel 2[1]) with Google Chrome (v73), and consider landing pages from the Alexa top 1000 sites; these pages are more likely to incorporate recommended best practices for enabling fast page loads.

Mobile web page loads often have very high compute. I record each page and then load it within the Mahimahi replay environment [163] over an emulated 4G network [37]; emulation was done using Chrome's network shaping feature. I focus my analysis on the 223 pages which experienced PLTs greater than 3 seconds (the *"Shaped, all cores"* line in Figure 4.1), since these loads are slower than user tolerance levels [107].

Since page loads consist of fetching resources over the network *and* processing those resources to display functional content, the observed load times could be high due to network

[1]We believe a more recent version of Google Pixel (e.g., Pixel 4) would show some, albeit limited, improvements in the total client-side computation time due to a slightly higher CPU clock speed [93].

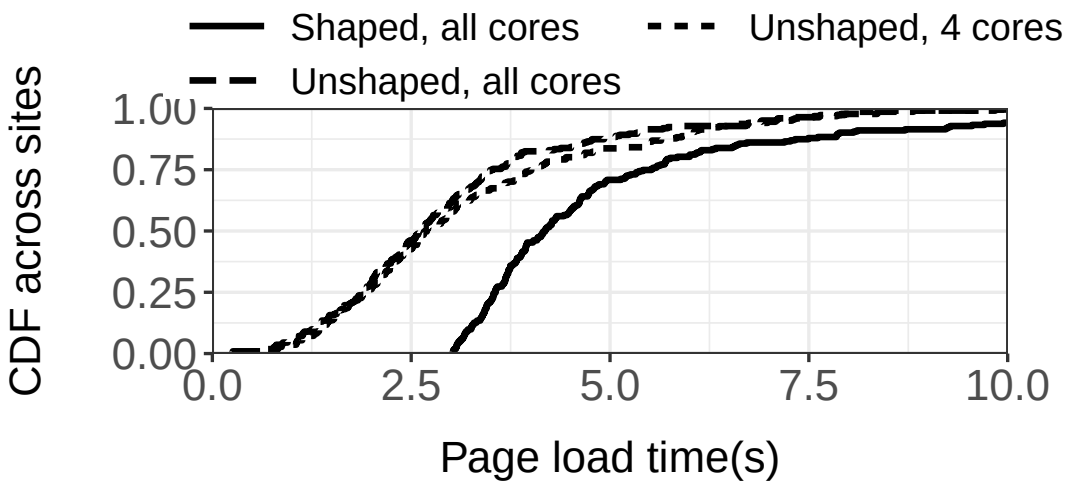


Figure 4.1: **Page load times, with or without network delays (shaped vs. unshaped) and when using all 8 CPU cores or only 4 of them. Results are for 223 landing pages with load times over 3 seconds with a shaped network.**

or computational delays. To distinguish between these two factors, I loaded the 223 slow landing pages again using Mahimahi, but this time with an unshaped network. I note that this represents the best case performance for prior (complementary) web optimizations that target network delays [197, 164, 86, 212]. As shown in the *"Unshaped, all cores"* line of Figure 4.1, 39% of these pages (i.e., 86 pages) continue to experience load times greater than 3 seconds, despite the lack of network delays; I call these "high-compute" pages. While client-side computation may not always be the primary bottleneck when these pages are loaded in the wild (i.e., network fetches may constitute the critical path), these results show that compute delays alone would slow down many web pages beyond user tolerance levels.

Our findings, while in line with recent work [187, 161], are in stark contrast to observations made by earlier studies. For example, a decade ago, Wang et al. [212] found that high network latency and the serialization of network requests in page loads are the key contributors to poor mobile web performance. Since then, the mobile web landscape has changed significantly in three ways. First, due to a 680% increase over the last 10 years in the number of bytes of JavaScript included on the median mobile page [43], the amount of client-side computation as part of web page loads has dramatically increased. Second, the quality of mobile networks has improved greatly, e.g., over the last 10 years, on the average mobile connection globally, bandwidth has increased from 1MBps to 19MBps and RTT has decreased from 700ms to 65ms [60, 55, 54]. Lastly, the increased adoption of HTTP/2 has reduced the serialization of network requests; while HTTP/2 did not exist a decade ago, the fraction of requests on the median page that are served over HTTP/2 is now up to 67% [28].

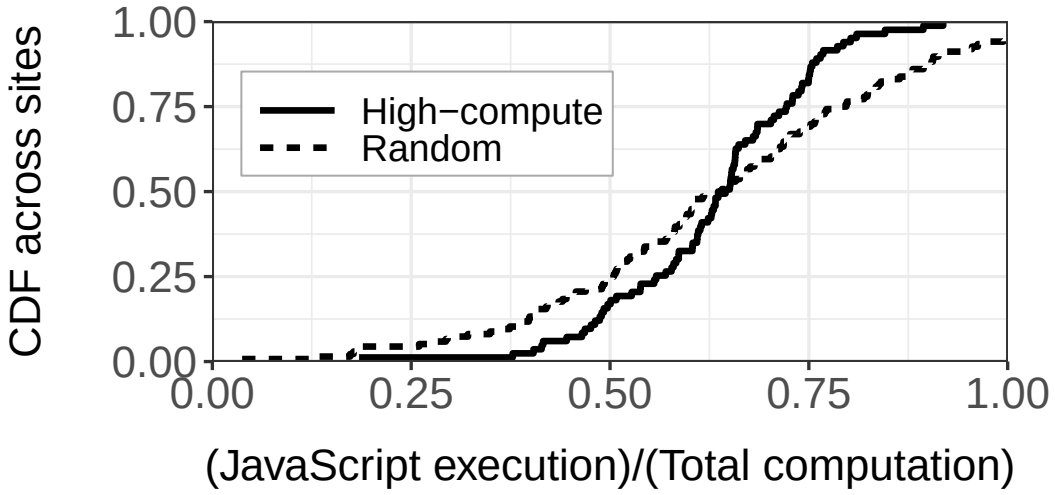


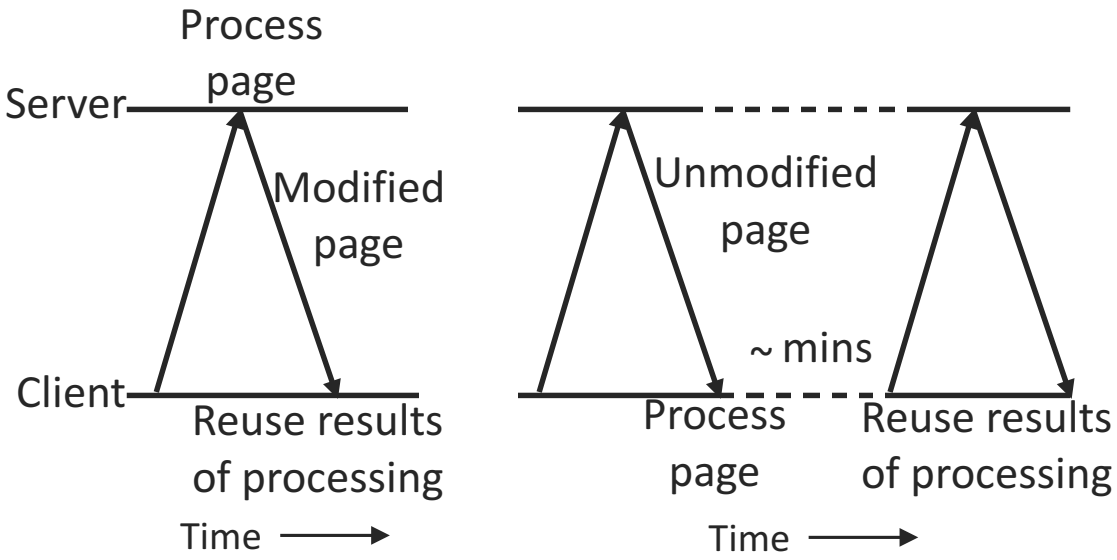
Figure 4.2: **Fraction of browser computation accounted for by JavaScript execution. Results are for 200 random landing pages in the Alexa top 1000 and 86 high-compute pages.**

Compute will continue to slow down page loads. Improvements in CPU performance generally come from increase in either the number of cores or the clock speed of each core. However, with mobile devices, improvements have been largely due to the former, with clock speeds increasing at a far slower rate, e.g., CPU clock speed on the Samsung Galaxy S series increased from 1.9GHz in 2013 to 2.73GHz in 2019; the number of CPU cores doubled during that time (from 4 to 8).

Unfortunately, this trend of increased cores provides little benefit to the mobile page load process. Web browsers are more dependent on clock speed than the number of cores [**?**] because they are unable to fully take advantage of all available CPU cores (described more below). Indeed, Figure 4.1's *"Unshaped, 4 cores"* line shows that PLTs are largely unchanged even when I disable 4 out of 8 CPU cores on the Pixel 2.

Cellular networks, on the other hand, are projected to continue to get significantly faster [16]. Given these trends, as well as the energy restrictions that hinder CPU speeds on mobile devices [183], client-side computation will likely continue to significantly contribute to load times.

JavaScript execution dominates computation delays. Computation delays in page loads stem from numerous tasks that browsers must perform, such as parsing and evaluating objects like HTML, CSS and JavaScript, and rendering content to the screen. Furthermore, the JavaScript engine in the browser spends time compiling and executing JavaScript, and doing garbage collection. I analyzed the loads of each page in my corpus to identify which of these compute tasks browsers spend the most time on. I consider two sets of pages: 200



(a) Server-side technique (b) Client-side technique

Figure 4.3: **(a) Server-side acceleration techniques send a processed page to the client. (b) Client-side acceleration requires initial web page loads to populate the computation cache, and then subsequent page loads to utilize this cache.**

random pages from the Alexa top 1000 sites, and the 86 high-compute pages from above.

I find that JavaScript execution is the primary contributor to browser computation delays in page loads. In particular, Figure 4.2 shows that JavaScript execution accounts for 64% and 65% of overall computation time for the median page in the two sets of pages, respectively.

This explains why page load performance does not benefit much from more cores, as I saw above. JavaScript execution in browsers is single-threaded and non-preemptive for each frame in a web page [185]. While this single-threaded model greatly simplifies web page development, it does so at the cost of degraded performance and resource utilization.

4.1.2 Client-side computation reuse

To overcome the practical limitations of prior systems, I advocate for a purely client-driven approach to reduce client-side computation in mobile page loads. Rather than having clients reuse the results of *server-side* or *proxy-side* page load processing, I envision each client reusing computations from *its own* page loads from the past. More specifically, like how web browsers cache objects to exploit temporal locality in a client's page loads [215] and eliminate redundant network fetches, I propose that browsers also reuse JavaScript executions from prior page loads.

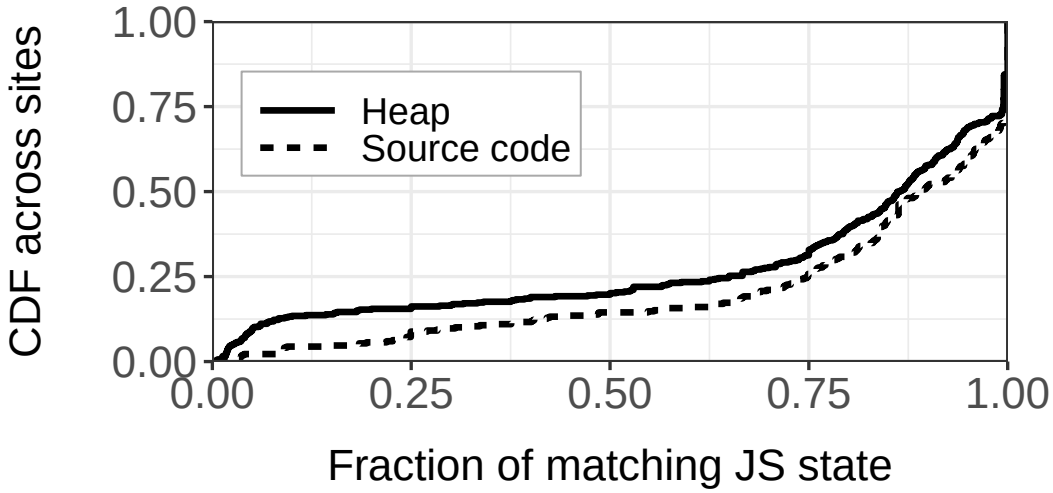


Figure 4.4: **Fraction of JavaScript that matches across two loads of the same page one hour apart.**

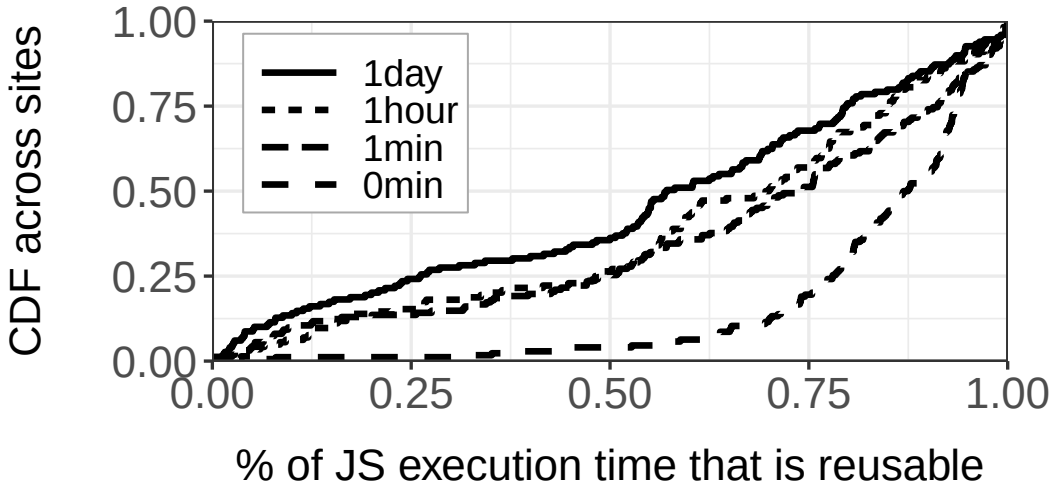
4.1.2.1 Need for fine-grained computation reuse

Although conceptually straightforward, my proposal necessitates a fundamentally new approach for how computation is reused. The primary difference between my vision and existing proxy-/server-side approaches is that of timing (Figure 4.3). In existing solutions, a proxy/server loads a page in response to a client request and returns a compute-optimized version, which the client applies a few seconds later. In this workflow, clients download a single post-processed object that reflects *all* of the state in the latest version of the page.

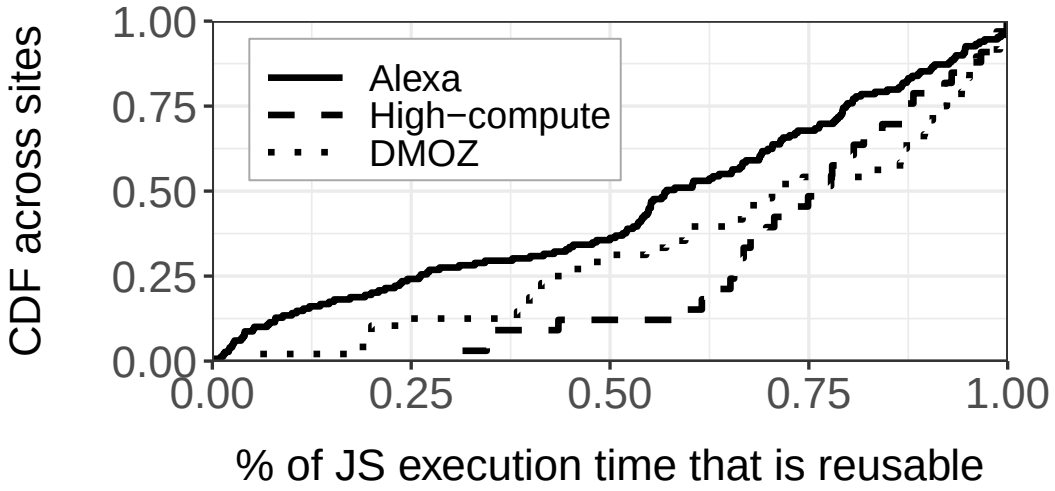
In contrast, if the browser locally caches computations from one of its page loads, the time gap until a subsequent load in which the cached computation is reused can be unbounded. As this time gap grows, operating the cache at the granularity of an entire page becomes increasingly suboptimal since even a small change to page content will render the page-level cached object unusable in subsequent page loads.

To better understand how often and in what ways web pages change over time, we load the landing pages of the Alexa top 500 sites twice, with a 1 hour time gap, and compare 1) the JavaScript source code fetched during each page load, and 2) the final `window` object that represents the constructed JavaScript heap. For source code, I compute the fraction of bytes that are identical, and for the heap, I compute the fraction of keys within the window object, whose values have the same *SHA-1 hash* in both loads. Figure 4.4 shows that both parts of web computation change over the course of an hour on most pages. A common reason behind this frequent change in page content is the increasing dynamism in modern web pages [162]; web servers often compute responses on-the-fly in order to deliver customized content catered to individual users.

4.1.2.2 Our proposal: function-level caching



(a) Landing pages of 150 sites out of Alexa top 500



(b) Pages loaded at a time gap of 1 day

Figure 4.5: **Reusable JavaScript execution across different time intervals and across different sets of sites.**

While the above results highlight that a page-level caching approach (i.e., a client entirely reuses computation results from a prior page load) will present minimal opportunities for computation reuse,[2] they also show that large parts of JavaScript content remain unchanged

[2]One approach to handling small changes in page content is to patch page-level caching data. However, a client-side caching solution precludes this approach because clients are unaware of content changes until they fully load the latest version of a page (thereby foregoing caching benefits).

across loads of a page. For the median page, Figure 4.4 shows that 86% of heap state and 89% of JavaScript source code match between two loads separated by an hour; also, JavaScript state changes by over 75% across an hour on only 25% of pages. Taken together, there exists significant potential for computation caching, but a fine-grained strategy is necessary to realize the savings.

Determining how fine a granularity to cache at (e.g., small or large code blocks) involves a tradeoff between potential benefits and storage overheads. Finer-grained caching would result in more cache entries and subsequently higher storage overhead, but would also offer more potential benefits (and be less susceptible to page changes).

I observe that a natural solution for balancing this tradeoff is to leverage the fact that most of the JavaScript code on a page is typically within JavaScript functions. On the median landing page among the Alexa top 500 sites, 96% of all the JavaScript code is inside functions. Furthermore, JavaScript functions represent a logical unit of compute as intended by the code developer. These properties naturally lend themselves to a function-level compute caching approach.

4.1.3 Benefits of client-side compute cache

Given the single-threaded nature of JavaScript execution (§4.2.1), if a JavaScript function is deterministic, then its execution is reusable when it is invoked with the exact same input state as one of its prior invocations, i.e., outputs from the prior invocation can be applied without executing the function again. In this section, I estimate the potential benefits of client-side computation caching by determining the percentage of JavaScript execution time that can be eliminated by reusing the results of function invocations from prior page loads.

4.1.3.1 Overview of JavaScript function state

A JavaScript function has access to a variety of web page state – global objects, local variables, function arguments, and closures – with the precise set being determined by web security policies (e.g., same-origin policy) and scope restrictions implicit to each state's definition. All of this state is mapped to objects on the JavaScript heap, DOM tree, and disk storage (like localStorage and sessionStorage). Given this, a JavaScript function execution can be summarized by the combination of its 1) *input state*, or the subset of the page's state that it consumes, and 2) *externally visible effects*, such as its impact on the page's global JavaScript heap, calls to internal browser APIs (e.g., DOM), and network fetches.

4.1.3.2 Quantifying potential for computation reuse

Methodology: I use the following approach to estimate the benefits of reusing computation from one page load in a subsequent page load. First, I identify all functions which make use of non-deterministic APIs (e.g., Math.random, Date, key traversal of dictionaries, and timing APIs [158]) and network APIs (e.g., XHR requests), and mark all such functions as uncacheable. For all remaining JavaScript functions, during both loads, I track the input state consumed by every invocation. Note that I do not include the input state of the nested functions in the parent functions, instead they are treated as separate function invocations. For each function, I then perform an offline analysis to determine which of its invocations in the later page load had the same input state as an invocation in the initial load; all matching invocations could be skipped via client-side computation caching. I then correlate this information with function execution times reported by the browser's profiler to compute the corresponding savings in raw computation time.

To employ the above methodology, I record web pages with Mahimahi [163] and then rewrite those pages using static analysis techniques [164]. The rewritten pages contain instrumentation code required to track and log function input state; it suffices to log only input states as my goal here is to only estimate the *potential* for reuse, and not to actually reuse prior computations. I then reload each instrumented page in Mahimahi and extract the generated logs. I run my experiments on three different corpora: landing pages for the Alexa top 500 sites, landing pages from 100 (less popular) sites in the 0.5 million-site DMOZ directory [20], and the 86 high-compute sites from Figure 4.2.

Reuse across loads of the same page. As discussed in Section 6.4, the time between the page load where the cache is populated and the load where the cache is used to skip function invocations is unbounded. I therefore compare input states for each function invocation across pages loads spaced apart by 1 minute, 1 hour, and 1 day.

Our dynamic tracing tool was successfully able to track input state of JavaScript invocations for 150 of the 500 Alexa pages and 33 of the 86 high-compute pages. Since the distribution of load times across these subsets of sites matches the overall distribution in the respective corpus, I expect my findings to be representative of other pages too. Figure 4.5(a) shows that, for the median Alexa page, 73% and 57% of JavaScript execution time can be skipped via computation caching across 1 minute and 1 day. From Figure 4.5(b), the fraction of JavaScript execution that is reusable a day later is higher (76%) for pages which are more in need of computation caching: the high-compute pages.[3] Since JavaScript execution

[3]To minimize the instrumentation overhead of my tracing tool, I exclude tracking of closure state for high-compute pages. On the subset of these pages which I are able to successfully load with closure state tracking enabled, I see that 59% of JavaScript execution time can be reused a day later on the median page.

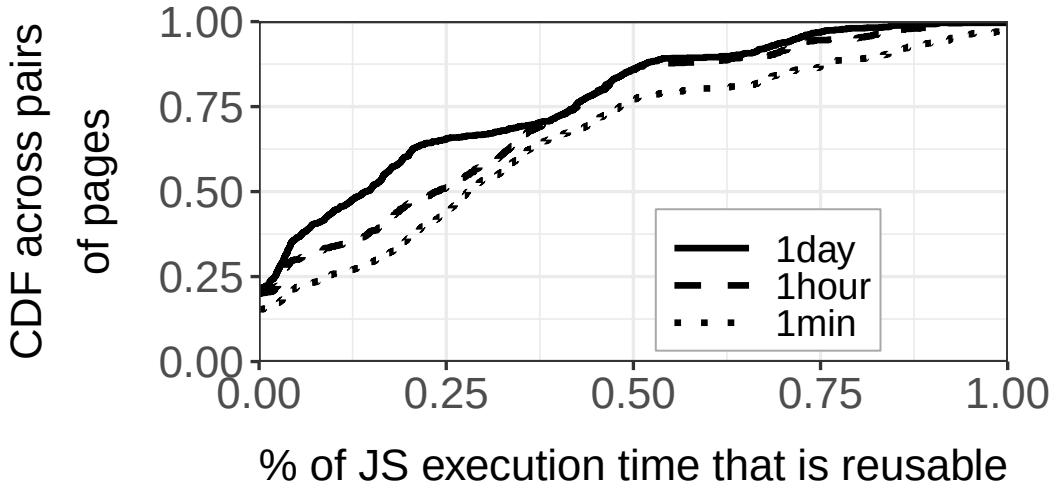


Figure 4.6: **Reusable JavaScript execution across pairs of pages.**

accounts for 65% of client-side computation on the median high-compute page (Figure 4.2), I estimate a net savings of 49% for the median page. This translates to 990ms of eliminated execution time even on the state-of-the-art Pixel 2 phone. I observe similar reuse potential for the DMOZ pages (48 out of the 100), with a median of 70% of Javascript execution time being reusable across a day.

Reuse across loads of different pages. Thus far, I have focused on reusing computations from a prior load of the exact same page. However, I observe that traditional browser caches support object reuse even across pages. More specifically, object caches are keyed by an object's resource URI, which may appear on multiple pages; this is a common occurrence for pages on the same site, e.g., a shared jQuery library. Inspired by this, I extend my analysis of computation caching to examine how cache entries can be reused across loads of *different* pages from the same site.

I sample 10 sites at random from the Alexa top 1000, and then select 20 random pages on each site. For example, *www.gamespot.com/3-2-1-rattle-battle/* and *www.gamespot.com/101-dinopets-3d/* constitute two pages on the same site. We then compare all pairs of pages on the same site to determine what percentage of JavaScript execution time can be reused between each pair. In other words, I load each of the 20 pages in a site once, load all of these pages again after a time gap, and evaluate how much of the JavaScript execution time on the latter load of a page can reuse executions from prior loads of the other 19 pages. My tool successfully tracked input states for 80 out of the 200 pages. Figure 4.6 shows, that for the median pair, 29% and 15% of JavaScript execution time can be reused across time intervals of 1 minute and 1 day, respectively. The % of reuse varies from site to site, with it being as high as 80% for *www.ci123.com* and as low as 9% for *www.prezi.com*.

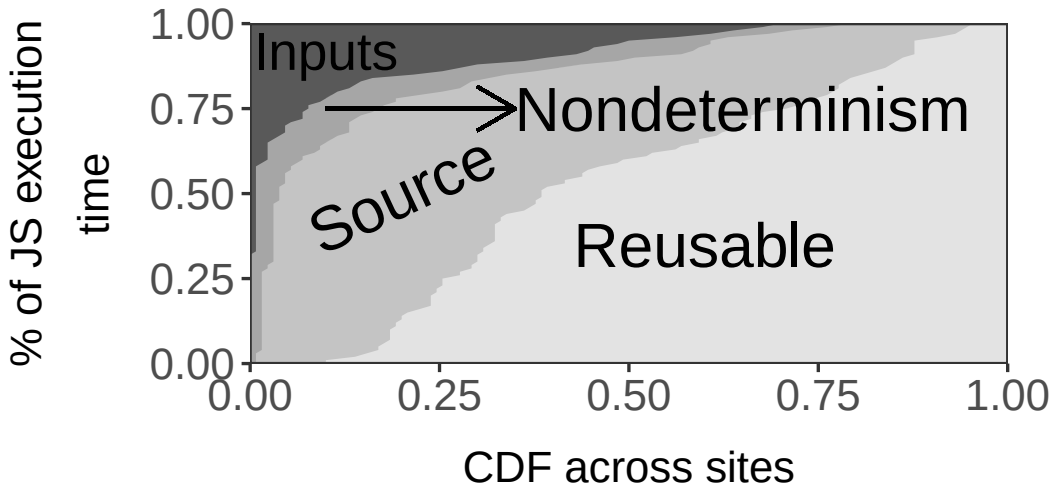


Figure 4.7: **JavaScript execution time breakdown for landing pages of 150 out of Alexa top 500 sites loaded at a time gap of 1 day. Reusable accounts for the execution time that can be reused. Source, Inputs, and Nondeterminism account for execution time which is non-reusable due to change in JavaScript source files, change in function inputs, and non-deterministic functions, respectively.**

Note that a user can benefit from all of the reductions in client-side computation that I estimate in this section, both across loads of a page and across loads of multiple pages on the same site, with only modifications to the user's web browser. In contrast, a large number of domains need to adopt prior server-side computation eliding strategies [165] in order for users to see benefits across all of the pages they visit. Furthermore, since JavaScript execution can be reused across pages as well, even the very first load of any page at a client can be sped up by reusing computation from that client's prior loads of other pages on the same site.

4.1.3.3 Characterizing computation cache misses

While my results above highlight the significant potential for computation reuse, I see that not all function invocations can be serviced by the cache. To investigate the reason for such misses, I analyze the data for the *"1 day"* line from Figure 4.5(a), and plot in Figure 4.7 the breakdown of the total execution time into reusable and non-reusable components (marking the reason precluding reuse).

Non-determinism. Functions which invoke non-deterministic APIs are not amenable to computation caching. Figure 4.7's *"Nondeterminism"* shaded area shows the amount of execution time marked non-reusable due to the presence of non-deterministic functions on the page. For the median page in my corpus, 0.5% of total non-reusable time was accounted

for by such non-deterministic functions.

Changes to JavaScript source code. For a function's execution to be reusable from an earlier page load, the source code of the JavaScript object file containing the function's definition should not have changed since that load.[4] As previously shown in Figure 4.4, 11% of JavaScript source code is subject to change within a time period of one hour. The longer the time gap between loads, the more susceptible are web pages to load object files with different source code [209].

Figure 4.7 shows that change in source code is the predominant reason for non-reusable computation. Further analysis reveals that for 13% of pages, change in source code was the only reason hindering reuse.

Changes to input state. Figure 4.7 also shows the percentage of non-reusable computation accounted for by changes in the input state of a function. A JavaScript function can observe different sets of inputs across invocations on different page loads, either due to changes in server-side state or due to non-determinism on the page.

Changes in server-side state can influence the responses sent to the client, which in turn can affect the input state of JavaScript invocations. For example, *www.cnblogs.com* sends a cookie known as *RNLBSERVERID* as a part of the response to a client request. This cookie value is used for server-side load balancing. Since the value of this cookie changes dynamically depending on server logic, a JavaScript function on this page reads different values for this cookie across loads spread out by 1 hour or more, causing this function to forego the use of results from prior invocations.

Inputs to a function can also change across loads due to non-determinism in functions which are invoked earlier in the page load. In order to distinguish between server-side state and non-determinism as the cause for change in input state, I load the same recorded set of webpages twice using Mahimahi. This eliminates any potential changes in the source code of JavaScript object files and server-side state. I then compare the input state across these two loads. Any mismatch could only be due to non-determinism on the page. Figure 4.5(a)'s *"0min"* line shows that for the median page, 18% of the total execution time is non-reusable due to such non-determinism.

4.1.4 Envisioned system

While my primary goal in this chapter is to motivate the need for client-side solutions and quantify the potential benefits of my proposed client-side reuse of JavaScript executions, I end

[4] Though it would suffice for only the source code within the function's boundaries to remain unchanged, my analysis considers cache entries for a function as non-reusable if the file containing the function changes.

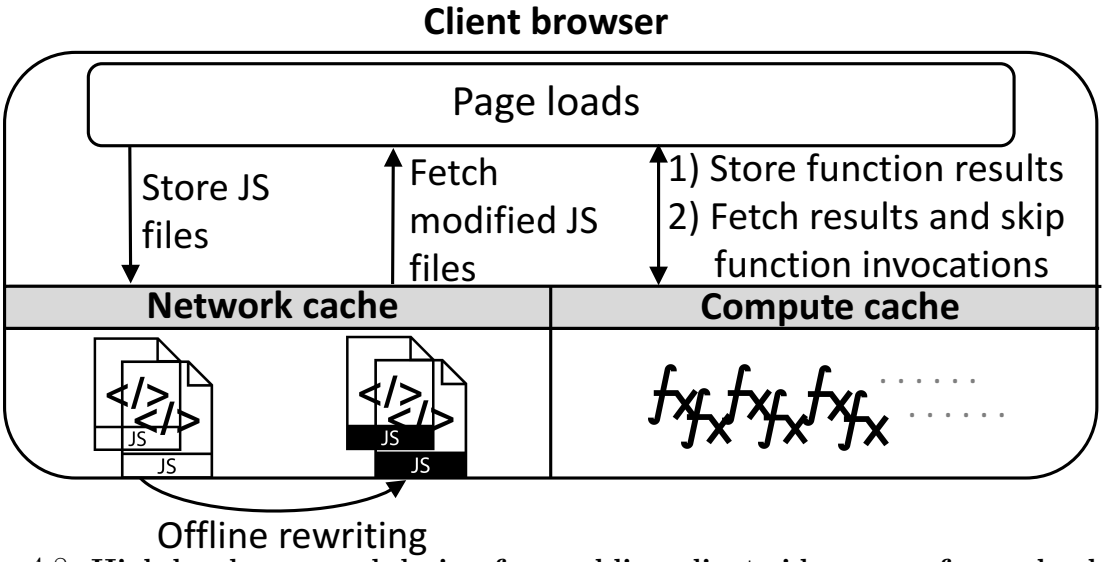


Figure 4.8: **High level proposed design for enabling client-side reuse of page load computations.**

by describing how web browsers might implement a computation cache and the challenges in doing so.

4.1.4.1 System workflow

Figure 4.8 illustrates my envisioned approach. When the user loads a page, the browser would fetch and cache previously un-cached resources, as it does today. The browser can then use (offline) static program analysis to instrument the cached scripts to track the state that each JavaScript function accesses or modifies (Section 4.1.3.1). Finally, when an instrumented script is executed in any page load, prior to executing every instrumented function, the browser can perform a computation cache lookup in search of a matching entry (i.e., a prior invocation of the same function with the same inputs); if a match is found, the browser would apply the effects from that cache entry instead of executing the function; otherwise, it would execute the function and add a new entry to the cache.

4.1.4.2 Practical challenges

Although my proposed design requires only browser modifications (easing deployability), my data collection in Section 4.1.3.2 reveals several overheads that hinder practicality.

- *Tracking:* The instrumentation required to track per-function state accesses and modifications can result in user-facing slowdowns of up to 200× [91]. To mask these overheads, tracking could be performed offline or using spare CPU cores (§4.2.1), but such

solutions would require special care (designed per page) to avoid negatively impacting persistent client- or server-side state (e.g., cookies) via, say, idempotent functions [210].

- *Cache management:* Similar to the browser's network cache, the computation cache would incur various caching overheads in terms of lookup times and storage. These overheads are exacerbated in the case of computation cache because of differences in the granularity at which the two caches operate: cache entry per resource in the network cache versus cache entry per function invocation in the compute cache. On the median landing page across the Alexa top 500 sites, the number of function invocations per page load (close to 9K) are more than two orders of magnitude higher than the number of resources fetched (30). Therefore, the overheads associated with the compute cache are likely to be much higher than with the network cache.

Solution. To address these overheads, I make the following key observations: 1) the distribution of execution time across JavaScript functions on a page is (typically) heavily skewed towards only a small fraction of functions, and 2) these small fraction of functions have relatively stable execution times across page loads. For the high-compute pages in my corpus, 80% of execution time on the median page is accounted for by only 27% of the functions. Across loads separated by an hour, the execution time for the median of these functions changed by only 15%. This implies that, to sidestep the aforementioned overheads without sacrificing most reuse benefits, only the subset of functions accounting for the bulk of the execution time should be considered for caching. Note that function-level execution time information can be easily extracted using built-in browser profilers [95] at low overheads.

4.1.5 Summary

Despite an abundance of proposed solutions, client-side computation continues to slow down mobile web page loads. To overcome this, I propose a purely client-side solution to elide web computations: augmenting web browsers with a computation cache. As a first step towards this vision, I empirically motivate the need for a finer granularity of computation reuse than prior systems, and I quantify the potential for client-side reuse of JavaScript function executions. I estimate that 49% of client-side computation can be reused across loads even a day apart on the median high-compute web page. I outlined a potential system architecture to realize my proposed client-side web computation cache and the challenges entailed in reaping the estimated benefits.

The mobile web landscape is highly dynamic, with the underlying technology changing at a rapid rate. In light of this dynamism, it is hard to predict the applicability of a technique,

such as ours, even a few years down the line. However, I believe that certain web trends will continue to hold, as they have in the past, and would therefore continue to enable JavaScript computation reuse. First, prior studies [215, 174] have shown that more than half the web pages browsed by users are revisits. I expect page revisits to be as common going forward. Second, despite the web's dynamism, it has been shown that large parts of web page content are fairly static [150, 209]. Taken together, I believe there will continue to exist significant opportunity for reusing JavaScript computations as a means to improve mobile web performance in the long run.

4.2 Automatic JavaScript Parallelism for Resource-Efficient Web Computation

In the previous section, I discussed how client-side computation alone results in poor quality of experience for mobile web users. To mitigate the impact of high computation, I proposed a system that automatically identifies opportunities for JavaScript computation caching and skips executing certain functions using this computation cache.

In this work, I propose a tangential solution to the same problem. Specifically, I observe that there exists a fundamental inefficiency in the computation model that browsers employ (§4.2.1). To simplify page development, JavaScript execution is single-threaded [168, 165], and worse, JavaScript and rendering tasks are forced to share a single "main" thread per frame in a page [116]. Consequently, browsers are unable to take advantage of the growing number of CPU cores available on popular phones in both developed and emerging regions [**? ?**]. This inefficiency will only worsen as, due to energy constraints, increased core counts have become the main source of compute resource improvements on phones [183, 109].

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 [153, 56] 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 [73, 127]. 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 [147, 61] is impractical, while reliance on browsers to speculatively make parallelism decisions [87, 154] 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.

In this work, I ask *How much parallelization potential exists in JavaScript execution on legacy web pages?*

4.2.1 Motivation

In this section I look at how the multiple cores on modern smartphones affect the total client-side JavaScript execution time.

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 [183, 109]. For example, since their inception in 2016, Google's Pixel smartphones (our developed region phone) have improved 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 [58] (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, I find that they fail to leverage available cores. To illustrate this, I iteratively disabled CPU cores on the phones in each setting and observed the impact on page load times. As shown in Figure 4.9, 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.

To determine the origin of these computation inefficiencies, I must consider the computation model that browsers use today. My discussion will be based on the Chromium framework [116], which powers the Chrome, Brave, Opera, and Edge browsers that account for 70% of the global market share [63, 59]. Figure 4.10 depicts Chromium's multi-process architecture. I 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, 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

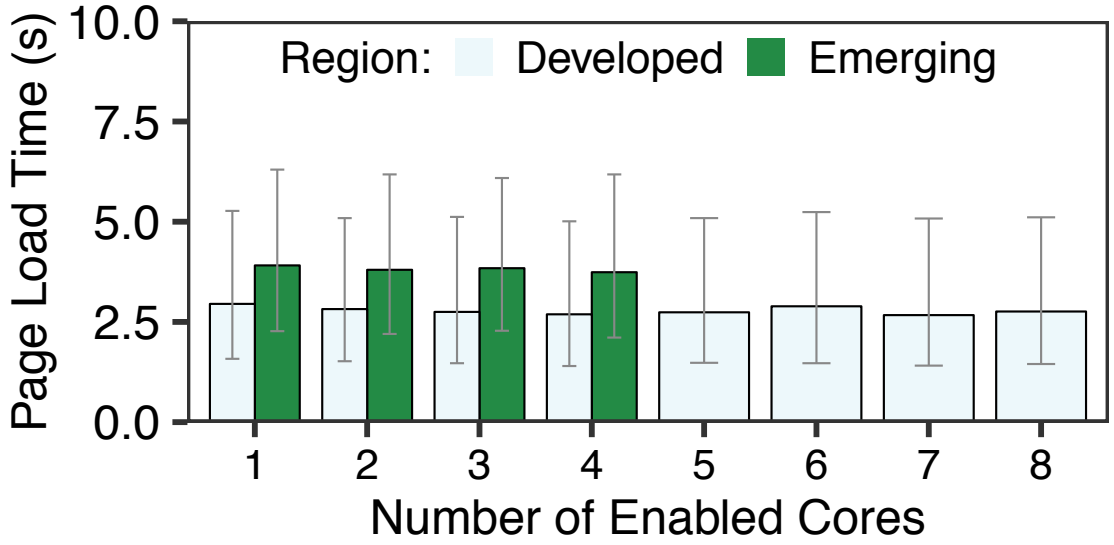


Figure 4.9: **Additional CPU cores have minimal impact on load times. The developed and emerging region phones have 8 and 4 cores. Bars list medians, with error bars for 25-75th percentiles.**

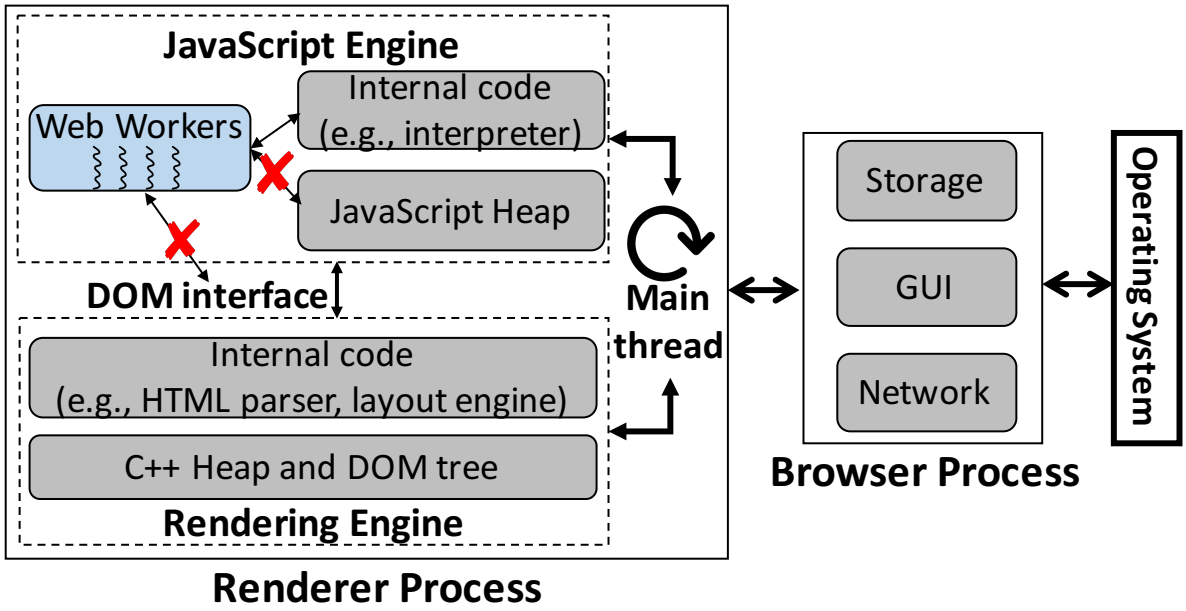


Figure 4.10: **Computation model for Chromium browsers.**

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.

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

Table 4.1: **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.**

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 [168, 165]. Worse, within a renderer process, all tasks across the two engines are coordinated to run on a single thread, called the *main thread*.[5] This lack of parallelism largely explains the poor use of CPU cores described earlier. 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 [62]). 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. I 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 [126].

4.2.2 Estimating benefits of parallelizing JavaScript execution

Given the results in §4.2.1, 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, I 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—I call this

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

property *safety*.

To estimate the potential benefits of parallelism with legacy pages, I analyzed the JavaScript code for each page in my corpus; in this section, I focus on page loads representative of those in developed regions. Since JavaScript functions account for 94% of the JavaScript source code on the median page, my 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, I convert all code outside of functions into anonymous functions. For each function, I 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.

I instrument the JavaScript source code to log all accesses to state in both the JavaScript heap and DOM tree; my instrumentation matches recent dynamic analysis tools [164, 169, 165], but with the following differences based on my parallelism use case.

- First, I care not just about the state that remains at the end of the page load [165], but also any state accessible by multiple functions during a page load. Hence, in addition to global heap objects, I track 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) [157].
- 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, I would log a read to `a.foo`, not `a`.
- For the DOM tree, I adopt a coarser approach than prior work. Instead of logging reads and writes to individual nodes in the DOM tree, I 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 I explain later, my design has the browser's main thread serialize all DOM operations.

Using these state access logs, I 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), my 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 4.1, 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 [**?**]. In this dissertation, I only leveraged my JavaScript analysis engine to identify and extract relevant state access information to accurately measure potential benefits from parallelizing JavaScript execution. Leveraging my analysis engine, Shaghayaghi et al built Horcrux [149], a web optimization system that actually reaps these parallelization opportunities to speed up web page loads.

4.2.3 Summary

In Horcrux [149], the authors implemented a web accelerator based on my findings of high JavaScript parallelization potential on modern web pages. Horcrux reduced the total client-side computation by around 40% on the median page in their corpus, using a smartphone with 8 cores. Since JavaScript execution accounts for 50% of client-side computation, this implies that Horcrux was able to speed-up JavaScript execution by around 80%. This result aligns exactly with my findings of the potential benefits feasible, discussed in this chapter.

The reason behind Horcrux's low overhead while offloading execution to multiple cores was twofold. First, Horcrux generates per-function signatures on the server-side, i.e., it relies on the participation of web servers. This eliminates the overhead of performing static analysis and extracting signature information, which is computationally expensive. Second, Horcrux only offloads functions at the granularity of root functions, minimizing the offloading overhead.

CHAPTER 5

Sprinter: Speeding Up High-Fidelity Crawling of the Modern Web

In this chapter, I highlight the importance of crawling the web at scale, and how little attention it has received from the research community in the recent years. I describe how current crawling practitioners have to choose from crawling techniques that sit at the opposite ends of the performance-fidelity trade-off; either crawl with high throughput, i.e., large number of pages per second, and sacrifice on the fidelity of the crawls, or crawl pages with perfect fidelity at the cost of significantly lower crawling throughput.

To address this trade-off, I present the design of Sprinter which combines browser-based and browserless crawling to get the best of both. The key to Sprinter's design is my observation that crawling workloads typically include many pages from every site that is crawled and, unlike in traditional user-facing page loads, there is significant potential to reuse client-side computations across pages. Taking advantage of this property, Sprinter crawls a small, carefully chosen, subset of pages on each site using a browser, and then efficiently identifies and exploits opportunities to reuse the browser's computations on other pages. Sprinter was able to crawl a corpus of 50,000 pages 5x faster than browser-based crawling, while still closely matching a browser in the set of resources fetched.

5.1 Problem Statement

To make the most of the enormous trove of information available on the web, all of us today rely upon a range of efforts. Web search engines help users find pages relevant to their needs. Data from the web serves as input to smart assistants such as Siri and Alexa, and is used to train generative AI models that can answer my questions. Web archives store repeated snapshots of web pages to document changes over time and to preserve the content of deleted

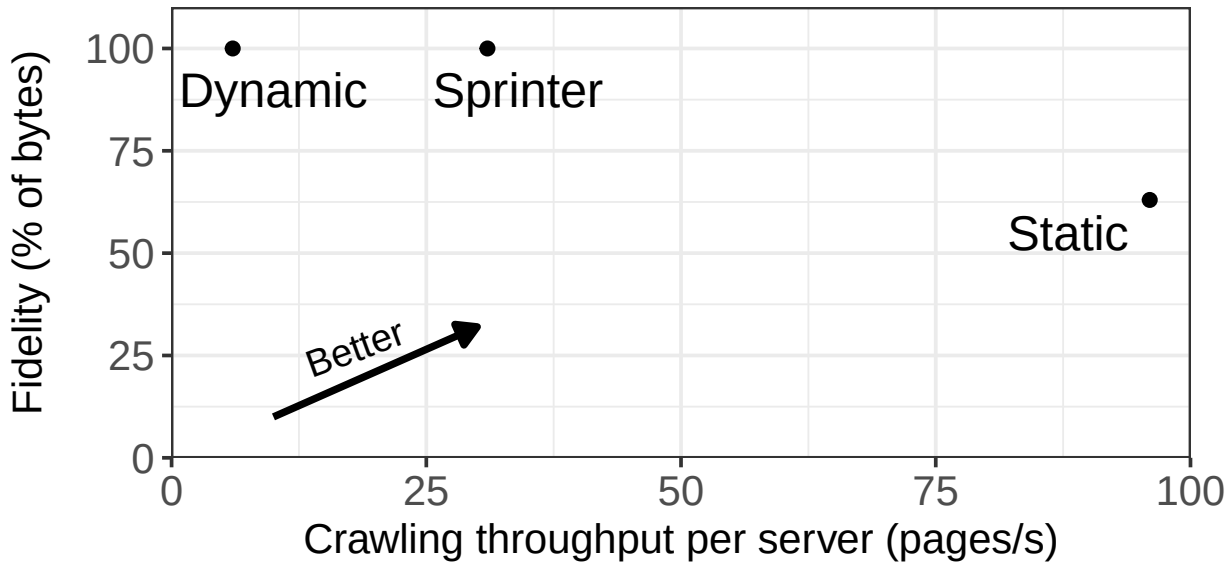


Figure 5.1: **Tradeoff between fidelity and performance with different crawlers.**

pages. Researchers continually study the web to help improve its performance and security.

A key enabler for all of the above is a capability that I take for granted today: the ability to crawl the web at scale. Web crawlers have traditionally crawled a page by first downloading the page's HTML, and then recursively fetching all embedded links to images, CSS stylesheets, scripts, etc. If one deploys many such so called static crawlers [118, 98] across a fleet of machines, the rate of crawling is limited by the network bandwidth of each machine.

Given that web crawlers have existed for over three decades, why revisit this topic now? Because, static crawlers no longer suffice. On today's web, the URLs of many of the resources on a page are determined at runtime, rather than being statically embedded in the page's source. To discover and fetch such resources, modern "dynamic" crawlers [11, 6, 10] leverage web browsers such as Chrome, Firefox, or Edge. However, due to the compute overheads associated with JavaScript (JS) execution and with browsers in general, the rate at which one can crawl pages drops by an order of magnitude relative to static crawling (Figure 5.1). Consequently, dynamic crawlers need to be deployed across a much larger number of servers in order to sustain the same crawling throughput as that feasible with static crawlers.

Thus, anyone seeking to crawl the web today has to make do either with the poor performance of dynamic crawlers or the incompleteness of static crawlers. Unfortunately, there is no easy fix. One could try to augment a static crawler with a lightweight JavaScript execution engine, but keeping up with constantly evolving web APIs is a challenge best left to the developers of widely used browsers. On the other hand, proposals that attempt

to mitigate the impact of client-side web computations on user-perceived web performance have little utility in the context of crawling. For example, overlapping the browser's computations with its network activity [164, 86, 208] or parallelizing the browser's execution of JavaScripts [149] can reduce page load times, but crawling throughput remains unchanged since the total amount of client-side computation is the same.

I address this undesirable status quo with Sprinter, a new crawler which judiciously combines browser-based and browserless crawling. Sprinter crawls pages at a much faster rate than dynamic crawlers while matching them in the resources fetched. Our main observation is that large-scale web crawling workloads typically include many pages from each site and there is significant potential to reuse client-side computations across pages (§5.3.1).To exploit this property, my design of Sprinter is based on three key principles.

First, when Sprinter crawls a page using a browser, it strives to minimize the amount of JS code executed. For every script file on a page, Sprinter attempts to reuse the browser's execution of that file on a previously crawled page. In user-facing page loads, execution of the same file is seldom exactly identical across multiple pages. In contrast, Sprinter can reuse JS execution at such a coarse granularity because it can skip executing a JS file as long as the URLs of the resources that file would fetch match those fetched during a prior execution of that file.

Second, even if none of the JS files on a page are executed, crawling the page with a browser imposes significant compute overhead. Therefore, on any site, Sprinter crawls the vast majority of pages on the site without a browser. To realize browserless crawling that does not sacrifice fidelity, I implement a lightweight page instrumentation framework that tracks the web APIs used on any page without support for executing these APIs. When it crawls a page without a browser, Sprinter uses this instrumentation to identify whether it can safely reuse JS executions from pages that it previously crawled with a browser.

Lastly, to maximize the fraction of pages that can be crawled without a browser, Sprinter crawls the pages on a site in a carefully chosen order. For any given site, Sprinter efficiently identifies a subset of pages such that most of the scripts seen on other pages are fetched as part of this subset. Sprinter crawls these pages first using a browser and captures the effects of JS executions. Most of the remaining pages can then be crawled without a browser, since Sprinter can identify all resources to be fetched on those pages without executing any JS code or web APIs.

I used Sprinter to crawl a corpus of 50,000 pages spread across a diverse collection of 100 sites. It offered a 5x speedup in crawling throughput compared to existing dynamic crawlers. When I recrawled the same corpus a week later, the rate at which Sprinter crawls pages improved by a further 78%. Importantly, Sprinter preserves almost all resource fetches

issued by a browser-based crawler, and it is compatible with legacy web servers. Sprinter's source code is available at https://github.com/goelayu/Sprinter.

5.2 Background and Motivation

I begin by describing common web crawling workloads and quantifying the limitations of existing strategies for supporting these workloads.

5.2.1 Target workloads

Web crawlers take as input a seed list of URLs to pages that need to be crawled. The input configuration to the crawler can specify a range of options such as timeout per page, retry policy, politeness constraints (i.e., time gap between crawls of pages on the same site), and whether other pages discovered while crawling the seed list should also be recursively crawled. Some crawlers provide the option of saving page screenshots [6] and triggering user interactions (e.g., scrolling or clicking) on rendered pages [10]. In this work, I focus on supporting the common need for crawlers to save the content of resources that are fetched on every page that is crawled. To not make any assumptions about what the crawls will be used for, I aim to fetch and save all page resources requested by a browser such as Chrome, rather than a subset that may suffice for a particular use case.

I focus on supporting workloads where pages are crawled from a large number of sites. This is the case in any large-scale system that relies on web crawls, e.g., to support web search, ChatGPT, and Siri, their providers aim to crawl the entire web. Even in more focused crawls, it is common to crawl many sites and many pages in each site. For example, after every presidential term in the US, the Internet Archive captures a snapshot of 1.3 million government websites, crawling roughly 700 pages on average per site [32]. Similarly, research studies attempting to understand the web's security vulnerabilities [180] have crawled roughly 2500 pages per site. When pages are crawled from a single site (e.g., a research study of pages on Facebook), the rate at which pages can be crawled is constrained by the rate limits imposed by the site being crawled.

5.2.2 Shortcomings of static crawling

As mentioned earlier, web crawling has traditionally relied on static crawlers, which identify all the resources to fetch on every page by extracting links embedded in the page's source. To demonstrate and quantify why static crawlers are now insufficient, we compile $Corpus_{10k}$, a collection of 10,000 pages comprising 100 randomly sampled pages from each of 100 sites:

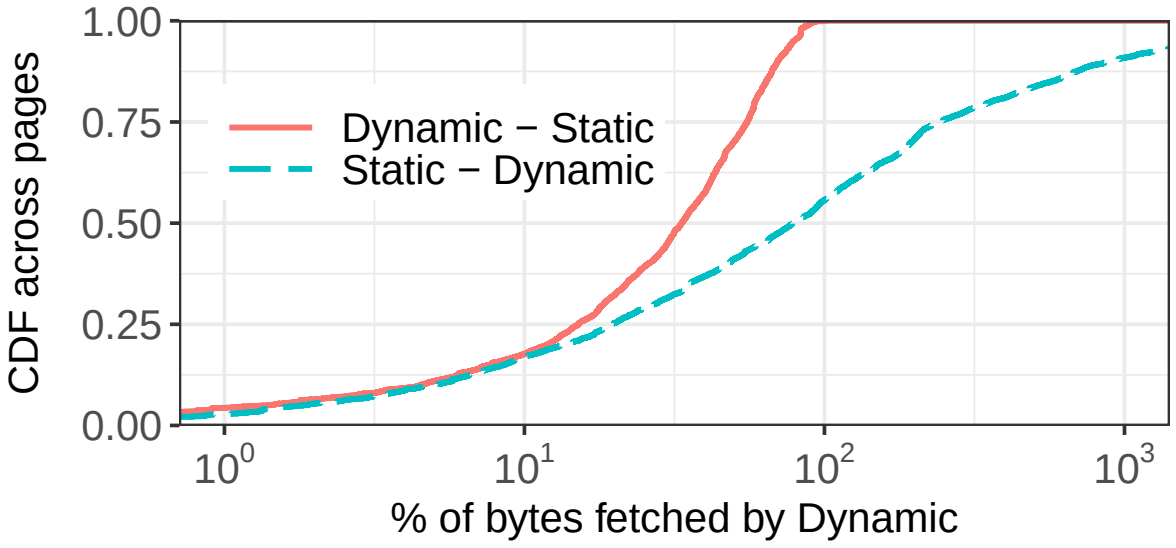


Figure 5.2: **Compared to a dynamic (i.e., Chrome-based) crawler, a static crawler both fails to fetch some resources and fetches many additional resources. Distribution shown is over 10,000 pages spread across 100 sites. Note logscale on x-axis.**

roughly 33 sites chosen at random from three ranges – [1, 1000], [1000, 100k], and [100k, 1m] – from Alexa's site rankings. This corpus spans a diverse collection of sites and is representative of real-world crawls in that it includes a large number of pages per site crawled.

On a server which has a 16-core 2.1 GHz Intel Xeon CPU, a 1 Gbps network connection, and a 500 GB SSD disk, I crawl every page in $Corpus_{10k}$ using a custom crawler which loads every page in Google Chrome but also fetches all URLs, both absolute and relative, that are embedded in text-based resources (i.e., HTML, CSS, and JS). I record all responses using a web record and replay tool [15]. I then separately crawl every page from my recorded copy once using Chrome and once using my custom static crawler (which mimics wget2 [24], a state-of-the-art static crawler), with network caching enabled in both cases, i.e., across all pages, each unique resource was only fetched once. Comparing the two types of crawlers in this manner eliminates any differences that might arise due to server-side non-determinism [136].

First, the *"Dynamic - Static"* line in Figure 5.2 shows that a static crawler fails to fetch 32% of bytes on the median page. This is because, on a modern web page, which resources are served to a client are often determined only when the client executes the scripts included on the page. Since a static crawler can identify the URLs of a page's resources only by parsing the source code of the page, it is blind to such resource fetches. Figure 5.3 shows an example.

9297.js

```
var EA = fetch("crazyegg.com/usnews.com.json")
// json contents:  {
       script_url: "crazyegg.com/commonscripts/759.js"
}
```

utag.js

```
const n = document.createElement("script");
n.src = EA.script_url;
const r = document.getElementsByTagName("script")[0];
r.parentNode.insertBefore(n, r);
```

Figure 5.3: **Snippet of JS code from www.usnews.com. The browser first fetches a JSON file, and then requests a JS file referenced inside the JSON.**

Second, Figure 5.2's *"Static - Dynamic"* line shows that, on the median page, the static crawler fetches 93% more bytes than fetched by Chrome; on some pages, this overhead is as high as 200x. These extra resource fetches arise because, within a single page, web developers often embed resources that are applicable across a large number of client device types, expecting the client browser to download the resources applicable to it. Examples include multiple resolutions of the same image, or different font files for the same HTML text. To enable the client to pick the appropriate version of any particular resource, modern pages either use media queries [47] or CSS selectors [19]; see Figure 5.4 for examples. A static crawler is unable to evaluate media queries and does not know which CSS selectors are dynamically applied during JavaScript execution. Therefore, it offers no control on whether to fetch only resources applicable to the machine used for crawling a page or to fetch every resource that might be requested in any load of the page.

5.2.3 Compute overheads of browser-based crawling

Given the shortcomings of static crawlers, state-of-the-art web browsers are often employed to crawl pages [6, 11, 10]. I observe that Chrome is the most widely used in browser-based crawling frameworks because of its better support for web APIs [51] and for automation capabilities [14]. For this reason, in the rest of this paper, I refer to Chrome[1] when discussing overheads of browser-based crawling.

I observe that the average number of pages that I could crawl per second with Chrome

[1]I use Chrome in a headless mode as it is known to be more compute efficient [193, 75].

index.html

```
<picture>
    <source srcset="ct.img/600x338" media="(min-width:768px)">
    <source srcset="ct.img/400x225" media="(min-width:0px)">
</picture>
```

style.css

```
.icon-calendar
      {font-family: {src: url("fonts/icomoon.woff")}}
```

widget.js

```
if (body.firstChild.hasAttr("data-widget")){
       var inode = document.createElement("i");
       inode.class = "icon-calendar";
       body.firstChild.insertBefore(inode)
}
```

Figure 5.4: **Code snippet from www.chicagotribune.com showing the two causes for a static crawler's extra resource fetches. (a) It will fetch both versions of the `ct.img` image, irrespective of the width of the client device's display. (b) It will fetch the font file `fonts/icomoon.woff`, whether or not the CSS selector `.icon-calendar` is used in the rest of the page. The CSS selector is only added if the HTML code contains a `data-widget` attribute.**

was only 12% of that achievable with the static crawler. The cause for this significant drop in crawling throughput is shown in Figure 5.5, which plots the average utilization of CPU, network, and disk with either crawler. Unlike the static crawler, which was limited by network bandwidth, the dynamic crawler ended up saturating all CPUs. If I were to use a 10 Gbps network, more than 5000 CPU cores would be necessary for the dynamic crawler to fully utilize the network, which is infeasible to accommodate on a single server.

I break down the reasons behind Chrome's high CPU usage using data from Chrome's in-built profiler [13]. I find three primary contributors: 1) the JavaScript engine, which is responsible for parsing and interpreting JS code, 2) inside the rendering engine, computation of the layout tree which specifies on-screen positions for page content, and 3) time spent inside Chrome's internal code, into which the profiler has no visibility. Together, these three sources of computation account for 96% of the compute delays on the median page, with JavaScript execution alone accounting for about half. Given the complex inter-dependencies between these three tasks, none of them can be simply eliminated to reduce Chrome's com-

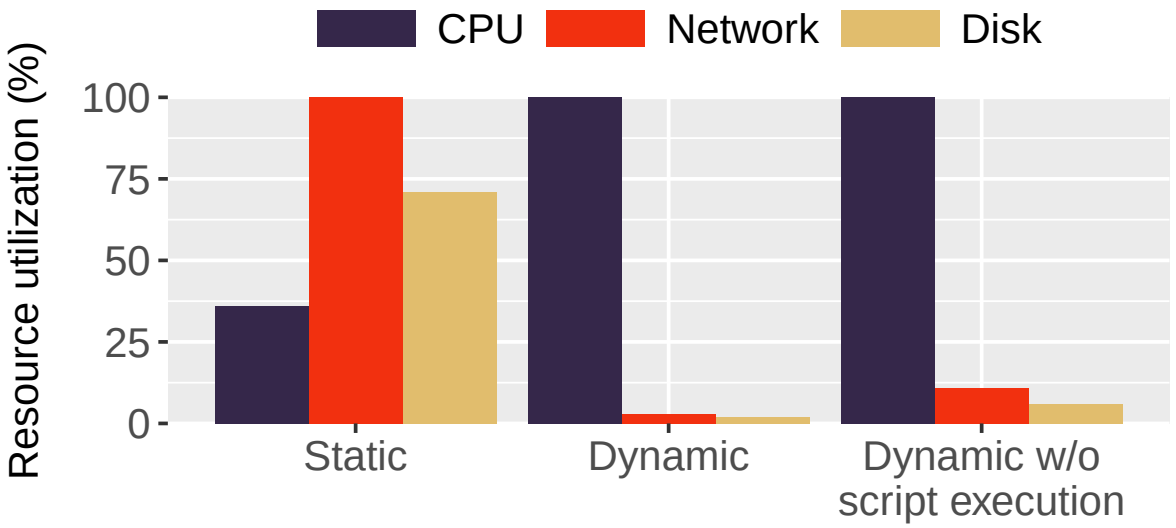


Figure 5.5: **A comparison of average CPU, network, and disk utilization by static and dynamic crawlers.**

putation overheads. For example, JavaScript execution queries layout information when scripts inspect the position of elements on the screen.

5.2.4 Minimizing browser's computation delays

The observation that the amount of client-side computation needed to load a web page has increased in recent times is not new. A large body of prior work [210, 165, 149, 164] has focused on addressing the impact of this overhead on user-perceived web performance. However, those solutions have little utility in the context of web crawling for two reasons.

First, many proposals for reducing the impact of client-side computation on page load times aim to either increase the overlap between the browser's use of the client CPU and network [164, 187] or parallelize the browser's execution of scripts on a page [149]. Such solutions can reduce the end-to-end latency of individual crawls, but the total amount of computation that the crawler needs to perform, and thus the crawling throughput, will remain unchanged.

Second, others [165, 210, 4] rely on server-/proxy-side support to ship processed versions of pages so as to minimize the amount of JavaScript that clients need to execute. Notwithstanding the fact that such solutions are not usable until they are adopted by millions of domains, I estimate their best case utility by crawling pages in $Corpus_{10k}$ with script execution in Chrome disabled. A comparison of *"Dynamic"* and *"Dynamic w/o script execution"* in Figure 5.5 shows that the latter marginally reduces the gap between CPU and network uti-

lization. However, client-side computation remains a significant bottleneck, thereby limiting crawling throughput to still be only 17 pages per second.

Alternatively, one could attempt to build a lightweight browser from scratch which only supports crawling, but does not enable users to visit web pages, i.e., has no graphical interface, does not support user interactions, etc. However, significant engineering effort would be required to constantly keep up with updates in HTML, CSS, and JavaScript APIs. For example, when I load the landing pages of the top 1000 Alexa sites using a version of Chrome from five years ago (v65), it fails to fetch 16% of the resources fetched by the most recent version of Chrome (v114). This is because certain JavaScript APIs that are commonly used today were not supported by Chrome v65, e.g., support for optional chaining [38] was only added in v80. It would be best for web crawlers to rely on widely used browsers which are well-maintained, instead of having to replicate the effort in a lightweight browser dedicated to crawling.

5.3 Overview

The takeaway from the previous section is that, today, operators of web crawlers are stuck with having to choose between two less than ideal options: use static crawlers and miss out on some resources, or make do with the poor performance of dynamic browser-based crawlers. I seek to resolve this quandary by enabling high-fidelity crawling at high throughput. I do so while respecting two constraints. First, I make sure to crawl all the resources on a page that a browser would fetch, but make it configurable whether to crawl only the resources relevant to the machine on which the crawler is executed. Second, to make my crawler compatible with the legacy web, I require no changes to web pages and the servers that host them.

5.3.1 Observations and approach

The high-level observation that guides my approach is that, on any site, there typically is significant overlap across pages both in the JavaScript code that they include and JavaScript-initiated fetches when a browser loads them. Figure 5.6 demonstrates this property on the pages in $Corpus_{10k}$.

First, for every site, out of all the unique JS files seen on at least one of the 100 pages on that site, I compute the fraction which are included in multiple pages; here, I consider the combination of a file's URL and a hash of its source code to be a unique identifier for a file. The *"Same source"* line plots the distribution of this fraction across the 100 sites in my corpus. For the median site, 72% of JS files were shared across multiple pages.

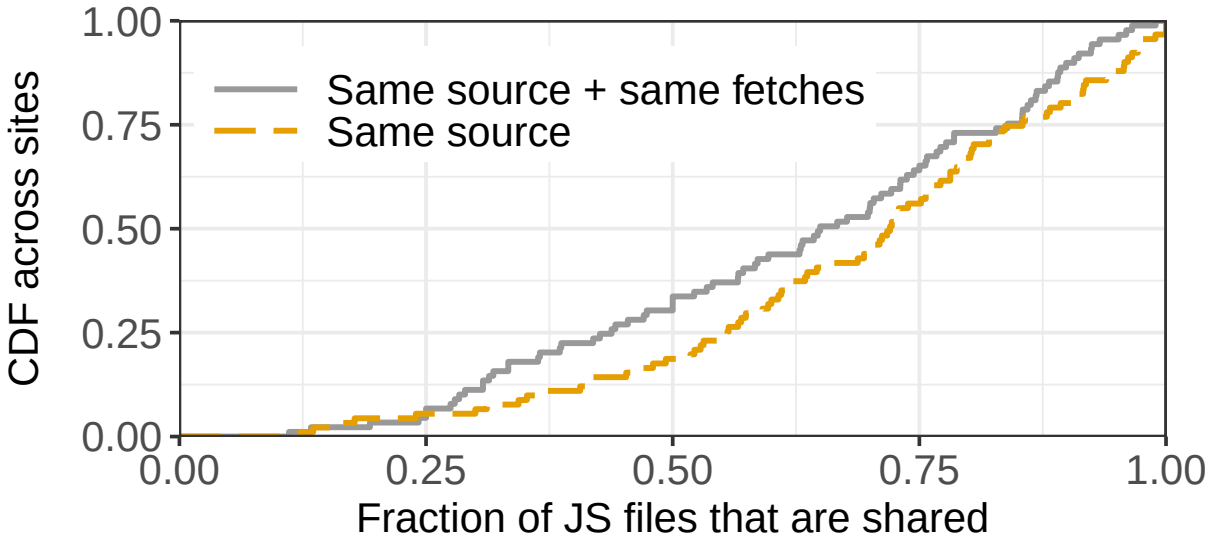


Figure 5.6: **For the sites in $Corpus_{10k}$, most JavaScript files appear on multiple pages and a script typically fetches the same resources on all the pages which include that script.**

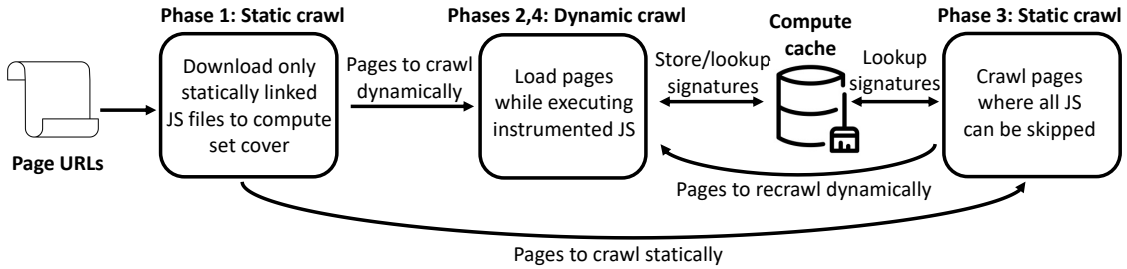


Figure 5.7: **Sprinter crawls the pages on any site in four phases which alternate between browserless and browser-based crawling.**

Next, I examine the likelihood that a JS file fetches the same set of resource URLs when it is executed on different pages. For this, I consider a script file's execution uniquely by the file's URL, the hash of its source, and the set of URLs it fetches. When I consider only those executions which result in at least one fetch, the *"Same source + same fetches"* line in Figure 5.6 shows that, on the median site, 65% of unique file executions – at least with respect to resource fetches – are repeated across multiple pages.

The takeaway from these observations, coupled with the property that web crawling workloads typically crawl a large number of pages per site (§5.2.1), is that there exists significant redundancy in a dynamic crawler's execution of JS files. When the browser used by the crawler executes a JS file that it had previously executed on a different page on the same site, the numbers from Figure 5.6 indicate that the same set of resources are often requested as on the previous page. The browser's network cache will ensure that it does not

have to waste network bandwidth in re-downloading those resources. But, the browser will still execute every JS file in its entirety just to identify these resources.

To improve crawling performance by reducing the crawler's computations, our approach aims to first eliminate redundant execution of JS files. Specifically, whenever my crawler, Sprinter, crawls any page, it skips executing a JS file if a) it has already executed that file while crawling a different page on the same site, and b) it identifies that, if executed, the file will fetch the same resources as it did on the previously crawled page. However, as observed earlier (§5.2.4), a browser imposes high compute overhead even when it is used to load pages with execution of scripts disabled. Therefore, second, on pages where it can reuse the executions of *all* JS files, Sprinter does not even employ a browser to crawl those pages. Put together, Sprinter uses a browser to crawl only a small subset of pages in each site and minimizes the browser's execution of JavaScripts.

5.3.2 Challenges

Realizing the above approach requires us to answer the following three questions.

- Whenever a script appears on multiple pages, it is not guaranteed to initiate the same resource fetches on all pages; in my corpus, 48% of repeated scripts had at least one execution where they fetched a different set of URLs than what they fetched in their first execution. Prior to executing a script, how can Sprinter efficiently identify that the script's execution will match a prior execution, and it is safe to skip executing it?
- Classic memoization involves storing the results of execution and using them to skip future executions of the same code in the same runtime context. In contrast, when Sprinter crawls a page without a browser, how can it reuse the browser's prior computations on other pages? Mimicking the entire browser runtime will significantly increase complexity and degrade performance.
- Finally, on each site, which subset of pages should Sprinter crawl using a browser? To minimize Sprinter's compute overheads, it is key that the subset be small. However, for Sprinter to crawl all the remaining pages on the site without a browser, I must ensure that the script executions on the pages crawled using a browser suffice to skip executing all the JS files on the remaining pages.

5.4 Design

As shown in Figure 6.10, Sprinter crawls a corpus of pages from any particular website in four phases. In the first phase, Sprinter identifies the subset of pages that need to be crawled with

vendor.js

```
var gKey = window.grumi.key; // “bfd2-4adc”
fetch(`https://www.geoedge.com/${gKey}/grumi-ip`)
var NYTD = {PageViewID:‘mubjhislka7867’};
window.NYTD = NYTD;
```

$\text{Signature}_{\text{vendor.js}}$

```
{
      Reads: [“window.grumi.key”,”bfd2-4adc”],
      Writes: [“window.NYTD”,”{PageViewID:‘mubjhislka7867’}”]
      Fetches: [“https://www.geoedge.com/bfd2-4adc/grumi-ip“]
}
```

Figure 5.8: **JavaScript code from www.nytimes.com which reads a global variable using the *window* object and, based on the property read, fetches a URL. It also writes to the window object. Signature for this includes the global state read and written (both the keys and the values) and the fetches initiated.**

a browser. It crawls those pages in the second phase while skipping JS executions whenever feasible. Next, Sprinter crawls the remaining pages on the site using its augmented static crawler. Finally, it recrawls some of the pages from the third phase using a browser. I present my design of Sprinter by first describing its operation in phases 2 (§5.4.1) and 3 (§5.4.2), and lastly, phases 1 and 4 (§5.4.3).

5.4.1 Memoizing JavaScript execution

Sprinter maintains a *compute cache* in order to take advantage of the opportunities to reuse JS executions across the pages on a site. On any page that Sprinter crawls with a browser, prior to executing JS on the page, the browser looks up the compute cache to determine whether the execution can be skipped. Upon a cache miss, the browser executes the JS code and logs a summary of its execution in the compute cache, for use on other pages.

Execution signatures to enable reuse. When JS code runs within a browser, it can read from or write to the JavaScript heap and HTML DOM object. It can also read the return values from various web APIs. Therefore, to enable reuse of JS executions without violating correctness, we associate the execution of every block of JS code with a *signature* which includes the values at the start of executing that block of code for all state from the heap or DOM that is read within that block. When the browser executes any block of JS code, its execution is guaranteed to result in the same externally visible effects (i.e., writes to

the DOM and heap, and URL fetches) as a prior execution which had the same signature, if the block does not invoke any non-deterministic APIs (e.g., `Date`, `Random` or `Performance`). Figure 5.8 shows an example block of code and the corresponding signature.

However, to construct code signatures, modern browsers provide no APIs to extract the necessary runtime information about JavaScript execution. To remedy this, Sprinter uses a custom JS instrumentation framework, similar to the ones used in prior work [112, 149, 165]. This instrumentation framework runs inside a man-in-the-middle (MITM) proxy which sits in front of the browser. For every new JS file requested by the browser, the proxy statically analyzes the code in the file and rewrites it by injecting code that tracks the state and APIs that are accessed when the file is executed. Sprinter's instrumentation tracks variables on the heap which are in either 1) the global scope, which is accessible using the `window` object, or 2) the closure scope, which is created within a function but persists after the function's execution if there exists a nested function declared in the same enclosed scope. For the DOM object, Sprinter tracks all APIs that can read from (e.g., `getElementById`) or write to (e.g., `appendChild`) the DOM.

Once the browser finishes loading a page, Sprinter's injected JS code compiles signatures for the scripts on the page and stores them in the compute cache which is co-located with the proxy. These signatures include both the above-mentioned information needed to identify the opportunity for reuse, and the writes to the heap and DOM that need to be executed when the corresponding code is skipped, along with any fetches initiated; see Figure 5.8. When a previously cached JS file is fetched in future page loads, the proxy embeds stored signatures for the code in this file directly into the file. When processing each JS file, the browser uses the embedded signatures to determine if any code within the file can be skipped.

Granularity of JS execution reuse. Given my results from §5.3.1, it is natural to try and reuse the browser's JS executions at the granularity of entire files, i.e., prior to processing any script file, the browser uses cached signatures for that file to determine whether to skip all the code in the file or execute all of it. However, as shown in the *"Full signature"* line in Figure 5.9, the cache hit rate is pretty poor. On the median site in $Corpus_{10k}$, less than 40% of JS file executions can be skipped.

To improve the hit rate, my key insight is that, unlike in user-facing page loads, I do not need to restrict Sprinter's skipping of a JS file's execution only when it is guaranteed to execute in a manner *exactly* identical to a previous execution of the same file. Rather, as long as I can guarantee that the code will fetch the same resource URLs, I can skip it. A crawler does not need to preserve other aspects of JavaScript execution, such as visual changes by modifying the DOM or functional changes by adding event handlers that allow users to interact with the page.

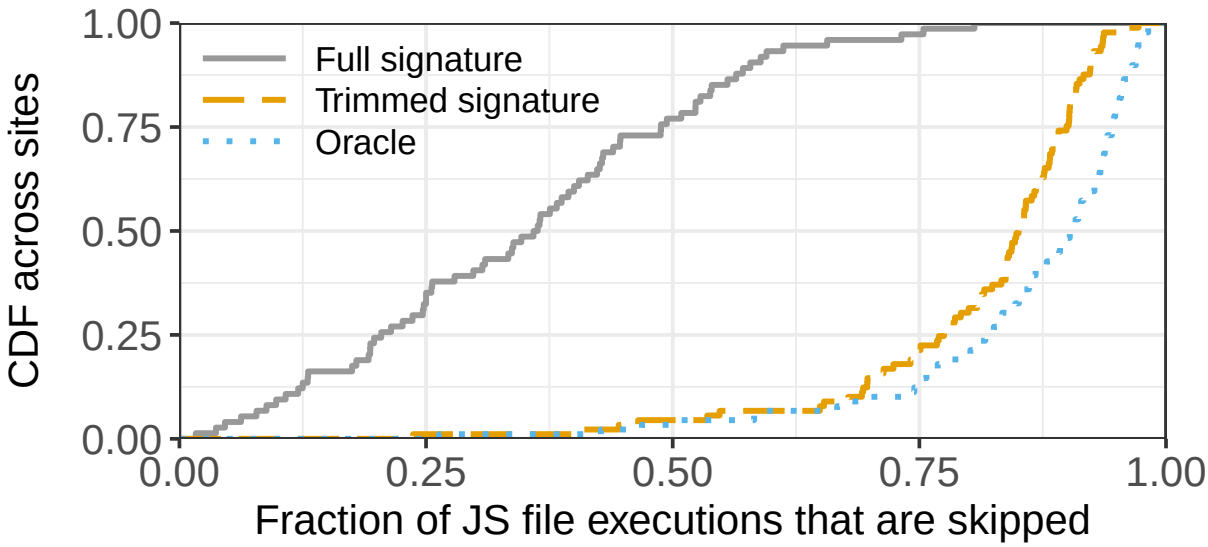


Figure 5.9: **Cache hit rate for JavaScript files that initiate fetches for other URLs.**

This observation enables us to trim file signatures and only include state that influences resource fetches. To identify this state, I turn to dynamic taint tracking [189]. Our instrumentation of any JS file marks all statements that initiate URL fetches (such as `XMLHTTPRequest.send`) and all DOM nodes with a *src* property as sinks. I also mark all control-flow statements as sinks. At the end of any file's execution, I include in the file's signature only those reads which propagate values to any of the sinks.

The *"Trimmed signature"* line in Figure 5.9 shows that trimming the signatures stored in Sprinter's compute cache improves the cache hit rate on the median site to over 80%. This is because a large fraction of reads performed by JS on the web does not influence the set of URLs fetched. Furthermore, I see that the cache hit rate with Sprinter is close to the best achievable hit rate, which I obtain via post-hoc analysis of JS executions to identify when the set of URLs fetched matched a prior execution. The gap between *"Trimmed signature"* and *"Oracle"* is due to Sprinter's conservative tracking of all control-flow dependencies, instead of only the ones that influence the URLs fetched.

5.4.2 Statically crawling pages

So far, I have discussed how Sprinter reuses execution across pages. However, as mentioned in §5.2.3, JS execution is only a part of the total compute overhead of web browsers. To maximize Sprinter's performance, I now discuss how it crawls pages without a browser in phase 3.

Crawling without a browser. I observe that the primary utility of crawling a page

within a browser is its implementation of the JavaScript heap and the DOM object, and its support of various APIs. However, if I are able to skip executing a file, I only need to compile the read state in its signature, for which I need a log of all the writes performed by previously executed or skipped JS files. I do not need to apply these writes to the browser's heap and DOM since there are no user interactions at the time of crawling.

Based on this insight, Sprinter's static crawler maintains a shadow heap, which is a key-value map from the properties of the heap to the corresponding values. It also maintains a shadow DOM, which it constructs by parsing the page's HTML at the start of every page load and offers the same read and write APIs as the ones provided by the browser.

For every page that it statically crawls in phase 3, Sprinter fetches the page's HTML, extracts all embedded resource URLs, and recursively fetches them. For every JS file fetched, the static crawler looks up the shadow heap and DOM to construct the file's signature. Upon a successful cache hit, Sprinter logs the writes included in the file's signature to the shadow heap and shadow DOM, and issues any resource fetches included in the signature. It repeats this process until all resources on the page have been fetched. Whenever there is a cache miss for a JS file, the static crawler is unable to execute the file, and it defers these pages for browser-based crawling in phase 4 (§5.4.3).

Handling additional fetches. Crawling pages as described above has the downside of fetching additional resources that a browser would not (as described in §5.2.2). For $Corpus_{10k}$, this increases the total number of bytes fetched by 3.5x. Unlike during dynamic crawling, when the network is severely underutilized (Figure 5.5), these additional fetches significantly degrade overall throughput when crawling without a browser.

If the input configuration to Sprinter specifies that only the resources relevant to the machine executing the crawler be downloaded, its static crawler does so by leveraging the browser's processing of pages crawled earlier in phase 2. First, during every page load executed within a browser, Sprinter adds to its compute cache the media queries evaluated and the corresponding value (true or false). For any media query encountered during browserless page loads, the static crawler fetches the corresponding URL if the compute cache either returns a true value or does not contain any entry for that media query. Similar to my observation of similarity in JS executions across pages, I find that, for the median site in $Corpus_{10k}$, 92% of all media queries occur on more than one page. Second, the static crawler uses the cached signatures for JS files to identify which selectors were applied when the browser executed those files. It fetches only the URLs contained within these selectors.

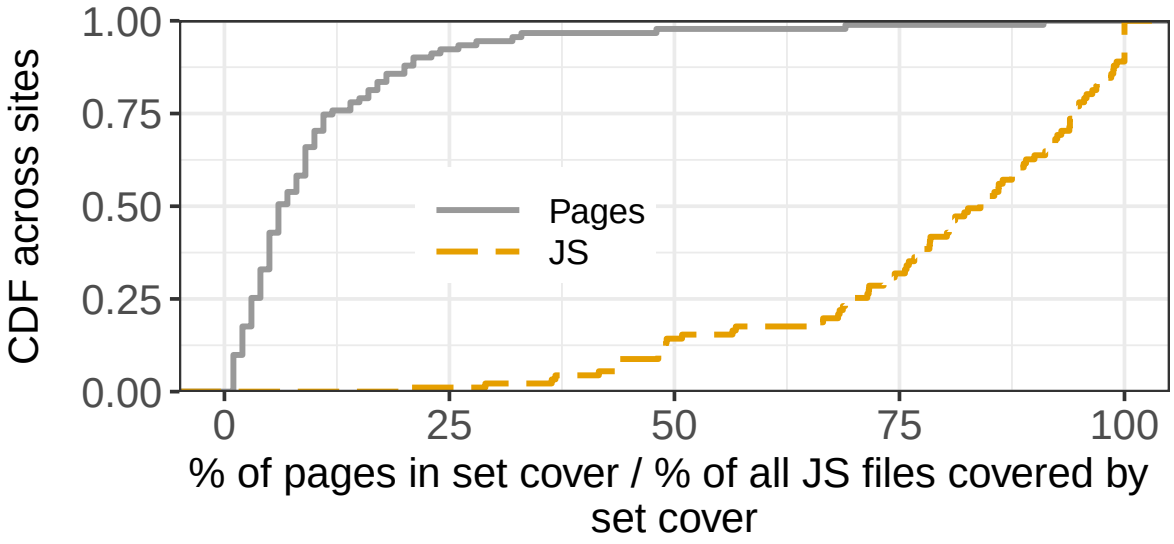


Figure 5.10: **Approximate set cover captures a large fraction of JS files ("JS"), while the number of pages in the set cover ("Pages") are a small fraction of the total corpus size.**

5.4.3 Scheduling page crawls

Given the high compute overhead of loading pages in a browser and extracting signatures, I must minimize the number of pages that Sprinter crawls using a browser. However, Sprinter can crawl a page without a browser only if it is able to skip executing *every* JS file on that page. Hence, the subset of pages on any site that Sprinter crawls without a browser in phase 3 should ideally be such that all of the JS files that appear on any of these pages also appear in at least one of the pages previously crawled with a browser in phase 2. This does not guarantee that the static crawler will find a compute cache entry with a matching signature for every JS file, but at least makes it possible.

Need for scheduling. Since the set of JS files on any page is not known apriori, Sprinter could use a browser to crawl the pages on any site in a random order and switch to browserless crawling once the set of JS files converges, i.e., once the union of JS files remains unchanged for n consecutive pages crawled. But, I find that there is no value of n that offers a good tradeoff between compute overheads and coverage of JS files. For example, with $n = 2$, I would need to crawl only 8% of pages on the median site in $Corpus_{10k}$ with a browser, but only 49% of the JS files seen across all the pages on this site appear on those pages. With $n = 10$, the fraction of JS files covered by browser-based loads increases to 82%; however, 38% of pages now need to be crawled using a browser.

Efficient identification of set cover. Sprinter takes an alternate approach of carefully

selecting which subset of pages on each site to crawl using a browser in phase 2. Though I cannot predict which JS files are on the remaining pages, I leverage my finding from §5.3.1 that the same JS file often fetches the same resources across pages of a site. Therefore, instead of finding a subset of pages that includes *all* the JS files used on that site, I find a subset that includes all the JS files that are statically embedded in the remaining pages. When these files are executed as part of the browser-based loads, all the JS files that are dynamically fetched on this site's pages will likely be fetched and executed.

Thus, in phase 1, Sprinter crawls all pages using a static crawler which only fetches the JS files that are directly linked. I then have a set of JS files for every page, and Sprinter computes the set cover, i.e., the subset of sets whose union matches the union of all sets. Since computing the optimal set cover is NP-complete [103], I use a greedy heuristic which runs in polynomial time and is known to closely approximate the optimal [198]. Figure 5.10 shows that, on the median site in $Corpus_{10k}$, Sprinter selects only 7% of pages to be crawled using a browser, yet these pages cover over 80% of all the JS files seen across all pages on the site.

Given this methodology for choosing which pages to crawl with a browser in phase 2, there are multiple reasons why Sprinter's static crawler may not find a matching compute cache entry for every JS file that it fetches. First, since the set cover is only based on statically linked JS files, some dynamically fetched JS files may not have been encountered in the browser-based loads. Second, even for files that were executed by the browser, those executions may not have had the same signature as that expected by the static crawler. If the static crawler runs into either issue on any page, Sprinter recrawls that page using a browser in phase 4.

5.5 Implementation

Our implementation of Sprinter has three components, which work together as shown in Figure 5.11.

Dynamic crawler. Sprinter's dynamic crawler is written in 1150 lines of NodeJS code. This dynamic crawler allows Sprinter to load pages using Chrome in phases 2 and 4. To automate Chrome, the crawler uses the Puppeteer library [40]. At the end of every page load, it uses Chrome's DevTools protocol [14] to collect runtime information, and it then sends the per-file signatures that it compiles to the man-in-the-middle (MITM) proxy.

Static crawler. The static crawler, used in phases 1 and 3, is written in 920 lines of Golang. It uses the goquery library [25] to create a virtual DOM for every HTML file and the cascadia library [12] to parse and query CSS selectors.

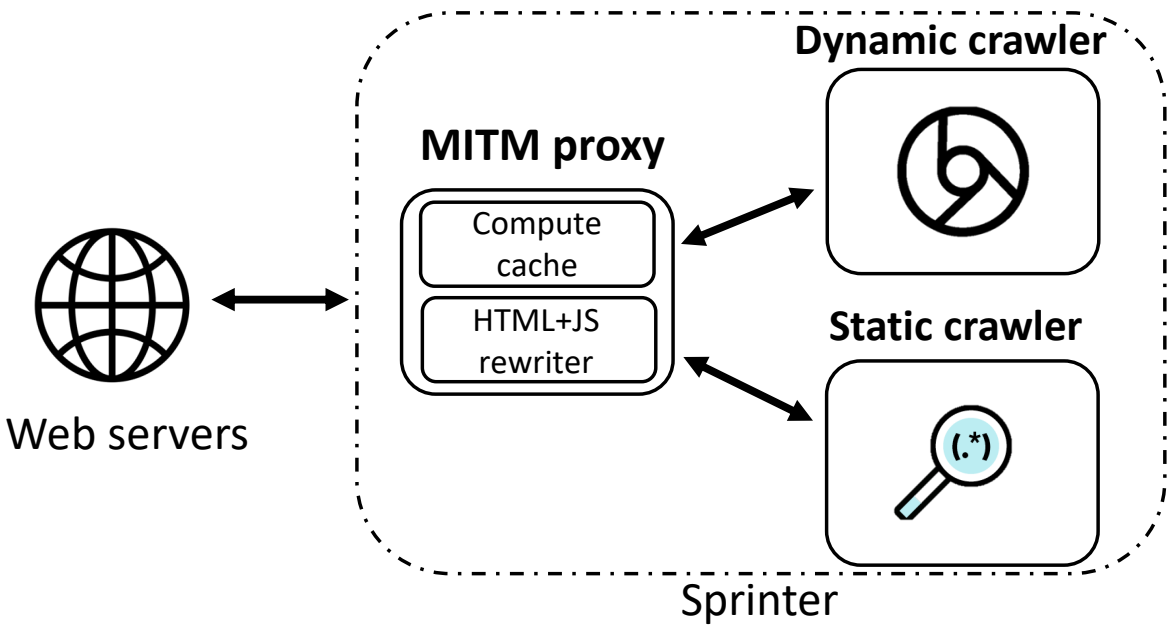
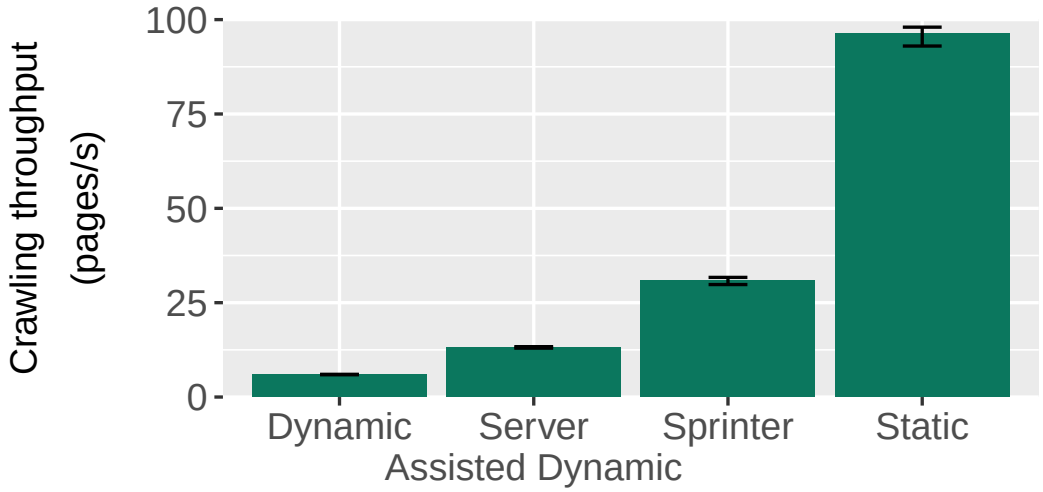


Figure 5.11: **Overview of my Sprinter implementation.**

MITM proxy. The MITM proxy is an HTTP proxy written in 350 lines of Golang. It intercepts requests and responses for every resource fetched by the static and dynamic crawlers. The proxy also statically analyzes and instruments JS files, for which it uses a static analyzer written in 1200 lines of NodeJS code. The static analyzer uses Babel [7], a JS transpiler, to create the abstract syntax tree for every JS file. While instrumenting JS files, the static analyzer enables tracking of the JS heap and DOM. Even though all web APIs should ideally be tracked since their return values can potentially influence URL fetches, my evaluation shows that the subset of APIs that my implementation currently supports – all values accessible from the `window` object directly, e.g., `window.navigator.userAgent` – largely suffices. Our finding is in accordance with prior work which studied the impact of web APIs on URL fetches [136]. The proxy also runs a gRPC server to receive signatures from the dynamic crawler.



(a)

(b)

Figure 5.12: **Comparison of (a) crawling throughput and (b) fidelity of Sprinter against the three baselines.**

5.6 Evaluation

I evaluate Sprinter with respect to the fidelity with which it crawls pages and its performance in terms of crawling throughput. I also estimate the effort that would be required to maintain its implementation over time as web APIs evolve. I compare it to various options that exist for web crawling today. Our key findings are as follows:

- When compared to dynamic browser-based crawlers, Sprinter improves crawling throughput by 5x while preserving over 99% of the bytes fetched.
- Even in comparison to prior web accelerators that rely on assistance from web servers to reduce in-browser computations when loading pages, Sprinter delivers 2.4x higher crawling throughput.
- Sprinter's performance benefits significantly improve as more pages are crawled per site, e.g., its crawling throughput increases by 2.1x when the number of pages per site goes up from 100 to 500.
- Sprinter's performance improves as the same corpus is crawled repeatedly over time, e.g., the second crawl of my corpus is 78% faster than the first run 1 week earlier.

5.6.1 Evaluation setup

Workload. I expand my corpus of pages to include 500 randomly sampled pages in each of 100 sites. This new corpus, which I refer to as $Corpus_{50k}$, retains the same properties as $Corpus_{10k}$: diverse set of sites, and representative of real-world crawling workloads in having a large number of pages per site. As described in §5.2.2, I crawl every page in this new corpus using a custom crawler which fetches the resources downloaded by either dynamic or static crawlers, and I use a record-replay tool [15] to record all request-response headers along with the corresponding payloads.

Hardware configuration and crawling methodology. I store the recorded pages in an SSD drive of a Linux server which hosts 450 web servers to concurrently service HTTP requests with the appropriate recorded content. The crawlers run on a different Linux server with a 16-core 2.1 GHz Intel Xeon CPU, 128 GB RAM, and a 1 Gbps Ethernet connection to the server housing recorded pages. Crawling performance in this setup matches what I see when crawling pages from the live web, but eliminates the impact of any server-side effects on my evaluation of performance and fidelity.

Baselines. Our primary baselines represent existing static and dynamic crawlers. For the static approach, I port wget2, a popular open-source crawler, to be compatible with my proxy-based setup; I verified that my static crawler is identical to wget2 in terms of fetched content. For the dynamic approach, I first considered three popular open-source crawlers: Archivebox [6], Browsertrix [10], and Brozzler [11]. However, my benchmark results for each revealed substantial performance drawbacks, likely because their primary goal was high fidelity, not necessarily high throughput. Specifically, undue overheads stem from spawning a new browser instance for each crawled page, using a CPU-intensive MITM proxy, and relying on an outdated Chrome automation framework. Therefore, I instead built

an in-house Chrome-based crawler that achieves 20%, 33%, and 250% higher throughput than Archivebox, Browsertrix, and Brozzler, respectively. I verified that my custom crawler fetches the same set of resources as Archivebox when used to crawl the landing pages for the 100 sites in $Corpus_{50k}$.

Our third baseline is representative of prior server-/proxy-assisted solutions to reduce client-side computations in user-facing page loads [165, 210]. To the best of my knowledge, none of these systems are open sourced, and we are unaware of any domains that have adopted these techniques. Therefore, to evaluate Sprinter against this prior work, I consider the best case outcome of these systems, where *all* client-side JS execution is eliminated. I mimic such a scenario by using my Chrome-based crawler to crawl a version of every page wherein I include links to all the resources fetched by JS files in the page's main HTML. The browser loads this modified HTML with JS execution disabled. I refer to this baseline as *server assisted dynamic* crawling.

Metrics. I measure the crawling throughput of each crawler as the average number of pages it can crawl per second on a single server. For each crawler, I run a sufficiently large number of instances so as to saturate either the CPU or the network. I expect crawling throughput to linearly increase with the number of servers. I run 5 trials for each experiment and plot the median value, with error bars plotting the minimum and the maximum values.

I consider the default goal of crawling to be to match a Chrome-based crawler. Therefore, I measure the fidelity offered by a crawler as the fraction of bytes it fetches of all the resources fetched by Chrome when crawling the same pages. When the goal is to crawl all resources that are relevant to any client device, I measure fidelity as the fraction of bytes fetched out of the union of the resources fetched by the static and dynamic crawlers.

5.6.2 Throughput and Fidelity

5.6.2.1 Comparison with baselines

To compare Sprinter with the three baselines, I load pages in $Corpus_{50k}$ using each of the four crawlers separately. I monitor the resources fetched by each crawler on every page, and the total time taken to finish crawling the entire corpus. I also monitor the CPU and network utilization to identify the bottleneck for each crawler.

Figure 5.12(a) plots the crawling throughput achieved with each crawler, and Figure 5.12(b) shows the fidelity achieved.[2] Static crawler achieves the best crawling throughput by far of 96 pages per second. However, it misses out on 37% of the bytes fetched by the

[2]I see no variation across runs in the resources fetched by each crawler because all of my crawls rely on one snapshot of every page crawled from the live web.

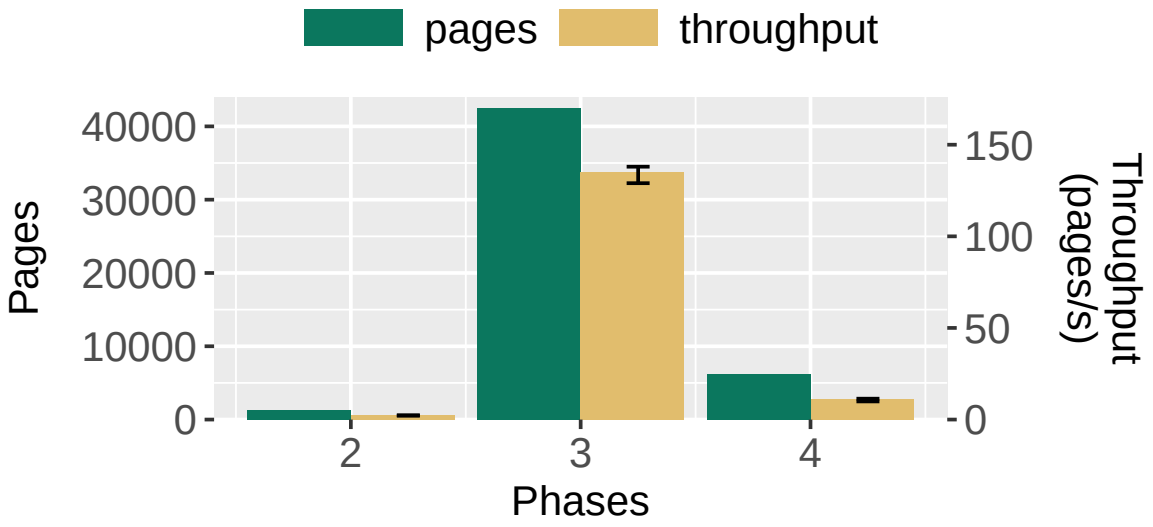


Figure 5.13: **Number of pages crawled during each of the different phases of Sprinter and the corresponding throughput achieved in each phase.**

dynamic crawler. In contrast, the dynamic crawler could only crawl at a rate of 6 pages per second. Since CPU utilization was at 100% throughout the entirety of the crawl with the dynamic crawler, throughput increased to 13 pages per second with the server-assisted dynamic crawler, which does not execute any JS.

Sprinter offers a significant additional speedup, improving crawling throughput to 31 pages per second, a 5x improvement relative to the dynamic crawler. Importantly, it does so without requiring any changes to the web and while preserving 99.2% of the bytes fetched by the dynamic crawler. The 0.8% of bytes that went unfetched stem from the incomplete support for all web APIs in my current implementation. 50% of these unfetched bytes correspond to JavaScript files, 27% to images, and 17% to HTMLs, with the remaining accounted for by CSS and other content types. While no resources went unfetched on the median page, the 90$^{\text{th}}$ percentile page was missing 1 resource.

5.6.2.2 Throughput in each phase

Sprinter's crawling throughput varies widely across phases. Figure 5.13 plots the number of pages crawled in each phase and the corresponding throughput. Whereas, Figure 5.14 shows a timeline of how Sprinter's crawling of the pages in $Corpus_{50k}$ proceeds over time.

- No page is fully crawled in phase 1; Sprinter only statically crawls the HTML files and embedded JavaScript files for every page so as to identify the subset of pages to be crawled with a browser in phase 2. Therefore, phase 1 finishes in 151s, the quickest of all four phases.

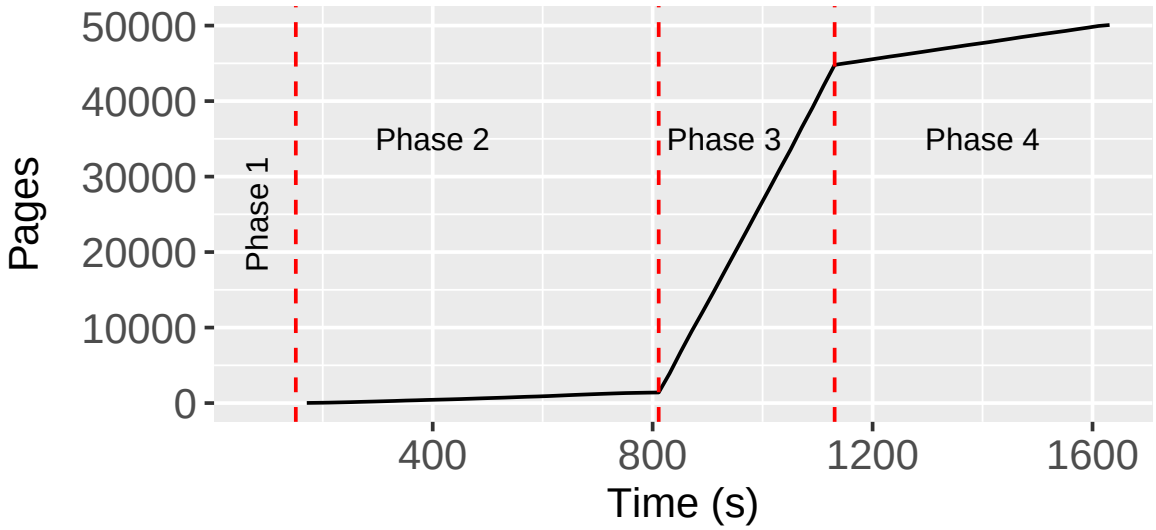


Figure 5.14: **A timeline of Sprinter's crawl of *Corpus*$_{50k}$, showing the duration and number of pages crawled in each phase.**

- Phase 2 is the slowest since Sprinter not only has to crawl pages with a browser, but it also has to incur the overheads of statically analyzing and rewriting every JavaScript file, executing these instrumented files inside Chrome, and processing the information it collects to generate and store per-file signatures. In this phase, Sprinter crawls 1413 pages in 620s, resulting in a crawling throughput of a little over 2 pages per second.
- Sprinter crawls the vast majority of pages in phase 3: 42497 pages in 316s. The average throughput of 135 pages per second in this phase is even higher than what a static crawler can achieve (96 pages per second, as shown in Figure 5.12(a)). This is because, unlike a static crawler, Sprinter leverages browser-based execution of media queries and CSS selectors in phase 2 to eliminate fetches of resources relevant only for other client types.
- In Phase 4, Sprinter recrawls the remaining 6090 pages with a browser; about a quarter of these are because they contained a JS file not executed in phase 2, and the remaining pages incurred at least one compute cache miss. The crawling throughput of 11 pages per second in this phase is better than in phase 2 because significantly fewer JS files need to be instrumented.

At the end of phase 4, Sprinter's compute cache had 3089 entries. The cache hit rate of 95.6% is the key enabler of Sprinter's throughput improvements as it could crawl a large fraction of pages in phase 3, without requiring a browser. I cannot further reduce the total crawl time by immediately spawning a browser to crawl any page that incurs a cache miss in phase 3 because both phases 3 and 4 are bottlenecked by the CPU.

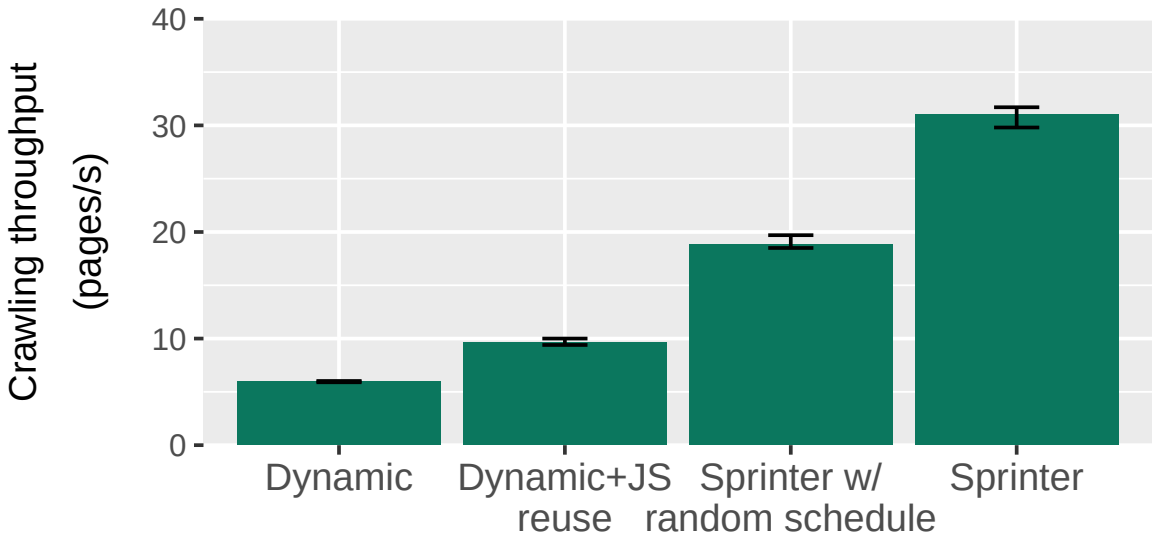


Figure 5.15: **Incremental benefit offered by each of the techniques used in Sprinter.**

5.6.2.3 Contribution of techniques

To understand the performance benefits of each of the techniques used in Sprinter, I incrementally add them to the dynamic crawler and measure crawling throughput.

First, I evaluate the benefits of only using JS memoization (§5.4.1) in Chrome, loading all pages in the corpus in a random order. Figure 5.15 shows that *"Dynamic+JS reuse"* provides a roughly 66% speedup over *"Dynamic"*.

Next, I crawl some of the pages with a browser and the rest using Sprinter's augmented static crawler (§5.4.2). To determine which pages to crawl using a browser, I consider the strawman approach (§5.4.3) wherein I transition to browserless crawling once the union of JS files remains unchanged for n consecutive pages. For $Corpus_{50k}$, I observe that $n = 25$ results in browser-based loads fetching the same fraction of all JS files as that covered by Sprinter's chosen set cover. Even this unsophisticated combination of dynamic and static crawling – *"Sprinter w/ random schedule"* in Figure 5.15 – roughly doubles the crawling throughput.

Finally, by efficiently choosing a carefully chosen subset of pages to crawl with a browser, Sprinter crawls 88% fewer pages using a browser in phase 2, resulting in a further 1.6x improvement in throughput.

5.6.3 Sensitivity to crawling parameters

I evaluate the impact of the following three configuration parameters on Sprinter's crawling throughput: 1) the number of pages crawled per site, 2) the time gap between repeated

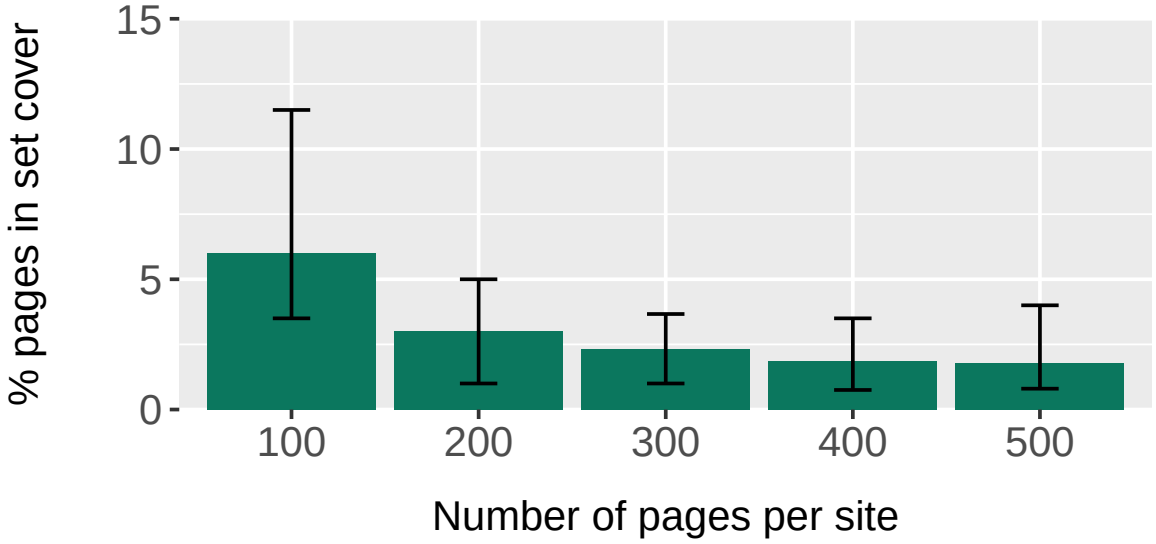


Figure 5.16: **Percentage of pages selected by Sprinter for browser-based crawling as a function of number of pages crawled per site. Bars show value for median site, with error bars for the 25th and 75th percentiles.**

crawls, and 3) whether fetching all statically embedded resource URLs is desired.

5.6.3.1 Number of pages per site

The key to Sprinter's high crawling throughput is its judicious partitioning of pages, crawling a small fraction using a browser and the remaining without. I examine how the fraction chosen for browser-based crawling varies as a function of the number of pages being crawled per site. For 5 different values of the number of pages per site, Figure 5.16 plots this fraction for the 25th, median, and 75th percentile sites. The percentage of pages in Sprinter's carefully selected "set cover" for the median site goes down from 6% with 100 pages per site to 1.6% with 500 pages per site. As a result, Sprinter is able to crawl a corpus of 10k pages at an average rate of 15 pages per second. But, for a 50k page corpus, its throughput improves to 31 pages per second (Figure 5.17). Akin to how a static crawler benefits more from network caching with more redundant resource fetches, Sprinter's compute cache enables it to reuse more client-side computations when it crawls more pages per site.

On the flip side, lower the number of pages per site, lower Sprinter's throughput. Figure 5.17 shows that, with 10 pages per site, Sprinter crawls 4 pages per second on average, which is slower than the dynamic crawler. For Sprinter to offer any benefit, I see that it must be asked to crawl at least 20 pages per site. As a result, workloads that only crawl landing pages of sites [17] will not benefit from Sprinter.

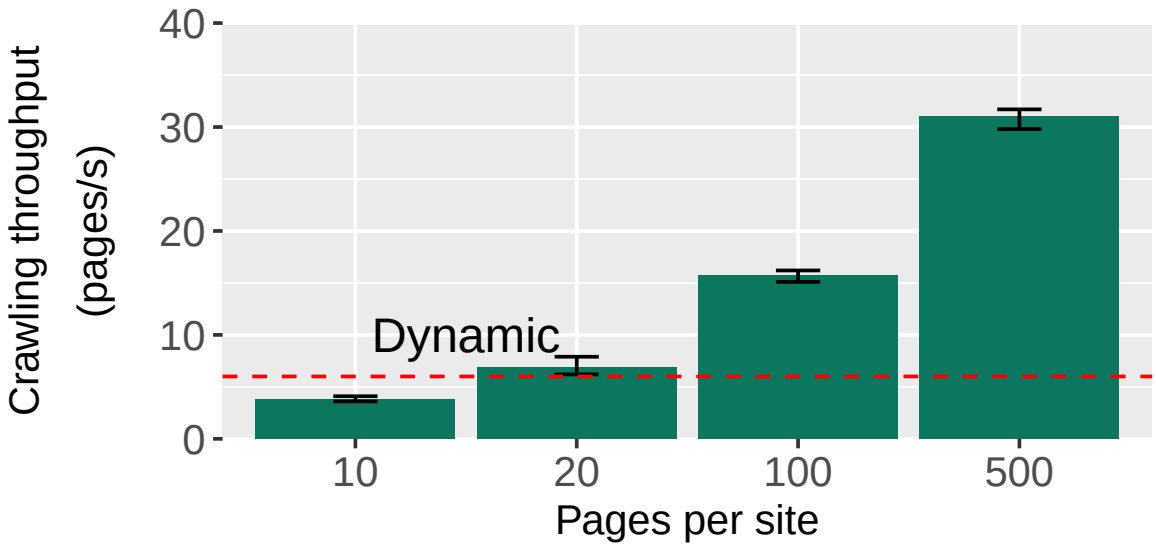


Figure 5.17: **Sprinter's crawling throughput as a function of the number of pages per site.**

5.6.3.2 Repeated crawling

In many web crawling workloads, the same corpus of pages is repeatedly recrawled. For example, a web search engine must ensure that its search index reflects the latest content on every page, and web archives must track changes to page content over time. In such cases, Sprinter will crawl the entire corpus in 4 phases the first time. However, when the corpus is recrawled, Sprinter can directly jump to crawling pages statically in phase 3, leveraging JS execution signatures from the previous crawls. Pages where no compute cache entry was found for at least one JS file would have to be recrawled with a browser in phase 4.

To measure the crawling throughput with Sprinter when the same corpus is crawled multiple times, I recrawl $Corpus_{10k}$ once three weeks after our initial crawl, and again a week later. I then use Sprinter in my replay setup to crawl pages from my last copy of the corpus. I run Sprinter once starting with an empty compute cache, once using signatures from the crawl a week before, and once using signatures from the crawl a month before.

Figure 5.18 shows that reusing signatures from a week ago improves Sprinter's throughput by 78% as compared to when no prior crawl existed. Reusing month-old signatures also speeds up Sprinter. But, since the compute cache entries are more stale and more previously unseen JS files are fetched, the benefits are significantly lower.

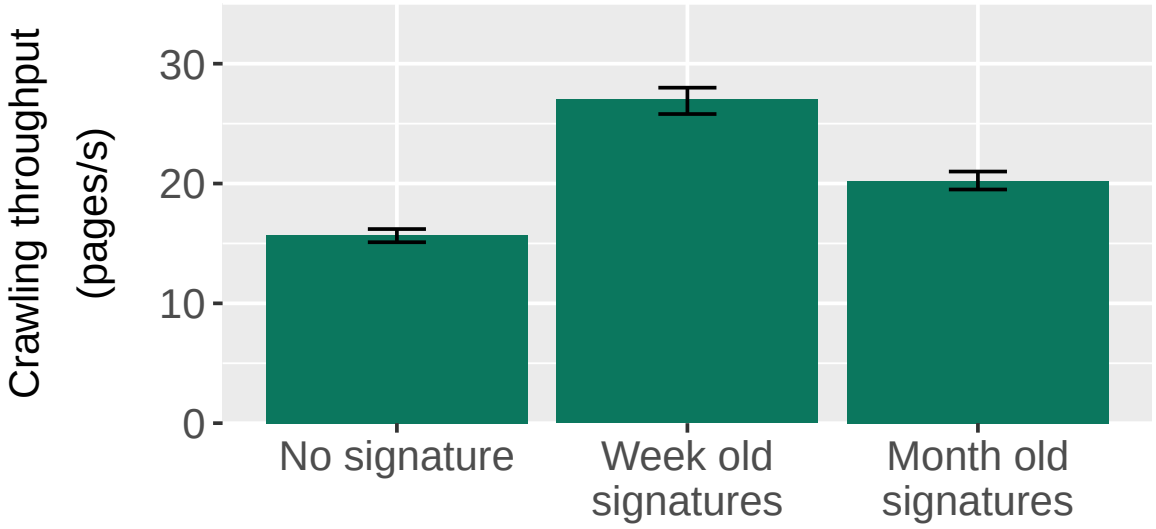


Figure 5.18: **Sprinter can crawl pages faster by leveraging signature information from previous crawls of the same corpus.**

5.6.3.3 Preserving static fetches

Thus far in my evaluation, I have considered the goal of crawling to be to fetch the same resources on every page as a dynamic crawler. However, in some cases, it might be desirable to also crawl all resources that would be fetched by a static crawler. For example, web archivists might want to preserve all versions of every image on a page, so as to be able to accurately render the preserved page irrespective of the client device used to visit this page in the future.

In these cases, Sprinter can be configured to not eliminate fetches using the techniques mentioned in §5.4.2. The resultant throughput of Sprinter drops to 28 pages per second which, though 9% lower than when it only tries to match the dynamic crawler, is still 4.6x faster than the dynamic crawler. This drop in throughput is because of Sprinter's static crawler having to fetch additional bytes in phase 3.

Note that the impact of this configuration option on Sprinter's throughput depends on the number of pages crawled per site. With more pages per site, phase 3 is able to achieve higher crawling throughput due to the benefits of network caching.

5.6.4 Maintainability

Web APIs and their specifications are constantly updated [142]. Web crawlers need to be correspondingly updated over time to ensure that web pages using the latest APIs are accurately crawled. Dynamic crawlers leveraging web browsers such as Chrome and Firefox

Chrome version	Lightweight browser		Sprinter	
	# of APIs added	# of files added/-modified	# of APIs added	LOC added
v108	4	41	1	9
v109	3	70	1	13
v110	4	58	1	6
v111	7	109	0	0
Total	18	278	3	28

Table 5.1: **Comparison of number of APIs that need to handled by Sprinter and a lightweight browser.**

simply need to update to the latest version of the browser, as these browsers are well-maintained and constantly updated to support most of the latest web APIs.

To get a measure of the effort that would be needed to maintain Sprinter or a lightweight browser such as phantomJS, I look at all the APIs added in the 4 most recent versions of Chrome (v108 to v111). For each API, I manually read its specification. Only a subset of these would need to be implemented by a lightweight browser designed for the purpose of crawling, e.g., any API that takes effect only during user interactions would not have to be handled. Sprinter's instrumentation of JS code would need to keep track of an even smaller subset of APIs, only those which influence execution signatures, i.e., any API that can read from or write to the global state.

Table 5.1 compares the number of APIs that need to be tracked and implemented by Sprinter versus a lightweight browser designed for crawling. Across the four versions, a lightweight browser would be required to implement 18 APIs; in Chrome's source, these APIs touch 278 files (Chrome's commit history only shows files added/modified, not the number of lines of code). In contrast, Sprinter needs to handle only 3 of these APIs, requiring 28 lines of code.

5.7 Summary

Over the years, crawling web pages with high fidelity has evolved from a workload that is limited by network bandwidth to a CPU-intensive one. In this chapter, I showed that the key to mitigating this new bottleneck is to strategically minimize the use of the web browser and its execution of JavaScripts. My design of Sprinter does so by efficiently identifying and exploiting opportunities to safely reuse the browser's computations across the pages on any site. I hope that my work will spur a new wave of innovation in scalable web crawling, a task that underlies many important systems in today's society.

CHAPTER 6

Jawa: Web Archival in the Era of JavaScript

In this chapter, I introduce the problems faced by web archives that are trying to preserve web pages and allowing individual users to access them at a later point. I investigate the root cause for two problems in particular: storage overhead of preserving page snapshots and the fidelity issues incurred while loading the archived pages. I then describe the design of Jawa, a new design for web archives which significantly reduces the storage necessary to save modern web pages while also improving the fidelity with which archived pages are served. Key to enabling Jawa's use at scale are our observations on a) the forms of non-determinism which impair the execution of JavaScript on archived pages, and b) the ways in which JavaScript's execution fundamentally differs between live web pages and their archived copies. On a corpus of 1 million archived pages, Jawa reduces overall storage needs by 41% when compared to the techniques currently used by the Internet Archive.

6.1 Introduction

URLs are brittle pointers to information on the web. Over time, a page may cease to exist at the URL where it was originally available [138, 200] or the content available at that URL might change due to the page being modified [172, 105].

Therefore, web archives play a key role in the web ecosystem, enabling users to lookup the content that existed at any particular URL at various times in the past. Web archives are used for a wide variety of use cases, such as web-data analytics, genealogical analysis, and even as legal evidence [129]. To support these uses, a number of organizations—cultural heritage institutions, national libraries, and public museums—operate web archives to ensure long-term preservation of content on the web. A recent survey estimates that there are 119

web archives in the United States alone [104].

The largest and most popular of these archives, Internet Archive (IA), has archived over 600 billion web pages to date, storing data in excess of 100 petabytes [31]. It repeatedly crawls web pages over time and saves many snapshots of every page. For every page snapshot, IA first downloads all resources (e.g., HTMLs, CSS stylesheets, JavaScripts, images) on the page). It stores these resources after rewriting all URL references to point to the copy hosted by the archive. When a user wants to later view any stored snapshot of a page, the user's browser loads the snapshot from IA in the same manner as it would load any page on the live web.

In this paper, I argue that this modus operandi no longer suffices due to the preponderance of JavaScript on modern web pages [43, 111, 162]. Specifically, the widespread use of JavaScript hinders web archives from satisfying two of their primary objectives: 1) to capture and save as much of the web as feasible, and 2) to ensure that archived page snapshots faithfully mimic the original page.

- **Higher operational costs:** First, the total number of bytes on the median web page has more than tripled over the last decade [27]. A significant contributor to this increase has been the increased usage of JavaScript. For example, across Internet Archive's copies of the home pages of 300 randomly sampled sites, I see that JavaScript accounts for 44% of the bytes on the median page in 2020, as compared to 20% in 2000 (§6.2). Since web archives are typically run by non-profit institutions with limited budgets, needing to store more bytes per page reduces the number of pages they can crawl and archive.
- **Poor page fidelity:** The archived copies of many JavaScript-heavy pages render with missing images and improperly laid out content (§6.2.1). This occurs due to the non-deterministic execution of JavaScript; when a user loads an archived copy of a page, the resource URLs requested by the user's browser can differ from those saved by the archive when it crawled the page. Consequently, the web archive returns errors for some of the requested resources. Due to the complex dependencies between the resources on a page [207, 86, 164], one failed resource fetch often has a cascading effect on the rest of the page load.

The challenge in holistically addressing both problems is that trying to reduce storage overheads by not saving some of the JavaScript found on crawled pages risks further degrading fidelity. A web archive could statically or symbolically analyze the JavaScript code on every page to identify what subset is necessary to preserve correctness in *all* potential loads of the page. However, the computational overheads of such methods [136, 145] render them impractical at the scale of a web archive, e.g., the Internet Archive crawls roughly 5000 pages per second [211]. To jointly address JavaScript's adverse impacts on storage and fidelity us-

ing computationally lightweight methods, I observe and leverage three fundamental ways in which JavaScript's execution on archived pages differs from that on the live web.

First, a significant fraction of JavaScript is dedicated to either sending user data to a page's origin servers or processing dynamically constructed server responses, e.g., to enable users to post comments or to push notifications. Any such functionality cannot work on archived pages, and therefore, the associated code need not be stored by web archives. Fortunately, the JavaScript code on any page is typically partitioned into several files, and I find that most of the code that will be non-functional in the context of a web archive is cleanly compartmentalized into a subset of these files that exhibit identifiable patterns in their URLs. Consequently, I show that web archives can efficiently, and safely, prune unnecessary JavaScripts by relying on URL-based filters to identify and discard JavaScript source files.

Second, many lines of JavaScript code are executed only in certain control flows, e.g., when a page is loaded on a smartphone, and not on a desktop. But, among the various sources of non-determinism that dictate whether or not a specific line might get executed, some sources are absent in loads of archived page snapshots; clients maintain no state across loads and server responses for the same request URL do not vary. Moreover, a web archive should actively eliminate those sources of non-determinism which can cause clients to request different resource URLs than those crawled. Thanks to the resulting reduction in non-determinism, I find that much of the JavaScript code on an archived page will never be exercised in any load of that page, making it moot for a web archive to store such code.

Lastly, a critical use of JavaScript is to enable users to interact with a page after the page's load has completed. On live pages, identifying *all* the code used to support such interactions is generally challenging because the code that is exercised varies based on how users interact with the page. For example, the input given to a search bar determines the server's response; based on the number of search results, JavaScript for paginating the results may or may not get executed. In contrast, I find that the subset of interactions that do work on archived pages (e.g., navigational menus and image carousels) distinctly differ from those that do not with respect to the properties of the page state they access. This greatly simplifies the task of identifying the code necessary to preserve post-load interactions.

Based on my three observations, I design and implement Jawa (JavaScript-aware web archive), a system for crawling and saving web pages. Jawa enables web archives to save many more pages than they could today for the same cost, e.g., it reduces the total amount of storage necessary to store a corpus of 1 million web pages by 41%. Importantly, Jawa enables this reduction both while increasing the rate at which pages can be crawled by 39% and significantly improving the fidelity of archived pages: for the vast majority of archived

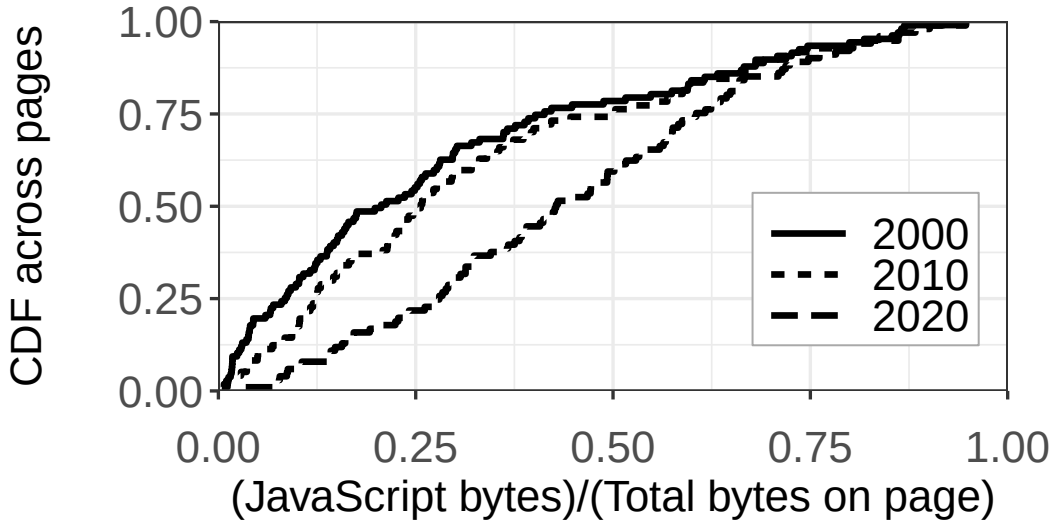


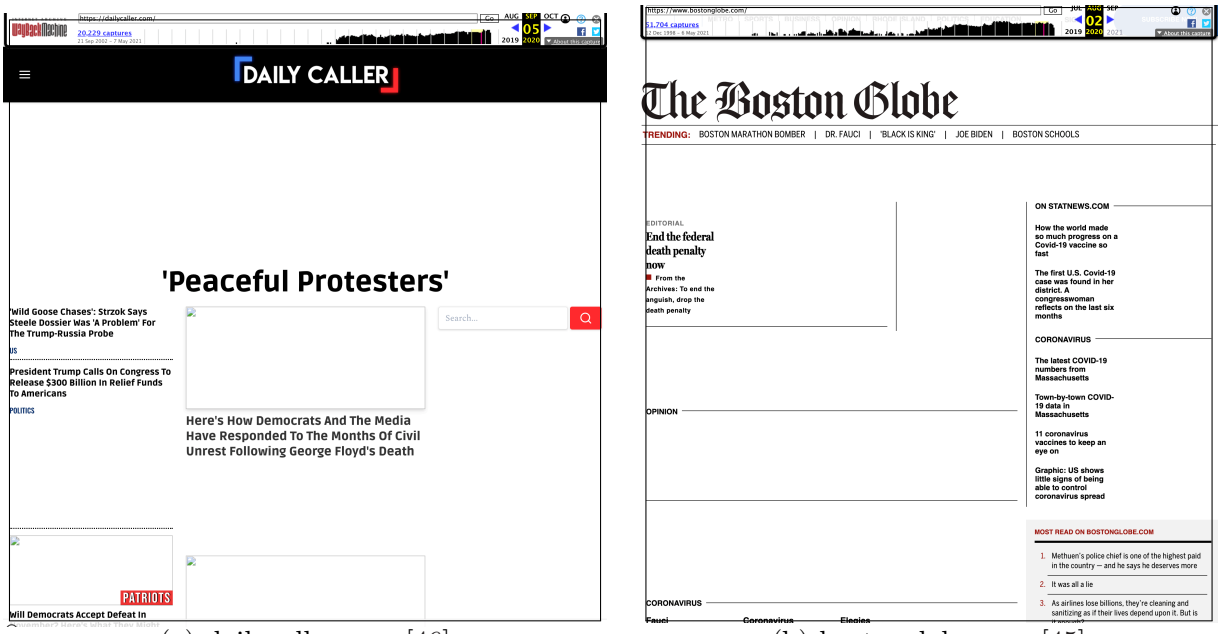
Figure 6.1: **Across the landing pages of 300 sites, distribution of fraction of bytes on the page accounted for by JavaScript.**

pages, Jawa ensures that the page is rendered in a manner identical to how it was when the page was crawled, and all page functionality that can possibly work on an archived page does work. Source code for Jawa, including scripts to reproduce the key results in the paper, are available at https://github.com/goelayu/Jawa.

6.2 Background and Motivation

As mentioned earlier, the Internet Archive (IA) is the largest and most popular web archive in the world today. For every page that it crawls, IA stores all the individual resources on that page (such as HTMLs, CSS stylesheets, JavaScript files, and images) in the Web ARChival format (also known as the WARC format [48]). Client browsers can load archived pages from IA's Wayback Machine [50] in a manner identical to how they do on the live web. When the Wayback Machine receives a request for any resource, it looks up an internal index to locate the WARC record for this resource and then responds along with relevant HTTP headers. IA rewrites all resource files so that all statically embedded URLs point to IA's web servers. For URLs which are dynamically generated via JavaScript, IA rewrites them on the fly using client-side API shims.

This architecture sufficed when IA began operating two decades ago. However, the web today is very different. In particular, JavaScript (JS) has become significantly more common. For example, Figure 6.1 shows that JS accounts for 44% of the bytes on the median page today; up from 20% in 2000. In this section, I show that this increase in JS hinders the ability of web archives to meet their two primary objectives: 1) to crawl and capture as



(a) dailycaller.com [46] (b) bostonglobe.com [45]

Figure 6.2: **Examples of page snapshots loaded from IA.**

much of the web as possible, and 2) to preserve page fidelity, i.e., when an archived page is loaded by a user, it should ideally match the page as it was crawled, both in visual (how the page looks) and functional (user interactions supported on the page) aspects.

To support my claims, in this section (and in the rest of the paper), I consider pages from 300 sites, comprising 100 randomly chosen sites from each of three ranges from Alexa's site rankings: [1, 1000], [1000, 100K], and [100K, 1M]. Using these 300 sites, I construct two corpuses. $Corpus_{3K}$ contains one of IA's copies from September 2021 for 1 landing and 9 internal pages per site. $Corpus_{1M}$ contains 3500 page snapshots for each site out of all of IA's page snapshots from September 2020. Note that both corpuses contain a mix of old and new pages. Though both corpuses contain page snapshots which were archived in the last couple of years, many of these pages were created before then. This is because IA recrawls pages over time to track changes to page content.

6.2.1 Poor fidelity due to JS non-determinism

When a user loads a web page, scripts on the page often dynamically construct the URLs for many of the resources on the page. In doing so, JS execution can leverage various sources of non-determinism: client-side state (e.g., cookies, local-storage), client-characteristics (e.g., user-agent), random number generators, etc. When a user loads an archived page, these sources of non-determinism can potentially lead to a different set of resource URLs being requested compared to what was crawled by the archive. This, in turn, leads to two significant

problems.

Failed fetches. First, IA returns a resource not found error (HTTP status code 404) for all resource URLs not stored at the archive, resulting in many archived pages rendering incorrectly. Figure 6.2 shows two examples of screenshots of page snapshots loaded from IA. In both cases, JS code on the page dynamically constructs the URLs of images to fetch by taking into account the screen size of the client loading the page. Since my client appears to differ from IA's crawler,[1] these pages end up being rendered incorrectly.

Runtime errors. Second, the execution of many scripts halts prematurely with runtime errors, which in turn leads to more resources going unfetched. I inspect the runtime logs generated by Chrome when loading the pages in $Corpus_{3K}$; specifically, the JavaScript console log and the network log. Figure 6.3 compares the number of errors seen in these logs during page loads from the web and when loading snapshots of these pages archived by the IA on the same day. Loading pages from IA results in more errors of both types. The total number of bytes that went unfetched because of these failed network requests cumulated to 5% and 45% of bytes on the median and 95$^{\text{th}}$ percentile page respectively.

6.2.2 High storage overhead

The more obvious downside of more JavaScript on web pages is that it increases a web archive's storage needs. To quantify this impact, I compute the total amount of storage required to store all the pages in $Corpus_{1M}$.Across all pages, I account for storing a single copy for every unique (resource URL, SHA-256 content hash) combination; IA applies similar deduplication to reduce storage overheads [49]. Despite the fact that scripts are often shared across pages (e.g., JavaScript libraries like jQuery are used by many sites and pages on the same site include a common set of scripts), JS accounts for 49% of all the bytes stored; resources of all other types (HTML, CSS, images, etc.) account for the remaining 51%.

Note that these numbers account for the size of textual resources such as HTML, CSS, and JS after compression. Also note that my corpus of a million pages appears to be large enough to approximate the utility of deduplication at scale because the fraction of total bytes accounted for by JavaScript plateaus after 750K pages as shown in Figure 6.4.

Overall, the fact that scripts roughly double the amount of storage that a web archive needs to deploy is concerning because web archives are largely reliant on donations to cover their operational expenses [21]. For example, IA spends around $18 million dollars each year in operational expenses and attributes over 60% of its earnings to donations [33]. Needing

[1]We look at the HTTP response headers of the archived resources to gather information about the client used by IA.

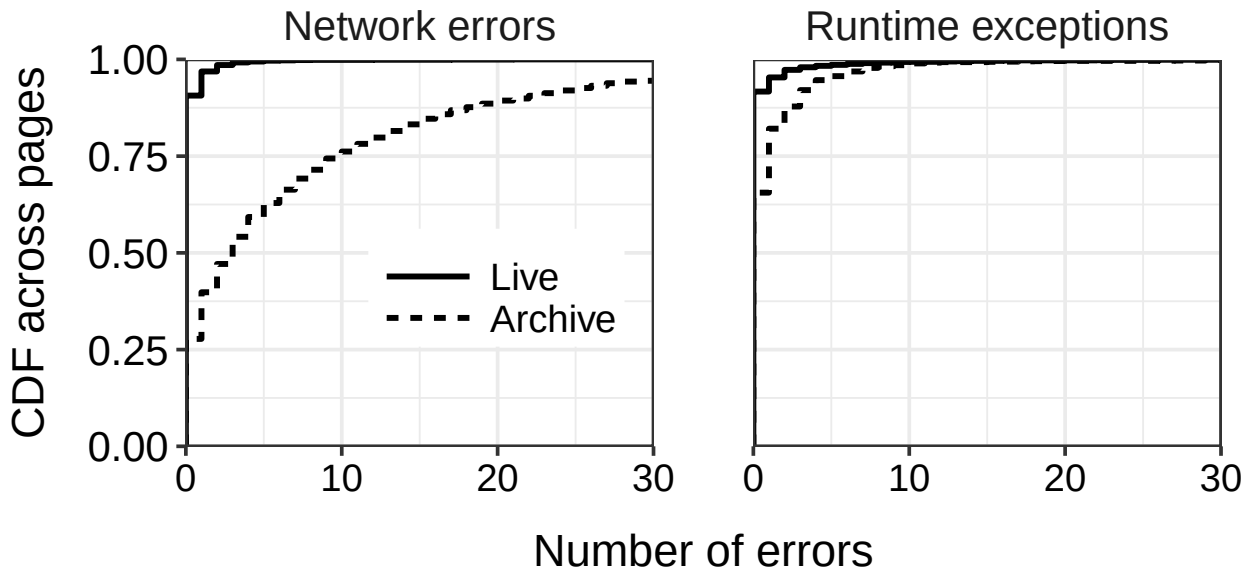


Figure 6.3: **Comparison of errors thrown during page loads from the web and from IA.**

to store more bytes per page means that an archive can store fewer pages for the same cost.

6.2.3 Downsides of alternate archival formats

To sidestep the shortcomings of IA that I have discussed thus far, a web archive could instead store and serve the end result of any page load, thereby preempting the need for clients to execute JavaScript.

Preserving post-load interactions. One such alternate archival format is to store a screenshot of the rendered page in the PNG or PDF format, as employed by private archiving institutions like Stillio [44] and PageVault [39]. However, many pages today enable users to interact with the content on the page, and storing screenshots of pages fails to preserve these post-load interactions [169]. Web developers enable such interactions by registering event handlers while a page is being loaded; these event handlers are triggered and executed when the user later interacts with the page. For example, modern pages often include carousels or sliders to display images and tabs to group information in separate categories; see, for example, the infographics on https://www.nytimes.com/interactive/2021/world/india-covid-cases.html. Event handlers are also used to enable users to navigate to other pages on the same site, e.g., the menu under the "Explore" button on https://www.coursera.org. Prior studies have shown that it is important to preserve such informational and navigational interactions even on archived pages [129].

I analyze the pages in $Corpus_{3K}$ to determine how many contain interactions that should work on archived copies. Specifically, I load every page after instrumenting all scripts so that

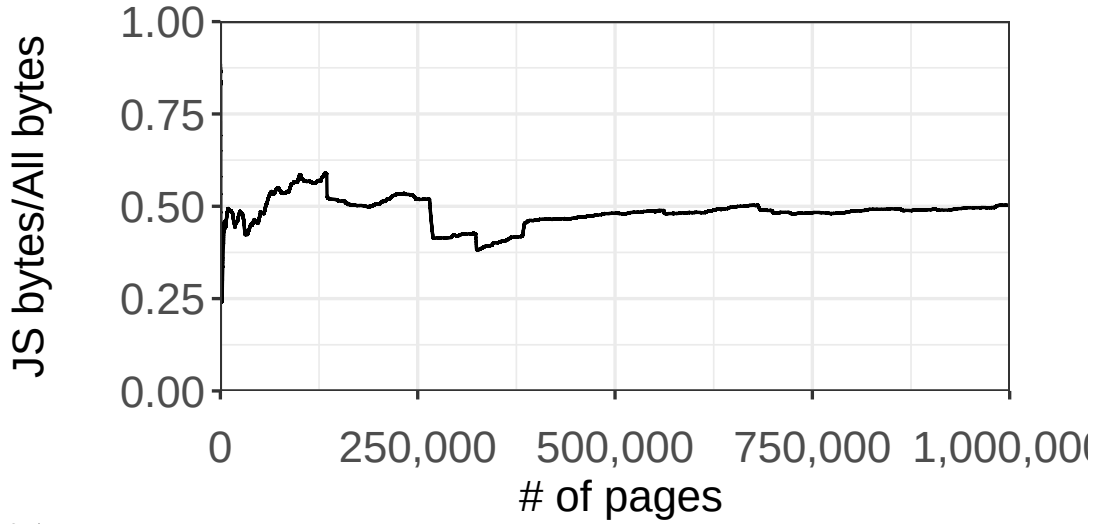


Figure 6.4: **Fraction of total bytes accounted for by JavaScript as a function of number of pages in my corpus.**

I can track all event handler registrations. I identify all event handlers which are associated with page elements whose attributes contain keywords such as menu, navbar, slider, carousel, dropdown, etc.; I consider 13 such keywords commonly associated with informational and navigational interactions. I find that 91% of the pages contained at least one such event handler.

Examples showing importance of event handlers.

Event handlers can be categorized into one of the three: 1) informational, 2) navigational, and 3) transactional. The third category of event handlers are not expected to work with archived pages since there in no back-end origin server. Figures 6.5 and 6.6 provide some screenshots of event handlers for the first two categories, which are essential for archived pages.

Overhead of capturing JavaScript heap. Alternatively, client-local interactions enabled by event handlers could be preserved by storing a) every page's final rendered HTML, b) all resources referenced from this HTML (such as CSS and images), and c) the JavaScript heap, which stores custom, page-defined JavaScript state as well as native JavaScript objects [166]. However, modern browsers do not expose the entire JavaScript heap [137]; only the global scope of the heap is accessible using the "window" object. The closure scope, which is a non-global scope that is defined by any function and is accessible only by the nested functions that execute in that function's enclosed scope [157], is not accessible. This is a key roadblock because event handlers often access closure state; 47% of the pages in $Corpus_{3K}$ contain at least one such handler (we describe how I perform the state tracking necessary to obtain this result in §6.4).

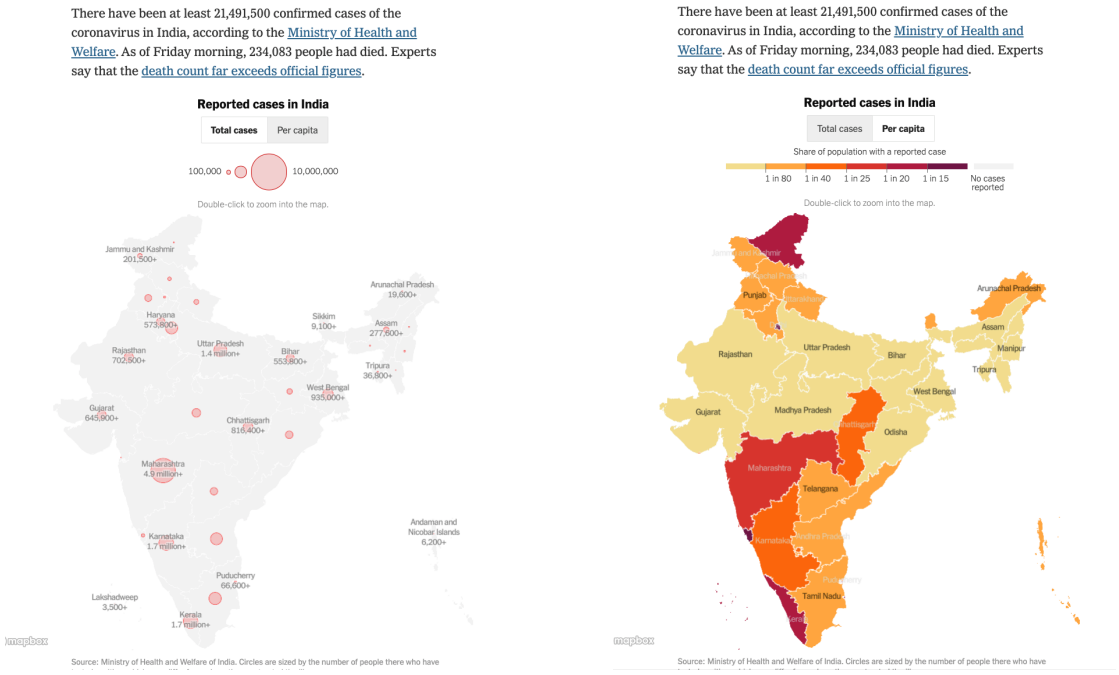


Figure 6.5: **www.nytimes.com screenshots show two different infographics, which can be toggled by clicking the button on top of the infographic.**

To access closure state, a web archive's crawler could statically analyze and rewrite the scripts on every page prior to executing them. However, I find that the combined overhead of performing the static analysis necessary to identify different scopes and running instrumented scripts inflates the time to crawl the median page in $Corpus_{3K}$ by 2x; this overhead increases to 6x at the 99th percentile. Such computational overhead will significantly increase costs for a web archive crawling thousands of pages every second [211].

Storage cost. Apart from the above shortcomings, these alternate storage formats also necessitate more complex deduplication across page snapshots which share resources but differ in the final outcome of the page load. If I use simple file-level deduplication, as is the case today in IA, Figure 6.7 shows that storing page snapshots as PNG screenshots or as DOM+Heap results in significant storage overhead for our corpus; for the median site, the former results in a 1.2x overhead in storage and the latter inflates storage needed by almost 32x.

6.3 Overview

To overcome the adverse impacts of JavaScript on web archival, my high-level insights stem from two key differences between the loads of live and archived pages. In this section, I describe these differences and outline the challenges entailed in leveraging these differences.

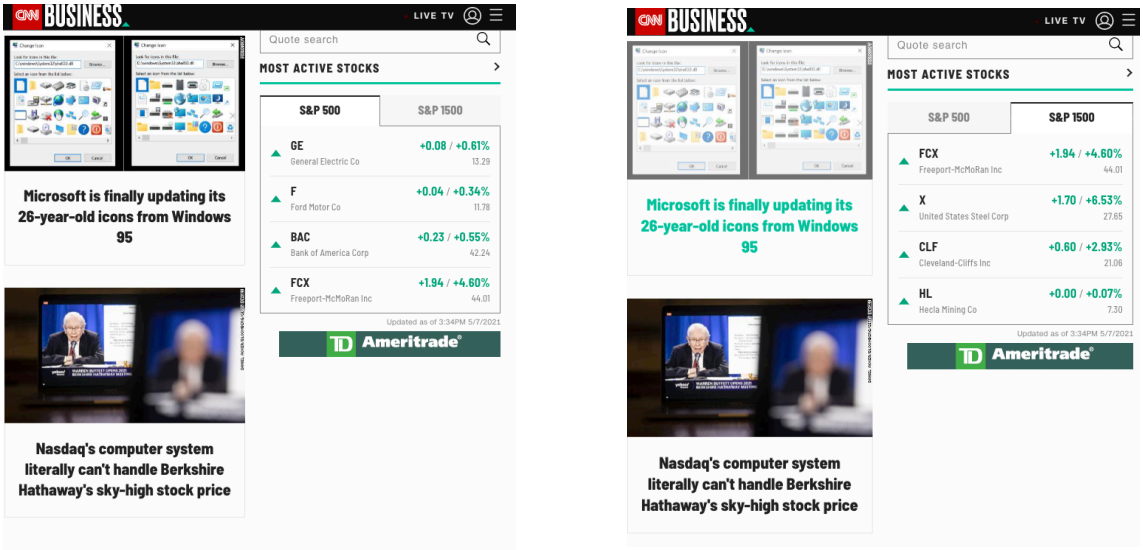


Figure 6.6: **www.money.cnn.com contains stock market information for S&P 500 and S&P 1500 which can be toggled by using the tab icon on top.**

6.3.1 Distinguishing properties of archived pages

No back-end origin server. Modern web pages include a range of functionalities which require communication with the page's origin servers, e.g., enabling users to post comments and having servers push updates to users while they are on a page. However, when a user loads an archived page snapshot, only that functionality on the page will work which can be served using the resources crawled when this snapshot was captured.

Limited sources of non-determinism. To deliver a dynamic user experience, many pages on the web adapt how they are rendered based on ① server-side state, ② client-side state (e.g., cookies, local storage), ③ client characteristics (e.g., user-agent, screen dimensions), and ④ "Date", "Random", and "Performance" APIs (we refer to these as *DRP* APIs for the sake of brevity). For example, after a script on a page fetches a JSON from the origin server, its subsequent control flow might depend on the contents of that JSON, which itself might be influenced by the contents of a client-side cookie. In loads of archived pages, the first two sources of non-determinism are absent: in response to the request for a particular resource in a specific page snapshot, a web archive will always serve the copy it fetched when crawling that snapshot; whereas, client browsers do not maintain any state across loads of archived pages.

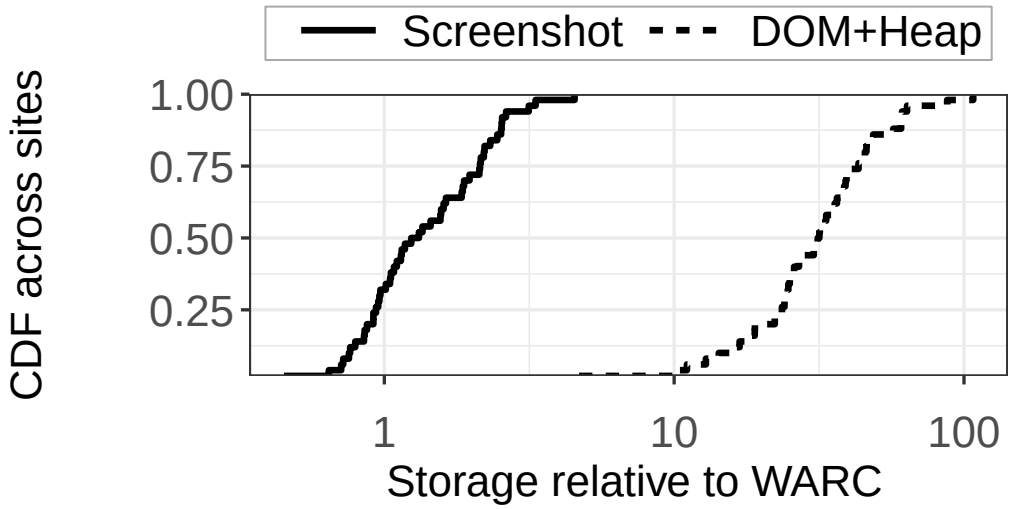


Figure 6.7: **Storage overheads of other formats**

6.3.2 Challenges

In order to leverage the above-mentioned differences to both improve page fidelity and reduce storage overhead in web archives, I need to answer several questions.

What are the causes of poor page fidelity? While some sources of non-determinism are absent in the loads of archived pages, the remaining sources – client characteristics, *DRP* APIs, and asynchronous execution of timer handlers and script fetches – still result in non-deterministic JS execution. Determining which of these factors is responsible for clients requesting different resource URLs than those crawled is key to eliminating failed resource fetches and the resultant runtime errors.

How to efficiently prune non-functional and unreachable code? In any page that it crawls, a web archive need not save any JS code that either relies on interactions with the page's origin servers or would never be executed in any load of the page (due to the absence of certain sources of non-determinism). One could potentially use methods like symbolic or concolic execution to perform reachability analysis and identify both unreachable code and non-functional code; the latter comprises code that is reachable from RPCs to origin servers. However, as reported in prior work [136, 149, 145], these methods for analyzing JS code are computationally expensive, requiring tens of minutes per page. Increasing the compute overheads of crawling to such a large extent would nullify any storage savings.

How to ensure code pruning does not hamper fidelity? While eliminating non-functional code reduces storage cost, doing so comes at the risk of inadvertently hurting fidelity. In particular, the code that is retained must function as it would if no code were discarded. Checking that any method identified for code elimination does preserve this

Goal	Observations	Section
Improve fidelity	APIs for client characteristics are the key cause for failed resource fetches	§6.4.1
	Differences in URLs due to *DRP* APIs can be resolved using server-side URL matching algorithms	
Prune non-functional code	Most of JS code which will not function on archived pages is in third-party source files, which can be identified based on their URLs	§6.4.2
	First-party scripts typically use third-party code cautiously, so that reliability of former is not dependent on availability of latter	
Prune unreachable code	*DRP* APIs typically have no impact on control flow	§6.4.3
	For event handlers associated with post-load interactions which work on archived pages, page state accessed is disjoint across handlers and user input does not influence control flow	

Table 6.1: **Overview of the main insights that influence my design of Jawa.**

property is non-trivial because browsers do not offer any APIs to extract runtime information that can be used to identify state dependencies between different scripts on any page.

6.3.3 Requirements

Based on all the considerations discussed thus far, I focus on three objectives.

- **High fidelity.** First, I seek to ensure that any archived page faithfully mimics the original page in two respects: 1) how the page is rendered, and 2) all functionality on the page which does not require communicating with the page's back-end servers works.
- **Low cost.** Second, I aim to enable a web archive to improve its coverage by reducing the amount of storage needed for any collection of page snapshots. In doing so, I seek computationally lightweight methods so as to minimize the cost overheads associated with maintaining the same rate of crawling pages as today.
- **Simplicity.** Lastly, my solutions must be simple to implement. In my discussions with the Internet Archive, they have emphasized that simplicity is key for any proposed changes to be viable in practice.

6.4 Design

I describe my design of Jawa in three parts. I begin by describing how Jawa improves page fidelity by eliminating the sources of non-determinism which result in failed resource fetches

while loading archived pages. Thereafter, I present the methods used by Jawa to identify what subset of crawled JS files need not be saved: first to eliminate non-functional code, and second to prune unreachable code while preserving post-load interactions. To enable Jawa's use at scale, the overriding principle that guides all aspects of my design is to minimize computational overheads by leveraging properties of JS typically found on the web; Table 6.1 provides an overview of my observations. Later (§6.7), I describe how a web archive which uses Jawa could potentially handle pages which do not satisfy these properties.

Analysis framework. Throughout this section, I use my custom JavaScript analysis framework (4.5K LOC) to study the properties of JavaScript found on pages in $Corpus_{3K}$. As in prior program analysis tools for JavaScript [149, 166, 111], our analysis framework first performs offline, static analysis of the JS in a page, converting each JS file into an abstract syntax tree (AST) representation. It then parses this AST to identify the different JS scope levels – local, block, closure, and global – and leverages this information to associate each JS variable to its corresponding scope. The framework also uses the AST to detect JS function invocations.

Building on these insights, my framework instruments pages with code that is triggered in each function invocation, and records the arguments to the function, all the closure and global scope variables read and written inside the function body, and the return value. Special care is taken to (1) record all accesses to the DOM, (2) track accesses of any global variable's properties via an alias, e.g., *"var a = window"* followed by a read of *"a.innerHeight"*, (3) identify DOM elements with registered event handlers and the corresponding handler functions, and (4) monitor and control the return values of browser APIs such as *"navigator.userAgent"*.

6.4.1 Improve fidelity by eliminating failed fetches

To ensure that users do not encounter failed resource fetches when they load archived pages, a web archive could rewrite every stored page to ensure that, when the page is loaded, the flow of execution and the return values of all browser APIs match those seen when the page was crawled.[2] If a web archive were to eliminate sources of non-determinism in this manner, I observe that fixing the schedule of execution cannot result in any loss of functionality; after all, developers of pages have no control over the client-side schedule of execution of asynchronous scripts. However, a page's developer can indeed ensure that code on the page behaves differently based on the results from browser APIs. Therefore, I seek to understand the impact of these APIs on resource URLs and eliminate only those sources of non-determinism which result in failed fetches during loads of archived pages.

[2]Alternatively, a web archive could crawl every page under all possible combinations of non-determinism. Doing so is not only impractical, but would dramatically inflate compute and storage overheads.

Impact of different sources of non-determinism. I measure the impact of each source of non-determinism as follows. I first load my locally stored copies of all pages in $Corpus_{3K}$ with a desktop client. I then reload these pages mimicking a different client ("iPhone 6"). Mimicking a different client allows me to exercise different values of most client characteristics, such as user-agent, screen dimensions, and OS. I reload all pages once more, this time matching the client characteristics used in the original load.

On 72% of pages, at least one different resource URL was requested in the second load compared to the first load; these two loads differ in the values for both APIs for client characteristics and *DRP* APIs. Whereas, when comparing the third load to the first, which differ only with respect to *DRP* API values, the corresponding fraction was 52%. Note that, in both cases, even one failed resource fetch can have a cascading effect, resulting in many other resources going unfetched.

Variance in resource URLs due to non-determinism results in failed network fetches only if a web archive (like IA) expects requests from clients to specify URLs which are identical to the ones crawled. However, across loads of a page, if the same resources are being requested using different URLs, it might suffice for the web archive to employ a better algorithm to match URLs requested to those crawled.

To check if this is the case, I consider two URL matching algorithms used in prior work: ① *querystrip*, where the query string in any URL (i.e., the portion of the URL beyond the delimiter '?') is stripped before initiating a match [163], and ② *fuzzy matching*, which leverages Levenshtein distance [141] to find the best match for any given URL [53]. Querystrip relies on the fact that query strings are typically used for updating server-side state, and they do not influence the content of the response. Fuzzy matching accounts for cases where non-determinism across loads results in simple string transformations of the URLs for the same resources. In any page load, I match URLs in the order they are requested, and I match any requested URL against those crawled URLs that have not already been matched.

Figure 6.8(a) shows that, on many pages, a significant fraction of URLs were unmatched with both algorithms, when APIs for client characteristics were a source for diverging URLs. This is because, when client characteristics differ, often the *number* of resources fetched on the same page changes. For example, www.nytimes.com fetches the JavaScript file *player-embedded.js* on mobile clients to enable video players, whereas it fetches no such scripts on desktop clients.

Digging deeper into *DRP* APIs. In contrast, when *DRP* APIs are the only source of non-determinism, Figure 6.8(b) shows that either URL matching algorithm suffices to eliminate almost all failed resource fetches. However, this might be the case only because I compare two loads of every page, and the return values of *DRP* API invocations did not

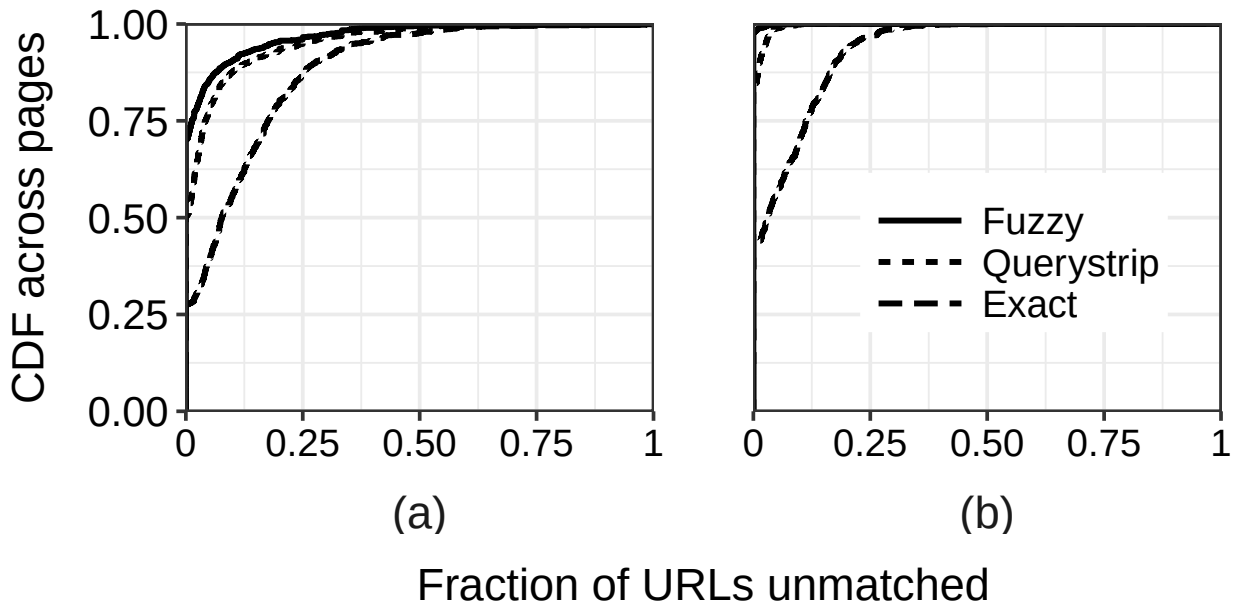


Figure 6.8: **For every page in *Corpus$_{3K}$*, fraction of resource requests which cannot be matched with any crawled resource. The impact of different URL matching algorithms is shown when the sources of non-determinism are (a) APIs for client characteristics as well as *DRP* APIs, and (b) only *DRP* APIs.**

sufficiently differ to have an impact.

To capture the effects of all possible return values of *DRP* APIs, I turn to concolic execution [110, 191, 136], a variant of symbolic execution which executes programs concretely (rather than symbolically) while ensuring complete coverage of all control flows. I modify a prior concolic execution tool [136] to only track control flows influenced by *DRP* APIs. I then randomly sample 300 pages from *Corpus$_{3K}$* because it takes around 20 minutes per page with this tool. On all pages, *DRP* APIs had no impact on control flow. Thus, comparing any two loads of a page suffices to examine the divergence in URLs across loads due to these APIs.

Takeaways. These results influence my design of Jawa in two ways. First, I instrument all scripts on any page so that, when clients execute these scripts, all APIs for client characteristics return the same values as when the page was crawled. Compared to a thin-client model where a web archive serves requests for pages by executing page loads on behalf of users [53], my approach of letting users execute page loads on their devices reduces server-side overheads. Second, I do not need to account for any differences across loads in *DRP* APIs because the impact of these differences can be accounted for with server-side matching of requested URLs to crawled URLs.

Note that I choose to patch all invocations of client characteristic APIs, and not just the ones which influence the URLs fetched. This is because, even if a particular invocation of an

API does not impact which URLs are fetched, it can impact the reachability of code which assumes that state dependent on the client's type has been setup earlier in the page load. Hence, if different API invocations return inconsistent values, this could exercise code which accesses uninitialized state, resulting in runtime errors.

6.4.2 Pruning non-functional code

I now turn my attention to reducing the storage overhead of JavaScript on web archives. Jawa's crawler uses two complementary approaches to take advantage of the two previously mentioned properties which distinguish archived page snapshots from pages on the web. The key consideration in both cases is to ensure that pruning any JavaScript code does not affect the execution of the remaining code.

Characteristics of non-functional code. Our first approach for pruning JavaScript code is based on two observations about the code which will not work on archived copies of pages, i.e., code which relies on clients interacting with origin servers. First, on a typical page, I find that most of such code is compartmentalized into a few files, rather than being evenly spread across all JavaScript source files on the page. As I will show later, these files do not contain any code that is worth preserving. Second, functionality which will not work on archived pages is largely implemented by third-party scripts. Even though some of the functionality which relies on communication with origin servers (e.g., intra-site search, login) is implemented by the first-party origin, I only focus on discarding third-party files, for reasons discussed shortly.

The implication of these observations is that, to identify most of the non-functional JavaScript code in archived pages, it is unnecessary to perform any complex code analysis. Instead, it suffices to assemble and use a "filter list" which captures the features distinctive to the URLs of scripts containing non-functional code; when crawling pages, a web archive would simply have to discard (and not even fetch) any script whose URL matches the filter list.

For example, via manual analysis of the URLs of all scripts seen in $Corpus_{1M}$, I assemble a filter list comprising 45 rules. I consider those script URLs which are included on many pages. For each such popular script, I first visit the domain on which the script is hosted to understand the services offered by that domain. In cases where a domain hosts scripts of many kinds, some of which are important to retain even on archived pages, I examine the script's content to determine its utility.

Every rule in my list matches URLs at one of three granularities: 1) domain, i.e., filter any file hosted on that domain (e.g., *"zephr.com"* enables support for user subscriptions),

2) file name, i.e., filter scripts if the file name matches, regardless of the domain hosting the script (e.g., *"jquery.cookie.js"* is used for cookie management), and 3) URL token, i.e., filter scripts if a specific keyword appears anywhere in their URL (e.g., *"pagesocial-sdk"* and *"recaptcha"*).

Recall that $Corpus_{1M}$ comprises page snapshots crawled from the Internet Archive, which already discards resources that users often block on the live web, e.g., ads. In contrast, my filter list aims to prune scripts which implement functionality that is important to preserve on the live web, but will not work on archived copies. Moreover, since a few popular third-party service providers are used by the vast majority of websites [143], I find that I only need to add 6 rules to my filter list to account for pages on 300 additional sites beyond the 300 sites included in $Corpus_{1M}$.

Filtering has no impact on fidelity. Discarding a subset of the JS files on a page might, however, break the execution of code in files that are retained. Therefore, I study the impact of filtering along two dimensions: 1) visual (i.e, does the page look the same?), and 2) functional (i.e, are post-load interactions that will work on archived pages unaffected?)

I load every page in $Corpus_{3K}$ with and without filtering enabled. I take a screenshot after every page load. Leveraging my JavaScript instrumentation described earlier, I also 1) identify all event handlers registered during each page load, 2) trigger all event handlers after the page load completes, and 3) track all values read or written from the JavaScript heap and DOM by these handlers.

First, when I compare the screenshots for every page with and without filtering, I observe that these screenshots differ in the value of at least one pixel for 109 of the 3000 pages in $Corpus_{3K}$. Upon manual examination of these 109 pages, I find that all differences are either due to animations or because *DRP* APIs result in a different timestamp on the page. Second, for all event handlers registered by the unfiltered files, I find 35 pages on which at least one value accessed by at least one of these event handlers differed across loads with and without filtering. Again, these differences were not consequential: they were due to differences in timing information, e.g., some event handlers log the times at which their execution starts and ends.

A key reason for these positive results, which show that Jawa's filtering has no impact on the fidelity of the code retained, is my explicit choice to only consider third-party source files for filtering. On the one hand, most third party scripts are self-encapsulated, i.e., the code in these files only interacts with itself or the files it subsequently fetches. On the other hand, as shown in Figure 6.9, first-party scripts typically invoke third-party code cautiously, so that the former is unaffected in the off chance that the latter fails to be fetched.

Note that one cannot simply eliminate *all* third-party scripts; that would render dysfunc-

```
<script src="https://js.sentry-cdn.com/7bc8b.min.js" </script>
<script>
        if (window.Sentry) {
            window.Sentry.onLoad(function() {
                window.Sentry.init({
                    maxBreadcrumbs: 30,
                environment: 'prd', });
            });
        }
</script>
```

Figure 6.9: **Code snippet from www.nytimes.com where the main frame first fetches a third-party JavaScript file hosted on www.js.sentry-cdn.com and then cautiously invokes a function from it inside an if condition.**

tional many post-load interactions which do work, and are important to preserve, on archived pages. As I show later in my evaluation (§6.6), while discarding files which match my carefully curated filter list enables significant storage savings, doing so preserves all navigational and informational interactions.

6.4.3 Prune unreachable code

In the Javascript files which do not match Jawa's filter list, many lines of code will never be executed in *any* page load. This is because 1) some sources of non-determinism are absent in loads of archived pages (§6.3.1), and 2) Jawa eliminates non-determinism caused by asynchronous execution and APIs for client characteristics (§6.4.1). Furthermore, I found that *DRP* APIs have no impact on control flow. Yet, identifying all reachable code remains challenging: beyond the code executed while crawling the page, I also need to retain the code for users' post-load interactions.

Challenges in preserving interactions. Post-load interactions are enabled via event handlers which are registered while a page is being loaded. Every event handler is associated with a specific DOM node on the page, and is bound to a specific action that would trigger the handler, such as a click, scroll, mouse hover, etc.

The code that is exercised when an event handler on a page is invoked can vary as a function of a) the order in which the user interacts with different elements on the page, b)

the inputs that the user provides to these events [74], and c) the return values of browser APIs. It is easy to see how the latter two can impact code reachability, e.g., in response to a search query, the number of search results can influence certain client type-specific UI features, such as the option of splitting the results across multiple pages. The order in which events are triggered can impact the execution of some handlers if the state read (from the DOM or JavaScript heap) by one handler could have been written to in a prior invocation of this or another handler. In particular, since I only care about identifying reachable code, only read-write dependencies which impact branch conditions are of interest.

I analyze the impact of these sources of non-determinism on the event handlers found on pages in $Corpus_{3K}$. I capture the state accessed by event handlers as described earlier in §6.4.2. For each event handler on a page, I check whether there is a read-write state overlap with itself or with any other event handler on the page, and if there was an overlap, whether this state is used in a branch condition. I also identify all handlers which accept user inputs; these include mouse events (e.g., click, mouseover, mouseon), keyboard inputs (e.g., keyup, keydown), and text inputs (e.g., "INPUT" or "FORM" DOM nodes). When I invoke each such handler, if a branch statement is executed, I conservatively conclude that the handler's inputs could impact the control flow of the handler.

On each page, I compare two sets of event handlers: those which work on the live version of these pages, and the subset which will work on archived copies. The former set comprises all handlers registered when I load the page without filtering. I identify the latter set of handlers by loading every page with Jawa's filtering enabled, and ignoring handlers which interact with origin servers (i.e., they are registered on either "INPUT" or "FORM" DOM nodes with a corresponding "action" attribute). On the median page, 14 event handlers work on the live page and 7 on the archived copy. At the 90^{th} percentile, the corresponding numbers are 170 and 44.

Impact of order. On 40 of the 3000 pages in $Corpus_{3K}$, at least one pair of handlers that work on the live page had a read-write dependency which could affect the control flow of one of these handlers. In contrast, I found no such case when focusing on the handlers which work on archived pages. This stark difference is because dependencies between handlers arise on live pages predominantly due to analytics, e.g., handlers registered with certain DOM nodes update locally maintained state to track if the user interacted with those nodes; when the user navigates away from the page, another handler reads this state and sends this information to back-end servers if the user did interact with those DOM nodes.

Impact of user input. Across all pages, none of the event handlers which work on archived copies read inputs which influenced branch predicates. Whereas, when I loaded all pages without filtering, 1134 of the 3000 pages had at least one handler which interacted

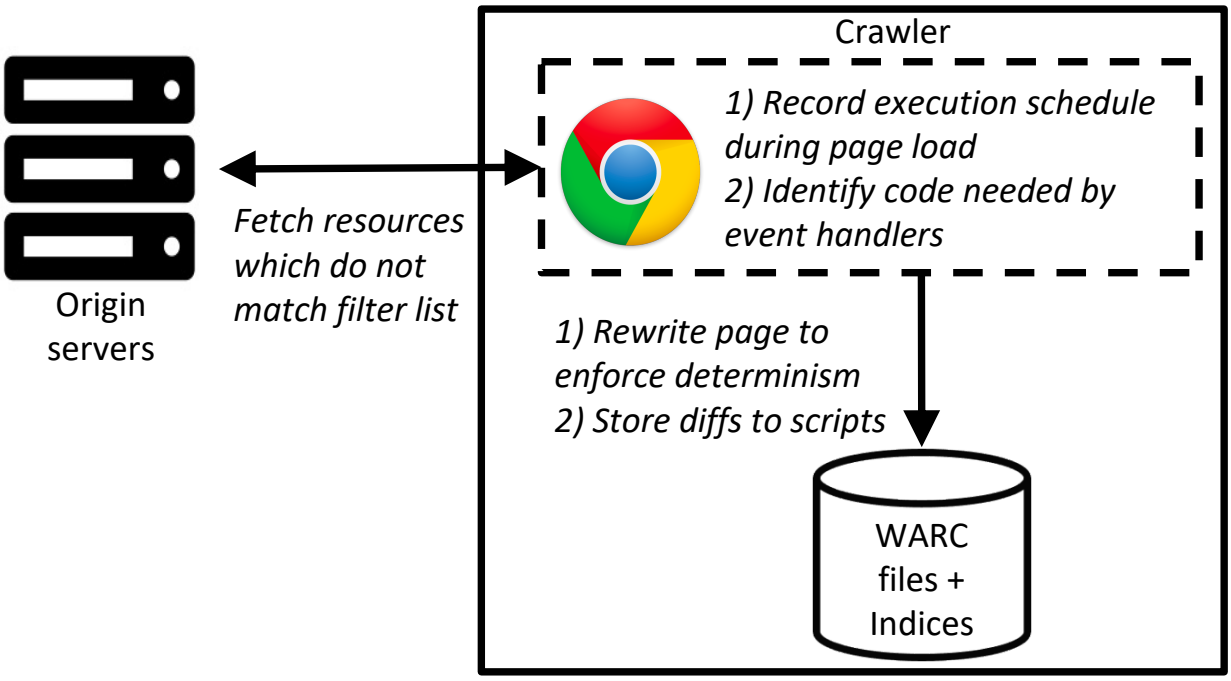


Figure 6.10: **High-level overview of Jawa.**

with back-end servers. In such cases, the responses from servers could potentially impact what code gets executed on the client.

Impact of browser APIs. As discussed earlier (§6.4.1), Jawa eliminates non-determinism caused by APIs for client characteristics. That leaves *DRP* APIs. Among the handlers that work in an archived context, 449 of the 3000 pages had at least one handler which invoked an API from either "Date" or "Math.random". All the invocations of "Math.random" APIs were due to the jQuery library assigning unique identifiers to elements inside its `ElementSelector` function [34]. Whereas, the "Date" API was used only for logging the start and end time of handler executions. Thus, in all of these cases, *DRP* APIs did not impact the reachable code for any event handler.

Takeaways. These results demonstrate why prior work which aims to identify code reachable by event handlers performs complex JavaScript program analysis [74, 188]. In contrast, I find that no source of non-determinism impacts the code executed by handlers which work on archived pages. Therefore, on any page, to identify the code necessary to retain for post-load interactions to work, it suffices for Jawa to invoke every handler once and save the code that is executed.

6.4.4 Summary

Put together, my observations on the differences between loads of archived and live pages enable Jawa to use a fairly simple methodology to crawl and save pages, as shown in Figure 6.10. For every page that it crawls, Jawa fetches all those resources which do not match its filter list. For the remaining files, it ① injects code to identify what code was executed during the page load and in what order, and ② triggers every registered event handler using default input values (e.g., the default x and y coordinates for a mouse click event is 0,0) and identifies the code executed. Finally, it stores those portions of the page that are exercised in either step above. It instruments the retained code so that, when users load the page, their browser follows the same execution schedule and uses the same client characteristics.

6.5 Implementation

Implementing a web archive involves several considerations which are outside the scope of this paper, e.g., distributing data across servers, detecting and coping with hardware failures, etc. Our implementation focuses on the aspects of a web archive addressed by Jawa (Figure 6.10), namely crawling and storing page snapshots. I also describe the impact of Jawa's design on serving page snapshots to users.

6.5.1 Crawling pages

When crawling a page, Jawa's crawler (1.2K LOC) uses a Node.js based man-in-the-middle proxy to interpose on all requests/responses. The proxy uses the Esprima [123] and BeautifulSoup [8] libraries to instrument JavaScript and HTML files as they are fetched. Jawa references the filter list for every outgoing request and, using regular expression matching, blocks the request for any resource whose URL matches any of the rules in the filter list. For all the remaining resources fetched, Jawa selectively instruments JS files prior to their execution. This instrumented code, upon execution, enables Jawa to 1) interpose on all browser APIs, 2) track the subset of JS code executed (in terms of JS functions), and 3) helps enumerate all event handlers registered on the page. The instrumentation overhead incurred by the crawler is significantly lower compared to when tracking all state accesses (§6.4).

Crawl index		
	Key	*Value*
IA	URL	List of (content hash, WARC file ID) tuples
Jawa	(URL, content hash)	List of (start byte offset, end byte offset, WARC file ID) tuples

Serving index		
	Key	*Value*
IA	(URL, timestamp)	(WARC file ID, byte offset)
Jawa	(URL, timestamp)	List of (WARC file ID, byte offset) tuples

Table 6.2: **Comparison of indices maintained by IA and Jawa.**

6.5.2 Storing page snapshots

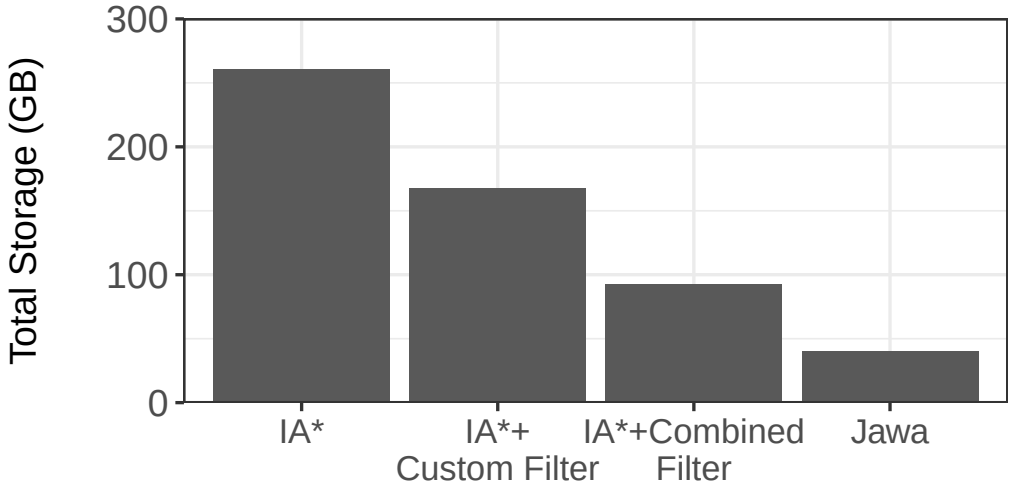
For every page that it crawls, Jawa saves only a subset of the JavaScript code on that page. Consequently, when the same JavaScript file (e.g., a library) is included on many pages, it is often the case that different subsets of this file need to be stored as part of different page snapshots, thereby preempting simple file-level deduplication, as used by the Internet Archive today [49].

Our solution is to store every unique file as a set of partitions; each partition represents a different disjoint subset of the file: from a specific start byte offset to an end byte offset. When Jawa crawls a new page snapshot, for every JavaScript file crawled that is not filtered, it identifies the subset of code in this file relevant for this snapshot. It then looks up the crawl index (Table 6.2) to determine if this subset is already covered by the byte ranges in this file that have previously been stored. The crawler creates new WARC records for portions of the file that have not been previously stored and appends new entries to the crawl index. The crawl index is processed asynchronously to produce the serving index (like is the case today with Internet Archive).

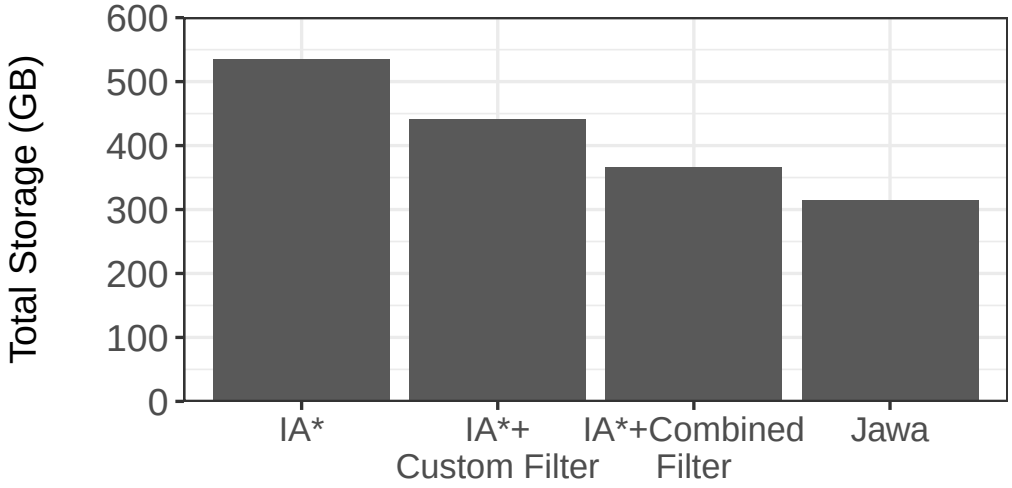
6.5.3 Serving page snapshots

The implication of storing any JavaScript file's contents as above is that, when a client requests for a file while loading a page snapshot, one does not know which of the partitions stored for this file are relevant for this particular snapshot. Instead, a web archive which uses Jawa can return the union of all stored partitions for the requested JavaScript file; after all, the portion of the file needed for any snapshot is a subset of the stored partitions. Since the size of this union is at most equal to the size of the original file, clients will have to fetch no more bytes than they do today.

6.6 Evaluation



(a) JavaScript resources



(b) All resources

Figure 6.11: **Total storage necessary to store corpus of 1 million page snapshots.**

I evaluate Jawa with three metrics: storage (to store crawled resources and to store indices), fidelity (similarity of archived page snapshots to the corresponding original pages) and performance (both for crawling and serving). In all cases, I compare against the corresponding techniques currently in use by the Internet Archive (§6.2), which I refer to as IA*.[3] In some

[3]IA* refers to me mimicking the techniques used by IA.

cases, I also break down the utility/overhead of each of Jawa's components. The key findings from my evaluation are as follows:

- Jawa reduces the storage needed for my corpus of 1 million page snapshots by 41%. This reduction stems from Jawa discarding 84% of JavaScript bytes.
- Despite this significant reduction in storage, on a random sample of pages, all event handlers that one would expect to function on archived pages continue to work.
- When I mimic loads of archived pages from IA, at least a quarter of resource fetches fail on more than 10% of pages. Whereas, on over 99% of pages, Jawa eliminates all failed network fetches and ensures that the set of resources requested from the archive match those crawled.
- Crawling throughput with Jawa improves by 39%, thanks to my use of lightweight techniques for code analysis and filtering of JavaScript files.

6.6.1 Storage

6.6.1.1 Storage for resources

To begin, I consider the total amount of storage needed to store the resources in my $Corpus_{1m}$ corpus. I crawl all of these page snapshots from IA using my crawler (§6.5). On each page, Jawa's crawler only fetches third-party JavaScripts which do not match its filter list. Apart from my manually curated filter list for pruning code which will not function on archived pages, I also leverage the open-source filter list from EasyList [22], which is widely used by many browser extensions to identify ads and analytics. In every script that it does fetch when crawling a page snapshot, Jawa's crawler identifies the subset of code necessary for this snapshot and stores the portion of this subset that is not covered by the subsets of this file previously stored.

Figure 6.11(a) shows that Jawa stores 40 GB of JavaScript across the 1 million pages, a reduction of 84% compared to IA*. Of course, to store the entire corpus, all resources on every page snapshot need to be saved, not only JavaScripts. For resources other than scripts (images, CSS, HTML, fonts), Jawa offers no storage benefits; it stores them exactly as IA*. Yet, I see a 41% reduction in total storage: 535GB with IA* to 314GB with Jawa (Figure 6.11(b)). This is because, as seen earlier in §6.2.2, JavaScript files account for 49% of all the bytes across all pages, even after file-level deduplication. Since 63% of the more than 140 PB of data stored by IA is devoted to web page snapshots [30, 31], I estimate that Jawa can reduce IA's storage needs by 35 PB.

Sources of storage benefits. Storage savings enabled by Jawa stem from a combination

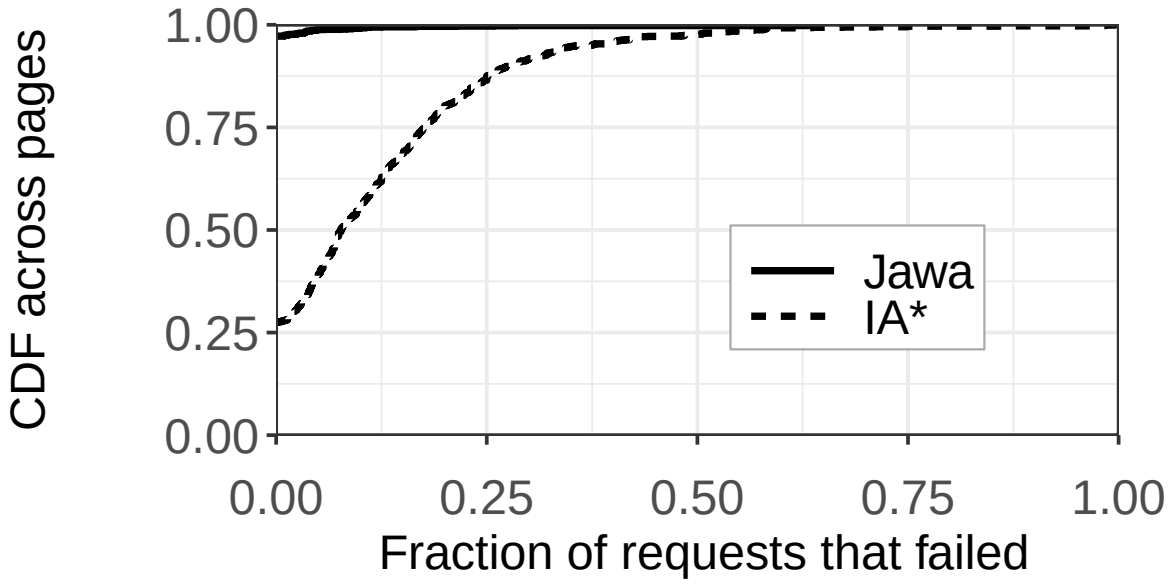
of not storing filtered files and pruning unreachable code. When I break down the impact of the filter lists I use, Figure 6.11(a) shows that my custom filter list alone reduces the total amount of JavaScript saved by 36%, and EasyList's rules result in a further reduction of 28%. Jawa also significantly reduces storage needs by eliminating unused code: the difference between the two right most bars in Figure 6.11.

6.6.1.2 Storage for indices

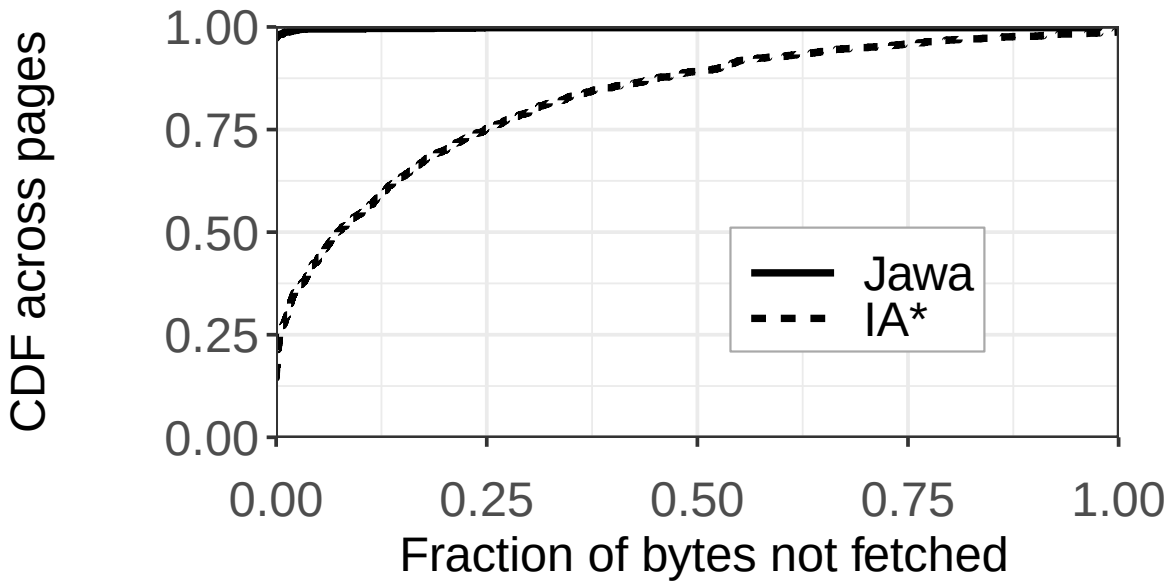
In addition to storing crawled resources, both IA* and Jawa also need to store the crawling and serving indices (Table 6.2). The former enables the crawler to not store duplicate content, whereas the latter enables lookups of requested resources when serving page snapshots. For my corpus of 1 million page snapshots, I find that size of both indices is marginally smaller (15%) with Jawa than with IA*. First, for most script files, Jawa ends up having to store a single WARC record; for such files, after the first time a subset of the file's code is stored, all subsequent page snapshots which include the same file end up needing the same subset. Second, the increase in index entries for other files (for which multiple subsets end up being stored) is offset by the elimination from the index of filtered files.

6.6.2 Fidelity

To evaluate Jawa's preservation of page fidelity, I crawl all 3000 pages in $Corpus_{3K}$ from the live web. I perform these crawls on a desktop, once with Jawa's crawler, and once without using any of its methods. I then load these pages from the two local copies, mimicking a different client ("iPhone 6"). When using page snapshots saved by Jawa, I match requested URLs to crawled URLs after stripping query strings.



(a)



(b)

Figure 6.12: **When snapshots of 3K pages are served, (a) number of resources requested by client which are not stored, and (b) fraction of resources stored for a snapshot which are not fetched by the client.**

Resource fetches. I first evaluate Jawa's impact on fidelity by examining the discrepancy between the set of resources stored for any snapshot and the set of resources fetched by a client when it loads that snapshot. Figure 6.12(a) shows that, while 7% of network requests return a 404 on the median page in loads of IA*, this fraction drops to 0% with Jawa. On

the 95th percentile page, the corresponding fractions are 36% with IA* and 0% with Jawa. Consequently, Figure 6.12(b) shows that, while 10% of stored resources are not fetched on the median page when mimicking loads from IA, this fraction drops to 0% with Jawa. On the 95th percentile page, the corresponding fractions are 75% with IA* and 0% with Jawa.

Visual analysis. To check if the pages served by Jawa are identical to the ones it crawled, I take a screenshot of every page both when crawling it and when I reload it from my local copy. I then compare every pair of screenshots to check if the value of every pixel matches. Apart from the visual differences accounted for by animations and non-determinism in 54 pages, both screenshots matched exactly for every other page when using Jawa. Since loads of IA* do not patch APIs for client characteristics, differences in screen dimensions between clients make it moot to compare screenshots.

Interactions. Finally, to evaluate Jawa's impact on post-load interactions, I randomly sample 150 pages. For each page, I load the versions that would be served by IA* and by Jawa. To isolate the impact of Jawa's techniques, I also consider an intermediate design point (Only filter) where I only use Jawa's filtering but do not prune unreachable code.

I categorize all event handlers on every page into three types: 1) navigational, i.e., they help in navigating either to a different page (e.g., a navigational bar) or within the page (e.g., a scroll-to-bottom button), 2) informational, i.e., they help make more information available (e.g., carousels or tabs), and 3) transactional (e.g., login or post buttons). On archived pages, transactional event handlers will not function. So, on each of the 150 sampled pages, I manually trigger all event handlers that belong to the first two categories. All 124 navigational interactions and 100 informational interactions worked as expected in all three loads: IA*, Only filter, and Jawa. Key to preserving these post-load interactions are Jawa's carefully curated filter list for discarding non-functional code, and its methods for identifying and retaining all reachable code. In contrast, if I discard all third-party files or if I use Jawa's filter list but save only the functions registered as handlers, then only 42% of these interactions work in the former case and 10% in the latter.

6.6.3 Performance

Crawling throughput. IA's production crawler is not public to the best of my knowledge. Therefore, I turn to two open-source crawlers: Brozzler [11] and ArchiveBox [6]. Brozzler is operated by IA, and used alongside their production crawler. Whereas, ArchiveBox is a very active and commonly used crawler by individual archivists (over 12K stars on GitHub). I find that Brozzler is 20% slower than ArchiveBox because of the latter's more efficient implementation of their headless Chrome interface. I also note, that on a server with 32

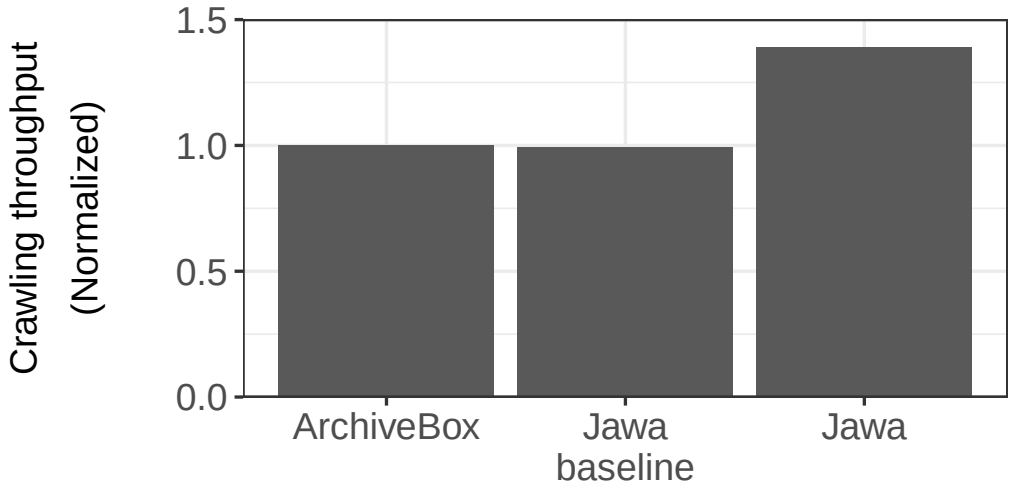


Figure 6.13: **Comparison of crawling throughput, normalized to that offered by ArchiveBox.**

cores and 128 GB RAM, I were able to crawl 5000 URLs in 15 minutes with ArchiveBox. With this crawling throughput, IA would need to dedicate 900 such servers for crawling pages, which is comparable to the number of servers they currently claim to use [29]. Therefore, I evaluate Jawa against ArchiveBox.

Figure 6.13 shows that Jawa's crawler offers throughput comparable to Archivebox when all of Jawa's techniques are disabled (Jawa baseline). Enabling all the methods in Jawa's design increases my crawler's throughput by 39%.

To breakdown the overheads, I measure the latency of each of the techniques used by Jawa's crawler in isolation, namely 1) filter: filtering JavaScript files, 2) code injection (CI): instrumenting the code in fetched scripts, 3) dynamic tracking (DT): dynamically tracking code execution and event handler registration, and finally 4) event triggering (ET): invoking event handlers and capturing the code executed. Figure 6.14 shows that not having to fetch filtered scripts completely offsets the overheads of all other techniques. Not only does Jawa's crawler not fetch any scripts which match its filter list, but all the resources that would have been fetched by the filtered files also go unfetched; this latter set of files often do not match the filter list.

Jawa also impacts crawling throughput by requiring more writes to the crawling index because, unlike IA*, it spreads the code in some script files across multiple WARC records. I cannot quantify the performance impact of doing so since my setup does not match a production archive like IA. However, I can quantify the number of additional writes that Jawa performs to the crawl index, compared to IA*. Table 6.3 shows that the number of writes to the crawl index *decrease* with Jawa; due to filtering, fewer files are crawled.

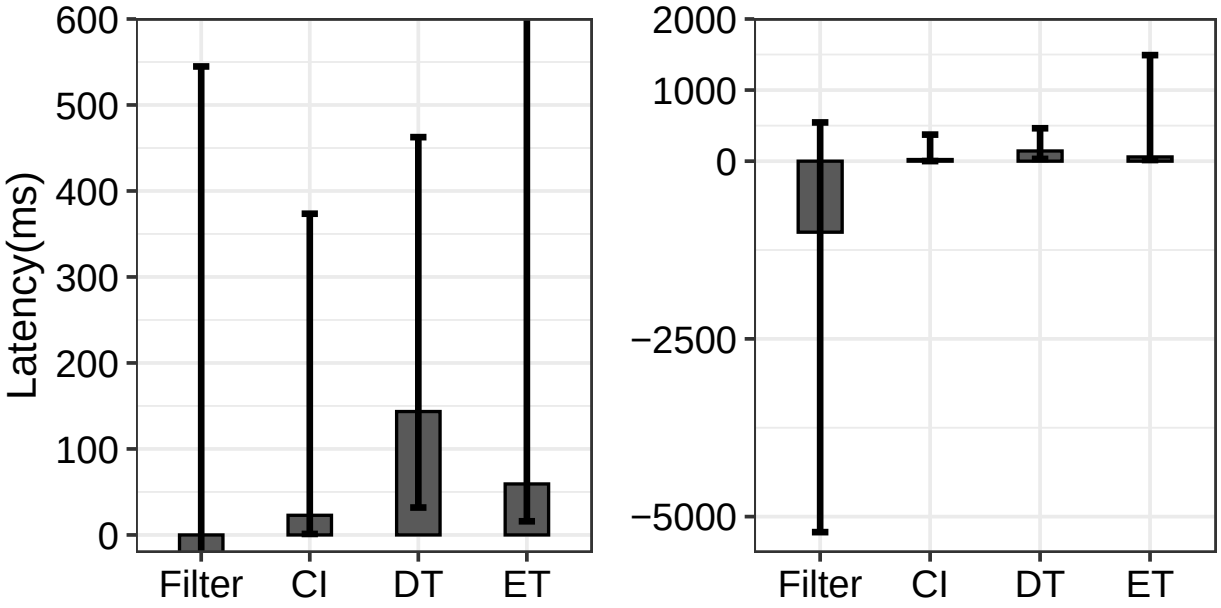


Figure 6.14: **Benchmarking the overhead of techniques used in Jawa's crawler. Bars plot median across pages, and whiskers plot 10^{th} and 90^{th} percentiles. Graph on the left zooms in on the yrange 0 to 500ms in the graph on the right.**

Index	**I/Os per page with IA***		**Reduction in I/Os per page with Jawa**	
	50^{th} %ile	90^{th} %ile	50^{th} %ile	90^{th} %ile
Crawling	3	15	1	5
Serving	41	107	1	3

Table 6.3: **Writes on crawling index and reads on serving index; values shown for 50^{th} and 90^{th} percentile page on median site.**

Serving performance. When serving page snapshots, Jawa's only overhead is in needing to potentially lookup multiple WARC records in order to respond to a request for a JavaScript file. I find that page load times on IA's Wayback Machine are proportional to the number of resources on the requested page snapshot, or equivalently, the number of WARC records that IA needs to lookup to serve the snapshot. Therefore, as a proxy for estimating Jawa's impact on user-perceived performance, I examine the increase due to Jawa in the number of WARC records read when serving page snapshots. Table 6.3 shows that the number of index lookups decrease with Jawa; again, thanks to filtering, a client has to fetch fewer files per snapshot.

6.7 Verifying Page Properties

Jawa's methods for pruning non-functional and unreachable code are based on three properties that I found to be true on archived web pages:

- *DRP* APIs have no impact on control flow
- Discarding third-party JavaScript files which match a manually curated filter list has no impact on fidelity
- For post-load interactions which work on archived pages, the event handlers which power them do not have read-write dependencies that influence branch conditions

All of these observations are rooted in my empirical analysis of a variety of web pages in $Corpus_{3K}$: 9 internal pages and 1 landing page in each of 300 sites, which span a wide range of rankings among Alexa's top million sites. However, I recognize that not all pages may abide by these properties. For example, consider a page which shows the time until a deadline and switches the font color when the time remaining is below a threshold; such a page would violate the first property listed above.

To handle such cases, I observe that web archives do not crawl every page just once; they repeatedly recrawl pages over time in order to capture changes to every page's content. For any given page, in some crawls of the page, a web archive can disable all of Jawa's methods and check if the properties expected to be true indeed hold on this page. For example, like the analysis I performed (§6.4), the web archive can instrument scripts to track state accesses, and then examine dependencies between event handlers and between files which do or do not match the filter list. It can also perform concolic execution to verify that *DRP* APIs have no impact on control flow.

The key to restricting the compute overheads of these heavyweight analyses is to run them on a sample of snapshots. To determine the sampling rate, a web archive can leverage properties that are stable across a page's snapshots. For example, upon analyzing all of IA's snapshots for 300 randomly chosen pages, I observe that the median page has the same number of runtime errors for an average of 53 snapshots. Therefore, once in every 53 crawls of any of these pages, a web archive can disable filtering and check if the number of runtime errors matches prior crawls where filtering had been used. If there is a mismatch, the web archive can disable the use of filtering for this page going forward. Since Jawa serves any JavaScript file to users as the union of all partitions of this file stored across crawls (§6.5), disabling filtering in one crawl of the page will also benefit all prior crawls of that page.

6.8 Discussion

How future proof is Jawa? In the immediate future, recent trends [43] indicate that the amount of JS on pages will continue to increase, making it important for web archives to adopt Jawa's techniques for pruning JS and for eliminating fidelity issues due to the non-determinism introduced by JS. In the long term, I expect that the principles that dictate Jawa's design will continue to hold: to serve pages with high fidelity, 1) archives must account for non-determinism, and 2) a large fraction of JS can be discarded with no risk.

Optimize already archived pages. Jawa's simple techniques make it highly amenable to be used with pages that have already been archived. First, a web archive can significantly reduce its storage needs by discarding all JS files that match Jawa's filter list. Second, the web archive can rewrite the HTML of every archived page to include a custom script which will enforce the same client characteristics as the crawler when users load the page. The only aspect of Jawa that would be hard to use on already archived pages is the elimination of unreachable code, as that requires invoking all event handlers on every page.

6.9 Artifact Appendix

Abstract

My open-source artifact contains the scripts and the data necessary to produce the key results from this paper. It also contains the code for the analysis framework which informed Jawa's design.

Scope The artifact can be used to confirm the three main benefits of Jawa: a) reduced storage overhead, b) improved fidelity by eliminating almost all failed network requests, and c) improved crawling throughput.

Contents The artifact contains all the code required to generate the key results with respect to three metrics: storage, fidelity and throughput. This includes a) Jawa's filter list and a NodeJS based crawler that leverages this filter list while loading web pages; b) a NodeJS based analyzer that injests JS files and instruments them to track all the JS functions executed at runtime, the set of event handlers registered, and the return values of browser APIs; and c) a set of scripts to automatically run the above code on a given corpus of pages. These scripts will produce the following results:

- **E1**: Reduced storage overhead using Jawa's two techniques: eliminating non-functional code using the filter list, and eliminating unused code by tracking the set of functions executed during the page load plus those required for enabling user interactions. This result will mimic the trend shown in Figure 6.11.

- **E2**: Improved page fidelity by eliminating almost all failed network requests. This result will reproduce the number of failed requests and the corresponding number of bytes not fetched, as shown in Figure 6.12.

- **E2**: Improved crawling throughput by reducing the number of IOs on the crawling index. This result will mimic the trend shown in the *"Crawling"* column of Table 6.3.

Apart from the scripts, the artifact contains a corpus of 3000 pages which is pre-recorded using the Mahimahi [163] tool. All scripts are run on this corpus of pages. Finally, the artifact also contains the JS analysis framework which was used to inform Jawa's design choices (§6.3).

Hosting The source code of the artifact is hosted on https://github.com/goelayu/Jawa with the corresponding commit ID: *"07e358eeed7cc054747271b19070b5563f3ff189"*. The corpus of pages is hosted on Google Drive.

Requirements

Software dependencies

The artifact has been tested on Ubuntu 16.04.7 LTS. It requires installing the following dependencies, in addition to the NodeJS dependencies included in the github repo (§6.9):

```
$ sudo apt-get install mahimahi google-chrome-stable parallel r-base r-
   base-core
$ sudo sysctl -w net.ipv4.ip_forward=1
```

Installations

Setting up the artifact involves three steps: a) downloading the source code and installing the NodeJS dependencies, b) patching the NodeJS dependencies to use the modified versions included in the github repo, and c) fetching and extracting the corpus of pages to run the analysis on.

Install the code

```
$ git clone https://github.com/goelayu/Jawa
$ cd Jawa
$ npm install
$ export NODE_PATH=${PWD}
```

Patch the dependencies

```
$ vim node_modules/puppeteer-extra-plugin-adblocker/dist/index.cjs.js
# add to line 73:
  return adblockerPuppeteer.PuppeteerBlocker.parse(fs.readFileSync('../
   filter-lists/combined-alexa-3k.txt', 'utf-8'));
```

Fetch the data

```
$ cd data
# download tarball from https://drive.google.com/file/d/17
   j6AYgaaXMhmV0VKWUmU_kMcHibMryVV/view?usp=sharing
$ tar -xf corpus.tar
```

Experiments workflow

As listed in §6.9, the artifact scripts will produce results corresponding to three metrics: storage, fidelity and crawling throughput.

Fidelity

I provide scripts and data to exactly reproduce Figure 6.12 (both a and b). The corpus used for this experiment contained 3000 pages. On a single core machine, it takes roughly 20–30 seconds for each page to load and, therefore, takes about 20 hours to load all 3000 pages once. I recommend to either run this experiment on a smaller corpus of pages (more details below) or to use a multi-core (16–32 cores) machine to speed up the overall execution time.

```
$ cd ../ae
# Usage: ./fidelity.sh <corpus_size> <num of parallel processes>
$ ./fidelity.sh 3000 1 # depending on the number of available cores on
   your machine, provide the 2nd argument
```

The output graphs will be generated in the same directory: *"count_fidelity.pdf"* and *"size_fidelity.pdf"*, corresponding to Figures 6.12(a) and 6.12(b), respectively.

Storage

Reproducing Figure 6.11 requires processing 1 million pages, which would take around a week (even with 128 CPU cores). I instead provide scripts to process 3000 pages, and demonstrate storage savings derived from both of Jawa's techniques. I provide preprocessed web pages, i.e., injected with instrumentation code to detect which functions are executed at runtime, and code to track event handlers. You can fetch the the instrumented pages as follows:

```
$ cd ../data
# download tarball from https://drive.google.com/file/d/16
   Pt4a2l1CNxC8UBwjalgEki-UlGAnFUm/view?usp=sharing
$ tar -xf processed.tar
```

You can now run the end-to-end storage analysis script:

```
$ cd ../ae
# Usage: ./storage.sh <corpus_size> <num of parallel processes>
$ ./storage.sh 3000 1 # depending on the number of available cores on your
    machine, provide the 2nd argument
```

The above script will print three storage numbers (in bytes) to the console. a) Total JS storage after deduplication (as incurred by Internet Archive); this mimics the *"IA*"* bar in Figure 6.11(a). b) Total JS storage after applying Jawa's filter; this mimics the *"IA*+Combined Filter"* bar in Figure 6.11(a). c) Total JS storage after removing unused JS functions; this mimics the *"Jawa"* bar in Figure 6.11(a).

Crawling throughput

I reproduce the throughput results from Table 6.3's *"Crawling"* column. The storage script above outputs the crawling index IOs as well. It prints the following two numbers: a) reductions in crawling IOs for the 50th percentile page, and b) reductions in crawling IOs for the 95th percentile page.

6.10 Summary

Since when the Internet Archive began operating in the late 1990s, a marked change on the web has been the increased use of JavaScript. In this chapter, I shined light on two significant problems caused by this change: broken rendering of archived pages, and petabytes of storage wasted on JavaScript which will either be non-functional or never be used. Our design of Jawa addresses these problems while emphasizing low overhead on both crawling and serving pages. As a result of my work, web archives will be able to archive many more pages than they can today for the same cost and ensure that archived pages more closely approximate their original versions.

CHAPTER 7

Conclusion

The past few decades have seen an exponential growth in the use of the internet, in particular the world wide web (WWW). We increasingly rely on the web to access information and services. Given this increased dependence on the web, a significant amount of effort has been spent on improving the web, such as initiative to reduce page load times, to bolster privacy and security policies, and others.

Despite these efforts, certain aspects of the web still fail to meet user expectations such as slow web page loads on mobile smartphones. Also, a number of relevant web domains such as crawling and archiving have been completely overlooked by systems researchers. In this dissertation, I have attempted to address this gap in the literature by tackling the issues of crawling and archiving the modern web and proposed two separate optimizations to significantly speed up web performance for users of the mobile web. I first described how modern web pages are highly amenable to JavaScript memoization technique in order to reduce the total client-side computations incurred during page loads. I then conducted similar analysis to show potential speed ups in JavaScript runtime that are achievable by offloading JS execution across multiple cores of modern smartphones. In chapter §5, I detailed the design of Sprinter, a web crawler that addresses the performance-fidelity trade-off of modern web crawlers. In chapter §6 I detailed the design of Jawa, a web crawler tailored towards web archives which crawls and stores web pages by leveraging differences between live and archived pages to reduce the storage overhead of preserving archived snapshots and eliminate any fidelity issues resulting from JavaScript's non-determinism.

In this rest of this chapter, I will summarize the main takeaways and lessons learned from my dissertation and conclude with some future work.

7.1 Lessons Learned

There are three main takeaways from the research summarized in this dissertation.

7.1.1 JavaScript's negative impacts on web pages are not limited to poor web performance

A significant amount of prior work on web performance has identified the negative impact of client-side computation on page load times. Solutions to mitigate these impacts have highlighted the need to reduce JavaScript execution. However, no prior work has characterized this client-side computation nor quantified the impact of JavaScript execution. In chapter §4, I breakdown client-side computation into each of its individual components—painting, rendering, HTML parsing and JavaScript compilation and execution—to identify how much JavaScript execution contributes to this overhead.

Although I was the first to precisely quantify the negative impacts of JavaScript execution on web page load times, this sentiment was already known to some extent as prior work had conducted analysis along similar lines [196]. In my dissertation, I uniquely identified a number of other negative impacts that JavaScript has on web pages. In chapter §6, I described two negative impacts that JavaScript has on web archiving. JavaScript doubles the storage overhead of web archives, despite using techniques like deduplication and compression. Moreover, the non-determinism manifested by JavaScript that helps customize web pages for the live web, degrades the fidelity of archived web pages by requesting URLs to be fetched that were never archived to begin with.

In chapter §5, I demonstrated how performance of web crawlers is dictated entirely by client-side computation overheads, and specifically JavaScript execution. This is different from end-user page loads, because while JS execution degrades end to end latency, it is not the only factor affecting performance. Network delays also play an important role. The reason behind JavaScript's bigger role in web crawling performance is the fact that crawlers care about crawling throughput, i.e., number of pages crawler per second, as opposed to the latency to download individual pages – the metric used to quality of experience for end-user page loads. I observed that compute resources on the client-side get starved way before any other resources (network, disk, memory) are saturated.

Although the use of JavaScript has become critical for web development, I believe that identifying and quantifying its negative impacts on multiple facets of web pages will hopefully lead to a more radical change in the way the web operates and call for more clean-slate web designs.

7.1.2 Differences in page loads in different contexts can be leveraged to overcome various issues with the web

While developing web pages, the first-order metric that page developers prioritize is the quality of experience for individual users loading pages on their client devices such as smartphones, laptops, etc. This involves improving the page loading time and better user experience by customizing pages for the needs of individual users.

As pointed out in my research, these features unfortunately don't bode well in other web loading contexts such as web archiving or large-scale web crawling. For example, support for dynamic rendering of web page loads by identifying what resources to fetch during the page load process itself aids customizing the page for individual users. However, in the case of web archiving, it results in poor fidelity of archived pages since the resultant non-determinism results in requesting resources that were never archived by the web archive.

The key to mitigating such issues in different contexts, as shown in my work, is to understand the differences in page loads in these different settings and leveraging these differences to optimize for the relevant metric. In Jawa, I described how archived pages are different from live pages in two ways. First, there is a lot of JavaScript code on an archived page that is non-functional since it needs to interact with a back-end server in order to function properly. For example, the ability to add comments to an article published on the web. Second, a large amount of code is un-reachable since certain branch conditions will never be taken. For example, if an authentication token is stored as a client-side cookie, the client-side code can print the user information, and if not, it can print the login button, on the screen. For archived pages, there will never be an authentication token stored in the cookies and therefore one of those branches will never be taken. Leveraging these differences, I was able to eliminate a large fraction of JavaScript code to reduce the total storage overhead incurred by web archives.

Similarly, in Sprinter, I identified how web crawlers and individual users care about different aspects of JavaScript execution. For individual users loading pages, everything that JavaScript does – add events to the page, DOM modifications to customize page for the user, identify what resources to fetch – matters. For crawlers, on the other hand, only what resources to fetch matters. As a result, I was able to exploit this difference to maximize reusing JavaScript execution across pages of the same site, and improve crawling throughput by over 5x.

7.1.3 Fine-grained analysis is feasible with the legacy web

The idea of fine-grained JavaScript analysis to optimize web performance was already introduced in prior work [165]. However, the proposed solution involved rewriting the web page on the server-side. This requirement of the participation by back-end web servers hindered the adoption of such techniques, since to be viable in practice a large number of web domains would have to adopt them.

In my dissertation, I showed that participation by back-end web servers is not a requirement to leverage the benefits of fine-grained analysis. As much diverse as the modern web has become, there still exist a number of properties or trends that hold true for a large variety of web pages. My work has capitalized on these trends in order to make fine-grained analysis of web computations feasible for legacy web pages. I have demonstrated the efficacy of this approach in two ways.

First, in Sprinter, I identified the trend of redundant JavaScript execution across pages of the same site. Even though identifying redundant executions itself required heavy weight fine-grained analysis which was computationally expensive, the fact that the degree of redundancy was so high helped amortize the compute overheads associated with such analysis. This meant that the analysis could be performed on client-side itself making it compatible with the legacy web.

In Jawa, I took an alternate approach. Instead of employing the fine-grained analysis in the system design itself, I performed offline analysis on a wide variety of web pages. Using the offline analysis, I extracted common properties that applied to almost all of the pages. Leveraging these properties, I designed Jawa. Clearly, not every web page is expected to be compatible with those properties, however the pervasiveness of those properties was high enough that it did not compromise on fidelity.

7.2 Future Work

The increasing ubiquity of the web has only fastened the rate at which it has evolved. The web stack—server-side web page development to the client browsers used to visit pages–is continuously evolving. For example, Google Chrome, the browser vendor with the largest market share releases a minor version update every 2 weeks. Simultaneously there are 100s of web features added/deprecated to the official web specifications [142]. As a result, it becomes unclear whether the problems with the web today will continue to persist in the future, and more importantly, whether the solutions presented today (including the ones in this dissertation) will continue to address those same problems in the future as well.

In the remainder of this chapter, I present a number of research problems relating to the web. I begin with outlining a clean-slate design for the web, and then discuss individual ideas with respect each of the web domains that were studied in this dissertation.

7.2.1 Clean slate design for the web

All the techniques and system designs presented in this dissertation are incremental changes to the status quo. This property of my solutions was not by accident. The systems research community often evaluates ideas on their practicality and ease of deployability. As a result, my systems were deliberately designed to work with legacy web pages, and unmodified browsers and required no manual work from web developers.

However, I believe that in order to overcome the array of problems tackled by web researchers, it is important that we rethink the design of the web from the ground up. Instead of proposing incremental changes compatible with the current web, I feel that there is a need for a more radical approach. As a first step towards clean-slate web architecture, I propose a couple of first principles that I believe the future web architects should base the design of the web on.

- **Versioning**. Web has become a collective repository of human knowledge. Any new contribution to this repository likely cites information created earlier, often by linking to different web pages, ranging from news articles, to social media posts on Twitter, Instagram etc. However, unlike other sources of information like books, information on the web can be easily modified. As a matter of fact, web pages are often updated over time, resulting in problems like content drift [131, 217].

  I believe that the web should inherently support versioning, analogous to software libraries and books. Each new revision gets annotated with a new version number, allowing users to refer to a particular snapshot of the text with the corresponding version number. This eliminates the need for web archives to repeatedly take snapshots of web pages to allow access to historic information on any page.

- **Crawling and archiving**. A key contribution of this dissertation was pointing out the often overlooked aspects of modern web pages – web crawling and web archiving. The current web is designed mostly to cater to the needs to end users loading pages using web browsers. As a result, both crawlers and web archives have to rely on a range of techniques to efficiently crawl the web, store it, and enable access to it for end-users.

  I believe that web pages should be designed to serve the needs for both end-users and

web crawlers. Depending on the user-agent of the client visiting a page, the web server for that page can respond with different versions of the same page. A more dynamic and customized version for end-users which will be optimized for user experience, high performance and supports all kinds of user interaction. For crawlers, it would serve a more static version of the page, ideally a single resource which contains all the relevant bits of the page. Finally, for web archives it might serve an even more stripped down version of the page, for example, by removing certain components such as search boxes which would not be functional when loaded from the web archive.

- **Resource identifier**. Web pages are identified using URLs, which contain information about the domain, host and directory on that host where the page is located. This model worked perfectly when a host corresponded to a single server hosting the web page and the URL path corresponded to the actual directory the web page resided in. Today, this is no longer true, since most web pages use complex content management systems which generate custom paths for each page.

  Instead, I propose that web pages be uniquely identified by their content. This proposal is based on the previous work on content centric networking which seeks to replace IP based networking, due to the numerous limitations it presents [181]. One possible way this could be implemented is by using content hashes to identify and locate each web page, similar to how IPFS operates [76].

- **Debloating**. A large number of performance and archiving issues stem from the increasing bloating of web pages. Content providers install a large number of analytics and advertising libraries on web pages to track users and monetize content on their page. Users, unfortunately, have no control over this, resulting in a large number of privacy violations [69] but also detrimental quality of experience with slower page loads.

  I propose that users preferences be included in what resources to serve as a part of the web page. Some browsers already have support for this, in the form of ad blockers and privacy trackers. However, I believe that web should natively support such options, instead of relying on browser vendors.

Next, I describe future work with respect to each of individual domains of web that I worked on.

7.2.2 Web performance

There is a long way to go before we achieve above par web performance for all users accessing all kinds of web pages on all sorts of client devices. To this end, I propose two research directions that haven't received enough attention.

Understanding utility of JavaScript . In this dissertation I pointed out the detrimental impacts of increasing JavaScript on web performance. Even though optimizing JavaScript execution has helped reduce this overhead to some extent, I believe that the problem will continue to exist as, given the web trends, the amount of JavaScript is only expected to increase over the next few years. A key question that naturally arises is *What is the utility of this increasing JavaScript on web pages?*

Currently we have a very limited understanding on the exact functionality or benefits achieved by the megabytes of JavaScript that is loaded as a part of most pages. If we could classify the existing JavaScript code into categories based on the functionality they provide, we could understand the root cause of this significant increase in JS bytes on pages. Also, by understanding what each of line JavaScript is essentially doing, we can make qualitative decisions on what lines are more critical than others so as to be able to trade off some JavaScript code for better performance.

System to study performance under different configurations . Each research study on web performance builds its own corpus of web pages to study any present limitations to better web performance and evaluates it on a test bed custom curated by the respective researchers based on a number of factors such as availability of resources or assumptions about the best smartphones, browser vendors etc. Moreover a lot of studies are restricted to particular geographic locations, with most of the top-tiered conference work mostly focusing on US-Europe market.

I believe that there is a rising need for a systematic way to study web performance given this heterogenous environment with an exponential number of parameters such as location, device type, browser type etc. I envision that ML models might be able to solve some of challenges that come with navigating such a large configuration space. ML models can be trained on the web performance metrics gathered by large browser vendors such as Google Chrome to identify what optimizations would benefit what kinds of users. Similar techniques can also be used to evaluate the efficacy of previous work on web performance to understand how generalizable those techniques are.

7.2.3 Crawling the web at scale

Utility of resource from crawlers perspective. In my work on improving the crawling throughput of web crawlers while ensuring high fidelity of resource fetches, I showed how existing crawlers sit at the opposite ends of performance-fidelity trade-off space. I assumed that from a crawlers perspective, perfect fidelity implies fetching all resources that would be fetched when a user loads a web page using a web browser. Even though this definition of fidelity might be applicable for some cases of crawling, for example, when researchers crawl the web to download all the resources of pages in order to study different aspects of pages, for a large number of crawling purposes, fetching only a subset of resources might suffice. For example, search engines that only rely on text based information to serve user queries can only download the HTML file for every page they crawl, if all the textual information is embedded in the HTML file itself.

I believe that there are two things missing from current crawling pipelines. First, crawlers should be capable of charaterizing page resources on their contribution towards the final rendered page. For example, would fetching a given JS file result in fetching more content (images, or text) that will be rendered as a part of the final page, or would fetching a given image file result in fetching of other resources due to resource dependencies. Second, crawlers should allow their users to explicitly state the intent of crawling, for example, is the user interested in fetching all resources, only text based resources or only images etc. Combined with the page resource characterization, the crawler must be able to decide how a given page should be crawled in order to fetch the relevant resources. If a given page relies on JavaScript execution to fetch all the images, and the user requires images to be fetched, then the crawler would load the page using a browser, so as to be able to execute the JavaScript and accurately identify image URLs to be fetched. On other hand, if the user doesn't care about images, then the crawler could use simple static parsing techniques to download only the relevant components of the page.

Automatic generation of crawlable versions . Given the increasing importance of crawlers to power modern services, specifically AI-based services that rely on data gathered from the web to train their respective models, I believe that web pages should provide native support for crawlers. As mentioned in my proposal for the clean-slate web, web servers should respond with the appropriate version of their site depending on what client is requesting the access. Since often times crawlers aren't going to interact with the page, or visually inspect the page, the web server can respond with a much simpler, stripped down version of the page for when requested by a crawler.

Currently, content providers don't have an automated means of creating such a stripped

down version of their web page. Large content providers have 100s of web pages hosted on their servers, which means manually creating another version of each page which would result in lots of developer effort. I propose an automated system that can take all the resources on a given page and produce a crawling-friendly version of the page, that ideally can be fetched as a single resource, without needing the crawler to use compute intensive methods like browser-based crawling. This automated system can be used not only across different pages of the same site, but also across different sites.

7.2.4 Efficient web archiving

Identifying what to archive. Increasing ephemerality of the web has resulted in lots of web archiving initiatives. However, web archives have limited resources which makes it infeasible for them to crawl and capture the entire web, which consists of trillions of web pages, adding up to zetabytes of resources. For example, even though Internet Archive, the largest web archive, has stored more than 600 billion web pages, it is still missing a large number of critical pages.

To remedy this, I propose that crawlers also account for *"freshness"* as a metric while measuring the efficiency of crawling. Instead of only focussing on crawling throughput, crawlers should identify which pages to crawl. A page has maximum utility or adds maximum freshness to the overall corpus being crawled if it contains the most unique content. This would require constructing some kind of semantic summary of web pages, and computing edit-distances on these semantic summaries to identify which pages are referring to similar information and therefore not worth crawling. For example, two news articles from two separate news organiziations covering the same news event are likely to report the same fact. Moreover, organization with similar political stance are likely to overlap in their coverage even more.

BIBLIOGRAPHY

[1] https://developers.google.com/search/docs/advanced/javascript/javascript-seo-basics.

[2] https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage.

[3] 2016 Q2 mobile insights report. http://resources.mobify.com/2016-Q2-mobile-insights-benchmark-report.html.

[4] Amazon silk. https://docs.aws.amazon.com/silk/latest/developerguide/what-is-silk.html.

[5] Archive unleashed. https://github.com/archivesunleashed/aut.

[6] Archivebox. https://github.com/ArchiveBox/ArchiveBox.

[7] Babel. https://babeljs.io/.

[8] BeautifulSoup. https://pypi.org/project/beautifulsoup4/.

[9] Browser market share worldwide. https://gs.statcounter.com/browser-market-share.

[10] Browsertrix crawler. https://github.com/webrecorder/browsertrix-crawler.

[11] Brozzler. https://github.com/internetarchive/brozzler.

[12] Cascadia. https://github.com/andybalholm/cascadia.

[13] Chrome cpu profiler. https://developer.chrome.com/docs/devtools/performance//.

[14] Chrome devtools protocol. https://chromedevtools.github.io/devtools-protocol/.

[15] Chrome web page replay. https://chromium.googlesource.com/catapult/+/HEAD/web_page_replay_go/README.md.

[16] Cisco annual Internet report highlights tool. https://www.cisco.com/c/en/us/solutions/executive-perspectives/annual-internet-report/air-highlights.html.

[17] Common crawl. https://commoncrawl.org/.

[18] Conifer. https://conifer.rhizome.org/.

[19] CSS selectors. https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors.

[20] Directory of the web. https://dmoz-odp.org/.

[21] Donate to the Internet Archive! https://archive.org/donate/.

[22] EasyList. https://easylist.to/.

[23] 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/.

[24] Gnu wget2. https://github.com/rockdaboot/wget2.

[25] Goquery. https://github.com/PuerkitoBio/goquery.

[26] Html tree syntax. https://www.w3.org/TR/html401/struct/global.html.

[27] HTTP Archive: State of the web. https://httparchive.org/reports/state-of-the-web#bytesTotal.

[28] HTTP/2 adoption. https://httparchive.org/reports/state-of-the-web#h2.

[29] IA infrastructure. https://archive.org/details/jonah-edwards-presentation.

[30] Inside wayback machine. https://thehustle.co/inside-wayback-machine-internet-archive/.

[31] Internet archive. https://www.archive.org/about/.

[32] Internet archive end of term 2020 web crawls. https://archive.org/details/EndOfTerm2020WebCrawls.

[33] Internet Archive tax return. https://projects.propublica.org/nonprofits/organizations/943242767.

[34] jQuery element selector. https://api.jquery.com/element-selector/.

[35] List of mime types. https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types.

[36] More than 9 million broken links on Wikipedia are now rescued. https://blog.archive.org/2018/10/01/more-than-9-million-broken-links-on-wikipedia-are-now-rescued/.

[37] Network latency numbers. https://github.com/WPO-Foundation/webpagetest/blob/master/www/settings/connectivity.ini.sample.

[38] Optional chaining. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Optional_chaining.

[39] Page-vault. https://www.page-vault.com/solutions/.

[40] Puppeteer. https://pptr.dev/.

[41] Size of the web. https://www.worldwidewebsize.com/.

[42] Smartphones are driving all growth in web traffic. https://www.vox.com/2017/9/11/16273578/smartphones-google-facebook-apps-new-online-traffic.

[43] State of JavaScript. https://httparchive.org/reports/state-of-javascript.

[44] Stillio. https://www.stillio.com/.

[45] The Boston Globe: Internet archive's copy from August 2, 2020. https://web.archive.org/web/20200802084355/https://www.bostonglobe.com/.

[46] The Daily Caller: Internet archive's copy from September 5, 2020. https://web.archive.org/web/20200905133311/https://dailycaller.com/.

[47] Using media queries. https://developer.mozilla.org/en-US/docs/Web/CSS/Media_Queries/Using_media_queries.

[48] The WARC format 1.0. https://iipc.github.io/warc-specifications/specifications/warc-format/warc-1.0/.

[49] WARC revisit tag. https://iipc.github.io/warc-specifications/specifications/warc-format/warc-1.0/#revisit.

[50] Wayback machine. https://www.archive.org/web.

[51] Web APIs. https://caniuse.com/?compare=chrome+114,firefox+113&compareCats=all.

[52] Webpack. https://webpack.js.org/guides/tree-shaking/.

[53] Webrecorder. https://webrecorder.net/.

[54] FCC: Broadband performance (OBI technical paper no. 4). https://transition.fcc.gov/national-broadband-plan/broadband-performance-paper.pdf, 2009.

[55] Third annual broadband study shows global broadband quality improves by 24% in one year. https://newsroom.cisco.com/press-release-content?type=webcontent&articleId=5742339, 2010.

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

[57] Seleniumhq browser automation. https://selenium.dev/, 2019.

[58] 10 Most Popular Phones in India in 2020 – Xiaomi and Samsung Rules. https://candytech.in/most-popular-phones-in-india/, 2020.

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

[60] GSMA: The State of Mobile Internet Connectivity 2020. https://www.gsma.com/r/wp-content/uploads/2020/09/GSMA-State-of-Mobile-Internet-Connectivity-Report-2020.pdf, 2020.

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

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

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

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

[65] 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. In *NSDI*, NSDI '15. USENIX, 2015.

[66] Fatemeh Ahmadi-Abkenari and Ali Selamat. An architecture for a focused trend parallel web crawler with the application of clickstream analysis. *Information Sciences*, 2012.

[67] Scott G Ainsworth, Ahmed Alsum, Hany SalahEldeen, Michele C Weigle, and Michael L Nelson. How much of the web is archived? In *Joint Conference on Digital Libraries*, 2011.

[68] Takuya Akiba, Keisuke Fukuda, and Shuji Suzuki. Chainermn: Scalable distributed deep learning framework. *arXiv preprint arXiv:1710.11351*, 2017.

[69] Istemi Ekin Akkus, Ruichuan Chen, Michaela Hardt, Paul Francis, and Johannes Gehrke. Non-tracking web analytics. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 687–698, 2012.

[70] Ahmed AlSum, Michele C Weigle, Michael L Nelson, and Herbert Van de Sompel. Profiling web archive coverage for top-level domain and content language. *International Journal on Digital Libraries*, 2014.

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

[72] Silviu Andrica and George Candea. WaRR: A tool for high-fidelity web application record and replay. In *DSN*, 2011.

[73] 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, page 351–367. Association for Computing Machinery, 2020.

[74] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Møller, and Frank Tip. A framework for automated testing of javascript web applications. In *ICSE*, 2011.

[75] Ajay Sudhir Bale, Naveen Ghorpade, S Rohith, S Kamalesh, R Rohith, and BS Rohan. Web scraping approaches and their performance on modern websites. In *International Conference on Electronics and Sustainable Communication Systems*, 2022.

[76] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[77] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A Acar, and Rafael Pasquin. Incoop: MapReduce for incremental computations. In *SoCC*, 2011.

[78] 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.

[79] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience*, 2004.

[80] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna. Bubing: Massive crawling for the masses. *ACM Transactions on the Web (TWEB)*, 2018.

[81] Kevin Boos, David Chu, and Eduardo Cuervo. Flashback: Immersive virtual reality on mobile devices via rendering memoization. In *MobiSys*, 2016.

[82] 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.

[83] Justin F Brunelle, Mat Kelly, Hany SalahEldeen, Michele C Weigle, and Michael L Nelson. Not all mementos are created equal: Measuring the impact of missing resources. *International Journal on Digital Libraries*, 2015.

[84] Justin F Brunelle, Mat Kelly, Michele C Weigle, and Michael L Nelson. The impact of javascript on archivability. *International Journal on Digital Libraries*, 2016.

[85] Justin F Brunelle, Michele C Weigle, and Michael L Nelson. Archival crawlers and javascript: discover more stuff but crawl more slowly. In *Joint Conference on Digital Libraries*. IEEE, 2017.

[86] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V. Madhyastha, and Vyas Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *NSDI*, NSDI. USENIX Association, 2015.

[87] 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.

[88] Soumen Chakrabarti, Martin Van den Berg, and Byron Dom. Focused crawling: a new approach to topic-specific web resource discovery. *Computer networks*, 31(11-16):1623–1640, 1999.

[89] 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.

[90] Junghoo Cho and Hector Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *VLDB*, 2000.

[91] Andrey Chudnov and David A Naumann. Inlined Information Flow Monitoring for JavaScript. In *CCS*, 2015.

[92] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable deterministic multithreading through schedule memoization. In *OSDI*, 2010.

[93] 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.

[94] Paul ME De Bra and RDJ Post. Information retrieval in the world-wide web: Making client-based searching feasible. *Computer Networks and ISDN Systems*, 27(2):183–192, 1994.

[95] Google Developers. Chrome DevTools. https://developers.google.com/web/tools/chrome-devtools/.

[96] Jenny Edwards, Kevin McCurley, and John Tomlin. An adaptive model for optimizing performance of an incremental web crawler. In *WWW*, 2001.

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

[98] Miklós Erdélyi, András A. Benczúr, Julien Masanés, and Dávid Siklósi. Web spam filtering in internet archives. In *Proceedings of the 5th International Workshop on Adversarial Information Retrieval on the Web*, 2009.

[99] 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.

[100] 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.

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

[102] Facebook. Prepack. https:/www.prepack.io, 2019.

[103] Uriel Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 1998.

[104] Zeon Trevor Fernando, Ivana Marenzi, and Wolfgang Nejdl. ArchiveWeb: Collaboratively extending and exploring web archive collections–how would you like to work with your collections? *International Journal on Digital Libraries*, 2018.

[105] Dennis Fetterly, Mark Manasse, Marc Najork, and Janet L Wiener. A large-scale study of the evolution of web pages. *Software: Practice and Experience*, 2004.

[106] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with coral. NSDI. USENIX, 2004.

[107] David F. Galletta, Raymond Henry, Scott McCoy, and Peter Polak. Web site delays: How tolerant are users? *Journal of the Association for Information Systems*, 2004.

[108] 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.

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

[110] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[111] Ayush Goel, Vaspol Ruamviboonsuk, Ravi Netravali, and Harsha V Madhyastha. Rethinking client-side caching for the mobile web. In *HotMobile*, 2021.

[112] Ayush Goel, Jingyuan Zhu, Ravi Netravali, and Harsha V Madhyastha. Jawa: Web archival in the era of JavaScript. In *OSDI*, 2022.

[113] Ayush Goel, Jingyuan Zhu, Ravi Netravali, and Harsha V Madhyastha. Sprinter: Speeding up high-fidelity crawling of the modern web. In *NSDI*, 2024.

[114] Daniel Gomes and Miguel Costa. The importance of web archives for humanities. *International Journal of Humanities and Arts Computing*, 8(1):106–123, 2014.

[115] Google. Accelerated mobile pages project (amp). https://www.ampproject.org/.

[116] Google. Chromium. https://www.chromium.org/Home.

[117] Google. Why performance matters? https://developers.google.com/web/fundamentals/performance/why-performance-matters.

[118] Gerhard Gossen, Elena Demidova, and Thomas Risse. ICrawl: Improving the freshness of web collections by integrating social web and focused web crawling. *JCDL*, 2015.

[119] M. Graham. Turn all references blue. https://archive.org/details/mark-graham-presentation.

[120] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A Thekkath, Yuan Yu, and Li Zhuang. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, 2010.

[121] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. JSFlow: Tracking Information Flow in JavaScript and Its APIs. In *SAC*, 2014.

[122] Allan Heydon and Marc Najork. Mercator: A scalable, extensible web crawler. *WWW*, 1999.

[123] Ariya Hidayat. Esprima. https://esprima.org/.

[124] Helen Hockx-Yu. Access and scholarly use of web archives. *Alexandria*, 25(1-2):113–127, 2014.

[125] Helge Holzmann, Vinay Goel, and Avishek Anand. Archivespark: Efficient web archive access, extraction and derivation. In *Joint Conference on Digital Libraries*, 2016.

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

[127] 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, page 207–221, New York, NY, USA, 2016. Association for Computing Machinery.

[128] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, Mahadev Satyanarayanan, Gregory R Ganger, Erik Riedel, and Anastassia Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *FAST*, 2004.

[129] International Internet Preservation Consortium. Access Working Group. Use cases for access to internet archives. *IIPC Report*, 2006.

[130] Mark Johnson. Memoization of top down parsing. *arXiv preprint cmp-lg/9504016*, 1995.

[131] Shawn M Jones, Herbert Van de Sompel, Harihar Shankar, Martin Klein, Richard Tobin, and Claire Grover. Scholarly context adrift: Three out of four URI references lead to changed content. *PloS one*, 2016.

[132] 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.

[133] 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.

[134] Mat Kelly, Michael L Nelson, and Michele C Weigle. A framework for aggregating private and public web archives. In *Joint Conference on Digital Libraries*, 2018.

[135] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Efficient patch-based auditing for web application vulnerabilities. In *OSDI*, 2012.

[136] Ronny Ko, James Mickens, Blake Loring, and Ravi Netravali. Oblique: Accelerating page loads using symbolic execution. In *NSDI*, NSDI, Berkeley, CA, USA, 2021. USENIX Association.

[137] Jin-woo Kwon and Soo-Mook Moon. Web application migration with closure reconstruction. In *WWW*, 2017.

[138] Steve Lawrence, Frans Coetzee, Eric Glover, Gary Flake, David Pennock, Bob Krovetz, Finn Nielsen, Andries Kruger, and Lee Giles. Persistence of information on the web: Analyzing citations contained in research articles. In *CIKM*, 2000.

[139] Hsin-Tsang Lee, Derek Leonard, Xiaoming Wang, and Dmitri Loguinov. Irlbot: scaling to 6 billion pages and beyond. *ACM Transactions on the Web (TWEB)*, 2009.

[140] Wonchan Lee, Elliott Slaughter, Michael Bauer, Sean Treichler, Todd Warszawski, Michael Garland, and Alex Aiken. Dynamic tracing: Memoization of task graphs for dynamic task-based runtimes. In *SC*, 2018.

[141] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 1966.

[142] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. How does web service API evolution affect clients? In *IEEE International Conference on Web Services*, 2013.

[143] Timothy Libert. Exposing the hidden web: An analysis of third-party HTTP requests on 1 million websites. *International Journal of Communication*, 2015.

[144] Jimmy Lin, Milad Gholami, and Jinfeng Rao. Infrastructure for supporting exploration and discovery in web archives. In *WWW*, 2014.

[145] Blake Loring, Duncan Mitchell, and Johannes Kinder. ExpoSE: Practical symbolic execution of standalone JavaScript. In *SPIN Symposium on Model Checking of Software*, SPIN 2017. ACM, 2017.

[146] 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.

[147] 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.

[148] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *WWW*, 2007.

[149] Shaghayegh Mardani, Ayush Goel, Ronny Ko, Harsha V. Madhyastha, and Ravi Netravali. Horcrux: Automatic javascript parallelism for resource-efficient web computation. In *OSDI*, 2021.

[150] Shaghayegh Mardani, Mayank Singh, and Ravi Netravali. Fawkes: Faster mobile page loads via app-inspired static templating. In *NSDI*. USENIX Association, 2020.

[151] Catherine C Marshall and Frank M Shipman. On the institutional archiving of social media. In *Joint Conference on Digital Libraries*, 2012.

[152] MDN. Web APIs. https://developer.mozilla.org/en-US/docs/Web/JavaScript, 2020.

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

[154] 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.

[155] Filippo Menczer and Richard K Belew. Adaptive information agents in distributed textual environments. In *Proceedings of the second international conference on Autonomous agents*, pages 157–164, 1998.

[156] Donald Michie. "Memo" functions and machine learning. *Nature*, 1968.

[157] James Mickens. Rivet: Browser-agnostic remote debugging for web applications. In *USENIX ATC*, USENIX ATC'12, USA, 2012. USENIX Association.

[158] James Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI*. USENIX Association, 2010.

[159] 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.

[160] 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, pages 1305–1315. International World Wide Web Conferences Steering Committee, 2016.

[161] Javad Nejati and Aruna Balasubramanian. An in-depth study of mobile browser performance. In *WWW*, 2016.

[162] Javad Nejati, Meng Luo, Nick Nikiforakis, and Aruna Balasubramanian. Need for mobile speed: A historical analysis of mobile web performance. In *TMA*, 2020.

[163] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate record-and-replay for HTTP. In *USENIX ATC*, Proceedings of ATC '15. USENIX, 2015.

[164] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *NSDI*, NSDI, Berkeley, CA, USA, 2016. USENIX Association.

[165] Ravi Netravali and James Mickens. Prophecy: Accelerating mobile page loads using final-state write logs. In *NSDI*, NSDI, Berkeley, CA, USA, 2018. USENIX Association.

[166] Ravi Netravali and James Mickens. Prophecy: Accelerating mobile page loads using final-state write logs. In *NSDI*, 2018.

[167] Ravi Netravali and James Mickens. Remote-Control Caching: Proxy-Based URL Rewriting to Decrease Mobile Browsing Bandwidth. In *HotMobile*, HotMobile '18, pages 63–68. ACM, 2018.

[168] Ravi Netravali and James Mickens. Reverb: Speculative debugging for web applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019.

[169] Ravi Netravali, Vikram Nathan, James Mickens, and Hari Balakrishnan. Vesper: Measuring time-to-interactivity for web pages. In *NSDI*, NSDI, Renton, WA, USA, 2018. USENIX Association.

[170] Ravi Netravali, Anirudh Sivaraman, James Mickens, and Hari Balakrishnan. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. In *MobiSys*, MobiSys '19, pages 430–443. ACM, 2019.

[171] Peter Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, 1991.

[172] Alexandros Ntoulas, Junghoo Cho, and Christopher Olston. What's new on the web? the evolution of the web from a search engine perspective. In *WWW*, 2004.

[173] 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.

[174] Hartmut Obendorf, Harald Weinreich, Eelco Herder, and Matthias Mayer. Web page revisitation revisited: implications of a long-term click-stream study of browser usage. In *CHI*, 2007.

[175] Opera. Opera Mini. http://www.opera.com/mobile/mini, 2018.

[176] Opera. Opera Turbo. http://www.opera.com/turbo, 2018.

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

[178] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *NSDI*, 2015.

[179] 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.

[180] Xiang Pan, Yinzhi Cao, Shuangping Liu, Yu Zhou, Yan Chen, and Tingzhe Zhou. CSPAutoGen: Black-box enforcement of content security policy upon real-world websites. In *CCS*, 2016.

[181] Diego Perino and Matteo Varvello. A reality check for content centric networking. In *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*, pages 44–49, 2011.

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

[183] Gavin Phillips. Smartphones vs. desktops: Why is my phone slower than my PC? https://www.makeuseof.com/tag/smartphone-desktop-processor-differences/.

[184] Brian Pinkerton. Finding what people want: Experiences with the webcrawler. In *Proc. of the 2nd Int. World Wide Web Conf., 1994*, 1994.

[185] Cosmin Radoi, Stephan Herhut, Jaswanth Sreeram, and Danny Dig. Are Web Applications Ready for Parallelism? In *PPoPP*, 2015.

[186] Alexander Rasmussen, George Porter, Michael Conley, Harsha V Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. Tritonsort: A balanced large-scale sorting system. In *NSDI*, 2011.

[187] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V. Madhyastha. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *SIGCOMM*. ACM, 2017.

[188] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for javascript. In *IEEE Symposium on Security and Privacy*, 2010.

[189] Prateek Saxena, R Sekar, and Varun Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO*, 2008.

[190] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *FSE*, 2013.

[191] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE*, 2005.

[192] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[193] Shahnaz Mohammedi Shariff, Heng Li, Cor-Paul Bezemer, Ahmed E Hassan, Thanh HD Nguyen, and Parminder Flora. Improving the testing efficiency of selenium-based load tests. In *International Workshop on Automation of Software Test*, 2019.

[194] Zejian Shi, Minyong Shi, and Weiguo Lin. The implementation of crawling news page based on incremental web crawler. In *2016 4th Intl Conf on Applied Computing and Information Technology/3rd Intl Conf on Computational Science/Intelligence and Applied Informatics/1st Intl Conf on Big Data, Cloud Computing, Data Science & Engineering (ACIT-CSII-BCD)*, pages 348–351. IEEE, 2016.

[195] 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 International Conference on Mobile Computing and Networking*, MobiCom. ACM, 2015.

[196] 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 *Mobicom*, Mobicom. ACM, 2017.

[197] 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 *CoNEXT*, CoNEXT '14, pages 325–336, New York, NY, USA, 2014. ACM.

[198] Petr Slavík. A tight analysis of the greedy algorithm for set cover. In *STOC*, 1996.

[199] Dolière Francis Some, Nataliia Bielova, and Tamara Rezk. On the content security policy violations due to the same-origin policy. In *Proceedings of the 26th International Conference on World Wide Web*, 2017.

[200] Diomidis Spinellis. The decay and failures of web references. *Communications of the ACM*, 46(1):71–77, 2003.

[201] Arjun Suresh, Erven Rohou, and André Seznec. Compile-time function memoization. In *Proceedings of the 26th International Conference on Compiler Construction*, 2017.

[202] Arjun Suresh, Bharath Narasimha Swamy, Erven Rohou, and André Seznec. Intercepting functions for memoization: A case study using transcendental functions. *ACM Transactions on Architecture and Code Optimization (TACO)*.

[203] Qingzhao Tan and Prasenjit Mitra. Clustering-based incremental web crawling. *ACM Trans. Inf. Syst.*, 2010.

[204] Yang Tang and Junfeng Yang. Secure deduplication of general computations. In *USENIX ATC*, 2015.

[205] Omer Tripp and Omri Weisman. Hybrid analysis for javascript security assessment. In *ESEC/FSE*, 2011.

[206] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, et al. Unity: Accelerating {DNN$}$ training through joint optimization of algebraic transformations and parallelization. In *OSDI*, 2022.

[207] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying page load performance with WProf. In *NSDI*. USENIX Association, 2013.

[208] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. How Speedy is SPDY? In *Proceedings of NSDI*. USENIX Association, 2014.

[209] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. How much can i micro-cache web pages? In *IMC*, 2014.

[210] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. Speeding up web page loads with Shandian. In *NSDI*. USENIX Association, 2016.

[211] Xinyue Wang and Zhiwu Xie. The case for alternative web archival formats to expedite the data-to-insight cycle. In *Joint Conference on Digital Libraries*, 2020.

[212] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. Why are web browsers slow on smartphones? In *HotMobile*, 2011.

[213] 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.

[214] Shiyi Wei and Barbara G Ryder. Practical blended taint analysis for javascript. In *International Symposium on Software Testing and Analysis*, 2013.

[215] Harald Weinreich, Hartmut Obendorf, Eelco Herder, and Matthias Mayer. Not quite the average: An empirical study of web use. *ACM Transactions on the Web*, 2008.

[216] Qinghua Zheng, Zhaohui Wu, Xiaocheng Cheng, Lu Jiang, and Jun Liu. Learning to crawl deep web. *Information Systems*, 2013.

[217] Jonathan L Zittrain, John Bowers, and Clare Stanton. The paper of record meets an ephemeral web: An examination of linkrot and content drift within the new york times. *Available at SSRN 3833133*, 2021.