Reining in Mobile Web Performance with Document and Permission Policies

Byungjin Jun[†] Fabián E. Bustamante[†] Ben Greenstein[⋆] Ian Clelland[⋆]
[†]Northwestern University [⋆]Google

ABSTRACT

The quality of mobile web experience remains poor, partially as a result of complex websites and design choices that worsen performance, particularly for users on suboptimal networks or with low-end devices. Prior proposed solutions have seen limited adoption due to the demand they place on developers and content providers, and the performing infrastructure needed to support them. We argue that *Document and Permissions Policies* – ongoing efforts to enforce good practices on web design – may offer the basis for a readily-available and easily-adoptable solution, as they encode key best practices for web development. In this paper, as a first step, we evaluate the potential performance cost of violating these well understood best practices and how common such violations are in today's web. Our analysis shows, for example, that controlling for *unsized-media* policy, something applicable to 70% of the top Alexa websites, can indeed significantly reduce Cumulative Layout Shift, a core metric for evaluating the performance of the web.

CCS CONCEPTS

• **Networks** → **Mobile networks**; **Network performance analysis**.

KEYWORDS

Mobile network, performance measurement, Document Policy

1 INTRODUCTION

Quality of experience (QoE) with the mobile web remains suboptimal [3], with the majority of pages taking several seconds to load [5], years after users have moved to mobile devices as their primary way to access the web [40]. For users connected over challenged networks with low-end devices, a common case in many developing countries [2, 7], this is painfully obvious.

Part of the problem is that much of the web has been designed, implicitly or not, for users on good networks and devices or, at least, without considerations of performance implications [12]. This has resulted in more complex websites [6], with heavy web fonts, external resources, large images, and animation that, while perhaps visually appealing for high-end users can be frustrating to the rest. This is exacerbated by a very permissive web that turns a blind eye to not following best practices for implementing performant websites, even if such violations drastically worsen the mobile experience.

This situation combined with the potential impact of poor web performance on user engagement (and profit) [11] have served as motivation for a range of industry and academic efforts (e.g., [9, 32, 37, 39]). Despite their demonstrated potential benefits, most solutions have seen limited adoption partially due to the demand they place on developers and content providers, and the performing infrastructure needed to support them. For instance, six years after its first release, the adoption of AMP, an effective [24] and popular solution, is still around 0.2% of all the websites [43].

We argue that *Document and Permission Policies* [14, 15] may offer a readily available, easily deployable, and effective way to help developers to improve their pages more performant under the poor conditions. Feature Policy, as the set of policies was originally named, is a specification that allows developers to control certain features and APIs within a browser [14]. The features referred to by these policies range from simple camera access and location information to image sizing and script execution.

While originally designed for the developers, users can enable these policies to identify when best practices for better mobile web performance, such as disabling the rendering of unsized images that may require a layout shift or excluding synchronous scripts that may block rendering, are not being followed. As such, they provide tools for developers to identify issues with their sites so that they may correct them. By operating while a page is rendering, they also provide an opportunity for the browser to intervene on the user's behalf to improve the website's implementation.

As a first step, in this paper, we evaluate the potential performance benefit of enforcing best practices and assess how common the violations of such policies are in today's web. We identify a number of performance-associated policies and rely on microbenchmarks to evaluate the performance benefits of applying them. In our test, Speed Index [22] can be improved by 1.4 seconds by applying *oversized-image* policy, while *font-display-late-swap* can boost Largest Contentful Paint [44] by 3 seconds[1]. To estimate the potential impact of this approach, we look at how frequently these policies, which encode best practices, are violated among the top one million most popular sites. We find that about 40% of Alexa top 100 pages and as many as 70% of Alexa top one million violate the *unsized-media* policy and more than 65% of the top one million pages include blocking scripts. Overall, the top one million pages violate 7.57 policies on average. These findings show the potential for new research on how to build mitigations for violations of best practices.

HotMobile '22, March 9–10, 2022, Tempe, AZ, USA

ACM ISBN 978-1-4503-9218-1/22/03.
https://doi.org/10.1145/3508396.3512878

[1]The best practices and the level of the improvement depend on devices and network constraints.

Our follow-up work will expand our initial findings presented in this paper, toward a solution that will use the best best practices encoded in the various policies as a way to trigger interventions by the browser to enforce best practices while retaining the intended experience of the site authors.

2 BACKGROUND

The relatively low QoE with the mobile web combined with its potential impact on user engagement (and profit) [11] have served as motivation for a range of industry and academic efforts that, despite their demonstrated benefits, have seen limited adoption. We argue that *Document and Permission Policies* may offer a readily-available and effective alternative.

2.1 Sources of poor performance and metrics

Many performance-related factors contribute to mobile web QoE, from paint latency to user input delay. In this paper, we focus on two, paint latency and reflow jank, because they weigh heavily on the overall experience, and because there are industry standard metrics to evaluate them.

By paint latency, we mean the time between when a navigation to a new page begins, and when content is painted to the screen. We use two metrics to measure paint latency, Largest Contentful Paint (LCP), which measures the time until the primary content of the page is painted, and Speed Index (SI), which measures how quickly page content becomes visible and stable.

Several sources contribute to paint latency, including unnecessarily large downloads like unoptimized images and large scripts, long chains of dependent downloads that block rendering, unimportant downloads that compete for bandwidth with the important ones, and script execution that blocks layout.

By reflow jank, our other focus, we mean the poor experience that occurs when elements of a page are painted to the screen, and are subsequently shifted or resized, a process called layout reflow. Layout reflows occur when, after painting some elements to the screen, the browser learns about elements and their sizes that affect the overall layout. Front-end design that does not consider reflows (e.g., using unsized media) may cause a series of unwanted layout shifts, which introduce an unsatisfactory visual experience as well as a slowing down of the rendering process. We evaluate reflow jank with Cumulative Layout Shift (CLS), which provides a score for the level of jankiness experienced due to layout changes occurring during a page load.

2.2 Limitations in previous approaches

There have been many industrial as well as academic efforts to resolve these issues. A content platform that can force the participants to follow best practices is a popular approach [4, 9, 17]. These can guarantee performant pages, but the benefit is limited to their participants, and to benefit, participants must rework their content to be compatible with the platform.

In order to reduce the time spent fetching and processing chains of dependent resources, particularly on slower networks and devices, RDR systems are proposed, which resolve the dependency graph of a page at a proxy [32, 33, 37, 39]. However, RDR solutions have not been widely adopted due to requiring proxy deployment and configuration, and being incompatible with HTTPS, unless the end-to-end security of the protocol is compromised.

In addition, some solutions adopt aggressive client-side caching [19, 26, 29], but require additional storage and primarily benefit only a handful of the most frequently visited pages.

2.3 Permissions and Document Policies

Permissions and Document Policies are specifications that allow developers to control certain features and APIs on a browser [14, 15], and are currently W3C working drafts. They were initially proposed as Feature Policy [10] in 2016 and are a standard being implemented by several browsers including Chrome, Firefox, and Safari.

A policy-controlled feature is an API or behavior which can be enabled or disabled in a document, or delegated to embedded documents, by referring to it in a permissions policy. The Permissions Policy for a document is a set of features and a list of allowed domains to which the features can be delegated. For example, Permissions Policy has a camera feature that allows or denies the usage of the camera by the browser. By enabling or disabling a policy, developers can ensure that their web applications behave as intended, even when third-party content is written in a way that does not; applied policy can report when a policy is violated, and can intervene on the entire page to force the page to be compliant.

There are about 35 features (5 others have been proposed) in Permission Policy[2] and they vary from simple features like *camera* and *geolocation* to performance-related ones such as *execution-while-out-of-viewport*.

Document Policy, unlike Permissions Policy which focuses on the permissions with binary choices (*i.e.,* yes or no), covers policies that are more related to performance aspects such as *synchronous scripts* and *oversized images* with possibly ranged parameters (e.g., 2.0X or 3.0X). Currently, there are 13 features in Document Policy[3]. When a feature in Document Policy is enforced, it typically blocks the element of the web page that violates the policy; we use the term "element" generically in this paper to refer to any component of a page, e.g., script, or an image, that can be governed by policy. When blocking happens, it may degrade the experience or break the result page depending on the nature of the policy (§ 3.1).

In order to apply policies to a webpage, two options are provided for developers; header policies and container policies. A header policy is a list of features to control, delivered via an HTTP header with the page response. This declares the given policies for the entire document. A container policy applies to a browsing context container (*i.e.,* iframes) with the "allow" attribute. In addition to header policies and container policies, there is a JavaScript API as well that can observe the policy states in the client-side code.

3 REPURPOSING POLICIES

While originally designed for the developers, both developers and users could use Document and Permission policies to enforce best practices for better mobile web performance. Leveraging these policies as a "scalpel," developers could adjust their content and/or the content made available by a third-party provider as they intend (e.g., to prevent third-party ads from blocking page rendering),

[2]The full list of policies can be found in their webpage [16]
[3]The full list of policies can be found in the Chromium code [13]

while end users or their user agents could control overall website behavior to suit local and network resource constraints.

Unlike all-or-nothing approaches, policies would allow content providers or users to selectively apply each option based on their preferences, allowing them to trade aspects other than performance such as design and usability.

Document and Permission policies demand little from developers or users, are very lightweight, and require little effort to adopt as they are readily available in some browsers (e.g., Chrome). In those browsers, simply manipulating the header of the server response can change these policy configurations when the site is loaded.

3.1 QoE-impacting Policies

Although there are nearly 50 policies, not all could have a significant impact on mobile web QoE. For instance, while controlling access to user's camera and audio are undoubtedly important features, they likely have little effect on the Speed Index of a website.

A careful review of the different features, however, reveals a subset of features potentially useful in terms of QoE, including: *oversized-images*, *unsized-media*, *layout-animations*, *sync-script*, and *font-display-late-swap*.

Oversized-images limits the maximum size images can be, under the given container (e.g., <img> tag) and viewport size, as unoptimized images are one of the main sources of wasted bandwidth. When an image violates this policy, it is replaced by a placeholder. Other similar image-related policies such as *lossy-images-max-bpp*, *lossless-images-max-bpp*, and *lossless-images-strict-max-bpp*, limit maximum bits per pixel and show a placeholder for the violations, potentially having a similar QoE impact to *oversized-images*.

Unsized-media applies to all media objects, images and videos, and sets them to a default size (300×150 pixels) if they do not have an explicitly stated size. This feature should impact QoE since when media above the fold are unsized, they may cause layout shifts that impact the user experience.

Web fonts can also block the loading of a page. In addition, fonts can commonly negatively impact QoE by loading in after the page has already rendered with a built-in font substitute, and triggering the page to re-render. *font-display-late-swap* prevents this by disabling the later swap of the font if its loading time is long (*i.e.,* over 100ms).

Layout-animations control animations that update layout since they consume significant CPU resources. When the policy is enabled, such animations just show the initial and final states of the animation and none of the animation's intermediate stages.

Finally, *sync-script* blocks synchronous Javascript (*i.e.,* Javascript code without async or defer attributes) when the feature is on.

4 POLICY EXPERIMENTATION SETUP

In the following paragraphs, we describe the experimental setup we have built to evaluate the potential impact of enabling different policies, on a variety of pages and, potentially, over a range of network conditions.

4.1 A Chrome Extension

To experiment with the performance impact of different policies, we built a dedicated Chrome extension that can manage the application

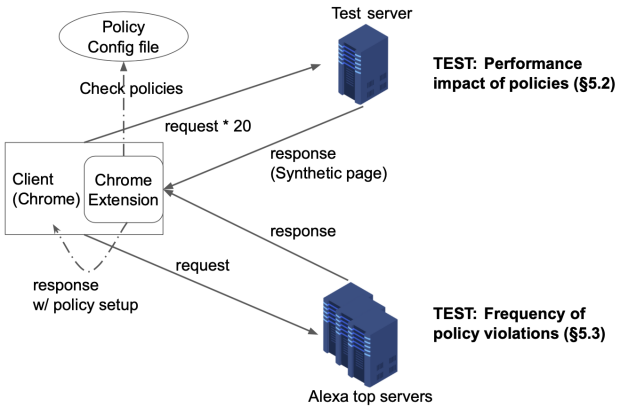


Figure 1: System diagram for the performance tests

of policies to the current page directly within the browser. This approach has a number of advantages: Chrome is the browser that most closely follows the evolution of policies; a client-side extension lets us test different policies on any web page we choose; it is relatively straightforward to build a prototype to run our analysis of a policy's impact on web QoE; and this model makes it available to run a network emulation beyond a mobile setting, so to expand our study to other configurations (e.g., in-flight wifi [38]).

While the extension is activated, whenever a web page is requested from a server, the extension modifies the header of the incoming HTTP response to add the given list of policies before the response begins to be processed., as Fig 1 presents. The overhead imposed by the extension is negligible, since modifying the incoming header is all this extension does, and further, this overhead is constantly added to every web request even when no policy is enabled (thus factoring out in the comparative analysis). With the policy header in the incoming server response, the browser imposes the enabled policies on the loaded page. While this may not be a desirable result for some users, the main purpose of this test is to learn the upper bound on the performance improvement expected from enforcing a policy, and second, users on challenged networks and/or low-end devices may prefer to access the content of the webpage with lower delays if at the cost of some design features.

4.2 Experimental setup

We carry out our measurements, emulating a mobile environment, on a MacBook Pro 2020 with a stable wired network. The automated testing is established with Puppeteer [21] which launches the actual Chrome browser without caching. Lighthouse [20] – a widely adopted web performance auditing tool – takes over control to configure the mobile environment and collect auditing results. In our tests, we set a mobile user agent and emulate a mobile viewport (Motorola G4) as some policies are affected by the viewport size (e.g., *oversized-images*). We also throttle mobile network and CPU configurations using the options provided by Lighthouse. Specifically, we use the "Slow 4G" configuration[4][5] that represents the bottom

[4]Regular 4G is pretty fast, so we targets the fastest setting among the "slow" network. The performance benefit would be larger in the poorer network.
[5]We follow connectivity configurations in WebPageTest, which is popular for the web performance testing [27].

25% of 4G connections and top 25% of 3G connections; the latency is 150ms, and upload/download throughput is 750Kbps/1.6Mbps respectively. A comparison between Lighthouse and Netem [18] throttling shows roughly similar results in many networking scenarios [8], with last-mile network emulation.

5 POLICY POTENTIAL

In the following paragraphs, we present evaluation results from an analysis of the potential impact of policies on mobile performance. We use both a set of synthetic and real pages in our evaluation. We employ synthetic pages to characterize the potential impact of individual policies and then carry out a study of policy violations on the one million most popular pages to estimate the potential value of this approach in the wild.

5.1 Tested Pages

The synthetic pages we employ have elements in the first mobile viewport, with each element included in violation of a particular policy. The images included are not optimized to show the performance impact of bad practices on webpages more clearly. They are 500-700KB, whereas optimized images typically are under 100KB [42].

We also use the Alexa top-ranked pages to evaluate how commonly pages in the wild violate best practices encoded in the policies under study. We take the top one million pages and cluster them into 5 groups: Top-100 pages (Alexa 100), pages ranked between 100 and 1,000 (Alexa 1K), 1,000 and 10,000 (Alexa 10K), 10,000 and 100,000 (Alexa 100K), and 100,000 to 1,000,000 (Alexa 1M). From each bin, 100 pages that load without failures and without interstitials are randomly sampled for our testing. We would expect that lower-ranked pages would have a larger number of violations, on average, than those higher in the ranking, since popular page owners (e.g., Amazon) likely have more resources to maintain their pages and enhance the performance.

5.2 Performance impact of policies

We carry out controlled experiments on synthetic pages with specific policy violations and analyze their impact on SI, LCP and CLS. Figure 2 presents box plots for each metric using 20 measurements for each selected feature. In addition to per-policy tests, the plots also include test results with all policies enabled (the all-policies configuration), and for a baseline, with all policies disabled (the no-policies configuration). Overall, we find that while different policy violations impact different performance metrics differently, the performance impact of policy violations – and thus the potential benefits of enforcing the associated best practices – is consistent across metrics.

The top plot shows SI. It is not surprising that SI improves significantly when large images, synchronous scripts and large fonts are not included. Interestingly, though, applying the layout-animation policy makes SI unexpectedly worse than no-policy. We suspect that this happens due to a bug in Chrome DevTools' frame recording that the SI algorithm in Lighthouse (Speedline [23]) relies upon. Chrome DevTools captures screenshots as page loading progresses until there is no further visual change in the first viewport, but when a CSS animation, which is quickly and constantly moving, exists

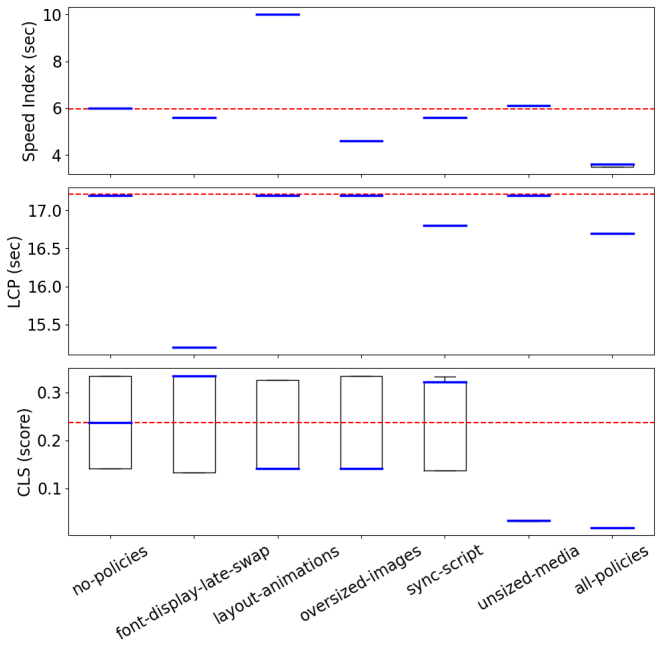


Figure 2: Performance impact of applying each policy. Boxes show the entire distribution. Bold blue denotes the median. Horizontal dashed red denotes the average for no-policies.

and the page is loaded under a slow network condition, screenshot capture stops before the page visually completes. This shortens the time for the last visual change, and thus makes SI much smaller than it is supposed to be. We filed this bug with the Chromium project.

The middle plot presents LCP. *Font-display-late-swap* is interesting as LCP with it enabled outperforms *all-policies*. We conjecture that this is due to the combination with other policies, particularly those associated with images, which are the key determinant of LCP. In other words, image-related policies lead to some unknown negative interactions when working with *font-display-late-swap*, thus *all-policies* reveal worse performance than *font-display-late-swap* only. As expected, we find that the combination of *font-display-late-swap* and *sync-script* present the best result in an additional test.

Unsized-media, which has little impact on other metrics, provides a greater improvement on CLS, simply because *unsized-media* prevents layout shift by forcing all media to be sized. Other test cases without the *unsized-media* policy enabled show two different scores (around 0.15 and 0.33), as they can have a layout shift or two. The one shift happens when the elements are loaded following the displaying order, and the page shifts twice when the element below the unsized media element is presented earlier.

Note that the performance gains we see from enforcing policies (e.g., blocking a large image) may be different from the gains we'd expect when we apply interventions aimed at conforming with policy while retaining the intended user experience (e.g., resizing the image before transferring it to the browser). The gains we present serve as an existence proof that enforcing policy can help

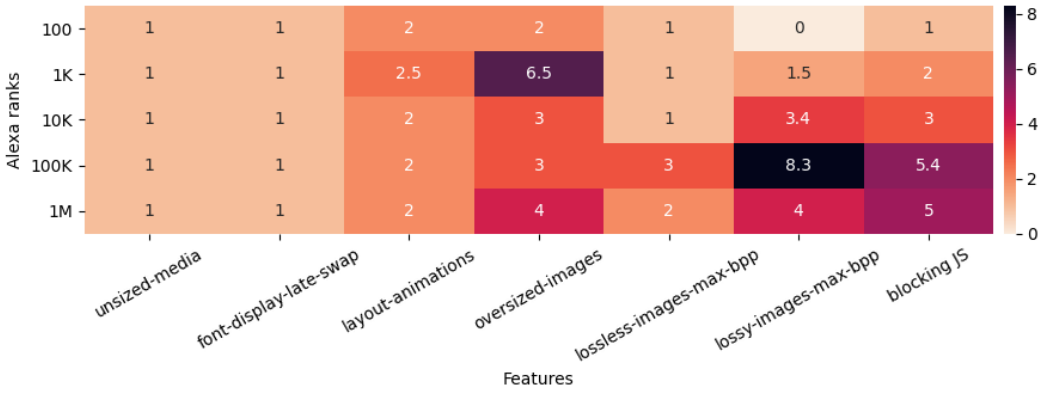


Figure 3: Common violations (90th percentile) per sampled cluster of Alexa pages

performance, and the magnitude of these gains is likely close to or at the best case for the violations encountered.

5.3 How common are policy violations?

Next, we collect the number of violations of best practices per page from five groups of sampled Alexa pages (§ 5.1). Figure 3 shows the number of violations of each policy per page at the 90th percentile. We include *lossless-images-max-bpp* and *lossy-images-max-bpp*, which we didn't include in § 5.2, to see their frequency in the wild. Note that the number of image-associated violations is a conservative estimate, as we count only the first policy violation we encounter on an element in a page, and ignore subsequent violations (e.g., an *oversized-image* violation, determined based on the container and viewport size, may hide an *unsized-image* violation). Moreover, here we look for blocking Javascript instead of synchronous Javascript. It is because, for pages in the wild, synchronous scripts are very common, but those on the critical path (a much smaller subset) are particularly harmful to the performance as they block rendering events and add significant delays to displaying content. We find pages pervasively violate best practices, and such violations are even more common in lower-ranked pages.

We explore the distribution of violations of each policy across the Alexa ranks. Figure 4(a) presents the results for blocking scripts across the Alexa top one million. We find as many as 20 blocking script violations in the worst case. More than 60% of the pages in Alexa 1M and over 20% of Alexa 100 pages have at least one blocking script. Figure 4(b) shows the results for *unsized-media*. The trend, in this case, is slightly different from blocking scripts - with fewer repeated violations per page but more pages with violations with 50% in top 100 and 70% in top one million pages showing a violation.

Figure 5 shows the SI improvement when enabling all policies considered (versus disabling them all) for pages in different sections of the Alexa top one million. Each Alexa subset shows a substantial improvement at the median (70%~85%) suggesting the approach has potential even for popular sites. In general, top Alexa pages show a higher improvement percentage than lower-ranked pages as they are already highly optimized, and thus a small absolute performance improvement can result in relatively large percentage improvement.

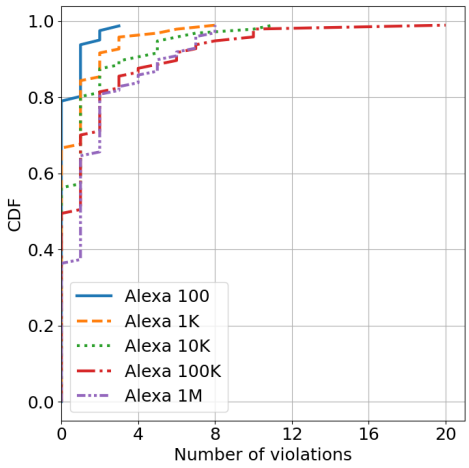
6 RELATED WORK

There is a significant body of prior work focused on improving mobile web performance; major areas include providing better options for developers, automatically restructuring pages, and predictive actions such as prefetching.

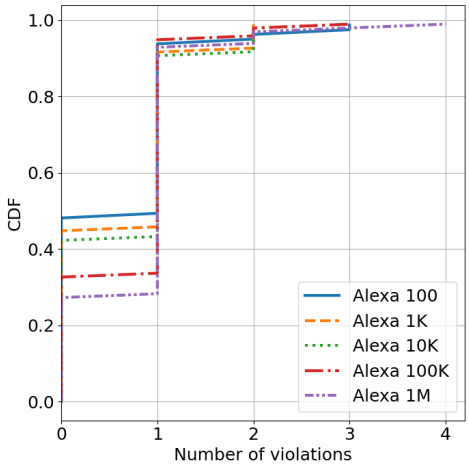
Frameworks and tools for developers: A popular approach for improving web page performance is to use a web development framework that incorporates best performance practices. Auditing tools for web pages such as Lighthouse [20], WebPageTest [28], and Mahimahi [31] allow developers to learn about the performance of a web page in order to improve it. In addition, JavaScript frameworks like React [35] make page rendering more efficient using a virtual DOM that helps the actual DOM reduce intermediate rendering events. Apple news [4], Facebook Instant Articles [17] and Google Accelerated Mobile Project (AMP) [9] incentivize participating content providers to generate content fitting their standards (that follow best practices) with the promise of boosting the readership of their pages. AMP, in particular, provides content creators with a stripped-down and optimized version of standard web development APIs so that the pages they publish can benefit from multiple performance-improving techniques such as preventing blocking JS, caching, and pre-rendering. While AMP pages yield significant performance benefits [24], reconstructing existing pages may be infeasible for minor content providers lacking developer resources. In contrast, our approach would be easily adoptable so that any developers can immediately benefit from it.

Interventions: Another popular option is automatically rewriting a web page or changing resources in-flight to make the page more performant. Prophecy [30] and Opera Mini [34] precompute expensive tasks on the server that are not required to be processed on the client, and serve a pre-processed version of the page. Researchers have proposed a variety of client-side interventions and optimizations such as reducing HTTP requests [29], using app-inspired static templating [26] and reducing redundant computation [19]. There are also proxy-based approaches like Flywheel [1] and FlexiWeb [41] that compress responses in-flight to reduce the data usage of users and expedite loading.

Interventions have the issue that they may harm the user experience because they may change the page to something other



(a) Blocking JavaScript



(b) Unsized media

Figure 4: The example of the distribution of violations over different Alexa ranks

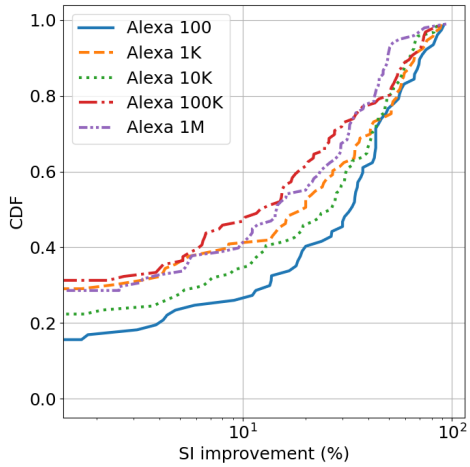


Figure 5: Speed Index improvement (%) per page in log scale.

than what the author intended. Server-driven interventions, in particular, have the drawback of requiring the deployment of servers. Our approach could help identify useful interventions and when to apply them in response to specific policy violations and current user's device/network conditions.

Prefetching and prerendering: Several approaches accelerate loading with prefetching and/or prerendering web content. Some efforts use remote dependency resolution (RDR), which resolves the dependency graph of a page at a proxy or server, to enable delivering dependent resources before the browser would otherwise request them, which reduces round trip delays. These efforts include the use of a server's aid instead of a proxy's [37], and recent work that enables RDR selectively [32] and focuses on third-party security [25]. Industry has proactively tried to provide their own solutions for better performance [33, 39] using specialized web browsers to get RDR's help. Google's AMP pre-renders above-the-fold page resources from the Google search result page before the user explicitly chooses to explore the page to make the page loading instantaneous. Prefetching and prerendering significantly improve performance, but they require powerful proxy or server, and prediction of the user's choice is not only challenging but privacy-concerning, and possibly leading to wasted energy and data usage [36].

We argue that our ongoing work on policies can form the basis of a readily deployable, server, or client-side solution, that requires no infrastructure, supports a gradual adoption of best practices, and can yield immediate benefits on mobile web performance.

7 CONCLUSION AND FUTURE WORK

We argue that *Document and Permissions Policies* may offer the basis for a readily-available, easily-adoptable way to improve mobile web performance. In this paper, we show the potential performance cost of violating the well-understood practices encoded in these policies, and how common such violations are in today's web.

Building on our preliminary findings, we envision the browser applying policies automatically to notify developers of issues and provide actionable feedback and to intervene on their behalf with performance-improving page changes for better QoE when developers do not address the issues we identify. Toward such performance improvements, we will determine what the browser should do to improve performance when policies identify violations of best practices and invent new interventions that are situation appropriate. The interventions should be carefully designed and (automatically) applied to strike the right balance between addressing the performance issues (which may be more severe under network and device constraints, and with some pages relative to others) while retaining enough (ideally all) of the intended user experience. We hope this work will provide the right incentives and mitigations to move the web ecosystem towards being more performant by following best practices.

ACKNOWLEDGEMENT

The work of Byungjin Jun is supported by 2021 Google PhD Fellowship.

REFERENCES

[1] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. 2015. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, 367–380.

[2] Sohaib Ahmad, Abdul Lateef Haamid, Zafar Ayyub Qazi, Zhenyu Zhou, Theophilus Benson, and Ihsan Ayyub Qazi. 2016. A View from the Other Side: Understanding Mobile Phone Characteristics in the Developing World. In *Proc. of ACM IMC*.

[3] Daniel An. 2018. Find out how you stack up to new industry benchmarks for mobile page speed. (2018). https://tinyurl.com/y6d38by3

[4] Apple. [n. d.]. Apple News+. ([n. d.]). Retrieved October 27, 2021 from https://www.apple.com/apple-news/

[5] HTTP Archive. [n. d.]. Report: Loading Speed. ([n. d.]). Retrieved October 27, 2021 from https://httparchive.org/reports/loading-speed

[6] HTTP Archive. [n. d.]. Report: Page Weight. ([n. d.]). Retrieved November 1, 2021 from https://httparchive.org/reports/page-weight

[7] Kalvin Bahia. 2020. The State of Mobile Internet Connectivity 2020. (2020). https://tinyurl.com/3u7k4xbk

[8] Debug Bear. 2019. Network throttling: DevTools vs. Lighthouse vs. Netem. https://tinyurl.com/ynbjmn3f. (2019).

[9] David Besbris. 2015. Introducing the Accelerated Mobile Pages Project, for a faster, open mobile web. (2015). Google: Official Blog.

[10] Eric Bidelman. 2018. Introduction to Feature Policy. (2018). Retrieved October 27, 2021 from https://tinyurl.com/235s8a2h

[11] Jake Brutlag. 2009. Speed matters for Google web search. http://tiny.cc/nn4usz. (2009).

[12] Michael Butkiewicz, Harsha V. Madhyastha, and Vyas Sekar. 2011. Understanding Website Complexity: Measurements, Metrics, and Implications. In *Proc. of ACM IMC*.

[13] Ian Clelland. [n. d.]. Document Policy Features. ([n. d.]). Retrieved October 27, 2021 from https://tinyurl.com/puedpwyz

[14] Ian Clelland. 2020. Permissions Policy, W3C Editor's Draft. (2020). Retrieved October 27, 2021 from https://tinyurl.com/2xwa6p7d

[15] Ian Clelland. 2021. Document Policy, W3C Editor's Draft. (2021). Retrieved October 27, 2021 from https://wicg.github.io/document-policy

[16] Ian Clelland. 2021. Policy Controlled Features. (2021). Retrieved October 27, 2021 from https://tinyurl.com/yyn5z8s

[17] Facebook. [n. d.]. Instant Articles. ([n. d.]). Retrieved October 27, 2021 from https://tinyurl.com/458w8jx3

[18] Linux Foundation. [n. d.]. Netem. ([n. d.]). Retrieved October 27, 2021 from https://wiki.linuxfoundation.org/networking/netem

[19] Ayush Goel, Vaspol Ruamviboonsuk, Ravi Netravali, and Harsha V. Madhyastha. 2021. Rethinking Client-Side Caching for the Mobile Web. In *Proc. of HotMobile*.

[20] Google. [n. d.]. Lighthouse. ([n. d.]). Retrieved January 31 2022 from https://developers.google.com/web/tools/lighthouse

[21] Google. [n. d.]. Puppeteer. ([n. d.]). Retrieved January 31 2022 from https://pptr.dev

[22] Google. 2019. Speed Index. https://web.dev/speed-index/. (2019).

[23] Paul Irish. [n. d.]. Speedline. ([n. d.]). Retrieved January 27, 2022 from https://github.com/paulirish/speedline

[24] Byungjin Jun, Fabián E. Bustamante, Sung Yoon Whang, and Zachary S. Bischof. 2019. AMP up your Mobile Web Experience: Characterizing the Impact of Google's Accelerated Mobile Project. In *Proc. of ACM MobiCom*.

[25] Ronny Ko, Blake Loring, Ravi Netravali, and James Mickens. 2021. Oblique: Accelerating Page Loads Using Symbolic Execution. In *Proc. of USENIX NSDI*.

[26] Shaghayegh Mardani, Mayank Singh, and Ravi Netravali. 2020. Fawkes: Faster Mobile Page Loads via App-Inspired Static Templating. In *Proc. of USENIX NSDI*.

[27] Patrick Meenan. [n. d.]. ([n. d.]). Retrieved January 18, 2022 from https://github.com/WPO-Foundation/webpagetest/blob/master/www/settings/connectivity.ini.sample

[28] Patrick Meenan. [n. d.]. WebPageTest. ([n. d.]). Retrieved January 31, 2022 from https://www.webpagetest.org

[29] James Mickens. 2010. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads. In *Proc. of USENIX WebApps*.

[30] Ravi Netravali and James Mickens. 2018. Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 249–266.

[31] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proc. of USENIX ATC*.

[32] Ravi Netravali, Anirudh Sivaraman, James Mickens, and Hari Balakrishnan. 2019. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. In *Proc. of ACM MobiSys*.

[33] Opera. [n. d.]. Opera Mini. ([n. d.]). Retrieved October 27, 2021 from https://www.opera.com/mobile/mini

[34] Opera. [n. d.]. Opera Mini. ([n. d.]). Retrieved January 31, 2022 from https://www.opera.com/browsers/opera-mini

[35] Meta Platforms. [n. d.]. React. ([n. d.]). Retrieved January 31, 2022 from https://reactjs.org/

[36] Lenin Ravindranath, Sharad Agarwal, Jitendra Padhye, and Christopher Riederer. 2013. Give in to Procrastination and Stop Prefetching. In *Proc. of ACM HotNets*.

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

[38] John P. Rula, Fabián E. Bustamante, James Newman, Arash Molavi Khaki, and Dave Choffnes. 2018. Mile High WiFI: A First Look At In-Flight Internet Connectivity Conference. In *Proc. of WWW*.

[39] Amazon Web Services. [n. d.]. What is Amazon Silk? ([n. d.]). Retrieved October 27, 2021 from https://tinyurl.com/4d9bppak

[40] Ronnie Simpson. 2016. Mobile and tablet internet usage exceeds desktop for first time worldwide. http://tiny.cc/jn4usz. (2016).

[41] Shailendra Singh, Harsha V. Madhyastha, Srikanth V. Krishnamurthy, and Ramesh Govindan. 2015. FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom '15)*. Association for Computing Machinery, 604–616.

[42] John Teague. [n. d.]. Web Almanac - Page Weight. ([n. d.]). Retrieved January 19, 2022 from https://almanac.httparchive.org/en/2021/page-weight#file-formats

[43] W3Techs. [n. d.]. Usage statistics of AMP for websites. ([n. d.]). Retrieved November 1, 2021 from https://tinyurl.com/y2x9vdkv

[44] Philip Walton and Milica Mihajlija. 2019. Cumulative Layout Shift (CLS). https://web.dev/cls/. (2019).