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

Shaghayegh Mardani[*], Mayank Singh[†], Ravi Netravali[*]
[*]UCLA, [†]IIT Delhi

Abstract

Despite the rapid increase in mobile web traffic, page loads still fall short of user performance expectations. State-of-the-art web accelerators optimize computation or network fetches that occur *after* a page's HTML has been fetched. However, clients still suffer multiple round trips and server processing delays to fetch that HTML; during that time, a browser cannot display any visual content, frustrating users. This problem persists in warm cache settings since HTML is most often marked as uncacheable because it usually embeds a mixture of static and dynamic content.

Inspired by mobile apps, where static content (e.g., layout templates) is cached and immediately rendered while dynamic content (e.g., news headlines) is fetched, we built Fawkes. Fawkes leverages our measurement study finding that 75% of HTML content remains unchanged across page loads spread 1 week apart. With Fawkes, web servers extract static, cacheable HTML templates for their pages offline, and online they generate dynamic patches which express the updates required to transform those templates into the latest page versions. Fawkes works on unmodified browsers, using a JavaScript library inside each template to asynchronously apply updates while ensuring that JavaScript code only sees the state that it would have in a default page load despite downstream content having already been loaded. Across a wide range of pages, phones, and live wireless networks, Fawkes improves interactivity metrics such as Speed Index and Time-to-first-paint by 46% and 64% at the median in warm cache settings; results are 24% and 62% in cold cache settings. Further, Fawkes outperforms recent server push and proxy systems on these metrics by 10%-24% and 69%-73%.

1 Introduction

Mobile web browsing has rapidly grown in popularity, generating more traffic than its desktop counterpart [18, 20, 57]. Given the importance of mobile web speeds for both user satisfaction [11, 12, 23] and content provider revenue [21], many systems have been developed by both industry and academia to accelerate page loads. Prior approaches have focused on pushing content to clients ahead of time [61, 70, 76, 19], compressing data between clients and servers [4, 67, 63], intelligent dependency-aware request scheduling [14, 42], offloading tasks to proxy servers [47, 65, 10, 6], and rewriting pages for the mobile setting (either by automatically serving post-processed objects to clients [71, 43, 51], or by manually modifying pages to follow mobile-focused guidelines [24, 34]). Yet despite these efforts, mobile page loads continue to fall short of user expectations in practice. Even on a state-of-the-art phone and LTE cellular network, the median page still takes over 10 seconds to load [7, 61].

Figure 1: **Comparing the mobile app and mobile web browser loading processes for BBC News over an LTE cellular network.**

Our key observation is that, while existing optimizations are effective at reducing network fetch delays and client-side computation costs during page loads, *they all ignore a large and fundamental bottleneck in the page load process: the download of a page's top-level HTML file*. To fetch a page's top-level HTML, a browser often incurs multiple network round trips for connection setup (e.g., DNS lookups, TCP and TLS handshakes), server processing delays to generate and serve content, and transmission time. These tasks can sum to delays of hundreds of milliseconds, particularly on high-latency mobile links.[1] Only after receiving and parsing a page's HTML object can the browser discover subsequent objects to fetch and evaluate, make use of previously cached objects, or render any content to the blank screen. Thus, from a client's perspective, the entire page load process is blocked on downloading the page's top-level HTML object. This is true even in warm cache scenarios, since HTML objects are most often marked as uncacheable [44] (§2).

Eliminating these early-stage inefficiencies would be fruitful for two reasons. First, overall load times would reduce since client-side computation and rendering tasks for cached content could begin earlier and be overlapped with network and server-side processing delays for new content; the CPU is essentially idle as top-level HTML files are fetched in traditional page loads. Second, and more importantly, browsers could immediately display static content, rather than showing only a blank screen as the HTML is fetched (Figure 1). This is critical as numerous web user studies and recent performance metrics highlight user emphasis on content becoming visible quickly and progressively [46, 25, 35, 49, 66].

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

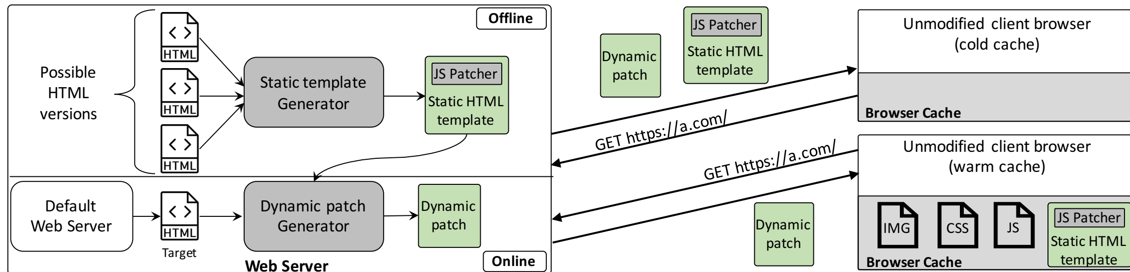


Figure 2: **Overview of cold and warm cache page loads with Fawkes. Servers return static, cacheable HTML templates, as well as uncacheable dynamic patch files that list the updates required to convert those templates into the latest page. Updates are performed dynamically using the Fawkes JavaScript patcher library that is embedded in the templates.**

To enable these benefits, we draw inspiration from mobile apps which, despite sharing many components with the mobile web (e.g., client devices, networks, content), are able to deliver lower startup delays (Figure 1). Apps reduce startup times by aggressively separating static content from dynamic content. At the start of executing a task (akin to loading a page), an app will issue a request for dynamic content *in parallel* with rendering locally cached content like structural templates, images, and banners. Once downloaded, dynamic content is used to patch the already-displayed static content.

Like apps, web pages already cache significant amounts of content across loads: 63% and 93% of the objects and bytes are cacheable on the median page. Yet startup times in warm cache page loads remain high due to download delays of top-level HTML files (§2). But why are HTML objects marked as uncacheable? The reason is that they typically bundle static content defining basic page structure with dynamic content (e.g., news headlines or search results). HTML which embeds dynamic content must be uncacheable so clients see the most recent page. Thus, at first glance, it appears that addressing this challenge with app-like templating would require a rethink of how web pages are written. However, our measurement study (§2) reveals that web pages are already highly amenable to such an approach given the large structural and content similarities for HTML objects across loads of a page. For instance, 75% of HTML tags on the median top 500 page have fixed attributes and positions across 1 week, and could thus be separated into static templates.

We present **Fawkes**, a new web acceleration system that modifies the early steps in the page load process to mirror that of mobile apps (Figure 2). Fawkes optimizes page loads in a two-step process. In the first phase, which is performed offline, web servers *automatically* produce static, cacheable HTML templates, which capture all content that remains unchanged across versions of a page's top-level HTML. The second phase occurs during a client page load; servers generate dynamic patches, which express the updates (i.e., DOM transformations) required to convert template page state into the latest version of a page. During cold cache page loads, browsers download precomputed templates while dynamic patches are being produced, and can quickly begin rendering template content and fetching referenced external objects as patches are pushed. In warm cache settings, browsers can immediately render/evaluate templates and referenced cached objects while asynchronously downloading the dynamic patch needed to generate the final page.

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

- On the server-side, generating templates is difficult: traditional tree comparison algorithms [75, 36, 16, 54, 55] do not consider invariants involving a page's JavaScript and DOM state, but templates execute to completion prior to patches being applied and thus must be *internally consistent*. For example, removing an attribute on an HTML tag can trigger runtime errors if downstream JavaScript code accesses that attribute; an acceptable template must keep or omit both of these components. In addition, graph algorithms are far too slow to be used for online patch generation. Instead, Fawkes uses an empirically-motivated heuristic which trades off patch generation time for patch optimality (i.e., number of operations; note that the final page is unchanged). Our insight is that tags largely remain on the same depth level of the tree as HTML files evolve over time. This enables Fawkes to use a breadth-first-search variant which generates patches 2 orders of magnitude faster (in 20 ms) with comparable content.
- On the client-side, each static template embeds a special JavaScript library which Fawkes uses to asynchronously download dynamic patches and apply the listed updates. The primary challenge is in ensuring *view invariance* for JavaScript code that is inserted via an update: that code must see the same JavaScript heap and DOM state as it would have seen during a normal page load. For example, consider an update which adds a `<script>` tag to the top of the HTML template. If that script executes a DOM method that reads DOM state, the return value may include DOM nodes pertaining to downstream tags in the template—this state is already loaded in a Fawkes page load, but not in a default one, and may trigger execution errors. To provide view invariance, Fawkes uses novel shims around DOM methods which prune the DOM state returned by native methods based on knowledge of page structure and the position of the script calling the method.

We evaluated Fawkes using more than 500 real pages, live wireless networks (cellular and WiFi), and two smartphone models. Our experiments reveal that Fawkes significantly accelerates warm cache mobile page loads compared to default browsers: median benefits are 64% for Time-to-first-paint (TTFP), 46% for Speed Index (SI), 26% for Time-to-interactive (TTI), and 22% for page load time (PLT). Despite targeting warm cache settings, Fawkes speeds up cold cache loads by 62%, 24%, 20%, and 17% on the same metrics. Fawkes also outperforms Vroom [61] and WatchTower [47], two recent mobile web accelerators, by 69%-73% and 10%-24% on warm cache TTFP and SI. Importantly, Fawkes is complementary to these approaches; Fawkes with Vroom achieves Fawkes's TTFP and SI benefits, while exceeding Vroom's PLT improvements by 22%. Source code and experimental data for Fawkes are available at https://github.com/fawkes-nsdi20.

2 Motivation

We begin with a range of measurements that illustrate the startup discrepancies between mobile apps and web pages (§2.1), and the amenability of web pages to app-like templating (§2.2). Results used the LTE setup described in §5.1.

2.1 Mobile Apps vs. Mobile Web

We compare the load process of mobile apps and web pages by analyzing equivalent tasks across 10 web services; services were selected by randomly choosing web pages from the Alexa US top 100 list [5], and discarding those without a corresponding mobile app. Our corpus includes news, recommendation platforms, search engines, and social media applications. For each service, we equate loading a homepage with a mobile browser to loading the home screen with the mobile app. When applicable, we also compare equivalent searches on both platforms. We load each task in the mobile app and website back to back, and for each task, we log the time until the first paint to the screen and collect screenshots for three events: the first time either platform displays content to the user (Time-to-first paint, or TTFP), an intermediate checkpoint with additional displayed content, and the time when both platforms reach their final visual state. Mobile app screenshots and paint events are captured via the Apowersoft Recorder [8] and Android Systrace [31] tools, respectively. Since apps have content cached during installation, for fair comparison, we consider warm cache page loads (back to back). Certain mobile apps operate by displaying a logo for several seconds during startup, prior to displaying the home screen. We do not consider such apps here.

Startup delays are far lower with apps than web pages. Across the corpus, our experiments reveal that TTFP values are between 3.5×–5.2× lower with mobile apps than mobile web pages. Figure 1 provides a representative example of loading the home page for BBC News. As shown, despite the high network latencies and potential server processing delays, the mobile app is able to quickly display static content that establishes the overall app layout and logos in under 300 ms. We verified that the reason for this is that the app quickly pulls this content from its local cache while asynchronously fetching dynamic news headlines. In contrast, the BBC web page remains blank as the browser establishes a connection to the backend and downloads the top-level HTML for the page. Only upon receiving the HTML object can the browser begin rendering any static or dynamic content to the screen–this does not begin until 1200 ms, 4× longer than the app.

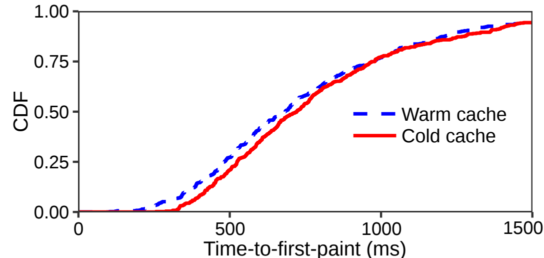


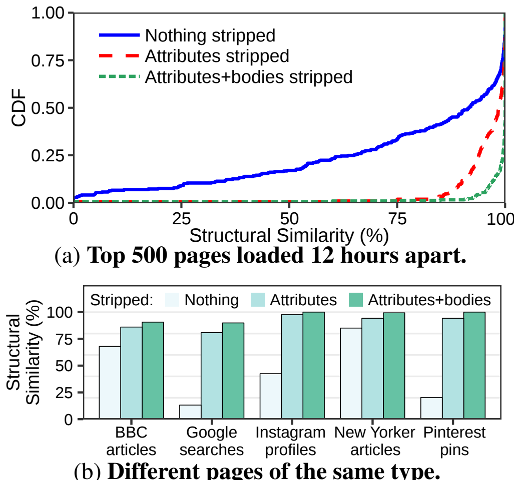
Figure 3: **Caching has minimal impact on time-to-first-paint since browsers cannot render cached content until they download the top-level HTML (typically uncacheable).**

Problem: uncacheable HTML limits caching benefits for the web. The above discrepancies between mobile apps and web pages are indicative of a fundamental difference in the startup tasks on the two platforms. Web HTML objects are used to set the context for the remainder of a page load, establishing render and JavaScript engine processes, creating a DOM tree (i.e., the browser's programmatic representation of the page's HTML) and JavaScript heap, and so on. However, most HTML objects are marked as uncacheable. For example, 72% are uncacheable across back to back loads of the top 500 pages; this number jumps to 85% for loads separated by 5 minutes. As a result, browsers are unable to make use of other objects marked as cacheable (e.g., images, CSS) until they download an HTML object; for reference, 53% and 93% of objects and bytes are cacheable on the median page. Figure 3 illustrates this point: median TTFP values are only 5.3% lower in warm cache scenarios than during cold cache page loads despite so many objects being cached.

2.2 Templating Opportunities for the Web

Motivated by the startup discrepancies between mobile app and web page loads described above, we investigated how amenable web pages are to app optimizations. Our analysis focuses on the feasibility of extracting static templates from HTML objects that can be cached across page loads. We consider two different sets of sites: the Alexa top 500 landing pages [5], and a smaller set which includes 10 pairs of different pages of the same type (e.g., different news articles or search results). More dataset details are provided in §5.

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



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

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

Figure 4: **Structural similarity for HTML files over time. Similarity is defined as the percentage of shared tag sequences (including tag attributes, bodies, and types).**

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

Opportunity: HTML structure and content is largely unchanged over time. Figure 4 shows that HTML objects exhibit high structural similarity. For example, for the median top 500 page, 92% of HTML tags remain identical across loads separated by 12 hours; these numbers jump to 98% and 100% when attributes are stripped alone or with bodies. These trends persist for different pages of the same type. For instance, two different Instagram profile pages exhibit structural similarity of 98% when only attributes are stripped. The trends also persist for other time windows. For example, median similarities in the 1-week setting are 75% and 95% with nothing and attributes stripped, respectively.

Key Takeaways:

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

3 DESIGN

Figure 2 shows the high-level design of Fawkes. Clients use unmodified browsers to load pages as normal. On the server side, websites must run Fawkes to handle incoming client HTTP(S) requests. The server-side Fawkes code performs two primary tasks. For a given page, Fawkes statically analyzes possible variants of the unmodified top-level HTML objects for the page and extracts a single **static HTML template** which maximally captures shared HTML content across versions. The generation of the static HTML template is performed offline, i.e., not during a client page load. Then, when a user loads the page, Fawkes compares the static HTML template to the target HTML, or the one that the default web server would have served without Fawkes, and generates a **dynamic patch**, which is a JSON file with an ordered list of DOM updates required to convert the template page into the target one.

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

3.1 Server-Side Operation

In order to generate static HTML templates, Fawkes's server-side component leverages state-of-the-art tree matching algorithms [54, 55]. The goal of these algorithms is to determine the minimum distance between two (or more) tree structures; recall that HTML files are structured as trees (§2). In particular, these algorithms take as input a set of trees whose nodes are assigned labels. The algorithms then compute a set of operations that, if applied, would efficiently transform the first input tree into the second. Operations typically comprise three primary types: *delete* operations remove a node and connect its children to its parent, *insert* operations add a node to a specific position in the tree, and *rename* operations do not change node positions but instead only alter a node's label. Algorithm execution works much like string edit distance techniques, using dynamic programming and assigning each operation a cost of 1.

Altering tree matching algorithms: Fawkes must alter existing tree matching algorithms in several ways to ensure that they are compatible with HTML and web semantics. First, existing algorithms require each tree node to be labeled with an individual string. However, HTML tags can include state beyond simple tag type (e.g., `<div>` or `<link>`), each of which could be shared across versions of a page. Properties

include attributes (e.g., class) that control the tag's behavior with respect to CSS styling rules and interactions with JavaScript code, and bodies such as inline JavaScript code or text to print. Failing to consider attributes during HTML comparison can result in either broken pages if attributes are incorrectly treated as equivalent, or suboptimal templates if shared attributes are not maximally preserved, i.e., any attribute discrepancy would require omitting a tag. Thus, Fawkes's tree comparison algorithm labels each HTML tag with a (type, [attributes], body) three-tuple.

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

Generating static HTML templates: Fawkes uses the above tree matching framework to generate static HTML templates from a set of HTML files. We describe how to get this input set in §3.3, and for simplicity, describe the approach assuming two input HTML files. To start, Fawkes runs its tree matching algorithm to generate a set of operations which, if applied, would convert *HTML1* to *HTML2*. Fawkes then iterates through HTML1 and selectively applies certain updates to only keep content that is shared across the inputs. Delete operations are directly applied to HTML1 as they represent content which is not shared across versions and thus should not be part of the static HTML template. Similarly, insert operations are ignored as they represent content that must be added to reach HTML2 and is thus not shared. Finally, Fawkes strips all content (tag attributes and bodies) referenced by merge operations as these highlight discrepancies between HTML versions.

While applying these operations and generating static HTML templates, Fawkes must be careful to preserve page semantics and not violate inherent dependencies between page state. In particular, Fawkes must ensure that static HTML templates are *internally consistent* and do not trigger JavaScript execution errors when parsed; this is important as templates are parsed to completion prior to any patch updates being applied. The key challenge is that altering an HTML tag's attributes or body can have downstream effects due to the shared state between JavaScript code and the DOM tree [42]. For example, a downstream <script> tag may access an upstream <p> tag's attribute. Deleting that <p> tag's attribute can thus trigger execution errors when the browser reaches the <script> tag. Similarly, different <script> tags can share state on the JavaScript heap. As a simple example, an upstream tag may define a variable which the downstream tag accesses. Thus, cutting the upstream tag's body can trigger downstream execution errors.

Existing tree matching algorithms are unaware of such dependencies and are agnostic to the HTML execution environment. Thus, Fawkes applies a post-processing step to ensure that such dependencies are not violated in the static HTML template. Fawkes essentially iterates through the static HTML template, and upon detecting an altered tag, cuts downstream <script> tags. Fawkes could leverage techniques like Scout [42] to more precisely characterize the dependencies between tags and JavaScript code in an effort to preserve more state in static HTML templates. However, accurately capturing such fine-grained dependencies would require web servers to also execute HTML content and load pages. Our empirical results motivate that templates derived from static tree analysis sufficiently keep the browser occupied with render and fetch tasks as dynamic patches are fetched, obviating the need for dynamic processing.

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

The tree comparison algorithm described above can produce the desired set of transformation updates that a patch must contain. However, such algorithms are far too slow for patch generation which, unlike template generation, must be performed online, during client page loads. Thus, Fawkes uses a tree comparison heuristic which trades off patch generation time for optimality in terms of number of operations in the patch. The key insight is empirically motivated: we observe that tags most often remain on the same depth level of a tree as HTML files evolve over time. Our analysis of HTML files for 600 pages over a week revealed that, at the median and 95th percentile, 0% and 1% of tags in target HTML files were at a different depth than they were a week earlier. This property favors a breadth-first-search approach over a depth-first-search one, and implies that we need not consider new positions for a tag outside of its current level (as traditional algorithms would). So, for each level in the target HTML file, Fawkes's algorithm works as follows:

1. Create hash maps for both the target and template HTML files that list all of the nodes for a given tag type in the order they appear on that level (from left to right).
2. Iterate through the template's level from left to right and handle each node in one of two ways. If the node's tag type exists in the same level of the target, match this node to the closest node of the same type with the same parent, and remove that node's entry from the target's hash map; if no node has a matching parent, match to the closest node of the same type. Record a *merge* op-

eration by comparing the attributes and bodies for the matched nodes. Else, if the tag type does not exist in the same level of the target, delete this node in the template and record a *delete* operation.

3. Once we reach the end of the template's level, apply *move* operations to order all matched nodes in the template in the same way as they appear in the target; objects which remain in the same position do not require any operation. Note that *move* operations (which simply change the position of a node) are not supported by traditional tree diffing algorithms, and are only enabled by our heuristic's "look ahead" hash maps. Also, note that moves made at this level are immediately reflected in lower levels of the tree as children are reordered.
4. Finally, from left to right, insert any remaining nodes listed in the target's hash map to the appropriate position. Record *insert* operations for these additions.

The key limitation of this heuristic is with respect to nodes moving across levels in the tree. Traditional algorithms can identify such cases, while Fawkes's approach would automatically require a delete at the original level and an insert at the new level. However, as noted above, such transformations are rare. In addition, matching nodes to their closest counterparts with the same parent could be suboptimal: an inserted node in the target can create cascading suboptimal rename and move operations for nodes of that type. Despite such potential inefficiencies, correctness of the final page load is unaffected. We compare this heuristic to standard tree comparison algorithms and other heuristics in §5.5.

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

3.2 Client-Side Operation

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

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

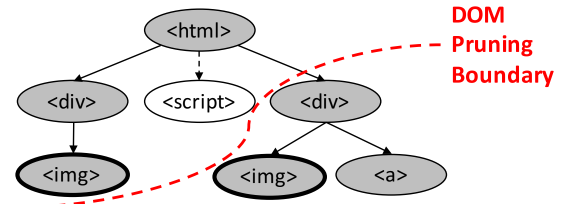


Figure 5: **Update challenge 1: provide view invariance to JavaScript code by hiding downstream DOM state. Shaded nodes are part of the static HTML template. The `<script>` tag is inserted via Fawkes's patcher and calls a DOM method to find `<img>` tags. The native DOM method would return the two nodes outlined in bold, even though the rightmost one would not be returned in the default page load; Fawkes's DOM shims prune the rightmost node from the return value.**

For insert operations, the patcher first creates a new DOM node using document.createElement(), sets the appropriate attributes with Element.setAttribute(), and then adds it to the appropriate position in the DOM tree by calling document.insertBefore(). Adding nodes to the DOM tree can have cascading effects with respect to rendering and layout tasks (both of which are expensive). To mitigate these overheads, Fawkes intelligently looks ahead in the update list to determine if subsequent updates reference the node being added by the current insert update [43, 22]. In these cases, Fawkes constructs a DOM subtree on the JavaScript heap prior to applying the entire subtree to the actual DOM. Fawkes uses similar techniques to handle merge and delete operations.

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

- The first issue is with providing *view invariance* for JavaScript code inserted via an update: that code must see the same JavaScript heap and DOM state as it would have seen during a normal page load. This is challenging since updates are not applied until after a page's static HTML template is entirely parsed. For example, consider Figure 5 where an inserted `<script>` tag invokes a DOM method to read `<img>` tags in the page. The return value for this method would include an `<img>` tag that is downstream in the page's HTML; this divergence from the default page load could trigger JavaScript execution errors or alter page semantics. To handle this, Fawkes shims all DOM methods which return a DOM node or a list of DOM nodes; examples include document.getElementById() and docu-

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

- The second issue is that JavaScript code can alter the DOM tree in ways that affect the child path ids for subsequent updates. The reason is that the child path ids listed in the dynamic patch are based on the static HTML, which does not consider JavaScript execution, but are applied to the dynamic DOM tree which JavaScript code can manipulate (Figure 6). To handle this, Fawkes shims DOM methods that affect DOM structure, either by adding, removing, or relocating nodes; example methods include document.appendChild() and document.insertBefore(). Each shim calls the native method, logs its effect on DOM structure (e.g., the child path of an added node), and then returns the value. When the patcher attempts to apply an update, it first checks this log, and modifies the child paths in the remaining updates based on the listed DOM changes. We note that JavaScript can also invalidate a listed update, e,g,. by replacing a `<div>` tag with an `<img>` tag in the same position. Fawkes's shims detect these alterations, and the patcher discards such updates since JavaScript takes priority over HTML for final page structure.

3.3 Identifying HTML Objects to Consider

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

Option 1: empirical analysis: One approach is for web servers to log the HTML objects that they would serve to clients over time without Fawkes. Fawkes can then periodically recompute a static HTML template based on the latest served HTML files to account for structural modifications that developers make to the page. An advantage of this approach is that the static HTML templates will inherently be based off of the popular pages that are actually served to clients. For instance, if a very rare state configuration would

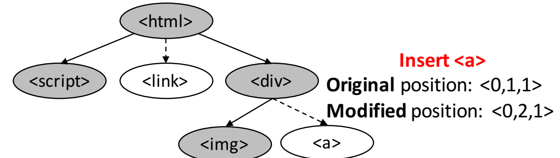


Figure 6: **Update challenge 2: revise update positional information to reflect JavaScript execution. Shaded nodes are part of the static HTML template. Upon execution, the `<script>` tag inserts an adjacent `<link>` tag into the page. Later, when the patcher tries to apply an update to insert an `<a>`, the listed child path id has gone out of date and must be updated.**

alter the structure of a page, most page loads would benefit from *not* considering this version in template generation.

Option 2: leveraging web frameworks: An alternative approach is to leverage the model-view-controller architecture that many popular web frameworks (e.g., Django, Ruby on Rails, and Express) use. In these systems, incoming requests are mapped to a controller function which generates a response by executing application logic code that combines application data and premade HTML blocks. Note that these blocks are small, spanning only a few tags each, and are significantly augmented with HTML tags generated by application logic–this precludes us from using these blocks as our templates. To leverage this structure, we can perform standard static program analysis [62, 37] on application code (particularly the controller for the URL under consideration) to determine the possible HTML block combinations and dynamically produced HTML code that could result for a page.

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

Case studies: Our evaluation (§5) primarily focuses on Option 1. However, to validate the feasibility of the remaining options, we analyzed the source code of two real open source web applications: Reddit [2] and ShareLatex [1]. Both applications follow the MVC model described above, with Reddit using the Python Pylons framework [58] and ShareLatex using NodeJS's Express [50]. For both applications, we wrote custom static analyzers which profile the controller for the sites' landing pages. The output of the profiler is an intermediate template that intertwines HTML code with Python (or JavaScript) logic that, when executed, reads in application variables and outputs a fully formed HTML file. Following branch conditions and unrolling loop bodies in the intermediate template revealed that a ShareLatex project page has 16 possible HTML structures, while Reddit can have over 150. We note that, for this analysis, any tag insertion/deletion or change in tag composition (e.g., an attribute value)

is treated as a new page structure. Consequently, despite the large number of potential HTML structures, both pages are highly amenable to large static templates.

3.4 Subtleties

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

In order to make use of hash information in client requests, Fawkes servers must maintain a mapping of hashes to past static HTML template files which covers the max duration over which the templates are cacheable, i.e., if the templates for a URL are set to be cacheable for 1 day, the Fawkes server must store an entry for each template version served over the past day. Importantly, we expect these storage overheads to be low as our results highlight that templates remain largely unchanged on the order of weeks, and across personalized versions of pages for different users (§2 and §5).

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

Static templates across URLs: In scenarios where static templates are generated for individual URLs by considering their possible HTML variants, templates can be cached directly under the page's URL. However, as we discuss in §2 and §5, Fawkes's template caching approach can provide significant benefits across different URLs of the same page type, e.g., different search result pages or news articles. To support such scenarios efficiently, browsers must slightly alter their caching approach to allow objects to be cacheable across multiple URLs. Websites can specify a regular expression that precisely covers the URLs for which the template applies, and browsers would use the same cached template for any load which matches that regular expression.

4 IMPLEMENTATION

On the server-side, Fawkes's template and dynamic patch generation code are written in 1912 and 462 lines of Python and C++ code, respectively. Both components are implemented as standalone modules for seamless integration with existing web servers and content management platforms [17, 73]. Module inputs are a set of HTML files, and outputs are full formed HTML and JSON files that can be directly shipped to client browsers. For template generation, Fawkes extended the APTED tree comparison tool [55, 56]. HTML parsing and modification are done using Beautiful Soup [60].

On the client-side, Fawkes's JavaScript patcher library consumes 3 KB when compressed with Brotli [27]. The patcher is written entirely using native DOM and JavaScript methods, and is thus compatible with unmodified web browsers. We note that the DOM shims are shared across pages, and thus could be cached as a separate object from each page's static template to reduce bandwidth costs.

5 EVALUATION

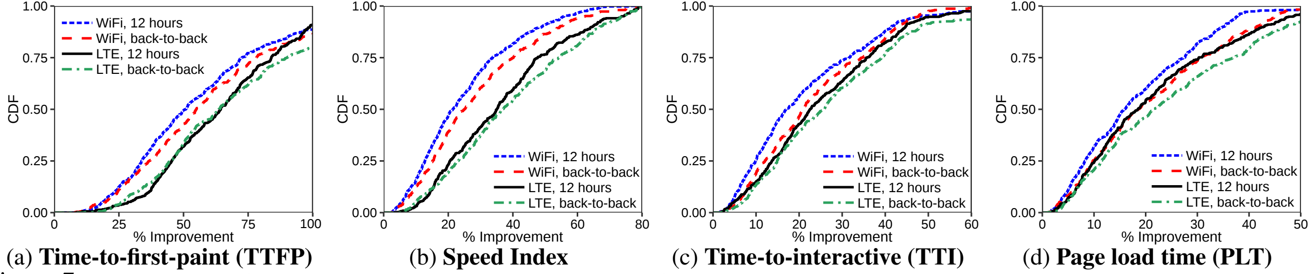
5.1 Methodology

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

Our experiments consider two different sets of pages:

- Alexa top 500 US landing pages [5]. We augment this list with 100 interior pages that were randomly chosen from a pool of 1000 pages generated by a monkey crawler [3] that clicked links on each site's landing page.
- a smaller set of 20 pages that includes pairs of different pages of the same type. Starting from the Alexa top 50 list, we identified page types that have many versions, and manually generated pairs for each one, e.g., two Google search results and two public Twitter profile pages.

In order to create a reproducible test environment and because Fawkes involves page modifications, our evaluation uses the Mahimahi web record-and-replay tool [48]. We recorded versions of each page in our corpus at multiple times to mimic different warm cache scenarios: back to back page loads, and page loads separated by 12 and 24 hours. Mobile-optimized (including AMP [24]) pages were used when available. To replay pages, we hosted the Mahimahi replay environment on a desktop machine. Mobile phones were connected to the desktop machine via both USB tethering and live wireless networks (Verizon LTE and WiFi) with strong signal strength. The desktop initiated page loads on the mobile device using Lighthouse [28], and all control traffic for this was sent over the USB connection. All web and DNS traffic were sent over the live wireless networks into Mahimahi's replay environment. We modified



(a) **Time-to-first-paint (TTFP)** (b) **Speed Index** (c) **Time-to-interactive (TTI)** (d) **Page load time (PLT)**

Figure 7: **Distributions of warm cache (back to back and 12 hour) per-page improvements with Fawkes vs. a default browser (i.e., using each page's default HTML) for 600 pages.**

Mahimahi to faithfully replay the use of HTTP/2 (including server push decisions) and server processing delays observed during recording; details of these modifications are listed in §A.1.

In accordance with §3, in all experiments, Fawkes's templates are generated a priori (i.e., offline). We apply server processing delays for a given template as the median delay observed for objects marked as cacheable in the default load of the page; these objects likely represent premade content. Note that this strategy ensures that templates experience the observed server-side delays that do not relate to content generation, e.g., delays due to high server load. Unless otherwise noted, templates are generated using the first and current versions of a page (i.e., a version at time 0, and the version in the back to back load, 12 hours later, etc.); we present results for other template generation strategies in §5.5. Dynamic patches are generated online by Mahimahi's web servers. Server processing times for patches include both the observed server processing time for the page's original HTML file, as well as the time taken to generate a patch.

We evaluate Fawkes on multiple web performance metrics. Page load time was measured as the time between the `navigationStart` and `onload` JavaScript events. We also consider three state-of-the-art metrics which better relate to user-perceived performance: 1) Speed Index (SI), which represents the time needed to fully render the pixels in the intial view of the page, 2) Time-to-first-contentful-paint (TTFP), which measures the time until the first DOM content is rendered to the screen, and 3) Time-to-interactive (TTI), which measures how quickly a page becomes interactive with rendered content, an idle network, and the ability to immediately support user inputs. All three metrics were measured using pwmetrics [32]. In all experiments, we load each page three times with each system under test, rotating amongst them in round robin fashion; we report numbers per system based on the load with the median page load time.

Correctness and limitations: To ensure a faithful evaluation, we analyzed the pages in our 600-page corpus to identify and exclude those that experience replay errors due to either Mahimahi's (22 pages) or Fawkes's limitations (17 pages). Details about our correctness checks are in §A.2.

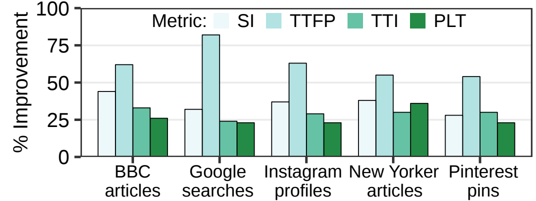


Figure 8: **Warm cache speedups for sites in our smaller corpus.**

5.2 Improving User-Perceived Web Performance

Warm Cache: Figure 7 illustrates Fawkes's ability to improve performance for our 600-page corpus, compared to a default browser, across a variety of web performance metrics and warm cache settings; we omit results for the 24 hour setting due to space constraints, but note that the trends were the same. Benefits with Fawkes are most pronounced on the metrics that evaluate visual loading progress, SI and TTFP. For example, in the 12 hour warm cache setting, median SI improvements are 38% and 22% on the LTE and WiFi networks, respectively. Improvements jump to 67% and 51% for TTFP; these benefits directly characterize Fawkes's immediate rendering of static HTML templates, compared to the lengthy blank screen in a default page load. Table 3 lists the raw time savings pertaining to these improvements.

Despite targeting quick visual feedback, Fawkes's results are also significant for more general web performance metrics like TTI and PLT: median improvements in the 24 hour scenario are 20% and 17% in the LTE setting. The reason is that in warm cache settings, Fawkes enables browsers to utilize network and CPU resources that go idle in standard page loads as HTML objects are being loaded. Browsers can immediately perform required rendering and processing tasks (which are non-negligible on mobile devices [43, 41, 69, 61]) of both template content and referenced cached objects; at the same time, browsers can issue requests for any referenced uncacheable objects to make use of the idle network.

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

As expected, benefits were consistently higher in the back-

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

Table 1: **Analysis of Fawkes's templates and dynamic patches across warm cache scenarios with different time windows. Results list median (95th percentile) values for each property.**

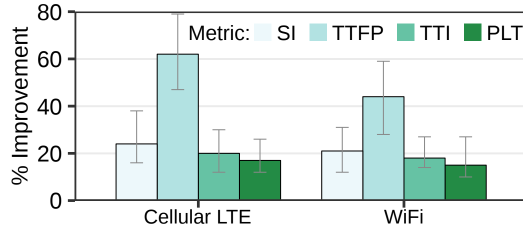


Figure 9: **Cold cache speedups with Fawkes versus a default browser on our 600-page corpus. Bars represent medians, and error bars span from the 25th to 75th percentile.**

to-back warm cache setting than when page loads were separated by 12 or 24 hours. This is because HTML objects undergo fewer changes across back-to-back loads, leading to larger templates and fewer updates (Table 1). Larger templates result in immediate feedback that more closely resembles the final page, as well as increased opportunities to utilize idle CPU and network resources. Note that patch sizes include page content (e.g., inline scripts) to be inserted.

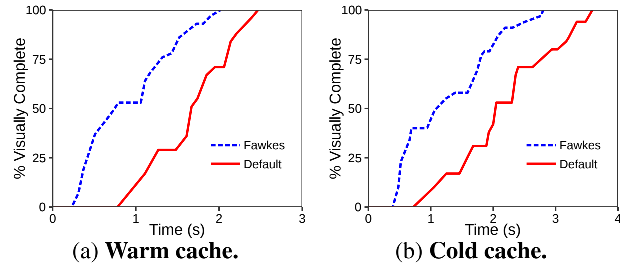
Figure 8 illustrates similar warm cache benefits (over the LTE network) for representative sites in our smaller corpus. Templates are made by consideringdifferent versions of the same page type. We note that TTFP benefits were highest for Google search pages because those pages incur the highest server processing times (for result generation).

Cold Cache: Although Fawkes's template-based approach primarily targets warm cache settings, benefits are significant in cold cache scenarios (Figure 9). For example, median SI and TTFP improvements were 24% and 62% for the LTE network. These results consider templates generated using HTML files generated 24 hours apart. The reason for these benefits mirror those in warm cache settings, but with smaller savings. Since static HTML templates are served faster than dynamic patches, browsers still have a window to perform template rendering and compute tasks with Fawkes while the default page load is blocked. Browsers largely use this time to quickly fetch referenced uncached objects, making better use of the idle network. Like in warm cache settings, SI and TTFP benefits drop to 21% and 44% on the WiFi network due to the decreased network round trip times.

5.3 Understanding Fawkes's Benefits

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

Figure 10 shows warm and cold cache results for a representative site in our corpus, the Yahoo homepage. In the



(a) **Warm cache.** (b) **Cold cache.**

Figure 10: **Visual progress with and without Fawkes for the Yahoo homepage. Warm cache loads were 12 hours apart.**

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

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

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

Patch generation: As described in §3, Fawkes opts to run a tree comparison heuristic rather than a state-of-the-art tree diffing algorithm. Fawkes's heuristic is designed to trade off

Algorithm	# of operations	Execution time (ms)
Fawkes's heuristic	2 (667)	20 (59)
Insert-first heuristic	2 (3013)	30 (70)
Delete-first heuristic	2 (3065)	30 (70)
State-of-the-art tree diffing algorithm (§3)	17 (136)	2717 (19702)

Table 2: **Fawkes's dynamic patch generation heuristic yields a desirable tradeoff between patch generation time and patch optimality, compared to other heuristics and a state-of-the-art tree diffing algorithm (which Fawkes uses for template generation). Results list median (95th percentile) values.**

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

Table 3: **Median warm (cold) cache raw times for our 600-page corpus on an LTE cellular network. All results are in seconds, and warm cache loads are spread by 12 hours.**

patch generation time for optimality (in terms of number of operations). To evaluate Fawkes's heuristic on this tradeoff, we compare it to the tree diffing algorithm Fawkes uses for template generation, and two additional heuristics: 'insert-first' and 'delete-first' breadth-first-search approaches where discrepancies discovered when comparing a level in the template and target are handled by first inserting the missing node or deleting the mismatched node, respectively, and then accounting for any remaining deltas (Table 2). As shown, Fawkes's heuristic runs 2 orders of magnitude faster than existing tree diffing algorithms. Median operations are lower with Fawkes's heuristic due to its *move* operation. 95th percentile operations are 5× worse with Fawkes's heuristic due to the inefficiences described in §3.1, but we note that this large gap is only present in 8% of pages.

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

5.4 Comparison with Vroom and WatchTower

We compared Fawkes with two recent mobile web optimization systems, Vroom [61] and WatchTower [47]. With Vroom, web servers user HTTP/2 server push to proactively send static resources that they own to clients ahead of future requests. In addition, Vroom servers send HTTP preload headers [68] to let clients quickly download resources that they will soon need from other domains. In contrast, WatchTower accelerates page loads by selectively using proxy servers based on page structural properties and network conditions. When enabled, a proxy loads a page locally using a headless browser and fast network links, and streams individual resources back to the client for processing. Our evaluation considers WatchTower's HTTPS-sharding mode, where each HTTPS origin runs their own proxy to preserve HTTPS security. Proxies were run on EC2 in California where the WatchTower paper reported the highest speedups.

Table 3 compares Fawkes with Vroom and WatchTower for both cold and warm cache loads of our 600 page corpus over an LTE network; trends were similar on the WiFi network. As shown, Fawkes is able to significantly improve performance on the interactivity-focused metrics compared to these systems. For example, median warm cache Speed Index values were 24% and 10% lower with Fawkes than with Vroom and WatchTower, respectively. Fawkes's TTFP benefits over these systems were 69%-73% since acceleration techniques with WatchTower and Vroom only take affect *after* incurring multiple network round trips and server processing delays to download HTML objects.

Our results also show that Vroom and WatchTower are more effective than Fawkes at reducing PLT; median benefits are 3.3% and 16.6%, respectively. The reason is that both Vroom and WatchTower can mask network round trips required to fetch external objects throughout the page load, including those triggered by non-HTML objects. Fawkes, on the other hand, focuses on early parts of a page load–indeed, targeting startup bottlenecks is what differentiates Fawkes from prior acceleration techniques. Importantly, we note that Fawkes's early-stage optimizations are largely complementary to prior techniques.To validate this, we reran the experiment above using a combination of Fawkes and Vroom. Vroom's server hints on the top-level HTML were sent along with Fawkes's dynamic patches. As shown in Table 3, this combination outperforms any tested system in isolation.

5.5 Additional Results

Stale HTML templates: Our warm cache evaluation considered static templates that were generated using the HTML object at time 0 and the one for the current time (e.g., 12 hours). Although this is possible using the techniques presented in §3.3 to generate representative HTML files for a page, it is not the sole practical deployment scenario. An alternative approach would be to generate static HTML templates with only back-to-back loads at a time 0, and use this for future warm cache loads. To evaluate the impact of such stale templates, we loaded the pages in our corpus using both stale (i.e., generated at time 0) and up-to-date (generated using HTML files at time 0 and the current time) templates. We considered staleness of 12 and 24 hours, and observed minimal performance degradations. For example, on the LTE network, median SI values dropped by only 4.2% and 6.8% for the 12 and 24 hour scenarios; TTFP values were unchanged.

Personalized pages: We selected 20 sites from our 600 page corpus that supported user accounts. For each site, we made two user accounts, selecting different preferences when possible, e.g., order results based on time or popularity. We then generated static templates from the HTML objects that each user account fetched. Finally, we loaded one of the user's pages 12 hours later with a warm cache, and compared per-

formance to that of a default browser. Fawkes was able to reduce SI by 27% and 18% on the LTE and WiFi networks, respectively. It is important to note that these trends may not hold for all personalization strategies. For example, pages like Facebook can display structurally diverse content over time and across users. However, our results illustrate that many pages do remain structurally similar across users.

Energy savings and other browsers: Fawkes reduces (per-page) energy usage by 7-18%, and its speedups persist across other browsers (e.g., Firefox). Details provided in §A.3.

6 Related Work

Server push systems: Numerous systems, including Vroom (§5.4), aim to accelerate mobile page loads by leveraging HTTP/2's server push feature [9], where servers proactively push resources to clients in anticipation of future requests [61, 19, 70]. Fawkes is largely complementary to server push systems as these approaches reduce fetch times for resources loaded after the top-level HTML. In contrast, Fawkes speeds up page startup times.

Proxy and backend accelerators: Compression proxies [4, 63, 67, 52] compress objects in-flight between clients and servers, while remote dependency resolution proxies [65, 47, 48, 64] perform object fetches on behalf of clients. Fawkes is orthogonal to these approaches, and can mask the network indirection and computation overheads associated with proxying. In addition, Fawkes preserves the end-to-end nature of the web, avoiding the security challenges of proxying.

More recently, Prophecy [43], Shandian [71], and Opera Mini [51] return post-processed versions of objects to reduce client-side computation and bandwidth costs. All three systems must incur the same network round trips and (more) server processing delays that default page loads to download top-level HTML objects–only then do their acceleration techniques help. These delays are exactly what Fawkes aims to alleviate. We also note that Fawkes's patcher and shims tackle a fundamentally different challenge than those in Prophecy and Shandian: Fawkes must execute JavaScript code in an environment with fast-forwarded DOM state.

Dependency-aware scheduling: Certain systems have improved the scheduling of network requests based on inherent dependencies in page content. Klotski [14] analyzes pages offline to identify high-priority objects in terms of user utility, and uses knowledge of network bandwidth to stream them to clients before they are needed. Polaris [42] uses a client-side request scheduler that reorders requests to minimize the number of effective round trips in a page load without violating state dependencies. However, both systems are unable to process or render content prior to an HTML download. Thus, these systems can work side by side with Fawkes.

JavaScript UI frameworks: Libraries like Vue.js [74], AngularJS [26], and React [22] efficiently update client-side page state during page loads. A key feature across these frameworks is the use of a virtual DOM, where JavaScript-based DOM updates are first performed on a lightweight DOM representation, and aggregate results (rather than intermediate layout and render events) are applied to the actual DOM. Using such efficient update strategies, these frameworks support client-side page rendering, whereby a page's top-level HTML embeds only a single JavaScript library that is responsible for downloading and rendering downstream page content. While these frameworks focus on efficiently updating content during a page load and require developers to rewrite pages, Fawkes operates on unmodified pages and aims to quickly display content shared across page loads. Further, unlike the client-side page rendering approach, Fawkes's static templates embed both the JavaScript patcher library *and* all of a page's static HTML content. This, in turn, ensures that Fawkes can render static content while fetching downstream (dynamic) content.

Accelerating HTML loading: Google's SDCH [15] allows web servers to specify cacheable components of HTML files; on subsequent loads, servers need only send new components or deltas to cached ones, thereby saving bandwidth. Unlike Fawkes, SDCH does not allow browsers to render cached HTML components *until the entire HTML is constructed*. Thus, SDCH does not face the view invariance challenges that Fawkes's patcher does, and SDCH is unable to reduce web startup times by rendering cached HTML content quickly for users. Other industry efforts have focused on dividing pages into modular components called "pagelets", which can be generated and processed independently and in parallel [13, 33]. Pagelets share Fawkes's goal of improving resource utilization to more quickly display content to users. However, unlike Fawkes, individual pagelets do not include a mechanism for automatically separating static and dynamic HTML, and instead use a single response that is shipped only after the pagelet's dynamic content is generated.

7 Conclusion

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

Acknowledgements: We thank Harsha Madhyastha and James Mickens for their valuable feedback on earlier drafts of the paper, as well as Ben Greenstein and Michael Buettner for useful discussions on Fawkes' practical implications. We also thank our shepherd, Ankit Singla, and the anonymous NSDI reviewers for their constructive comments.

REFERENCES

[1] Overleaf: A web-based collaborative latex editor. https://github.com/overleaf/overleaf.

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

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

[4] V. Agababov, M. Buettner, V. Chudnovsky, M. Cogan, B. Greenstein, S. McDaniel, M. Piatek, C. Scott, M. Welsh, and B. Yin. Flywheel: Google's Data Compression Proxy for the Mobile Web. NSDI '15. USENIX, 2015.

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

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

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

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

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

[10] D. Bhattacherjee, M. Tirmazi, and A. Singla. A Cloud-based Content Gathering Network. In *Proceedings of HotCloud*, 2017.

[11] N. Bhatti, A. Bouch, and A. 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.

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

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

[14] M. Butkiewicz, D. Wang, Z. Wu, H. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2015.

[15] J. Butler, W.-H. Lee, B. McQuade, and K. Mixter.

[16] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. In *Proceedings of the 34th International Conference on Automata, Languages and Programming*, ICALP'07, pages 146–157, Berlin, Heidelberg, 2007. Springer-Verlag.

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

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

[19] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan. Towards a SPDY'Ier Mobile Web? *IEEE/ACM Trans. Netw.*, 23(6):2010–2023, Dec. 2015.

[20] D. 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.

[21] T. Everts and T. Kadlec. WPO stats. https://wpostats.com/, 2019.

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

[23] D. F. Galletta, R. Henry, S. McCoy, and P. Polak. Web Site Delays: How Tolerant are Users? Journal of the Association for Information Systems, 2004.

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

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

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

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

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

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

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

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

[32] P. Irish. pwmetrics: Progressive web metrics. https://github.com/paulirish/pwmetrics, 2019.

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

[34] B. Jun, F. E. Bustamante, S. Y. Whang, and Z. 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.

[35] C. Kelton, J. Ryoo, A. Balasubramanian, and S. R. Das. Improving User Perceived Page Load Time Using Gaze. In *Proceedings of the 14th USENIX Confer-*

ence on Networked Systems Design and Implementation, NSDI'17, pages 545–559. USENIX Association, 2017.

[36] P. N. Klein. Computing the edit-distance between unrooted ordered trees. In *Proceedings of the 6th Annual European Symposium on Algorithms*, ESA '98, pages 91–102, London, UK, UK, 1998. Springer-Verlag.

[37] G. Li, E. Andreasen, and I. Ghosh. Symjs: Automatic symbolic testing of javascript web applications. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 449–459, New York, NY, USA, 2014. ACM.

[38] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. 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.

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

[40] Monsoon Solutions Inc. Power monitor software. http://msoon.github.io/powermonitor/, 2018.

[41] J. Nejati and A. 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.

[42] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX, 2016.

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

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

[45] R. Netravali and J. Mickens. Reverb: Speculative Debugging for Web Applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 428–440, New York, NY, USA, 2019. Association for Computing Machinery.

[46] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring Time-to-Interactivity for Web Pages. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX, 2018.

[47] R. Netravali, A. Sivaraman, J. Mickens, and H. Balakrishnan. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. In *Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '19, pages 430–443. ACM, 2019.

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

[49] J. Newman and F. Bustamante. The Value of First Impressions: The Impact of Ad-Blocking on Web QoE". In D. Choffnes and M. Barcellos, editors, *Passive and Active Measurement - 20th International Conference, PAM 2019, Proceedings*, pages 273–285, Germany, 1 2019. Springer Verlag.

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

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

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

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

[54] M. Pawlik and N. Augsten. Rted: A robust algorithm for the tree edit distance. *Proc. VLDB Endow.*, 5(4):334–345, Dec. 2011.

[55] M. Pawlik and N. Augsten. Efficient Computation of the Tree Edit Distance. *ACM Trans. Database Syst.*, 40(1):3:1–3:40, Mar. 2015.

[56] M. Pawlik and N. Augsten. APTED algorithm for the Tree Edit Distance. https://github.com/DatabaseGroup/apted, 2018.

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

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

[59] L. Ravindranath, S. Agarwal, J. Padhye, and C. Riederer. Give in to Procrastination and Stop Prefetching. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII. ACM, 2013.

[60] L. Richardson. Beautiful Soup. https://www.crummy.com/software/BeautifulSoup/bs4/doc/, 2019.

[61] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM. ACM,

2017.

[62] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[63] S. Singh, H. V. Madhyastha, S. V. Krishnamurthy, and R. Govindan. FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers. In *Proceedings of the 21st Annual Conference on Mobile Computing and Networking*, MobiCom. ACM, 2015.

[64] A. Sivakumar, C. Jiang, S. Nam, P. Shankaranarayanan, V. Gopalakrishnan, S. Rao, S. Sen, M. Thottethodi, and T. Vijaykumar. Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, Mobicom. ACM, 2017.

[65] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. Parcel: Proxy assisted browsing in cellular networks for energy and latency reduction. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 325–336, New York, NY, USA, 2014. ACM.

[66] M. Varvello, J. Blackburn, D. Naylor, and K. Papagiannaki. EYEORG: A Platform For Crowdsourcing Web Quality Of Experience Measurements. In *Proceedings of the 12th Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16. ACM, 2016.

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

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

[69] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2013.

[70] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How Speedy is SPDY? In *Proceedings of NSDI*, NSDI'14, pages 387–399, Berkeley, CA, USA, 2014. USENIX Association.

[71] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding Up Web Page Loads with Shandian. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI. USENIX Association, 2016.

[72] Z. Wang, F. X. Lin, L. Zhong, and M. 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.

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

[74] E. You. Vue.js: The Progressive JavaScript Framework. https://vuejs.org/, 2019.

[75] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, Dec. 1989.

[76] T. Zimmermann, B. Wolters, O. Hohlfeld, and K. Wehrle. Is the Web ready for HTTP/2 Server Push? In *Proceedings of the 14th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT. ACM, 2018.

A APPENDIX

A.1 Mahimahi Modifications

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

A.2 Correctness and Limitations

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

On the remaining pages, correctness with Fawkes was evaluated by forcing determinism upon JavaScript execution (e.g., using fixed return values for calls to `Math.Random()`) [39, 45], and comparing loads with and without Fawkes in three ways: 1) a pixel-by-pixel analysis of the final page (using pwmetrics' screenshots and visual analysis tools [32]), 2) the number of registered JavaScript event handlers (logged using shims on the `addEventListener` mechanism and by iterating over the DOM tree after the `onload` event fired [46]), and 3) the browser console errors printed during the page load. We excluded the 17 pages that differed on any of these three properties from our evaluation. Further investigation revealed two key reasons for such discrepancies, which are limitations with Fawkes:

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

A.3 Additional Results

Energy consumption: To examine the impact that Fawkes has on energy usage, we connected a Nexus 6 phone to a Monsoon power monitor [40] and loaded our 600 page corpus. During cold cache loads, Fawkes's speedups reduce median per-page energy usage by 11% and 7% compared to a default browser on the LTE and WiFi networks, respectively. Benefits jump to 18% and 11% in warm cache settings (12 hours apart). In both cases, benefits are higher on LTE due to the higher network latencies and the fact that LTE radios consume more energy than WiFi hardware when active [65].

Additional browsers: Since Fawkes does not require any browser modifications, we also evaluated Fawkes with Firefox (v68) using our 600 page corpus and the same experimental setup from §5.1. Benefits in the 12 hour warm cache setting were quite comparable, despite Firefox using a different rendering engine than Chrome. Fawkes reduced median SI by 21% and 34% on the WiFi and LTE networks.

A.4 Additional Related Work

Mobile-optimized pages: Certain sites cater to mobile settings by serving pages that involve less client-side computation, fewer bytes, and fewer network fetches. For example, Google AMP [24, 34] is a recent mobile web standard that requires developers to rewrite pages using restricted forms of HTML, JavaScript, and CSS. Unlike AMP, Fawkes accelerates legacy pages without needing developer effort. Further, Fawkes provides complementary benefits and can lower AMP startup delays: Fawkes's TTFP and SI reductions were 58% and 27% for the 23 AMP pages in our corpus.

Prefetching: Prefetching systems predict user browsing behavior and optimistically download objects prior to user page loads [53, 38, 72]. Unfortunately, such systems have witnessed minimal adoption due to challenges in predicting what pages a user will load and when; inaccurate page and timing predictions can waste device resources or result in stale page content [59]. By rendering static templates as soon as a user navigates to a page, Fawkes is able to achieve comparable TTFP reductions without the issues of prefetching.

Progressive Web Apps (PWAs): Google recently proposed PWAs [29], applications that are written using standard web languages (e.g., HTML, JavaScript), can be loaded by a standard web browser, but are installed as an application on a user device. PWAs use service workers [30] which employ aggressive caching and custom update logic to run offline and support push notifications from servers. Fawkes shares the idea of improving use of web caching and app-like update logic. However, in contrast to PWAs which require developer effort for creation (and potentially maintenance), Fawkes transparently applies app-like templating to legacy pages.