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

Ravi Netravali, ***MIT CSAIL;*** **James Mickens,** ***Harvard University***

https://www.usenix.org/conference/nsdi18/presentation/netravali-prophecy

This paper is included in the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18).

April 9–11, 2018 • Renton, WA, USA

ISBN 978-1-939133-01-4

Open access to the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation is sponsored by USENIX.

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

Ravi Netravali*, James Mickens†
*MIT CSAIL, †Harvard University

ABSTRACT

Web browsing on mobile devices is expensive in terms of battery drainage and bandwidth consumption. Mobile pages also frequently suffer from long load times due to high-latency cellular connections. In this paper, we introduce Prophecy, a new acceleration technology for mobile pages. Prophecy simultaneously reduces energy costs, bandwidth consumption, and page load times. In Prophecy, web servers precompute the JavaScript heap and the DOM tree for a page; when a mobile browser requests the page, the server returns a write log that contains a single write per JavaScript variable or DOM node. The mobile browser replays the writes to quickly reconstruct the final page state, eliding unnecessary intermediate computations. Prophecy's server-side component generates write logs by tracking low-level data flows between the JavaScript heap and the DOM. Using knowledge of these flows, Prophecy enables optimizations that are impossible for prior web accelerators; for example, Prophecy can generate write logs that interleave DOM construction and JavaScript heap construction, allowing interactive page elements to become functional immediately after they become visible to the mobile user. Experiments with real pages and real phones show that Prophecy reduces median page load time by 53%, energy expenditure by 36%, and bandwidth costs by 21%.

1 INTRODUCTION

Mobile browsing now generates more HTTP traffic than desktop browsing [18]. On a smartphone, 63% of user focus time, and 54% of overall CPU time, involves a web browser [56]; mobile browsing is particularly important in developing nations, where smartphones are often a user's sole access mechanism for web content [12, 23]. So, mobile page loads are important to optimize along multiple axes: bandwidth consumption, energy consumption, and page load time. Reducing bandwidth overhead allows users to browse more pages without violating data plan limits. Reducing energy consumption improves the overall lifetime of the device, because web browsing is a significant drain on battery power [8, 11, 48, 55, 56]. Improving page load time is important because users are frustrated by pages that take more than a few seconds to load [15, 17, 29, 50].

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

Earlier attempts at applying precomputation to web sites have suffered from significant practical limitations (§6), in part because these systems used serialized graphs instead of write logs. Serialized graphs hide data flows that write logs capture; analyzing these data flows is necessary to perform many optimizations. For example, Prepack [16] cannot handle DOM state, and is unable to elide computation for some kinds of common JavaScript patterns. Shandian [51] does not support caching for the majority of a page's content, does not support immediate page interactivity (§3.5), and does not work on unmodified commodity browsers; furthermore, Shandian exposes all of a user's cookies to a single proxy, raising significant privacy concerns. In contrast, Prophecy works on commodity browsers, handles both DOM and JavaScript state, preserves traditional same-origin policies about cookie security, and supports byte-granularity caching (which is *better* than HTTP's standard file-level caching scheme). Prophecy can also prioritize the loading of interactive state; this feature is important for sites that load over high-latency links, and would otherwise present users with rendered GUIs that may not actually be functional. Many of Prophecy's advantages are enabled by having fine-grained, variable-level understanding of how a page load unfolds.

Experiments with a Nexus 6 phone, loading 350 web pages on real WiFi and LTE networks, reveal Prophecy's significant benefits: median energy usage drops by 36%, median bandwidth consumption decreases by 21%, and median page load time decreases by 53% (2.8 seconds). Prophecy also helps page loads on desktop browsers, reducing median bandwidth usage by 18%, and median

page load time by 38% (0.8 seconds). These benefits are 2.2×–4.6× better than those enabled by Polaris [36], another state-of-the-art web accelerator. Thus, Prophecy represents a significant advance in web optimization.

2 Background

A single web page consists of one or more frames. Each frame is defined by an associated HTML file. The HTML describes a tree structure that connects individual HTML tags like `<title>` and `<div>`. Most kinds of tags can embed visual style attributes directly in their HTML text (e.g., `<h1 style='color:blue;'>`). However, best practice is for developers to place style information in separate CSS files, so that a frame's basic visual structure (as defined by HTML) can be defined separately from a particular styling approach for that structure.

Some tags, like `<script>` and `<img>`, support a `src` property which indicates the URL from which the tag's content should be downloaded. Alternatively, the content for the tag can be inlined. For example, a `<script>` tag can directly embed the associated JavaScript code. CSS information can also be inlined using a `<style>` tag. Inlining a tag's content eliminates an HTTP-level RTT to fetch the associated data. However, inlining prevents a browser from caching the associated object, since the browser cache interposes on explicit HTTP requests and responses, using the URL in the HTTP request as the key for storing and retrieving the associated object.

A frame uses JavaScript code to perform computation. JavaScript code is single-threaded and event-driven, with managed memory allocation; so, between the execution of event handlers, a frame's only JavaScript state resides in the managed heap. There are two kinds of JavaScript objects: application-defined and native. Application-defined objects are composed of pure JavaScript-level state. In contrast, native objects are JavaScript wrappers around native code functionality defined by the JavaScript engine, the HTML renderer, or the network engine. Examples of native objects include `RegExps` (which implement regular expressions) and `XMLHttpRequests` (which expose HTTP network connections).

DOM nodes [34] are another important type of native object. As the HTML parser scans a frame's HTML, the parser builds a native code representation of the HTML tree; this tree is reflected into the JavaScript runtime as the DOM tree. There is a 1-1 correspondence between HTML tags and DOM nodes. Using the DOM interface, JavaScript code can programmatically add, remove, or update DOM nodes, changing the visual content which is shown to a user. DOM changes often require the browser to recalculate the layout and styles of DOM nodes, and then repaint the DOM nodes. These calculations are computationally expensive (and therefore energy-intensive as well) [8, 25, 27, 56].

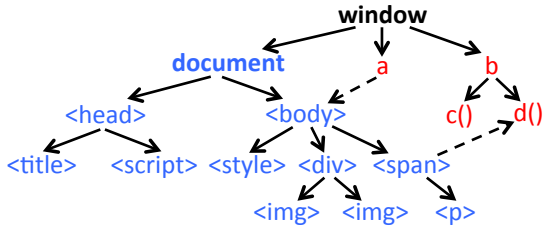


Figure 1: A simple frame's JavaScript heap and DOM tree. The JavaScript heap is red; the DOM tree is blue.

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

Browsers define two main types of client-side storage. A cookie [5] is a small, per-origin file that can store up to 4 KB of data. When a browser issues an HTTP request to origin X, the browser includes any cookie that the browser stores on behalf of X. When the server receives the cookie, the server can generate personalized content for the HTTP response. The server can also use special HTTP response headers to modify the client-side cookie. Cookies are often used to hold personal user information, so cookie sharing has privacy implications.

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

3 Design

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

[1] A headless browser lacks a GUI, but otherwise performs the normal duties of a browser, parsing and rendering HTML, executing JavaScript, and so on.

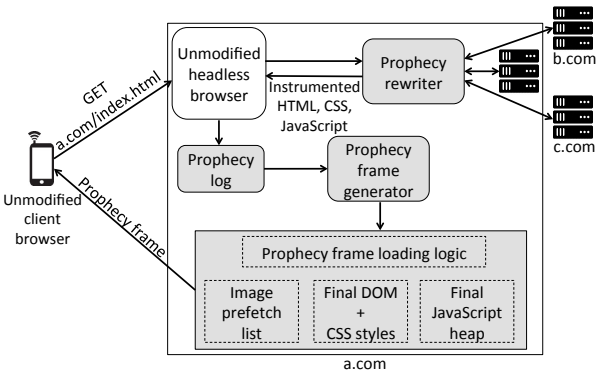


Figure 2: The Prophecy architecture.

mentation which tracks how the frame manipulates the JavaScript heap and the DOM tree.

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

- a **write log for the JavaScript heap**, containing an ordered set of writes that a client executes to recreate the frame's final heap state;
- a **write log for the DOM**, containing HTML tags with precomputed styles, such that the client can immediately resurrect the DOM with minimal layout and rendering overheads;
- an **image prefetch log**, describing the images that the browser should fetch in parallel with the construction of the JavaScript heap and the DOM; and
- the **Prophecy resurrection library**, a small piece of JavaScript code which orchestrates client-side reconstruction of the frame (§3.2), optimizing the reconstruction for a particular load metric (§3.5).

During a warm cache load (§3.3), the three logs are diffs with respect to the client's cached logs. By applying the diffs and then executing the patched logs, a client fast-forwards its view of the frame to the latest version.

3.1 Generating a Prophecy Frame

Prophecy enables web acceleration at the granularity of a frame. However, web developers create the content for a particular frame in a Prophecy-agnostic way, using a normal workflow to determine which objects (e.g., HTML, CSS, JavaScript, and images) should belong in a frame. The process of transforming the normal frame into a Prophecy variant is handled automatically by Prophecy. The transformation can happen online (i.e., at the time of an HTTP request for the frame), or offline (i.e., before such a request has arrived). In this section, we describe the transformation process; later, we describe the tradeoffs between online and offline transformation (§3.4).

After fetching the frame's HTML, Prophecy's server-side component loads the frame in a headless browser. As the frame loads, Prophecy tracks the reads and writes that the frame makes to the JavaScript heap and to the DOM. Prophecy's design is agnostic as to how this tracking is implemented. Our concrete Prophecy prototype uses Scout [36], a frame rewriting framework, to inject logging instrumentation into the loaded frame, but Prophecy is compatible with in-browser solutions that use a modified JavaScript engine and renderer to log the necessary information. Regardless, once the frame has loaded, Prophecy analyzes the reads and writes to create the three logs which represent the Prophecy version of a frame.

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

The write log must pay special attention to functions. In JavaScript, a function definition can be nested within an outer function definition. The inner function becomes a closure, capturing the variable scope of the outer function. To properly handle these functions, Prophecy rewrites functions to explicitly expose their closure scope [30, 32, 51]. At frame load time on the server, this allows Prophecy's write tracking to explicitly detect which writes involve a function's closure state. Later, when a mobile browser needs to recreate a closure function, the replayed write log can simply create the function, then create the scope object, and then write to the scope object's variables.

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

The DOM write log: Once Prophecy's server-side component has loaded a frame, Prophecy generates an HTML string representation for the frame using

the browser's predefined `XMLSerializer` interface. Importantly, the HTML string that is returned by `XMLSerializer` does not contain styling information for individual tags; the string merely describes the hierarchical tag structure. To extract the style information, Prophecy iterates over the DOM tree, and uses `window.getComputedStyle(domNode)` to calculate each node's style information.[2] Prophecy then augments the frame's HTML string with explicit style information for each tag. For example, a tag in the augmented HTML string might look like `<div style='border-bottom-color: rgb(255, 0, 0);border-left-color: rgb(255, 0, 0);'>`. Prophecy modifies all CSS-related tags in the augmented HTML string, deleting the bodies of inline `<style>` tags, and setting the `href` attributes in `<link rel='stylesheet'>` tags to point to the empty string (preventing a network fetch). Prophecy also modifies the `src` attribute of `<script>` tags to point to the empty string (since all JavaScript state will be resurrected using the JavaScript write log).

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

The image prefetch log: This log is a JavaScript array that contains the URLs for the images in the loaded frame. The associated `<img>` tags may have been statically declared in the frame's HTML, or dynamically injected via JavaScript. Note that the write log for the DOM tree contains the associated `<img>` tags; however, as we explain in Section 3.2, the image prefetch list allows the mobile browser to keep its network pipe busy as the CPU is parsing HTML and evaluating JavaScript.

The Prophecy frame consists of the three logs from above, and a small JavaScript library which uses the logs to resurrect the frame (§3.2). Since the three logs are expressed as JavaScript variables, the Prophecy server can just add those variables to the beginning of the resurrection library. So, the Prophecy frame only contains one HTML tag—a single JavaScript tag with inline content.

3.2 Loading a Prophecy Frame

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

Phase 1 (DOM Reconstruction): The resurrection library passes the DOM write log to the browser's pre-existing `DOMParser` interface. `DOMParser` returns a `document` object, which is a special type of DOM node that represents an entire DOM tree. The resurrection library updates the frame's live DOM tree by splicing in the `<head>` and `<body>` DOM subtrees from the newly created `document`. After these splice operations complete, the entire DOM tree has been updated; note that the browser has avoided many of the traditional computational overheads associated with layout and rendering, since the resurrection library injected a pre-styled DOM tree which already contains the side effects of load-time JavaScript calls to the DOM interface. As the browser receives the asynchronously prefetched image data, the browser injects the pixels into the live DOM tree as normal, without assistance from the resurrection library; note that the browser will not "double-fetch" an image if, at DOM reconstruction time, the browser encounters an `<img>` tag whose prefetch is still in-flight.

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

Phase 3 (Fixing Cross-references): At this point, the DOM tree and the JavaScript heap are largely complete. However, DOM objects can refer to JavaScript heap objects, and vice versa. For example, an application-defined JavaScript object might have a property that refers to a specific DOM node. As another example, the event handler for (say) a mouse click is an application-defined JavaScript function that must be attached to a DOM node via `DOMnode.addEventLister(evtType, func)`. In Phase 3, the resurrection library fixes these dangling references using information in the JavaScript write log. During the initial logging of reads and writes in the frame load (§3.1), Prophecy assigned a unique id to each JavaScript object and DOM node that the frame created. Now, at frame reconstruction time on the mobile browser, the resurrection library uses object ids to determine which object should be used to resolve each dangling reference. As hinted above, the library must resolve some dangling references in DOM nodes by calling specific DOM functions like `addEventListener()`. The library also needs to

[2] Prophecy uses additional logic to ensure that the extracted style information includes any default tag styles that apply to the DOM node. These default styles are not returned by `getComputedStyle()`.

invoke the relevant timer registration functions (e.g., `setTimeout(delay, callback)`) so that timers are properly resurrected.[3]

At the end of Phase 3, the frame load is complete, having skipped intermediate JavaScript computations, as well as intermediate styling and layout computations for the DOM tree. A final complication remains: what happens if, post-load, the frame dynamically injects a new DOM node into the DOM tree? Remember that Prophecy's write log for the DOM tree contains no inline `<style>` data, nor does it contain `href` attributes for `<link rel='stylesheet'>` tags (§3.1). So, as currently described, a Prophecy frame will not assign the proper styles to a dynamically created DOM node.

To avoid this problem, the resurrection library contains a string which stores all of the frame's original CSS data. The resurrection code also shims [31] DOM interfaces like `DOMnode.appendChild(c)` which are used to dynamically inject new DOM content. Upon the invocation of such a method, the Prophecy shim examines the frame's CSS rules (and the live style characteristics of the DOM tree) to apply the appropriate inline styles to the new DOM node. After applying those styles, Prophecy can safely inject the DOM node into the DOM tree.

3.3 Caching, Personalization, and Cookies

To enable frame content to be personalized, we extend the approach from Sections 3.1 and 3.2. At a high level, when a server receives an HTTP request for a frame, the server looks inside the request for a cookie that bears a customization id. If the server does not find such a cookie, then the server assumes that the mobile browser has a cold cache; in this case, the server returns the Prophecy frame as described in Section 3.1, placing a cookie in the HTTP response which describes the frame's customization id. If the server does find a customization id in the HTTP request, then the server assumes that the client possesses cached write logs for the frame. The server computes the write logs for the latest customization version of the frame. The server then calculates the diffs between the latest write logs and the ones that are cached on the phone. Finally, the server returns the diffs to the mobile browser. The mobile browser applies the diffs to the cached write logs, and then recreates the frame as described in Section 3.2.

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

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

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

In the cold cache case, the server generates the appropriate write logs for the latest frame version v, and then diffs the latest logs against the baseline logs. The server stores the diffs in $diffTable[v]$, and then returns the latest write logs to the client as in Section 3.1, setting the customization id in the HTTP response to v. The client rebuilds the frame as in Section 3.2, and then stores the three write logs in DOM storage.

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

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

[3] During the instrumented frame load on the server, Prophecy shims timer registration interfaces to track timer state [31].

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

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

3.4 Online versus Offline Transformation

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

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

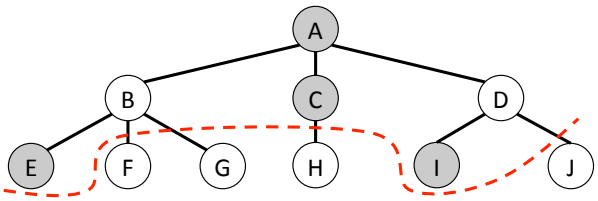


Figure 3: An example of how Prophecy determines the interactive DOM subtree to build before the rest of the DOM nodes are recreated. The shaded circles represent DOM nodes that are 1) above-the-fold, and/or 2) are manipulated by the event handlers of above-the-fold DOM nodes. The interactive subtree resides above the red line.

3.5 Defining Load Time

To fully load a traditional frame, a browser must fetch and evaluate the frame's HTML, and then fetch and evaluate the external objects that are referenced by that HTML. The standard definition for a frame's load time requires all of the external objects to be fetched and evaluated. A newer load metric, called Speed Index [21], measures how quickly a browser renders a frame's above-the-fold[4] visual content. Using write logs, Prophecy improves frame-load time (FLT) by eliding unnecessary intermediate computations and inlining all non-image content. However, as described in Section 3.2, Prophecy completely renders the DOM before constructing the JavaScript heap and then patching cross-references between the two. So, Prophecy gives higher priority to visual content, much like Speed Index (SI).

Both FLT and SI have disadvantages. FLT does not capture the notion that users desire above-the-fold content to appear quickly, even if below-the-fold content is still loading. However, at FLT time, *all* of a frame's content is ready; in contrast, SI ignores the fact that a *visible* DOM element does not become *interactive* until the element's JavaScript event handler state has been loaded. The difference between visibility and interactivity is especially apparent when a web page loads over a high-latency link; in such scenarios (which are common on mobile devices), slow-loading JavaScript can lead to `<button>` tags that do nothing when clicked, or `<input>` tags that do not offer autocompletion suggestions upon receiving user text. The median page in our test corpus had 113 event handlers for GUI interactions, so optimizing for interactivity is useful for many mobile pages.

To optimize for interactivity, Prophecy can explic-

[4] Above-the-fold content refers to the visual portion of a frame that lies within the browser GUI at the beginning of a frame load, before the user has scrolled down.

itly target a newer load metric called Ready Index [37]. Ready Index (RI) declares a frame to be ready when its above-the-fold content is both visible and interactive. To optimize for RI, Prophecy feeds its server-side log of reads and writes (§3.1) to Vesper [37]. Vesper uses load-time read/write logs, as well as read/write logs generated by active, Vesper-driven triggering of event handlers, to identify the frame's interactive state. The interactive state consists of:

- the above-the-fold DOM nodes,
- the JavaScript state which defines the event handlers for above-the-fold DOM nodes,
- the DOM state and the JavaScript state which is manipulated by those event handlers.

Given the DOM nodes in a frame's interactive state, Prophecy finds the minimal HTML subtree, rooted by the top-level `<html>` tag, which contains all of the interactive DOM nodes. Figure 3 shows an example of this interactive DOM subtree. Prophecy then represents a frame using two HTML write logs (one for the interactive subtree, and one for the remaining HTML subtrees), and two JavaScript write logs (one for the state which supports above-the-fold interactive DOM nodes, and another write log for the remaining JavaScript state). Prophecy keeps a single image prefetch log, but places above-the-fold images first in the log. To load a frame on the client browser, Prophecy first renders the above-the-fold DOM nodes, and then builds the JavaScript state which supports interactivity for those DOM nodes. After patching cross-references, the frame is interactive. Prophecy then attaches the below-the-fold DOM nodes, creates the remaining JavaScript state, and patches a final set of cross-references.

By optimizing for RI, Prophecy can minimize the likelihood that attempted user interactions will fail. However, Prophecy cannot eliminate all such problems. For example, if a user issues GUI events before the first set of write logs are applied, the events may race with the browser's creation of above-the-fold DOM elements and interactive JavaScript state. Such race conditions are present during regular, non-Prophecy page loads [40]; by optimizing for RI, Prophecy reduces the size of the race window, but does not completely eliminate it.

3.6 Privacy

A frame from origin X may embed content from a different origin Y. For example, X's frame may embed images or JavaScript from Y. When the mobile browser sends an HTTP request for X's frame, the browser will only include cookies from X, since the URL in the HTTP request has an origin of X. As Prophecy's server-side code loads the frame and generates the associated logs (§3.1), the server from X will fetch content from Y. However, in the HTTP requests that the server sends to Y, the server will not include any of Y's cookies that reside on the mobile browser—the server never received those cookies from the client. This policy amounts to a "no third-party cookie" approach. Variants of this policy are already being adopted by some browsers for privacy reasons, since third party cookies enable users to be tracked across different sites [43]. So, in Prophecy, a server from X only sees cookies that belong to X, and a frame load does not send third party cookies to any external origin Y.

3.7 Discussion

Prophecy is compatible with transport protocols like HTTP/2 [26] and QUIC [7] that pipeline HTTP requests, leverage UDP instead of TCP to transmit data, or otherwise try to optimize a browser's HTTP-level network utilization. Prophecy is also compatible with proxy-based web accelerators like compression proxies [1, 44] or split-browsers [3, 38, 39]. From the perspective of these technologies, the content in a Prophecy frame is no different than the content in a non-Prophecy frame.

Prophecy is also compatible with HTTP/2's server-push feature [7]. Server-push allows a web server to proactively send an HTTP object to a browser, prewarming the browser's cache so that a subsequent fetch for the object can be satisfied locally. Prophecy-enabled frames use cookies to record the versions of locally-DOM-cached frames (§3.3). So, imagine that a web server would like to push frames. When the server receives an HTTP request for frame f_i, the server can inspect the cookies in the request and determine, for some different frame f_j to push, whether to push a cold-cache or warm-cache version of the frame.

A Prophecy web server does not track any information about a client's DOM storage (besides diffs for the write logs that reside in that DOM storage). Since the server does not track client-side DOM storage, the final result of a frame load should not depend on the client's non-write-log DOM storage—this state will not be available to the server-side frame load that is used to generate write logs. To the best of our knowledge, all web accelerators that use server-side load analysis [3, 38, 39, 51] assume empty client-side DOM storage, since mirroring all of that storage would be expensive, and developer best practice is to use DOM storage as a soft-state cache.

4 Implementation

On the server-side, Prophecy uses a modified version of Scout [36] to rewrite frame content and track reads and writes to the JavaScript heap and the DOM tree. Prophecy extends Scout's JavaScript translator to rewrite closure scopes (so that Prophecy can efficiently resurrect closure functions). Prophecy also extends the translator to log the classes of objects created via the `new` operator

(so that Prophecy can determine the appropriate instance objects to create in the write log for the JavaScript heap).

When rewriting a frame's HTML, Prophecy injects JavaScript source code for a timer that fires in response to the `load` event. This timer serializes the DOM as described in Section 3.1, using `XMLSerializer` to generate the basic HTML string, and using Beautiful Soup [41] to parse and edit the string, e.g., to inject precomputed CSS styles, and to extract the image `src` URLs to place in the image prefetch log.

To support client-side caching, Prophecy servers use the `google-diff-patch-match` library [20] to generate diffs. The scaffolding for Prophecy's server-side logic is implemented as a portable CGI script for Apache.

On the client-side, Prophecy's resurrection code is 1.3 KB in size. The code uses the `google-diff-patch-match` library [20] to perform diffing, and a modified version of the `CSSUtilities` framework [28] to apply styles to dynamically-created DOM nodes (§3.2).

5 RESULTS

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

To create a reproducible test environment, we used Mahimahi [38] to record the content in the Alexa Top 350 pages [2], and later replay that content to test browsers. For pages which defined both a mobile version and a desktop version, we recorded both. Later, at experiment time, Mahimahi always returned the desktop version of a page to the desktop browser; when possible, Mahimahi returned the mobile version of a page to the mobile browser. At replay time, Mahimahi used HTTP/2 for pages that employed HTTP/2 at recording time. Server-push events that were seen at recording time were applied during replay.

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

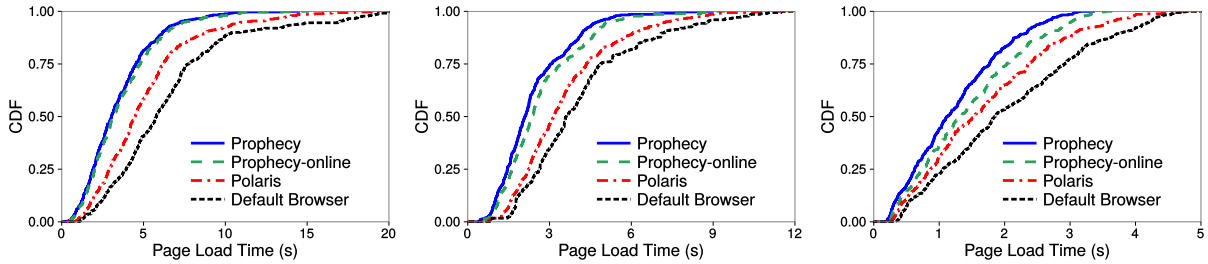
The mobile phone was connected to the desktop via both USB tethering and a live wireless connection (Verizon LTE or WiFi) with excellent signal strength. The desktop ran the test driver, initiating mobile page loads by sending commands through the USB connection. HTTP and DNS traffic between the phone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

In each of our experiments, we considered two versions of Prophecy: an *offline* version in which Prophecy frames were computed before clients requested them, and an *online* version in which the write logs were computed on-demand, in the critical path of each HTTP request (§3.4). Throughout this section, we refer to the offline version as Prophecy, and the online version as Prophecy-online. We compared the versions to default Chrome page loads, and to page loads that used Polaris [36], a state-of-the-art web accelerator. Polaris uses a client-side JavaScript library to schedule the fetching and evaluation of a page's objects. Polaris improves page load time through parallel use of the CPU and the network, and by prioritizing the fetching of objects along the dynamic critical path in a page's dependency graph. However, Polaris does not inline content or apply precomputation.

We evaluated each system on several metrics. Page load time (PLT) is the page-level equivalent of FLT (§3.5). In other words, PLT measures the time required for a browser to fetch all of the content in all of a page's frames. We evaluated Prophecy using PLT instead of FLT because PLT better captures a human's notion of a page being loaded when all of the page's frames are loaded. To measure PLT, we recorded the time between the JavaScript `navigationStart` and `onload` events. RI was computed using Vesper [37], and SI was measured using Speedline [24]. In each experiment, we loaded every page in our corpus 5 times for each system listed above, recording the median value for each load metric. Unless otherwise specified, all experiments used cold browser caches and DNS caches. In experiments with a mobile phone, energy savings were recorded by directly connecting the phone's battery leads to a Monsoon power monitor [33].

5.1 Reducing PLT

Figure 4 illustrates Prophecy's ability to reduce PLT for both mobile devices and desktop machines. Prophecy's benefits are the largest on mobile devices; for example, when using a phone to load a page over an LTE network, Prophecy reduces median PLT by 53%, and 95th percentile PLT by 67%. Prophecy helps mobile devices more for two reasons.



(a) Mobile: 4G LTE cellular network (b) Mobile: Residential WiFi network (c) Desktop: 25 Mbps link, 10 ms RTT

Figure 4: Distribution of page load times with Prophecy, Prophecy-online, Polaris, and a default browser.

- First, mobile devices suffer from higher CPU overheads for page loads, compared to desktop machines [35, 52]. So, Prophecy's elision of intermediate computation (including reflows and repaints) is more impactful on mobile devices.
- PLT is much more sensitive to network latency than to network bandwidth [1, 6, 46, 47]. Cellular links typically exhibit higher latencies than wired or WiFi links. Prophecy's aggressive use of inlining allows clients to fetch all frame content in a single HTTP-level RTT. Such RTT elision unlocks disproportionate benefits in cellular settings.

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

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

As expected, PLT improvements with Prophecy-online are lower than with Prophecy, since Prophecy-online generates a frame's write logs on-demand, upon receiving a request for that frame. However, Prophecy-online still reduces median PLT by 49% on the LTE connection.

5.2 Reducing Bandwidth

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

- A Prophecy frame contains a write log which generates the final, precomputed JavaScript heap for the frame. The JavaScript write log is typically smaller than the frame's original JavaScript source code; although the write log must recreate the original function declarations, the log can omit intermediate function *invocations* that would incrementally create frame state.
- The HTML write log for a frame consists of an augmented HTML string that contains precomputed, inline styles for the appropriate tags. The HTML write log tends to be *larger* than a frame's original HTML string, since traditional CSS declarations can often cover multiple tags with a single CSS rule.
- The image prefetch log does not change the size of images. The log is simply a list of image URLs.

Setting	System	Bandwidth Savings (KB)
Mobile	Prophecy	262 (587)
Mobile	Polaris	-37 (-5)
Desktop	Prophecy	336 (695)
Desktop	Polaris	-41 (-12)

Table 1: Median (95th percentile) per-page bandwidth savings with Prophecy and Polaris. The baseline was the bandwidth consumed by a normal page load. The average mobile page in our test corpus was 1519 KB large; the average desktop page was 2388 KB in size.

Table 1 depicts the overall bandwidth savings that Prophecy enables; note that bandwidth savings are identical for Prophecy and Prophecy-online. Table 1 shows that Prophecy's large reductions in JavaScript size outweigh the small increases in HTML size, reducing overall bandwidth requirements by 21% in the mobile setting, and 18% in the desktop setting. In contrast, Polaris *increases* page size by a small amount. This is because a Polaris page consist of a page's original objects, plus the client-side scheduler stub and scheduler metadata.

5.3 Energy savings

Figure 5 demonstrates that Prophecy significantly reduces the energy consumed during a mobile page load. Median reductions in per-page energy usage are 36% on an LTE network, and 30% on a WiFi network. For both networks, Prophecy eliminates the same amount

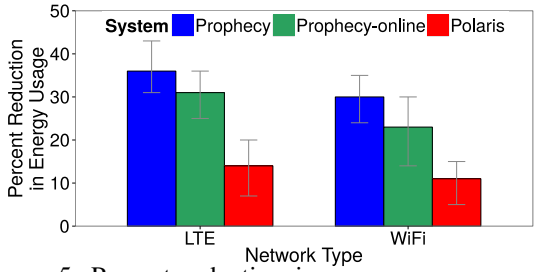


Figure 5: Percent reduction in per-page energy usage with Prophecy, Prophecy-online, and Polaris, relative to a default page load. Bars show median values, and error bars range from the 25th to the 75th percentiles. Results were collected using a Nexus 6 smartphone.

of browser computation, the same number of HTTP-level RTTs, and the same amount of HTTP-level transfer bandwidth. However, LTE hardware consumes more energy in the active state than WiFi hardware [46]; thus, reducing network traffic saves more energy on an LTE network than on a WiFi network.

Prophecy provides more energy reductions than Prophecy-online—36% versus 31% for LTE, and 30% versus 23% for WiFi. The reason is that, in Prophecy-online, server-side request handling takes longer to complete. As a result, the client-side phone must keep its network hardware active for a longer period.

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

5.4 Reductions in SI and RI

As described in Section 3.5, SI and RI only consider the loading status of above-the-fold state. SI tracks the visual rendering of above-the-fold content, whereas RI considers both visibility and functionality. Our exploration of SI used Prophecy's default configuration. In contrast, the RI experiments used the version of Prophecy which explicitly optimizes for Ready Index (§3.5).

Speed Index: As shown in Figures 6a and 6b, Prophecy actually reduces SI more than it reduces PLT. For mobile browsing over an LTE network, the median SI reduction is 61%; for desktop browsing over a 25 Mbps link with a 10 ms RTT, the reduction is 52%. Recall that, by default, a Prophecy frame reconstructs the entire DOM tree before resurrecting the JavaScript heap (§3.1). Prioritizing DOM construction results in better SI scores, since the browser totally dedicates the CPU to rendering pre-computed HTML before replaying the JavaScript write log. As with prior experiments, the synchronous computational overheads of Prophecy-online result in slightly worse performance compared to Prophecy—57% SI reduction versus 61% in the mobile scenario, and 45% SI reduction versus 52% in the desktop setting. However, the benefits are still significant, and Prophecy-online has several advantages over Prophecy with respect to server-side overheads (§3.4).

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

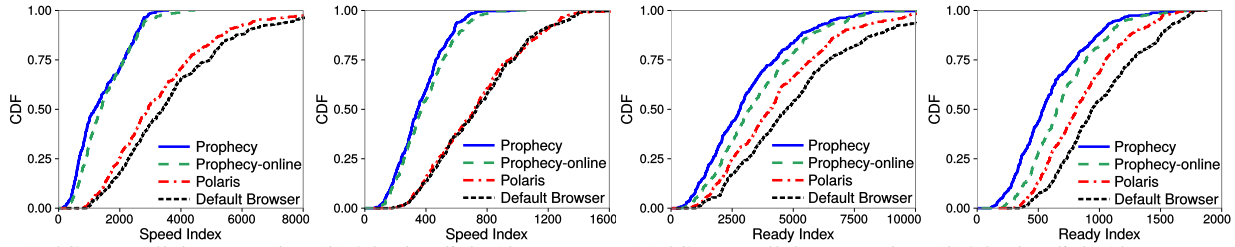
Ready Index: Figures 6c and 6d show that when Prophecy explicitly optimizes for RI (§3.5), Prophecy reduces median RI by 43% in a mobile browsing scenario, and 40% in a desktop setting. User studies indicate that, when users load a page with the expectation of interaction, optimizing for RI leads to happier users [37]. Of course, not all sites have interactive content, or a typical engagement pattern that involves immediate user input. These sites can use the standard Prophecy configuration and enjoy faster PLTs and SIs.

5.5 The Sources of Prophecy's Benefits

Prophecy optimizes a frame load in several ways:

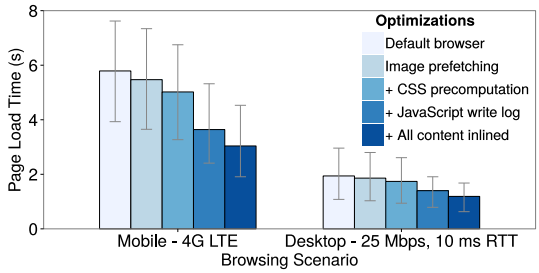
- Image prefetching: The resurrection library issues asynchronous fetches for images before constructing the DOM tree and the JavaScript heap. The asynchronous fetches keep a client's network pipe busy as the CPU works on constructing the rest of the frame.
- CSS precomputation: The DOM write log contains precomputed CSS styles for all DOM nodes, including ones that were dynamically injected by JavaScript code. Precomputation reduces client-side CPU overheads for styling, layout, and rendering.
- JavaScript write log: By only writing to each JavaScript variable once, a browser avoids wasting time and energy on unnecessary JavaScript computations.
- All content inlined: Prophecy consolidates all of the frame content into a single inlined JavaScript file which stores all of the information that is needed to rebuild the frame. Thus, a browser can fetch the entire frame in one HTTP-level round trip, as opposed to needing multiple RTTs to fetch multiple objects.

To better understand how the individual optimizations affect Prophecy's performance, we loaded each page in our corpus with a subset of the optimizations enabled. Our experiments considered mobile browsing using LTE or WiFi networks; we also tested a desktop browser with a 25 Mbps, 10 ms RTT link. In all scenarios, we used a cold cache and measured PLT. In the mobile settings, we

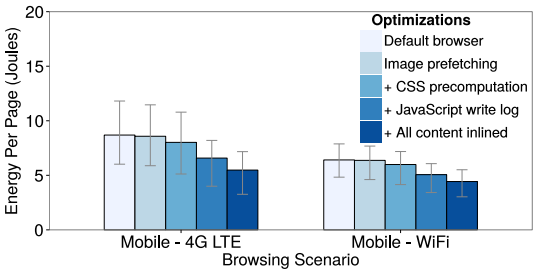


(a) 4G LTE cellular network (b) 25 Mbps link, 10 ms RTT (c) 4G LTE cellular network (d) 25 Mbps link, 10 ms RTT

Figure 6: Evaluating Prophecy using Speed Index and Ready Index.



(a) Breakdown of Prophecy's page load time reductions.



(b) Breakdown of Prophecy's energy savings.

Figure 7: Breakdown of the performance benefits enabled by individual optimizations. Optimization bars begin with image prefetching, and incrementally add new optimizations until "All content inlined," which represents Prophecy's default configuration.

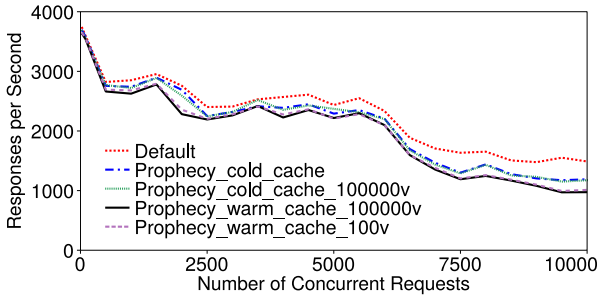


Figure 8: Prophecy-online's impact on server response throughput.

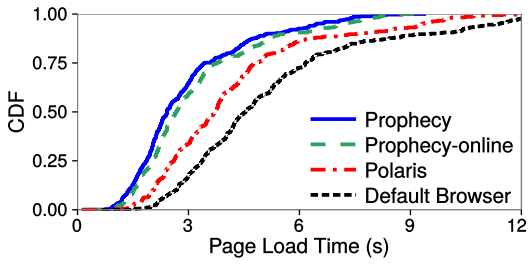
also measured energy usage. Note that Prophecy's bandwidth savings are primarily from the JavaScript heap log; image prefetching and CSS precomputation do not have a significant impact on bandwidth usage (§5.2).

As shown in Figure 7, Prophecy's largest reductions in page load time and energy consumption are enabled by the JavaScript write log optimization. The next most critical optimization is content inlining; saving round trips not only reduces load time, but also reduces the amount of time that a mobile device must actively listen to the network (thereby reducing energy consumption).

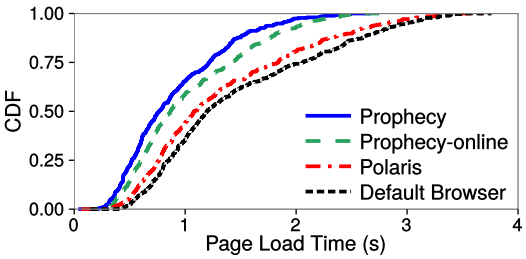
5.6 Server-side overhead

When a Prophecy-online server receives a request for a frame, the server must load the requested frame and generate the necessary write logs on-demand. If the client has a warm cache, then the server must also calculate write log diffs on-demand. Figure 8 depicts the impact that these online calculations have on server response throughput. We used the Apache benchmarking tool `ab` [4] to scale client load and measure response times. The server and `ab` ran on the same machine, to isolate the computational overheads of Prophecy-online. We evaluated five server-side configurations: a default server which returned a frame's normal top-level HTML; Prophecy_cold_cache and Prophecy_cold_cache_v100000, in which clients had cold caches, and the server had either an empty *diffTable* or one that had 100,000 54 KB entries; and Prophecy_warm_cache_v100 and Prophecy_warm_cache_v100000, in which clients had warm caches and the server had the indicated number of *diffTable* entries. For warm cache experiments, we orchestrated `ab` so that all frame versions were accessed with an equal random likelihood. In all experiments, the baseline frame was the top-level frame in the `amazon.com` homepage, and the diff was an empirically-observed diff from two snapshots of the frame that were captured a day apart.

As shown in Figure 8, the performance differences grow as client load grows. Up to 6,500 concurrent requests, all server variants are within 12.1% of each other, but at 10,000 concurrent requests, the difference between the default server and the warm-cache servers is 31.4%. Performance overheads with Prophecy are mostly due to online write log generation. Also note that the CPU overhead of diffing, not the memory overhead of a *diffTable*, leads to the degraded response throughput.



(a) Mobile: 4G LTE cellular network



(b) Desktop: 25 Mbps link, 10 ms RTT

Figure 9: Distribution of warm cache page load times with Prophecy, Prophecy-online, Polaris, and a default browser. Warm cache page loads were performed 1 hour after their cold cache counterparts.

5.7 Additional Results

Due to space restrictions, we defer a full discussion of our remaining experiments to the appendix. Here, we briefly summarize the results of those experiments:

Caching: Roughly 50% of users have an empty browser cache for a particular page load [53, 54]. So, load optimizers should provide benefits if caches are warm *or* cold. The results in this section have assumed a cold cache, but Section A.1 describes the performance of Prophecy in warm cache scenarios. Unsurprisingly, Prophecy's benefits are smaller, but as shown in Figure 9, the gains are still significant, with PLT decreasing by a median of 43% in a mobile setting, and 34% in a desktop setting. Prophecy also maintains its performance advantages over Polaris with respect to SI, RI, bandwidth consumption, and energy expenditure.

Non-landing pages: Our main test corpus consisted of the landing pages for the Alexa Top 350 sites. We also tested Prophecy on 200 interior pages that were linked to by landing pages. As described in Section A.2, Prophecy performs slightly *better* on interior pages, since they tend to have more complicated structures than landing pages.

Diff sizes: We empirically analyzed snapshots of live pages, measuring how large diffs would be for clients with warm caches (Section A.3). For clients with a day-old warm cache, the median diff size was 38 KB, with a 95th percentile size of 81 KB. So, diffs are small enough for a server's *diffTable* to store many of them.

6 Related Work

6.1 Prepack

Prepack [16] is a JavaScript-to-JavaScript compiler. Prepack scans the input JavaScript code for expressions whose results are statically computable; Prepack replaces those expressions with equivalent, shorter sequences that represent the final output of the elided expressions. If JavaScript code contains dynamic interactions with the environment (e.g., via calls to `Date()`), Prepack leaves those interactions in the transformed code, so that they will be performed at runtime.

Prepack does not handle the DOM, or interactions between HTML, CSS, and JavaScript. Prepack is also unaware of important desiderata for web pages, like object cacheability and personalization (§3.3), and incremental interactivity (§3.5). Thus, Prepack is insufficiently powerful to act as a general-purpose web accelerator. Prepack's ability to elide intermediate JavaScript computations is shared by Prophecy, but Prophecy's elision is more aggressive. Prepack uses symbolic execution [10, 14] and abstract interpretation [13] to allow the results of environmental interactions to live in post-processed JavaScript as abstract values; in contrast, Prophecy evaluates all environmental interactions on the server-side, allowing all of the post-processed data to be concrete. This aggressive elision is well-suited for Prophecy's goal of minimizing client-side power usage. For example, if environmental interactions occur in a loop, Prophecy only outputs the final results, whereas Prepack often has to output an abstract, finalized-at-runtime computation for each loop iteration.

6.2 Shandian

Shandian [51] uses a proxy to accelerate page loads. The proxy uses a modified variant of Chrome to load a requested page and generate two snapshots:

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

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

At the architectural level, the key difference between Prophecy and Shandian is that Prophecy tracks fine-grained reads and writes during a server-side page load. Shandian does not. This design decision has cascading ramifications for performance, deployability, and robust-

ness, as described in great detail in Section A.4. For example, Shandian cannot optimize for RI; more generally, Shandian cannot interleave the resurrection of JavaScript code and the DOM tree. The reason is that Shandian lacks an understanding of how the JavaScript heap and the DOM tree interact with each other, so Shandian cannot make interleaved reconstruction safe. The specific lack of write logs for the DOM tree and the JavaScript heap also makes it difficult for Shandian to resurrect state and support caching. JavaScript is a baroque, dynamic language, and the lack of write logs forces Shandian's resurrection logic to use complex, overly conservative rules about (for example) which JavaScript statements are idempotent and which ones are not. The complicated logic requires in-browser support to get good performance, and makes caching semantics sufficiently hard to get right that Shandian does not try to support caching for load-time state (and Shandian only supports a limited form of caching for post-load state). In contrast, Prophecy's use of read/write tracking enables straightforward diff-based caching, safe interleaving of DOM construction and JavaScript resurrection, and browser agnosticism (since Prophecy's write logs are just JavaScript variables). Prophecy also enforces traditional privacy policies for cookies, unlike Shandian (§A.4).

Shandian's source code is not publicly available, and there are no public Shandian proxies. So, we could not perform an experimental comparison with Prophecy. Based on the performance numbers in the Shandian paper, we believe that Prophecy's PLT savings are roughly equivalent to those of Shandian, but Prophecy's bandwidth savings are roughly 20% better. The Shandian paper did not evaluate energy consumption, but we believe that Prophecy will consume less energy due to a simpler resurrection algorithm and less network traffic at resurrection time. Prophecy provides these benefits while enabling a constellation of important features (e.g., cacheability, optimization for interactivity) that Shandian does not provide. We refer the interested reader to Section A.4 for a more detailed discussion of Shandian.

6.3 Split browsers

In a split-browser system [38, 45, 46], a client fetches the top-level HTML in a page via a remote proxy. The proxy forwards the request to the appropriate web server. Upon receiving the response, the proxy uses a headless browser to load the page; as the proxy parses HTML, executes JavaScript, and discovers external objects in the page, the proxy fetches those objects and then forwards them to the client. Since the proxy has fast, low-latency network paths to origin servers, the time needed to resolve a page's dependency graph [11, 36] is mostly bound by proxy/origin RTTs (which are small), not the last-mile client/proxy RTTs (which may be large).

Prophecy is compatible with such approaches—a Prophecy frame can be loaded by a split-browser proxy. However, the only external objects that the proxy would discover are images, since a Prophecy frame inlines the (final effects of) external CSS and JavaScript objects. Also note that the goal of a split-browser is to hide the network latency associated with a client's object fetches; split browsers cannot identify *client-side computations* that may be elided. Prophecy does find such computations, while also eliminating fetch RTTs via inlining.

6.4 Mobile web optimizations

Klotski [9] is a mobile web optimizer that uses server-push (§3.7). When a browser fetches HTML for a particular page, the Klotski web server returns the HTML, and also pushes high-priority objects which are referenced by the page (and will later be requested by the browser). Klotski identifies high-priority objects in an offline phase using a utility function (e.g., that prioritizes above-the-fold content). Prophecy is compatible with server-push, but at the granularity of entire frames, not individual objects, since Prophecy inlines content (§3.7). Inlining, combined with final-state patching, allows Prophecy to both lower load time and decrease energy consumption. In contrast, a Klotski page elides no computation. VROOM [42] is similar to Klotski, except that clients prefetch data instead of receiving server pushes; a VROOM server uses link preload headers [22] in returned HTTP responses to hint to clients which objects can be usefully prefetched.

AMP [19] accelerates mobile page loads by requiring pages to be written in a restricted dialect of HTML, CSS, and JavaScript that is faster to load. For example, AMP forces all external `<script>` content to use the `async` attribute so that the browser's HTML parse can continue as the JavaScript code is fetched in the background. AMP forces a page to have at most one CSS file, which must be an inlined `<style>` tag whose contents are less than 50 KB in size. Prophecy is designed to support arbitrary pages that use arbitrary HTML, CSS, and JavaScript. However, Prophecy can be applied to AMP pages since those pages are just HTML, CSS, and JavaScript.

7 Conclusion

Prophecy is a new acceleration system for mobile page loads. Prophecy uses precomputation to reduce (1) the amount of state which must be transmitted to browsers, and (2) the amount of computation that browsers must perform to build the desired pages. Unlike current state-of-the-art systems for precomputation, Prophecy handles all kinds of page state, including DOM trees, and supports critical features like object caching, incremental interactivity, and cookie privacy. Experiments show that Prophecy enables substantial reductions in page load time, bandwidth usage, and energy consumption.

REFERENCES

[1] 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. In *Proceedings of NSDI*, 2015.

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

[3] Amazon. What Is Amazon Silk? https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html, 2018.

[4] Apache Software Foundation. ab: Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2018.

[5] A. Barth. HTTP State Management Mechanism. RFC 6265. https://tools.ietf.org/html/rfc6265, April 2011.

[6] M. Belshe. More Bandwidth Doesn't Matter (Much). Google. https://goo.gl/PFDGMi, April 8, 2010.

[7] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2. RFC 7540. https://tools.ietf.org/html/rfc7540, May 2015.

[8] D. H. Bui, Y. Liu, H. Kim, I. Shin, and F. Zhao. Rethinking Energy-Performance Trade-Off in Mobile Web Page Loading. In *Proceedings of Mobicom*, 2015.

[9] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proceedings of NSDI*, 2015.

[10] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of OSDI*, 2008.

[11] Y. Cao, J. Nejati, M. Wajahat, A. Balasubramanian, and A. Gandhi. Deconstructing the Energy Consumption of the Mobile Page Load. In *Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS, New York, NY, USA, 2017. ACM.

[12] J. Chernofsky. Why emerging markets are dominating mobile browsing. The Next Web. https://thenextweb.com/insider/2016/04/07/first-world-problems-emerging-markets-dominating-mobile-browsing/, April 7, 2016.

[13] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of POPL*, 1977.

[14] P. D. Coward. Symbolic Execution Systems: A Review. *Software Engineering Journal*, 3(6):229–239, November 1988.

[15] T. Everts. New findings: For Top Ecommerce Sites, Mobile Web Performance is Wildly Inconsistent. https://blog.radware.com/applicationdelivery/wpo/2014/10/2014-mobile-ecommerce-page-speed-web-performance/, October 22, 2014.

[16] Facebook. Prepack: Partial evaluator for javascript. https://prepack.io/, 2017.

[17] D. F. Galletta, R. Henry, S. McCoy, and P. Polak. Web Site Delays: How Tolerant are Users? *Journal of the Association for Information Systems*, 5(1), 2004.

[18] S. Gibbs. Mobile web browsing overtakes desktop for the first time. The Guardian. https://www.theguardian.com/technology/2016/nov/02/mobile-web-browsing-desktop-smartphones-tablets, November 2, 2016.

[19] Google. Accelerated Mobile Pages Project - AMP. https://www.ampproject.org/, 2018.

[20] Google. google-diff-match-patch. https://github.com/google/diff-match-patch, February 13, 2018.

[21] Google. Speed Index: WebPagetest Documentation. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index, 2018.

[22] I. Grigorik and Y. Weiss. Preload. https://www.w3.org/TR/preload/, October 26, 2017.

[23] GSMA Intelligence. Global Mobile Trends 2017. https://www.gsmaintelligence.com/research/?file=3df1b7d57b1e63a0cbc3d585feb82dc2&download, September 2017.

[24] P. Irish. Speedline. https://github.com/paulirish/speedline, November 21, 2017.

[25] P. Irish. What forces layout/reflow. https://gist.github.com/paulirish/5d52fb081b3570c81e3a, February 6 2018.

[26] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of SIGCOMM*, 2017.

[27] P. Lewis. Avoid Large, Complex Layouts and Layout Thrashing. Google Developers. https://developers.google.com/web/fundamentals/performance / rendering / avoid - large - complex - layouts-and-layout-thrashing, May 12, 2017.

[28] H. Lindqvist. CSSUtilities. http://www.brothercake.com/site/resources/scripts/cssutilities/, April 4, 2010.

[29] B. McQuade, D. Phan, and M. Vajolahi. Instant Mobile Websites: Techniques and Best Practices. Google I/O Conference presentation. http://goo.gl/DfhPJT, May 16, 2013.

[30] J. Mickens. Rivet: Browser-agnostic Remote Debugging for Web Applications. In *Proceedings of USENIX ATC*, 2012.

[31] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic Capture and Replay for Javascript Applications. In *Proceedings of NSDI*, 2010.

[32] J. Mickens, J. Elson, J. Howell, and J. Lorch. Crom: Faster Web Browsing Using Speculative Execution. In *Proceedings of NSDI*, 2010.

[33] Monsoon Solutions Inc. Power monitor software. http://msoon.github.io/powermonitor/, 2015.

[34] Mozilla Developer Network. Document Object Model (DOM). https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model, August 29, 2017.

[35] J. Nejati and A. Balasubramanian. An In-depth Study of Mobile Browser Performance. In *Proceedings of WWW*, 2016.

[36] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *Proceedings of NSDI*, 2016.

[37] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring Time-to-Interactivity for Modern Web Pages. In *Proceedings of NSDI*, 2018.

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

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

[40] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race Detection for Web Applications. In *Proceedings of PLDI*, 2012.

[41] L. Richardson. Beautiful Soup. http://www.crummy.com/software/BeautifulSoup/, February 17, 2016.

[42] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of SIGCOMM*, 2017.

[43] S. Shankland. Ad industry attacks Safari's effort to protect your privacy. CNET. https://www.cnet.com/news/ad-industry-attacks-safaris-effort-to-protect-your-privacy/, September 15, 2017.

[44] S. Singh, H. V. Madhyastha, S. V. Krishnamurthy, and R. Govindan. FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers. In *Proceedings of Mobicom*, 2015.

[45] A. Sivakumar, C. Jiang, Y. S. Nam, P. Shankaranarayanan, V. Gopalakrishnan, S. Rao, S. Sen, M. Thottethodi, and T. Vijaykumar. NutShell: Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Proceedings of MobiSys*, 2017.

[46] 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 CoNEXT*, 2014.

[47] S. Sundaresan, N. Feamster, R. Teixeira, and N. Magharei. Measuring and Mitigating Web Performance Bottlenecks in Broadband Access Networks. In *Proceedings of IMC*, 2013.

[48] N. Thiagarajan, G. Aggarwal, A. Nicoara, D. Boneh, and J. P. Singh. Who Killed My Battery?: Analyzing Mobile Browser Energy Consumption. In *Proceedings of WWW*, 2012.

[49] J. Vesuna, C. Scott, M. Buettner, M. Piatek, A. Krishnamurthy, and S. Shenker. Caching Doesn't Improve Mobile Web Performance (Much). In *Proceedings of USENIX ATC*, 2016.

[50] T. Vrountas. How Slow Mobile Page Speeds Are Ruining Your Conversion Rates. https://instapage.com/blog/optimizing-mobile-page-speed, August 5, 2017.

[51] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding Up Web Page Loads with Shandian. In *Proceedings of NSDI*, 2016.

[52] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why Are Web Browsers Slow on Smartphones? In *Proceedings of HotMobile*, 2011.

[53] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How Far Can Client-only Solutions Go for Mobile Browser Speed? In *Proceedings of WWW*, 2012.

[54] YUI Team. Performance Research, Part 2: Browser Cache Usage - Exposed! https://yuiblog.com/blog/2007/01/04/performance-research-part-2/, January 4, 2007.

[55] Y. Zhu and V. J. Reddi. High-performance and energy-efficient mobile web browsing on big/little systems. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[56] Y. Zhu and V. J. Reddi. WebCore: Architectural Support for Mobileweb Browsing. In *Proceeding of the International Symposium on Computer Architecuture (ISCA)*, 2014.

A Appendix

A.1 Warm Browser Caches

The experiments in Section 5 assumed a cold browser cache. Here, we explore the performance of Prophecy when caches are warm, finding that Prophecy still unlocks significant decreases in page load time, bandwidth consumption, and energy expenditure.

For each page in our test corpus, we used Mahimahi

Setting	System	Bandwidth Savings (KB)
Mobile	Prophecy	176 (441)
Mobile	Polaris	-37 (-5)
Desktop	Prophecy	298 (571)
Desktop	Polaris	-41 (-12)

Table 2: Median (95th percentile) per-page bandwidth savings with Prophecy and Polaris, using warm browser caches. The baseline was the bandwidth consumed by a default browser with a warm cache. The average warm cache mobile page load in our test corpus consumed 664 KB; the average desktop page used 973 KB.

Update frequency	Pages	Frames
<= 1 hour	124	313
1-2 hours	77	114
2-4 hours	41	76
4-8 hours	19	20
8-24 hours	49	109
>= 24 hours	41	229

Table 3: Update frequencies for the pages and Prophecy frames in our corpus.

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

PLT reductions: Figure 9 corroborates prior caching studies which found that mobile caching is less effective than desktop caching at reducing PLT [49]. However, Figure 9 demonstrates that Prophecy still provides substantial benefits compared to both Polaris and a default page load. For example, Prophecy enables median PLT reductions of 43% in the mobile setting, and 34% in the desktop setting. An important reason for Prophecy's persistent benefit is that, even in a warm-cache Prophecy frame (§3.3), Prophecy elides computation that must be incurred by Polaris and a default page load.

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

Bandwidth savings: Table 2 demonstrates that Prophecy reduces per-page bandwidth consumption by 26% (176 KB) for mobile browsing, and 30% (298 KB) for desktop browsing. The raw savings are less than the cold cache scenarios for obvious reasons. However, since Prophecy can cache at byte granularity, not file granularity (§3.3), Prophecy downloads fewer network bytes than either Polaris or a default load.

Energy savings: Prophecy's energy savings decrease in warm cache page loads. The reason is that caching is more effective at reducing energy costs than page load time [11]; having an object cached will always avoid the battery drain associated with a network fetch, but may not decrease PLT much if the cached object is not on the critical path in the page's dependency graph [11, 36]. Regardless, Prophecy still provides substantial energy savings, reducing median and 95th percentile consumption by 17% and 29% for an LTE network. Prophecy-online's energy savings are lower than Prophecy (12% and 21%), but are higher than those of Polaris (6% and 12%).

A.2 Additional Sites

In addition to the 350 site corpus that we used for our main experiments, we also evaluated Prophecy on two additional sets of sites. First, using a web monkey, we generated a list of 200 additional pages by performing clicks on the pages in our original corpus; we generated 4 clicks per page, and then randomly selected 200 pages from the 1400 page list. These pages represented interior pages for websites, rather than the landing pages which are provided by the Alexa lists. We performed the same PLT experiments as described in Section 5.1, loading pages with a mobile phone over an LTE network. The trends were similar to those in our primary corpus. Median speedups with Prophecy increased to 57%, while Prophecy-online and Polaris accelerated PLT by 53% and 26%, respectively.

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

A.3 Diff Characteristics

To understand how large diffs would be in practice, we recorded 6 versions of each page in our corpus: a baseline version (at time t=0), and versions recorded at t values of 1 hour, 2 hours, 4 hours, 8 hours, and 24 hours. We then computed Prophecy frames for each version of each page. Finally, we computed diffs for each version of each frame, comparing against the baseline frame from

t=0. The server's diff calculations were fast: across all versions of the page, the median computation time was 4.6 ms, and the 95th percentile time was 8.8 ms. The median size for the largest diff across all frame versions was 38 KB; the 95th percentile largest diff size was 81 KB.

As shown in Table 3, 35% of the pages in our corpus require diff updates at least once an hour. In contrast, 12% of the pages do not require any diff updates within a single day. Similarly, some frames must be updated frequently, and some rarely change.

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

A.4 Detailed Discussion of Shandian

In Section 6.2, we provided a high-level comparison of Shandian and Prophecy. Here, we provide more technical detail about how Shandian works, and why we believe that Prophecy's write log approach is advantageous.

Robustness: Shandian's load-time snapshot is just serialized HTML and CSS. However, Shandian's post-load snapshot cannot contain the page's unmodified JavaScript code, since client-side execution of the code would encounter a different DOM environment than what would have been seen in a normal page load. Thus, resurrecting the JavaScript state is challenging. Shandian's approach is to create a post-load snapshot which contains (1) a serialized graph of JavaScript objects minus their methods, and (2) a set of JavaScript function definitions that Shandian extracted from the page's original JavaScript code. Splicing this post-load state into the client-side environment requires complex, subtle reasoning about idempotency and ordering. For example, in a JavaScript program, a single function definition can be evaluated multiple times, with each evaluation binding to a different set of closure variables that are chosen using dynamic information. Some of the closure variables may themselves be functions. Thus, Shandian requires careful logic to generate a function evaluation order that results in the desired final state; using the lexical order of function definitions in the original source code is insufficient. Our personal experience writing JavaScript heap serializers [30, 32] has convinced us that serialization-based approaches are fragile and difficult to make correct. Prophecy's ability to track writes dramatically simplifies matters. With knowledge of the final state of each function, object, and primitive property, Prophecy can apply a straightforward three-pass algorithm to recreate an interconnected DOM tree and JavaScript heap (§3.2). Thus, we believe that a write log approach is simpler and more robust than a serialization-based approach.

Liveness: Shandian lacks a fine-grained understanding of interactions between the JavaScript heap and the DOM, so Shandian cannot safely interleave DOM construction and JavaScript evaluation. As a result, Shandian must restore all JavaScript state at once, after the DOM has been constructed. This limitation prevents Shandian from making pages incrementally interactive (§3.5). Deferring JavaScript execution has other disadvantages, like timer-based animations not starting until the associated JavaScript code has been fetched and evaluated. In contrast, Prophecy can identify related clusters of DOM nodes and JavaScript state, enabling safe, interleaved construction of a page's DOM tree and JavaScript heap. Prophecy also returns all of the page state to the client in a single HTTP round trip, unlike Shandian, which requires multiple RTTs.

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

Caching: Shandian provides no caching support for the content in the initial snapshot. So, if just a single byte in the initial snapshot changes, the client must download an entirely new snapshot, spending precious energy and network bandwidth. Shandian supports caching for the post-load data, but the content in that snapshot is dependent on the content in the load-time snapshot! Thus, if the load-time snapshot changes, then cached post-load content is invalidated. In contrast, Prophecy provides a straightforward caching scheme that supports byte-level diffing (§3.3), maximizing the amount of cached content that can be used to reconstruct new versions of a page. Prophecy's caching approach is naturally suggested by Prophecy's use of write logs—these write logs are easily diffed using standard algorithms. In contrast, given Shandian's complex resurrection approach, it is not immediately clear how Shandian could be extended to support traditional caching semantics.

CSS: On the client browser, Shandian evaluates load-time CSS rules twice: once during the initial load, and again during the evaluation of a page's post-load CSS. As the Shandian paper states, the result is "additional energy consumption and latencies." We cannot quantify the costs due to lack of access to a Shandian system. Thus, we merely observe that Prophecy's inlining of CSS styles avoids *all* client-side CSS parsing for load-time DOM nodes—the associated CSS rules are evaluated *zero* times on the client. Note that, post-load, a Prophecy page can immediately style dynamically-created DOM nodes (§3.2). In contrast, Shandian will either have to wait for post-load CSS styles to be fetched (which may take a long time on a slow mobile link), or style the node immediately, but possibly incorrectly (leading to broken page state).

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