Vesper: Measuring Time-to-Interactivity for Web Pages

Ravi Netravali[†*], Vikram Nathan[†*], James Mickens[‡], Hari Balakrishnan[†]
[†]MIT CSAIL, [‡]Harvard University

Abstract

Everyone agrees that web pages should load more quickly. However, a good definition for "page load time" is elusive. We argue that, for pages that care about user interaction, load times should be defined with respect to *interactivity*: a page is "loaded" when above-the-fold content is visible, and the associated JavaScript event handling state is functional. We define a new load time metric, called *Ready Index*, which explicitly captures our proposed notion of load time. Defining the metric is straightforward, but actually measuring it is not, since web developers do not explicitly annotate the JavaScript state and the DOM elements which support interactivity. To solve this problem, we introduce Vesper, a tool that rewrites a page's JavaScript and HTML to automatically discover the page's interactive state. Armed with Vesper, we compare Ready Index to prior load time metrics like Speed Index; across a variety of network conditions, prior metrics underestimate or overestimate the true load time for a page by 24%–64%. We introduce a tool that optimizes a page for Ready Index, decreasing the median time to page interactivity by 29%–32%.

1 Introduction

Users want web pages to load quickly [31, 40, 42]. Thus, a vast array of techniques have been invented to decrease load times. For example, browser caches try to satisfy network requests using local storage. CDNs [9, 27, 36] push servers near clients, so that cache misses can be handled with minimal network latency. Cloud browsers [4, 29, 34, 38] resolve a page's dependency graph on a proxy that has low-latency links to web servers; this allows a client to download all objects in a page using a single HTTP round-trip to the proxy.

All of these approaches try to reduce page load time. However, an inconvenient truth remains: none of these techniques directly optimize the speed with which a page becomes *interactive*. Modern web pages have sophisticated, dynamic GUIs that contain both visual and programmatic aspects. For example, many sites provide a search feature via a text input with autocompletion support. From a user's perspective, such a text input is worthless if the associated HTML tags have not been rendered; however, the text input is also crippled if the JavaScript code that implements autocompletion has not been fetched and evaluated. JavaScript code can also implement animations or other visual effects that do not receive GUI inputs directly, but which are still necessary for a page to be ready for user interaction. As shown in Figure 1, pages often contain *hundreds* of event handlers that drive interactivity.

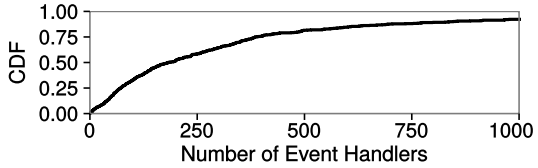


Figure 1: For the Alexa US Top 500 sites, we observed the median number of GUI event handlers to be 182.

In this paper, we propose a new definition for load time that directly captures page interactivity. We define a page to be fully loaded when:

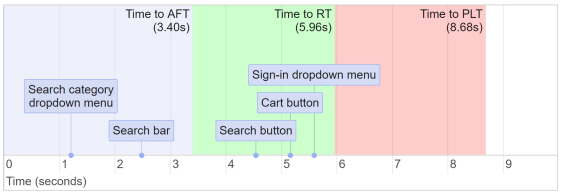
(1) the visual content in the initial browser viewport[1] has completely rendered, and
(2) for each interactive element in the initial viewport, the browser has fetched and evaluated the JavaScript and DOM state that supports the element's interactive functionality.

Prior definitions for page load time overdetermine or underdetermine one or both of those conditions (§2), leading to inaccurate measurements of page interactivity. For example, the traditional definition of page load time, as represented by the JavaScript `onload` event, captures when *all* of a page's HTML, JavaScript, CSS, and images have been fetched and evaluated; however, this definition is overly conservative, since only a subset of that state may be needed to allow a user to interact with the content in the initial viewport. Newer metrics like above-the-fold time [21] and Speed Index [14] measure the time that a page needs to render the initial viewport. However, these metrics do not capture whether the page has loaded critical JavaScript state (e.g., event handlers that respond to GUI interactions, or timers that implement animations).

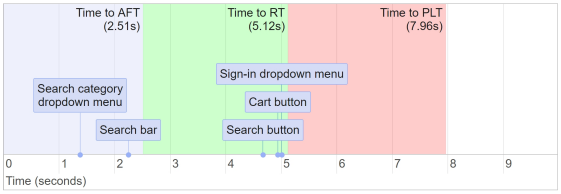
To accurately measure page interactivity, we must determine when conditions (1) and (2) are satisfied. Determining when condition (1) has been satisfied is relatively straightforward, since rendering progress can be measured using screenshots or the paint events that are emitted by the browser's debugger interface. However, determining when condition (2) has been satisfied is challenging. How does one precisely enumerate the JavaScript state that supports interactivity? How does one determine when this state is ready? To answer these questions, we introduce a new measurement framework called *Vesper*.

[*] These authors contributed equally to this work.

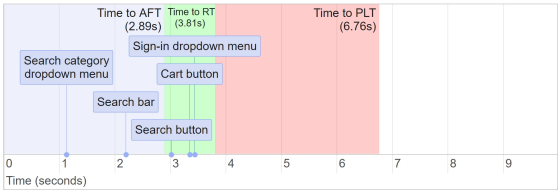
[1] The viewport is the region of a page that the browser is currently displaying. Content in the initial viewport is often called "above-the-fold" content.



(a) Loading the normal version of the page.



(b) Loading a version that optimizes for above-the-fold time.



(c) Loading a version that optimizes for Ready Time.

Figure 2: Timelines for loading `amazon.com`, indicating when critical interactive components become fully interactive. Note that Ready Time best captures when the site is interactive; furthermore, optimizing for Ready Time is the best way to decrease the page's time-to-interactivity. The client used a 12 Mbit/s link with a 100 ms RTT (§5.1).

RTT	PLT	RT	AFT
25 ms	1.5 (3.9)	1.1 (2.9)	0.8 (1.9)
50 ms	3.4 (7.2)	2.5 (5.8)	1.9 (4.7)
100 ms	6.1 (12.5)	3.9 (9.1)	2.9 (7.0)
200 ms	9.2 (20.6)	5.6 (12.8)	3.8 (8.9)

Figure 3: Median (95th percentile) load time estimates in units of seconds. Each page in our 350 site corpus was loaded over a 12 Mbit/s link.

Vesper rewrites a page's JavaScript and HTML; when the rewritten page loads, the page automatically logs paint events as well as reads and writes to individual JavaScript variables and DOM elements.[2] By analyzing these logs, Vesper generates a progressive load metric, called *Ready Index*, which quantifies the fraction of the initial viewport that is interactive (i.e., visible and functional) at a given moment. Vesper also outputs a derived metric, called Ready Time, which represents the exact time at which *all* of the above-the-fold state is interactive.

Using a test corpus of 350 popular sites, we compared our new load metrics to traditional ones. Figure 2(a) provides a concrete example of the results, showing the differences between page load time (PLT), above-the-fold time (AFT), and Ready Time (RT) for the `amazon.com` homepage when loaded over a 12 Mbit/s link with a 100 ms RTT. AFT underestimates time-to-full-interactivity by 2.56 seconds; PLT overestimates the time-to-full-interactivity by 2.72 seconds. Web developers celebrate the elimination of *milliseconds* of "load time," claiming that a slight decrease can result in millions of dollars of extra income for a large site [6, 8, 41]. However, our results suggest that developers may be optimizing for the wrong definition of load time. As shown in Figure 3, prior metrics inaccurately forecast time-to-full-interactivity under a variety of network conditions, with median inaccuracies of 24%–39%; as shown in our user study (§6), users with interactive goals prefer websites that actually prioritize the loading of interactive content.

The differences between load metrics are particularly stark if a page's dependency graph [25, 37] is deep, or if a page's clients are stuck behind high-latency links. In these scenarios, the *incremental interactivity* of a slowly-loading page is important: as the page trickles down the wire, interactive HTML tags should become visible and functional as soon as possible. This allows users to meaningfully engage with the site, even if some content is missing; incremental interactivity also minimizes the time window for race conditions in which user inputs are generated at the same time that JavaScript event handling state is being loaded [30]. To enable developers to build incrementally-interactive pages with low Ready Indices, we extended Polaris [25], a JavaScript framework that allows a page to explicitly schedule the order in which objects are fetched and evaluated. We created a new Polaris scheduler that optimizes for Ready Index; the resulting scheduler improves RI by a median of 29%, and RT by a median of 32%. Figure 2(c) demonstrates the scheduler's performance on the `amazon.com` homepage. Importantly, Figure 2(b) shows that optimizing for above-the-fold time does *not* optimize for time-to-interactivity.

Of course, not all sites have interactive content, and even interactive sites can be loaded by users who only look at the content. In these situations, pages should optimize for the rendering speed of above-the-fold content. Fortunately, our user study shows that pages which optimize for Ready Index will substantially reduce user-perceived rendering delays too (§6). If desired, Vesper enables developers to automatically optimize their pages solely for rendering speed instead of Ready Index.

In summary, this paper has four contributions. First, we define a new load metric called Ready Index which quantifies a page's interactive status (§3). Determining how interactivity evolves over time is challenging. Thus, our second contribution is a tool called Vesper that automates the measurement of Ready Index (§4). Our third contribution is a study of Ready Index in 350 real pages.

[2]Each HTML tag in a web page has a corresponding DOM element. The DOM element is a special JavaScript object that JavaScript code can use to manipulate the properties of the underlying HTML tag.

By loading those pages in a variety of network conditions, we explain the page characteristics that lead to faster interactivity times (§5). Our fourth contribution is an automated framework for optimizing a page's Ready Index or pure rendering speed; both optimizations are enabled by Vesper-collected data. User studies demonstrate that pages which optimize for Ready Index provide better support for immediate interactivity (§6).

2 BACKGROUND

In this section, we describe prior attempts to define "page load time." Each metric tracks a different set of page behaviors; thus, for a given page load, different metrics may provide radically different estimates of the load time.

The Original Definition: The oldest metric is defined with respect to the JavaScript `onload` event. A browser fires that event when all of the external content in a page's static HTML file has been fetched and evaluated. All image data must be present and rendered; all JavaScript must be parsed and executed; all style files must be processed and applied to the relevant HTML tags; and so on. The load time for a page is defined as the elapsed time between the `navigationStart` event and the `onload` event. In the rest of the paper, we refer to this load metric as PLT ("page load time").

PLT was a useful metric in the early days of the web, but modern web pages often dynamically fetch content after the `onload` event has fired [12, 13]. PLT also penalizes web pages that have large amounts of statically-declared below-the-fold content. Below-the-fold content resides beneath the initial browser viewport, and can only be revealed by user scrolling. PLT requires static below-the-fold content to be fetched and evaluated before a page load is considered done. However, from a user's perspective, a page can be ready even if its below-the-fold content is initially missing: the interactivity of the initial viewport content is the primary desideratum.

Time to First Paint: Time to First Paint (TTFP) measures when the browser has received enough page data to render the first pixels in the viewport. Time to First Meaningful Paint [33], or TTFMP, measures the time until the biggest layout change, using the intuition that the associated paint event is the one that matters most. TTFP and TTFMP try to capture the earliest time that a human could usefully interact with a page. For a given PLT, a lower TTFP or TTFMP is better. However, decreasing a page's PLT is not guaranteed to lower the other metrics, and vice versa [1]. For example, when the HTML parser (which generates input for the rendering pipeline) hits a `<script>` tag, the parser may need to synchronously fetch and evaluate the JavaScript file before continuing the HTML parse [25]. By pushing `<script>` tags to the end of a page's HTML, render times may improve; however, careless deferral of JavaScript evaluation may hurt interactivity, since event handlers will be registered later, animation callbacks will start firing later, and so on.

Above-the-fold Time: This metric represents the time that the browser needs to render the final state of *all* pixels in the initial browser viewport. Like TTFP, above-the-fold time (AFT) is not guaranteed to move in lockstep with PLT. Measuring AFT and TTFP requires a mechanism for tracking on-screen events. WebKit-derived browsers like Chrome and Opera expose paint events via their debugging interfaces. Rendering progress can also be tracked using screenshots [16, 19].

If a web page contains animations, or videos that automatically start playing, a naïve measurement of AFT would conclude that the page never fully loaded. Thus, AFT algorithms must distinguish between *static pixels* that are expected to change a few times at most, and *dynamic pixels* that are expected to change frequently, even once the page has fully loaded. To differentiate between static and dynamic pixels, AFT algorithms use a threshold number of pixel updates; a pixel which is updated more often than the threshold is considered to be dynamic. AFT is defined as the time that elapses until the last change to a static pixel.

Speed Index: AFT fails to capture the progressive nature of the rendering process. Consider two hypothetical pages which have the same AFT, but different rendering behavior: the first page updates the screen incrementally, while the second page displays nothing until the very end of the page load. Most users will prefer the first page, even though both pages have the same AFT.

Speed Index [14] captures this preference by explicitly logging the progressive nature of page rendering. Intuitively speaking, Speed Index tracks the fraction of a page which has not been rendered at any given time. By integrating that function over time, Speed Index can penalize sites that leave large portions of the screen unrendered for long periods of time. More formally, a page's Speed Index is $\int_0^{end} 1 - \frac{p(t)}{100}\,dt$, where *end* is the AFT time, and $p(t)$ is the percentage of static pixels at time t that are set to their final value. A lower Speed Index is better than a higher one.

Strictly speaking, a page's Speed Index has units of "percentage-of-visual-content-that-is-not-displayed milliseconds." For brevity, we abuse nomenclature and report Speed Index results in units of just "milliseconds." However, a Speed Index cannot be directly compared to a metric like AFT that is actually measured in units of time. Also note that TTFP, AFT, and Speed Index do not consider the load status of JavaScript state. As a result, these metrics cannot determine (for example) when a button that has been rendered has actually gone live as result of the associated event handlers being registered.

User-perceived PLT: This metric captures when a user believes that a page render has finished [20, 35]. Unlike Speed Index, User-perceived PLT is not defined programmatically; instead, it is defined via user studies which empirically observe when humans think that enough of a page has rendered for the page load to be "finished." Like Speed Index, User-perceived PLT ignores page functionality (and thus page interactivity). User-perceived PLT also cannot be automatically measured, which prevents developers from easily optimizing for the metric.

TTI: Several commercial products claim to measure a page's time-to-interactivity (TTI) [28, 32]; however, these products do not explicitly state how interactivity is defined or measured. In contrast, Google is currently working on an open standard for defining TTI [15]. The standard's definition of TTI is still in flux. The current definition expresses interactivity in terms of time-to-first-meaningful-paint, the number of in-flight network requests, and the utilization of the browser's main thread (which is used to dispatch GUI events, execute JavaScript event handlers, and render content). TTI defines an "interactive window" as a period in which the main thread runs no tasks that require more than 50 ms; in other words, during an interactive window, the browser can respond to user input in at most 50 ms. A page's TTI is the maximum of:

(1) the time when the `DOMContentLoaded` event has fired, and
(2) the start time of the first interactive window that has at most two network requests in flight for 5 consecutive seconds.

This definition for load time has several problems. First, it could declare a page to be loaded even if the page has not rendered all of the content in the initial viewport. Second, condition (2) does not consider whether a network request is for above-the-fold, interactive content; a window with many outstanding network requests may represent an interactive page if those network requests are for below-the-fold state. Similarly, this TTI definition makes no explicit reference to the JavaScript state that supports above-the-fold event handlers, and the JavaScript state that does not. User-perceived interactivity requires the former state to be loaded, but not the latter.

Summary: Traditional metrics for load time fail to capture important aspects of user-perceived page readiness. PLT does not explicitly track rendering behavior, and implicitly assumes that all JavaScript state is necessary to make above-the-fold content usable. AFT, Speed Index, User-perceived PLT, and TTFP/TTFMP consider visual content, but are largely oblivious to the status of JavaScript code—the code is important only to the extent that it might update a pixel using DOM methods [23]. However, AFT, Speed Index, User-perceived PLT, and TTFP/TTFMP completely ignore event handlers (and the program state that event handlers manipulate). Consequently, these metrics fail to capture the interactive component of page usability. Google's TTI also imprecisely captures above-the-fold, interactive state.

3 READY INDEX

In this section, we formally define Ready Index (RI). Like Speed Index, RI is a progressive metric that captures incremental rendering updates. Unlike Speed Index, RI also captures the progressive loading of JavaScript state that supports interactivity.

Defining Functionality: Let T be an upper-bound on the time that a browser needs to load a page's above-the-fold state, and make that state interactive. This upper-bound does not need to be tight; in practice (§5), we use a static value of 30 seconds.

Let E be the set of DOM elements that are visible in the viewport at T. For each $e \in E$, let $h(e)$ be the set of all event handlers that are attached to e at or before T. Let t_e be the earliest time at which, for all handlers $h \in h(e)$, h's JavaScript function has been declared, and all JavaScript state and DOM state that would be accessed by h's execution has been loaded. Given those definitions, we express the *functionality progress* of e as

$$F(e,t) = \begin{cases} 0 & t < t_e \\ 1 & t \geq t_e \end{cases} \quad (1)$$

Intuitively speaking, Equation 1 states that a DOM node is not functional until all of the necessary event handlers have been attached to the node, and the browser has loaded all of the state that the handlers would touch if executed.

Defining Visibility: An element e may be the target of multiple paint events, e.g., as the browser parses additional HTML and recalculates e's position in the layout. We assume that e is not fully visible until its last paint completes. Let $P(e)$ be the set of all paint events that update e, and let $P_t(e) \subseteq P(e)$ be the paint events that have occurred by time t. The *visibility progress* of e is

$$V(e,t) = \frac{|P_t(e)|}{|P(e)|} \quad (2)$$

Similar to how Speed Index computes progressive rendering scores for pixels [14], Equation 2 assumes that each paint of e contributes equally to e's visibility score. Note that $0 \leq V(e,t) \leq 1$.

Defining Readiness: Given the preceding definitions for functionality and visibility, we define the readiness

of an element e as

$$R(e,t) = \frac{1}{2}F(e,t) + \frac{1}{2}V(e,t) \quad (3)$$

such that the functionality and visibility of e are equally weighed,[3] and $0 \leq R(e,t) \leq 1$. The readiness of the entire page is then defined as

$$R(t) = \sum_{e \in E} A(e)R(e,t) \quad (4)$$

where $A(e)$ is the area (in pixels) that e has at time T.

Putting It All Together: An element e is *fully ready* at time t if $R(e,t) = 1$, i.e., if e is both fully visible and fully functional. A page's Ready Time (RT) is thus the smallest time at which all of the above-the-fold elements are ready. A page's Ready Index (RI) is the area above the curve of the readiness progress function. Thus, RI is equal to

$$RI = \int_0^T 1 - \frac{R(t)}{R(T)}\,dt \quad (5)$$

4 VESPER

Vesper is a tool that allows a web developer to determine the RI and RT for a specific page. Vesper must satisfy three design goals. First, Vesper must produce *high coverage*, i.e., Vesper must identify all of a page's interactive, above-the-fold state. Second, Vesper's instrumentation must have *minimal overhead*, such that instrumented pages have RI and RT scores that are close to those of unmodified pages. Ideally, Vesper would also be *browser-agnostic*, i.e., capable of measuring a page's RI and RT without requiring changes to the underlying browser.

These design goals are in tension. To make Vesper browser-agnostic, Vesper should be implemented by rewriting a page's JavaScript code and HTML files, not through modification of a browser's JavaScript engine and rendering pipeline; unfortunately, the most direct way to track interactive state is via heavyweight instrumentation of all reads and writes that a page makes to the JavaScript heap, the DOM, and the rendering bitmap. Vesper resolves the design tension by splitting instrumentation and log analysis across two separate page loads. Each load uses a differently-rewritten version of a page, with the first version using heavyweight instrumentation, and the second version using lightweight instrumentation. As a result, the second page load injects minimal timing distortion into the page's true RI and RT scores. Figure 4 provides an overview of Vesper's two-phase workflow. We provide more details in the remainder of this section.

[3]The use of equal weights reflects our assumption that functionality and visibility are equally important. However, future empirical research may suggest better weighting schemes.

4.1 Phase 1

The goal of this phase is to identify the subset of DOM nodes and JavaScript state that support above-the-fold interactivity.

Element Visibility: For most pages, only a subset of all DOM nodes will have bounding boxes that overlap with the initial viewport. Even if a node is above-the-fold, it may not be visible, e.g., due to CSS styling which hides the node. Vesper injects a JavaScript timer into the page which runs at time T. When the timer function executes, it traverses the DOM tree and records which nodes are visible. In the rest of the section, we refer to this timer as the Vesper timer.

Event Handlers: Developers make a DOM element interactive by attaching one or more event handlers to that element. For example, a `<button>` element does nothing in response to clicks until JavaScript code registers `onclick` handlers for the element. To detect when such handlers are added, Vesper shims the event registration interfaces [22]. There are two types of registration mechanisms:

- DOM elements define JavaScript-accessible properties and methods that support event handler registration. For example, assigning a function `f` to a property like `DOMnode.onclick` will make `f` an event handler for clicks on that DOM node. Invoking `DOMnode.addEventListener("click", f)` has similar semantics. Vesper interposes on registration mechanisms by injecting new JavaScript into a page that modifies the DOM prototypes [22]; the modified prototypes insert logging code into the registration interfaces, such that each registered handler is added to a Vesper-maintained, in-memory list of the page's handlers.
- Event handlers can also be defined via HTML, e.g., `<img src=... onload=handler()/>`. At T, the Vesper timer iterates through the page's DOM tree, identifying event handlers that were not registered via a JavaScript-level interface, and adding those handlers to Vesper's list.

The Vesper timer only adds a handler if the handler is attached to a visible DOM element that resides within the initial viewport.

Event Handler State: When a handler fires, it issues reads and writes to program state. That state may belong to JavaScript variables, or to DOM state like the contents of a `<b>` tag. As the handler executes, it may invoke other functions, each of which may touch an additional set of state. The aggregate set of state that the call chain may touch is the *functional state* for the handler. Given a DOM element e, we define e's functional state as the

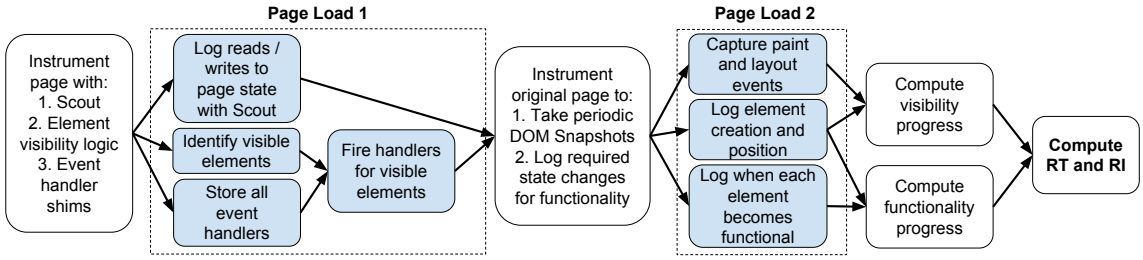


Figure 4: Vesper's two-phase approach for measuring RI and RT. Shaded boxes indicate steps that occur during a page load. Clear boxes represent pre- and post-processing steps.

union of the functional state that belongs to each of *e*'s event handlers.

If *e* resides within the initial viewport, then *e* is not functional until two conditions have been satisfied:

1. all of *e*'s event handlers must be registered, and
2. all of *e*'s functional state must be loaded.

At any given moment during the page load, none, either, or both of these conditions may be satisfied. For example, if *e*'s event handlers are defined in a `<script>` tag, but key functional state is defined by downstream HTML or `<script>` tags, then after evaluation of the first `<script>` tag, condition (1) is true, but condition (2) is not.

To identify a page's functional state, Vesper instruments the HTML and JavaScript in a page, such that, when the instrumented page loads, the page will log all reads and writes to JavaScript variables and DOM state. When the Vesper timer runs, it actively invokes the event handlers that were captured by event registration shimming. As those handlers fire, their call chains touch functional state. By post-processing the page's logs, and looking for reads and writes that occurred after the Vesper timer began execution, Vesper can identify a page's functional state. In particular, Vesper can associate each handler with its functional state, and each DOM element with the union of the functional states of its handlers.

To fire the handlers for a specific event type like `click`, the Vesper timer determines the minimally-sized DOM subtree that contains all handlers for the `click` event. Vesper then constructs a synthetic `click` event, and invokes the built-in `DOMnode.dispatchEvent()` method for each leaf of the subtree. This approach ensures that synthetic events follow the same dispatch path used by real events.

Some event types are logically related to a single, high-level user interaction. For example, when a user clicks a mouse button, her browser generates `mousedown`, `click`, and `mouseup` events, in that order. Vesper is aware of these semantic relationships, and uses them to guide the generation of synthetic events, ensuring a realistic sequence of handler firings.

Implementation: To instrument a page, Vesper could modify the browser's renderer and JavaScript engine to track reads and writes to DOM objects and JavaScript variables. However, our Vesper prototype leverages Scout [25] instead. Scout is a browser-agnostic rewriting framework that instruments a page's JavaScript and HTML to log reads and writes. A browser-agnostic approach is useful because it allows Vesper to compare a page's Ready Index across different browser types (§5.4).

The instrumentation that tracks element visibility and handler registration adds negligible overhead to the page load process. However, tracking all reads and writes to page state is more costly. Across the 350 pages in our test corpus, we measured a Scout-induced load time increase of 4.5% at the median, and 7.6% at the 95th percentile. Thus, trying to calculate RI and RT directly in Phase 1 would lead to inflated estimates. To avoid this problem, we use the outputs of Phase 1 as the inputs to a second phase of instrumentation. This second phase is more lightweight, and directly calculates RI and RT.

4.2 Phase 2

In Phase 1, Vesper discovers the DOM nodes and JavaScript variables that support above-the-fold interactivity. In Phase 2, Vesper tracks the rendering progress of the above-the-fold DOM elements that were identified in Phase 1. Vesper also tracks the rate at which functional JavaScript state is created. This information is sufficient to derive RI and RT.

4.2.1 Measuring Functionality Progress

A DOM element becomes functional when all of its event handlers have been registered, and all of the functional state for those handlers has been created. An element's functional state may span both the JavaScript heap and the DOM. Vesper uses different techniques to detect when the two types of state become ready.

JavaScript state: By analyzing Scout logs from Phase 1, Vesper can determine when the last write to each JavaScript variable occurs. The "last write" is defined as a source code line and an execution count for that line. The execution count represents the fact that a source code

line can be run multiple times, e.g., if it resides within a loop body.

At the beginning of Phase 2, Vesper rewrites a page's original JavaScript code, injecting a logging statement after each source code line that generates a final write to functional JavaScript state. The logging statement updates the execution count for the line, and only outputs a log entry if the final write has been generated.

DOM state: An event handler's functional state may also contain DOM nodes. For example, a `keypress` handler may assume the existence of a specific DOM node whose properties will be modified by the handler. At the beginning of Phase 2, Vesper rewrites a page's original HTML to output the creation time for each DOM node. The rewriting is complicated by the fact that, when a browser parses HTML, it does not trigger a synchronous, JavaScript-visible event upon the creation of a DOM node. Thus, Vesper rewrites a page's HTML to include a new `<script>` tag after *every* original HTML tag. The new `<script>` tag logs two things: the creation of the preceding DOM node, and the bounding boxes of all DOM nodes which exist at that moment in the HTML parse. The `<script>` tag then removes itself from the DOM tree (so that at any point in the HTML parse, non-Vesper code that inspects the DOM tree will see the original DOM tree which does not contain Vesper's self-destructing tags). *DOM snapshots* using self-destructing JavaScript tags are by far the most expensive part of the Phase 2 instrumentation; however, they only increase page load times by 1.9% at the median, and 3.9% at the 95th percentile. Thus, we believe that the overhead is acceptable.

After the initial HTML parse, DOM nodes may be created by asynchronous event handlers. Vesper logs such creations by interposing on DOM methods like `DOMnode.appendChild()`. This interpositioning has negligible overhead and ensures that Vesper has DOM snapshots after the initial HTML parse.

4.2.2 Measuring Visibility Progress

DOM snapshots allow Vesper to detect when elements are created. However, a newly-created element will not become *visible* until some point in the future, because the construction of the DOM tree is earlier in the rendering pipeline than the paint engine. Browsers do not expose layout or paint events to JavaScript code. Fortunately, Vesper can extract those events from the browser's debugging output [11]. Each layout or paint message contains the bounding box and timestamp for the activity. Unfortunately, the message does not identify which DOM nodes were affected by the paint; thus, Vesper must derive the identities of those nodes.

After the Phase 2 page load is complete, Vesper collates the DOM snapshots and the layout+paint debugging events, using the following algorithm to determine the layout and paint events that rendered a specific DOM element e:

1. Vesper finds the first DOM snapshot that contains a bounding box for e. Let that snapshot have a timestamp of t_d. Vesper searches for the layout event that immediately precedes t_d and has a bounding box that contains e's bounding box. Vesper defines that layout event L_{first} to be the one which added e to the layout tree.
2. Vesper then rolls forward through the log of paint and layout events, starting at L_{first}, and tracking all paint events to e's bounding box. That bounding box may change during the page load process, but any changes will be captured in the page's DOM snapshots. Thus, Vesper can determine the appropriate bounding box for e at any given time.

As described in Equation 2, each paint event contributes equally towards e's visibility score. For example, if e is updated by four different paints, then e is 25% visible after the first one, 50% visible after the second one, and so on.

In summary, the output of the Phase 2 page load is a trace of a page's functionality progress and visibility progress. Using that trace, and Equations 4 and 5, Vesper determines the page's RT and RI. Note that, for a given version of a page (i.e., for a particular set of HTML, CSS, and JavaScript files), Phase 1 only needs to run once, on the server-side, with Phase 2 running during the live page loads on clients in the wild.

4.3 Discussion

The PLT metric is natively supported by commodity browsers, meaning that a page can measure its own PLT simply by registering a handler for the `onload` event. Newer metrics that lack native browser support require 1) browsers to install a special plugin (the SI approach [10]), or 2) page developers to rewrite content (the approach used by our Vesper prototype). Vesper is amenable to implementation via plugins or native support; either option would enable lower instrumentation overhead, possibly allowing Vesper to collapse its two phases into one.

As a practical concern, a rewriting-based implementation of Vesper must deal with the fact that a single page often links to objects from multiple origins. For example, a developer for `foo.com` will lack control over the bytes in linked objects from `bar.com`. As described in Section 5, our Vesper prototype uses Mahimahi [26], a web replay tool, to record *all* of the content in a page; Vesper rewrites the recorded content, and then replays the modified content to a browser that runs on a machine controlled by the `foo.com` developer. In this manner, as with the browser plugin approach, a developer can mea-

sure RI and RT for any page, regardless of whether the developer owns all, some, or none of the page content.

All load metrics are sensitive to nondeterministic page behavior. In the context of Vesper, such behavior may result in a page having different interactive state across different page loads. For example, an event handler that branches on the return value of `Math.random()` might access five different DOM nodes across five different loads of the page. Even if a page's state is deterministic, Vesper's synthetic event generation (§4.1) is not guaranteed to exhaustively explore all *possible* event handler interleavings—instead, Vesper tests the *most likely* event sequences based on how a realistic human user would generate GUI events. Vesper could use symbolic execution [7] to increase path coverage, but we believe that Vesper's current level of coverage is sufficiently high for two reasons. First, from the empirical perspective, the pages in our large test corpus do not exhibit nondeterminism that results in different functional state across different loads. Second, the Vesper timer does not fire synthetic events until a page is fully loaded; thus, "unexpected" event-level race conditions arising from partially-loaded content [30] should not arise.

5 Evaluation

In this section, we compare RI and RT to three prior metrics for page load time (PLT, AFT, and Speed Index). We do not evaluate Google's TTI because the metric's definition is still evolving.

Across a variety of network conditions, we find that PLT overestimates the time that a page requires to become interactive; in contrast, AFT and Speed Index underestimate the time-to-interactivity (§5.2 and A.1.1). These biases persist when browser caches are warm (§A.1.2). Furthermore, the discrepancies between prior metrics and our interactive metrics are large, with median and 95th percentile load time estimates often differing by multiple *seconds* (Figures 3 and 6). Thus, Ready Index and Ready Time provide a fundamentally new way of understanding how pages load.

5.1 Methodology

We evaluated the various load metrics using a test corpus of 350 pages. The pages were selected from the Alexa US Top 500 list [2]. We filtered out sites using deprecated JavaScript statements that Scout [25] does not rewrite. We also filtered sites that caused errors with Speedline [19], a preexisting tool for capturing SI.

To measure PLT, we recorded the time between the JavaScript `navigationStart` and `onload` events (§2). RT and RI were measured with Vesper; we set T to 30 seconds. We also used Vesper to measure AFT and SI.[4] Calibration experiments showed that Vesper's estimates of SI were within 2.1% of Speedline's estimates at the median, and within 3.9% at the 95th percentile.

Measuring PLT is non-invasive, since unmodified pages will naturally fire the `navigationStart` and `onload` events. Capturing the other metrics requires new instrumentation, like DOM snapshots (§4.2.1). To avoid measurement biases due to varying instrumentation overheads, each experimental trial loaded each page five times, and in each of the five loads, we enabled all of Vesper's Phase 2 instrumentation, such that each load metric could be calculated. Enabling all of the instrumentation increased PLT by 1.9% at the median, and 3.9% at the 95th percentile.

We used Mahimahi [26] to record the content in each test page, and later replay the content via emulated network links. With the exception of the mobile experiments (§A.1.1), all experiments were performed on Amazon EC2 instances running Ubuntu 14.04. Unless otherwise specified, each page load used Google Chrome (v53) with a cold browser cache and remote debugging enabled so that we could track layout and paint events.

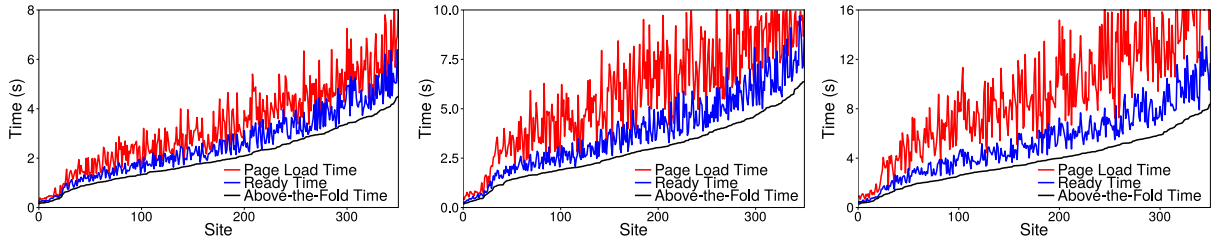
5.2 Cross-metric Comparisons

On computationally-powerful devices like desktops and laptops, network latency (not bandwidth) is the primary determinant of how quickly a page loads [1, 5, 25, 34]. So, our first set of tests used a t2.large EC2 VM with a fixed bandwidth of 12 Mbit/s, but a minimum round-trip latency that was drawn from the set {25 ms, 50 ms, 100 ms, 200 ms}. These emulated network conditions were enforced by the Mahimahi web replay tool.

Figure 3 summarizes the results for PLT, RT, and AFT. Recall that these metrics are non-progressive, i.e., they express a page's load time as a single number that represents when the browser has "completely" loaded the page (for some definition of "completely"). As expected, PLT is higher than RT because PLT requires all page state, including below-the-fold state, to be loaded before a page load is finished. Also as expected, AFT is lower than RT, because AFT ignores the load status of JavaScript code that is necessary to make visible elements functional.

The surprising aspect of the results is that the differences between the metrics are so noticeable. As shown in Figures 2(a) and 3, the differences are large in terms of percentage (24.0%–64.3%); more importantly, the differences are large in terms of absolute magnitude, equating to hundreds or thousands of milliseconds. For example, with a round-trip latency of 50 ms, RT and PLT differ by roughly 900 ms at the median, and by 1.4 seconds at the 95th percentile. For the same round-trip latency, RT and

[4] To compute SI, Vesper only considers element visibility, assigning zero weight to functionality.



(a) 50 ms RTT (b) 100 ms RTT (c) 200 ms RTT

Figure 5: Comparing RT, PLT, and AFT. Results used emulated links with a bandwidth of 12 Mbit/s.

RTT	Ready Index	Speed Index
25 ms	714 (1522)	568 (1027)
50 ms	1759 (3846)	1325 (3183)
100 ms	2737 (6174)	2054 (4549)
200 ms	4252 (9719)	3071 (6913)

Figure 6: Median (95th percentile) load time estimates (see Section 2 for a discussion of the units). Results used our entire 350 page corpus. Content was loaded over a 12 Mbit/s link.

AFT differ by approximately 600 ms at the median, and by 1.1 seconds at the 95th percentile.

The discrepancies increase as RTTs increase. This observation is important, because cellular and residential networks often have RTTs that exceed 100 ms [3, 18]. For example, in our emulated network with an RTT of 100 ms, RT differed from PLT by *2.2 seconds* at the median; RT differed from AFT by 1 second at the median. From the perspective of a web developer, the differences between RT and AFT are particularly important. Users frequently assume that a visible element is also functional. However, visibility does not necessarily imply functionality, and interactions with partially-functional elements can lead to race conditions and broken page behavior [30]. In Section 6, we describe how developers can create incrementally-interactive pages that minimize the window in which a visual element is not interactive.

Figure 5 compares the RT, PLT, and AFT values for each page in our 350 site corpus. Pages are sorted along the x-axis in ascending AFT order. Figure 5 vividly demonstrates that PLT is an overly conservative definition for user-perceived notions of page readiness. The spikiness of the RT line also demonstrates that pages with similar AFT values often have very different RT scores. For example, consider an emulated link with a 100 ms round-trip time. Sites 200 (`mashable.com`) and 201 (`overdrive.com`) have AFT values of 3099 ms and 3129 ms, respectively. However, the sites have RT values of 4418 ms and 3970 ms, a difference of over 400 ms. In Section 5.3, we explain how the relationships between a page's HTML, CSS, and JavaScript cause divergences in RT and AFT.

Figures 6 and 7 compare the two progressive metrics. The results mirror those for the non-progressive metrics. A page's SI is lower than its RI, because SI does not consider the load status of JavaScript code that supports interactivity. Furthermore, pages with similar SIs often have much different RIs.

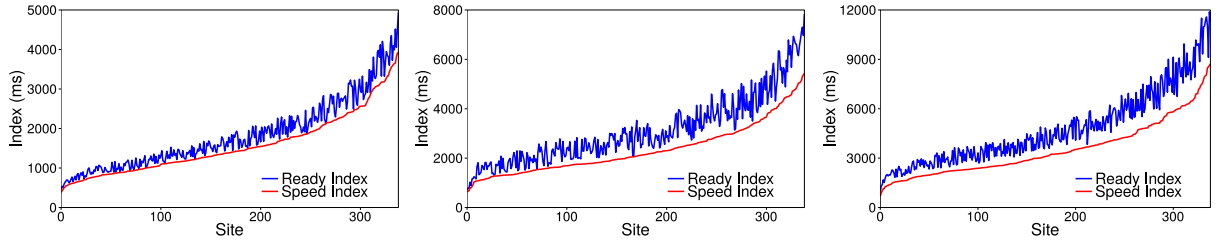
5.3 Case Studies

Figure 8 uses two randomly-selected pages to demonstrate how interactivity evolves. Figure 8(a) describes the homepage for Bank of America, whereas Figure 8(b) describes the homepage for WebMD. Using the terminology from Section 3, each graph plots the visual progression of the page ($\sum_{e \in E} V(e,t)A(e)$) and the readiness progression of the page ($R(t)$); in the graphs, each data point is normalized to the range [0.0,1.0]. At any given moment, a page's readiness progression is less than or equal to its visual progression, since visual progression does not consider the status of functional state.

The gaps between the red and blue curves indicate the existence of visible, interactive DOM elements that are not yet functional. If users try to interact with such elements, then at best, nothing will happen; at worst, an incomplete set of event handlers will interact with incomplete JavaScript and DOM state, leading to erroneous page behavior. For example, the Bank of America site contains a text input that supports autocompletion. With RTTs of 100 ms and above, we encountered scenarios in which the input was visible but not functional. In these situations, we manually verified that a human user could type into the text box, have no autosuggestions appear, and then experience the text disappear and reappear with autosuggestions as the page load completed.

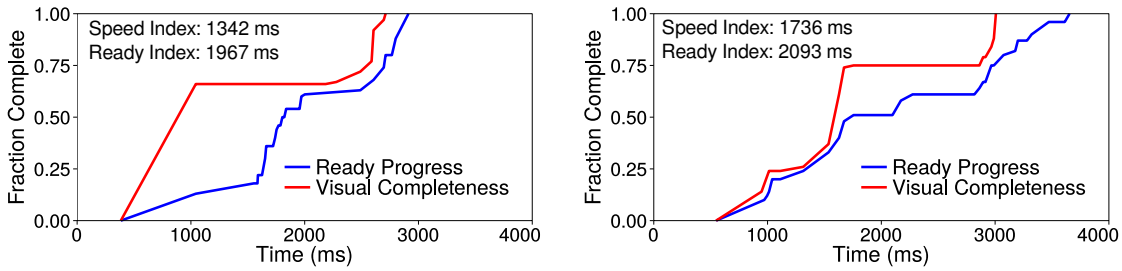
Both the red and blue curves contain stalls, i.e., time periods in which no progress is made. For example, both pages exhibit a lengthy stall in their visual progression—for roughly a second, neither page updates the screen. Both pages also contain stretches that lack visual progress or readiness progress. During these windows, a page is not executing any JavaScript code that creates interactive state.

Functionality progression stalls when the `<script>` tags supporting functionality have not been fetched, or have been fetched but not evaluated. Visual progression may stall for a variety of reasons. For example, the browser might be blocked on network fetches, waiting on



(a) 50 ms RTT (b) 100 ms RTT (c) 200 ms RTT

Figure 7: Comparing the progressive metrics (Ready Index versus Speed Index). Results used emulated links with a bandwidth of 12 Mbit/s.



(a) https://www.bankofamerica.com (b) http://www.webmd.com

Figure 8: Exploring how visibility and functionality evolve for two different pages. The client had a 12 Mbit/s link with an RTT of 100 ms. Remember that a progressive metric like Ready Index is calculated by examining the area that is *above* a curve.

HTML data so that new tags can be parsed and rendered. Browsers also use a single thread for HTML parsing, DOM node rendering, and JavaScript execution; thus, executing a `<script>` tag blocks parsing and rendering of downstream HTML. As described in Section 6, developers can use automated tools to minimize these stalls and improve a page's Ready Time and Ready Index.

5.4 Other Page Load Scenarios

In Section A.1, we analyze how Ready Index evolves in three additional scenarios: mobile page loads, page loads that use a warm browser cache, and page loads on two different browsers (namely, Chrome versus Opera). Due to space restrictions, we merely provide a summary here:

Mobile page loads: Mobile page loads exhibit the same trends that we observed on more powerful client devices. For example, on a Nexus 5 phone running on an emulated Verizon LTE cellular link, the median PLT is 35.2% larger than the median RT; the median RI is 29.7% larger than the median Speed Index.

Warm cache loads: The results from earlier in this section used cold caches. However, clients sometimes have a warm cache for objects in a page to load. As expected, pages load faster (for all metrics) when caches are warm. However, the general trends from Section 5.2 still hold. For example, on a desktop browser with a 12 Mbit/s, 100 ms RTT link, the median warm-cache PLT is 38.2% larger than the median RT. The median RT is 26.0% larger than the median AFT.

Chrome vs. Opera: Since our Vesper implementation is browser-agnostic, it can measure a single page's load metrics across different browser types. For example, we compared RI on Chrome and Opera. With cold browser caches and a 12 Mbit/s, 100 ms RTT link, Chrome's RI values were 6.5% lower at the median, and 11.9% lower at the 95th percentile. Since Vesper's logs contain low-level information about reads and writes to interactive state, browser vendors can use these logs to help optimize the internal browser code that handles page loading.

6 Optimizing for Interactivity

To minimize a page's Ready Time and Ready Index, browsers must fetch and evaluate objects in a way that prioritizes interactivity. In particular, a browser should:

1. maximize utilization of the client's network connection;
2. prioritize the fetching and evaluating of HTML files that define above-the-fold DOM elements;
3. prioritize the fetching and evaluating of `<script>` tags that generate interactive, above-the-fold state; and
4. respect the semantic dependencies between a page's objects.

By maximizing network utilization (Goal (1)), a browser minimizes the number of CPU stalls that occur due to synchronous network fetches; ideally, a browser would

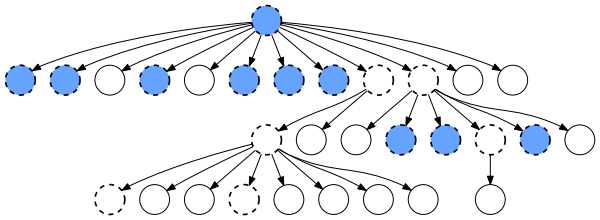


Figure 9: The dependency graph for `priceline.com`. OPT-PLT assigns equal weights to all nodes. OPT-SI prioritizes the shaded objects. OPT-RI prioritizes the objects with dashed outlines.

fetch each piece of content *before* that content is desired by a parsing/evaluation engine. Goals (2) and (3) directly follow from the definitions for page readiness in Section 3. However, Goal (4) is in tension with the others: fetching and evaluating objects in a way that satisfies Goals (1), (2), and (3) may break page functionality. For example, two JavaScript libraries may have shared state, like a variable that is written by the first library and read by the second. Invalid reads and other problems will arise if a browser evaluates the two libraries "out-of-order" with respect to the lexical order of their `<script>` tags in the page's HTML.

Web pages contain a variety of additional dependencies that constrain the order in which objects can be fetched and evaluated. Polaris [25] is a load optimizer that uses Scout to extract all of these dependencies and generate an explicit dependency graph (i.e., a partial ordering that specifies how certain objects must be loaded before others). Polaris then rewrites the page so that the page is *self-assembling*. The rewritten page uses a custom JavaScript library to schedule the fetching and evaluating of objects in a way that satisfies Goals (1) and (4).

At any given moment in a page load, the *dynamic critical path* is the path in the dependency graph that has the largest number of unfetched objects. The default Polaris scheduler prioritizes the fetching of objects along the dynamic critical path. This policy minimizes PLT, but may increase or decrease RT, depending on whether interactive, above-the-fold state is created by objects along the dynamic critical path.

We created a new scheduling policy, called OPT-RI ("optimize RI"), which prioritizes the loading of interactive content. Let $O_{interactive}$ be the objects (e.g., HTML files, JavaScript files) that Vesper identifies as generating interactive, above-the-fold state. Given $O_{interactive}$ and the dependency graph from Scout, OPT-RI assigns node weights of zero to nodes that do not reside in $O_{interactive}$; for a node in $O_{interactive}$, OPT-RI finds all of the above-the-fold elements that the node affects, and then weights the node by the fraction of the initial viewport area that those elements cover. During the actual page load, the OPT-RI scheduler prioritizes objects along the *weighted* dynamic critical path.

We also defined OPT-SI, which only considers visual progress. Nodes that do not lead to the creation of visible, above-the-fold DOM elements receive a weight of zero. For each remaining node, OPT-SI finds the DOM elements that the node influences, and assigns a node weight that is proportional to the fraction of the viewport that the elements cover. OPT-SI will not prioritize JavaScript files that only define event handler state; however, OPT-SI will prioritize JavaScript files that dynamically create above-the-fold content via DOM methods like `document.appendChild()`. Figure 9 provides an example of a real dependency graph, and the nodes that are prioritized by the various schedulers.

Figure 10 compares the performance of the schedulers. OPT-RI and OPT-SI reduce all load metrics, *but the targeted metrics decrease the most*. Thus, sites that want to decrease time-to-interactivity must explicitly target RI and RT, not preexisting metrics like SI and PLT. For example, consider the search button in Figures 2(b) and 2(c). OPT-RI makes the button interactive 1.5 seconds earlier than OPT-SI. Differences of that magnitude have significant impacts on user satisfaction and site revenue [6, 8, 41].

As shown in Figure 10, OPT-RI reduces RI by a median of 29%, and RT by a median of 32%; PLT, AFT, and SI also drop, but not as much (by 23%, 15%, and 12%, respectively). Interestingly, the default Polaris scheduler (OPT-PLT) improves PLT, RT, and RI, but actually *hurts* AFT and SI by -4% and -7% at the median. The reason is that JavaScript files often form long dependency chains; evaluating one JavaScript file in the chain leads to the fetching and evaluation of additional JavaScript files. These long dependency chains tend to lie along the dynamic critical paths that are preferentially explored by OPT-PLT. By focusing on those chains, OPT-PLT increases the speed at which event handling state is loaded. However, this approach defers the loading of content in short chains. Short chains often contain images, since images (unlike HTML, CSS, and JavaScript) cannot trigger new object fetches. Deferring image loading hurts AFT and SI, though RT and RI improve, and the likelihood of broken user interactions (§5.2 and §5.3) decreases.

User Study 1: Do User-perceived Rendering Times Actually Change? The results from Figure 10 programmatically compare OPT-PLT, OPT-SI, and OPT-RI. We now evaluate how the differences between these optimization strategies are perceived by real users. We performed a user study in which 73 people judged the load times of 15 randomly-selected sites from our corpus, each of which had three versions (one for each optimization strategy). We used a standard methodology for evaluating user-perceived load times [20, 35]. We pre-

Weights	PLT	RT	AFT	SI	RI
OPT-PLT	**36% (51%)**	13% (22%)	-4% (5%)	-7% (4%)	8% (17%)
OPT-RI	23% (34%)	**32% (48%)**	15% (26%)	12% (20%)	**29% (35%)**
OPT-SI	10% (19%)	18% (31%)	**27% (39%)**	**18% (28%)**	14% (23%)

Figure 10: Median (95th percentile) load time improvements using our custom Polaris schedulers and the default one (OPT-PLT). Results used our entire 350-page corpus. Loads were performed on a desktop Chrome browser that had a 12 Mbit/s link with an RTT of 100 ms; the performance baseline was a regular (i.e., non-Polaris) page load. The best scheduler for each load metric is highlighted.

sented each user with 10 randomly-selected pages that employed a randomly-selected optimization target; we injected a JavaScript `keypress` handler into each page, so that users could press a key to log the time when they believed the page to be fully loaded. In all of the user studies, content was served from Mahimahi on a Macbook Pro, using an emulated 12 Mbit/s link with a 100 ms RTT.

Unsurprisingly, users believed that OPT-PLT resulted in the slowest loads for all 15 pages. However, OPT-SI did not categorically produce the lowest user-perceived rendering times; users thought that OPT-RI was the fastest for 4 pages, and OPT-SI was the fastest for 11. Across the study, median (95th percentile) user-perceived rendering times with OPT-RI were within 4.7% (10.9%) of those with OPT-SI. Furthermore, the performance of OPT-RI and OPT-SI were closer to each other than to that of OPT-PLT. At the median (95th percentile), OPT-RI was 14.3% (25.3%) faster than OPT-PLT, whereas OPT-SI was 17.4% (32.9%) faster.

These results indicate that a page that only wants to decrease rendering delays should optimize for SI. However, optimizing for RI results in comparable decreases in rendering time. Our next user study shows that optimizing for RI also decreases user-perceived time-to-interactivity.

User Study 2: Does OPT-RI Help Interactive Sites? Unlike the first user study, our second one asked users to interact with five well-known landing pages: Amazon, Macy's, Food Network, Zillow, and Walmart. For each site, users completed a site-specific task that normal users would be likely to perform. For example, on the Macy's page, users were asked to hover over the "shopping bag" icon until the page displayed a pop-up icon that listed the items in the shopping bag. On the Walmart site, users were asked to search for "towels" using the autocompleting text input at the top of the page; they then had to select the autocompleted suggestion. To avoid orientation delays, users were shown all five pages and the location of the relevant interactive elements at the beginning of the study. This setup emulated users who were returning to frequently-visited sites.

The study had 85 users interact with three different versions of each page: a default page load, a load that was optimized with OPT-SI, and one that was optimized with OPT-RI. For each page, users were presented with the three variations in a random order and were unaware of which variant they were seeing. Users were asked to select the variant that enabled them to complete the given task the fastest; if users felt that there was no perceivable difference between the loads, users could report "none."

Load method	Preference %
OPT-RI	83%
OPT-SI	4%
Default load	7%
None	6%

Figure 11: The results of our second user study. OPT-RI leads to human-perceivable reductions in the completion times for interactive tasks.

As shown in Figure 11, OPT-RI was overwhelmingly preferred, with 83% of users believing that OPT-RI led to the fastest time-to-interactivity. For example, on the Macy's page, OPT-RI made the shopping bag icon fully interactive *1.6 seconds faster* than the default page load, and *2.1 seconds faster* than the OPT-SI load. Time-to-interactivity differences of these magnitudes are easily perceived by humans. Thus, for pages with interactive, high-priority content, OPT-RI is a valuable tool for reducing time-to-interactivity (as well as the time needed to fully render the page). Optimizing for interactivity is particularly important for web browsing atop mobile devices with poor network connectivity. In these scenarios, users often desire to interact with pages as soon as relevant content becomes visible [17].

7 Conclusion

A web page is not usable until its above-the-fold content is both visible and functional. In this paper, we define Ready Index, the first load time metric that explicitly quantifies page interactivity. We introduce a new tool, called Vesper, that automates the measurement of Ready Index. Using a corpus of 350 pages, we show that Ready Index captures interactivity better than prior metrics like PLT and SI. We also present an automated page-rewriting framework that uses Vesper to optimize a page for Ready Index or pure rendering speed. User studies show that pages which optimize for Ready Index support more immediate user interactions with less user frustration.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd Jon Howell for their helpful feedback. We also thank Tim Dresser at Google for his insights about how Google's TTI metric is currently defined. This research was partially supported by NSF grant 1407470 and by a Google Research Award.

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] M. Allman. Comments on Bufferbloat. *SIGCOMM Comput. Commun. Rev.*, 43(1):30–37, January 2012.

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

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

[6] J. Brutlag. Speed Matters. Google Research Blog. https://research.googleblog.com/2009/06/speed-matters.html, June 23, 2009.

[7] 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.

[8] K. Eaton. How One Second Could Cost Amazon $1.6 Billion In Sales. https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales, March 15, 2012.

[9] M. J. Freedman. Experiences with CoralCDN: A Five-year Operational View. In *Proceedings of NSDI*, 2010.

[10] Google. Perceptual Speed Index. https://developers.google.com/web/tools/lighthouse/audits/speed-index, December 14, 2017.

[11] Google. Chrome Debugging Protocol. https://chromedevtools.github.io/devtools-protocol/, 2018.

[12] Google. Reduce the size of the above-the-fold content. PageSpeed Tools Documentation. https://developers.google.com/speed/docs/insights/PrioritizeVisibleContent, January 9, 2018.

[13] Google. Remove Render-Blocking JavaScript. PageSpeed Tools Documentation. https://developers.google.com/speed/docs/insights/BlockingJS, January 25, 2018.

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

[15] Google. Time to Interactive (TTI). https://github.com/WPO-Foundation/webpagetest/blob/master/docs/Metrics/TimeToInteractive.md, January 12, 2018.

[16] Google. WebPagetest: Website Performance and Optimization Test. https://www.webpagetest.org/, 2018.

[17] B. Greenstein. Delivering the Mobile Web to the Next Billion Users. Keynote speech: Workshop on Mobile Computing Systems and Applications (HotMobile), 2018.

[18] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. A Close Examination of Performance and Power Characteristics of 4G LTE Networks. In *Proceedings of MobiSys*, 2012.

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

[20] C. Kelton, J. Ryoo, A. Balasubramanian, and S. Das. Improving User Perceived Page Load Times Using Gaze. In *Proceedings of NSDI*, 2017.

[21] P. Meenan. How Fast is Your Web Site? *ACM Queue*, 11(2), March 2013.

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

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

[24] Mozilla Developer Network. HTTP Caching FAQ. https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching_FAQ, January 4, 2018.

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

[26] 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.

[27] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai Network: A Platform for High-performance Internet Applications. *SIGOPS Oper. Syst. Rev.*, 44(3), August 2010.

[28] D. Oksnevad. Time to Interact: A New Metric for Measuring User Experience. https://blog.dotcom-monitor.com/web-performance-tech-tips/time-to-interact-new-metric-measuring-user-experience/, February 3, 2014.

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

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

[31] T. Poss. How Does Load Speed Affect Conversion Rate? Oracle: Modern Marketing Blog. https://blogs.oracle.com/marketingcloud/how-does-load-speed-affect-conversion-rate, January 14, 2016.

[32] T. Russo. Why Your Website Dev Team Should Care About Revenue. https://www.bluetriangletech.com/performance-insider/your-website-dev-team-should-care-about-revenue/, June 10, 2016.

[33] K. Sakamoto. Time to First Meaningful Paint. Chromium draft document. https://bit.ly/ttfmp-doc, June 6, 2016.

[34] 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.

[35] M. Varvello, J. Blackburn, D. Naylor, and K. Papagiannaki. EYEORG: A Platform For Crowdsourcing Web Quality Of Experience Measurements. In *Proceedings of CoNEXT*, 2016.

[36] L. Wang, K. S. Park, R. Pang, V. Pai, and L. Peterson. Reliability and Security in the CoDeeN Content Distribution Network. In *Proceedings of USENIX ATC*, 2004.

[37] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying Page Load Performance with WProf. In *Proceedings of NSDI*, 2013.

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

[39] K. Winstein, A. Sivaraman, and H. Balakrishnan. Stochastic Forecasts Achieve High Throughput and Low Delay over Cellular Networks. In *Proceedings of NSDI*, 2013.

[40] S. Work. How Loading Time Affects Your Bottom Line. Kissmetrics Blog. https://blog.kissmetrics.com/loading-time/, April 28, 2011.

[41] Z. Yang. Every Millisecond Counts. https://www.facebook.com/note.php?note_id=122869103919, August 28, 2009.

[42] W. Young. The Need For Speed: 7 Observations On The Impact Of Page Speed To The Future Of Local Mobile Search. Search Engine Land. http://searchengineland.com/need-speed-7-observations-impact-page-speed-future-local-mobile-search-243128, February 29, 2016.

A APPENDIX

A.1 Additional Evaluation Results

In this section, we elaborate on the experimental results from Section 5.4, discussing how Ready Index applies to mobile page loads (§A.1.1), warm cache page loads (§A.1.2), and loads on different browser types (§A.1.3).

A.1.1 Mobile Page Loads

Mobile browsers run on devices with limited computational resources. As a result, mobile page loads are typically compute-bound, with less sensitivity to network latency [5, 34]. To explore RI and RT on mobile devices, we USB-tethered a Nexus 5 phone running Android 5.1.1 to a Linux desktop machine that ran Mahimahi. Mahimahi emulated a Verizon LTE cellular link [39] with a 100 ms RTT. The phone used Google Chrome v53 to load pages from a test corpus. The corpus had the same 350 sites from our standard corpus, but used the mobile version of each site if such a version was available. Mobile sites are reformatted to fit within smaller screens, and to contain fewer bytes to avoid expensive fetches over cellular networks.

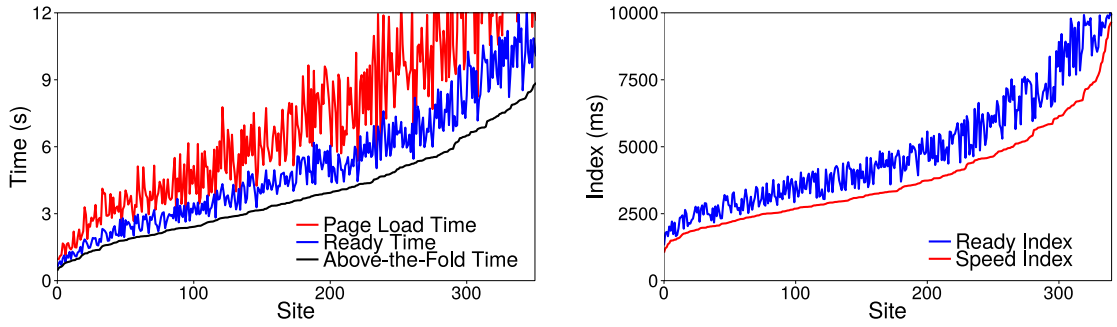
As shown in Figure 12, mobile page loads exhibit the same trends that we observed on more powerful client devices. For example, the median PLT is 35.2% larger than the median RT; the median RI is 29.7% larger than the median Speed Index. These differences persist even when considering only the mobile-optimized pages in our corpus. For that subset of pages, the median PLT is 27.4% larger than the median RT, and the median RI is 25.3% larger than the median Speed Index.

A.1.2 Browser Caching

Our prior experiments used cold browser caches, meaning that, to load a particular site, a browser had to fetch each of the constituent objects over the network. However, users often visit the same page multiple times; different sites also share objects. Thus, in practice, browsers often have warm caches that allow some object fetches to be satisfied locally.

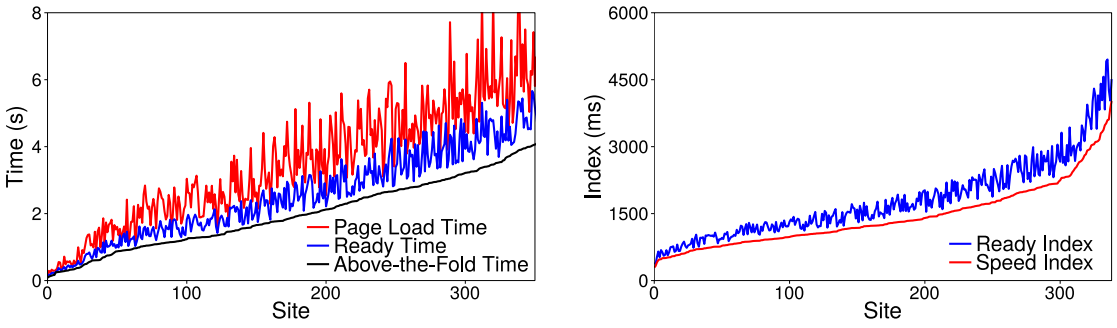
To determine how warm caches affect page loads, we examined the HTTP caching headers [24] for each object in our corpus. For each object that was marked as cacheable, we rewrote the headers to indicate that the object would be cacheable forever. We then loaded each page in our corpus twice, back to back; the first load populated the cache, and the second one leveraged the prewarmed cache. Figure 13 shows the results for a desktop browser which used a 12 Mbit/s link with a 100 ms RTT.

As expected, pages load faster when caches are warm. However, the general trends from Section 5.2 still hold. For example, the median PLT is 38.2% larger than the median RT, which is 26.0% larger than the median AFT. The correlations between various metrics also continue



(a) RT vs. PLT vs. AFT (b) RI vs. SI

Figure 12: Comparing the load metrics for mobile pages loaded on a Nexus 5 phone. The network used an emulated Verizon LTE link with a 100 ms RTT.



(a) RT vs. PLT vs. AFT (b) RI vs. SI

Figure 13: Page loads with warm browser caches. The desktop browser used a 12 Mbit/s link with a 100 ms RTT.

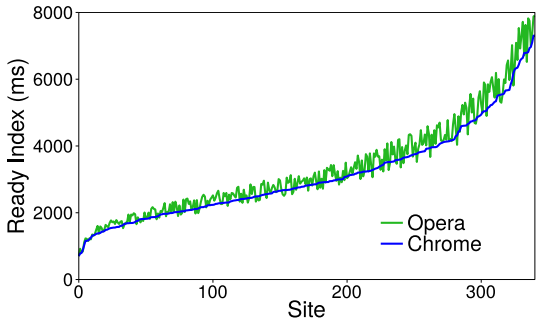


Figure 14: The Ready Index for each page in our corpus, as measured on Chrome and Opera. Pages are sorted on the x-axis by increasing Ready Index on Chrome. The results were collected using cold browser caches and a 12 Mbit/s link with an RTT of 100 ms.

to be noisy. For example, SI increases from 1147 ms to 1168 ms between sites 134 (`duckduckgo.com`) and 135 (`nexusmods.com`); however, RI decreases from 1601 ms to 1228 ms.

A.1.3 Cross-Browser Comparisons

Different browsers are built in different ways. As shown in Figure 14, those architectural variations impact page load times. Figure 14 compares Ready Index on Chrome v53 and Opera v42. Chrome and Opera both use the WebKit rendering engine and the V8 JavaScript runtime. However, the browsers' code is sufficiently different to produce noticeable biases in RI values: Chrome's RI values are 6.5% lower at the median, and 11.9% lower at the 95th percentile.

To understand the causes for such discrepancies, developers must analyze the steps that a browser takes to load a page. Tools like WProf [37] and the built-in Chrome debugger allow developers to examine coarse-grained interactions between high-level activities like HTML parsing, screen painting, and JavaScript execution. However, Vesper's logs describe how interactive state loads *at the granularity of individual JavaScript variables and DOM nodes.* For example, Vesper allows a developer to associate a dynamically-created text input with the specific code that creates the input and registers event handlers for the input; Vesper also tracks the JavaScript variables that are manipulated by the execution of the event handlers. None of this information is explicitly annotated by developers, nor should it be: for a large, frequently-changing site, humans should focus on the correct implementation of desired features, not the construction of low-level bookkeeping details about data and code dependencies. Thus, automatic extraction of these dependencies is crucial, since, as we demonstrate in Section 6, a fine-grained understanding of those dependencies is necessary to minimize a page's time-to-interactivity.