# Polaris: Faster Page Loads Using Fine-grained Dependency Tracking

Ravi Netravali*, Ameesh Goyal*, James Mickens†, Hari Balakrishnan*
*MIT CSAIL †Harvard University

## Abstract

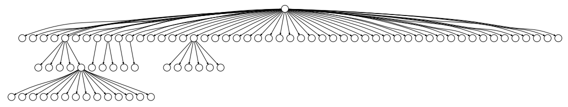
To load a web page, a browser must fetch and evaluate objects like HTML files and JavaScript source code. Evaluating an object can result in additional objects being fetched and evaluated. Thus, loading a web page requires a browser to resolve a *dependency graph*; this partial ordering constrains the sequence in which a browser can process individual objects. Unfortunately, many edges in a page's dependency graph are unobservable by today's browsers. To avoid violating these hidden dependencies, browsers make conservative assumptions about which objects to process next, leaving the network and CPU underutilized.

We provide two contributions. First, using a new measurement platform called Scout that tracks fine-grained data flows across the JavaScript heap and the DOM, we show that prior, coarse-grained dependency analyzers miss crucial edges: across a test corpus of 200 pages, prior approaches miss 30% of edges at the median, and 118% at the 95th percentile. Second, we quantify the benefits of exposing these new edges to web browsers. We introduce Polaris, a dynamic client-side scheduler that is written in JavaScript and runs on unmodified browsers; using a fully automatic compiler, servers can translate normal pages into ones that load themselves with Polaris. Polaris uses *fine-grained dependency graphs* to dynamically determine which objects to load, and when. Since Polaris' graphs have no missing edges, Polaris can aggressively fetch objects in a way that minimizes network round trips. Experiments in a variety of network conditions show that Polaris decreases page load times by 34% at the median, and 59% at the 95th percentile.

## 1 Introduction

Users demand that web pages load quickly. Extra delays of just a few milliseconds can result in users abandoning a page early; such early abandonment leads to millions of dollars in lost revenue for page owners [5, 6, 10]. A page's load time also influences how the page is ranked by search engines—faster pages receive higher ranks [12]. Thus, a variety of research projects [17, 23, 33, 34] and commercial systems [1, 21, 22, 31] have tried to reduce page load times.

To load a page, a browser must resolve the page's *dependency graph* [8, 18, 37]. The dependency graph captures "load-before" relationships between a page's HTML, CSS, JavaScript, and image objects. For example, a browser must parse the HTML `<script>` tag for a JavaScript file before that file can be fetched. Similarly, the browser must execute the JavaScript code in that file to reveal which images should be dynamically fetched via `XMLHttpRequests`. The overall load time for a page is the time that the browser needs to resolve the page's dependency graph, fetch the associated objects, and evaluate those objects (e.g., by rendering images or executing JavaScript files). Thus, fast page loads require efficient dependency resolution.


(a) The dependencies captured by traditional approaches.

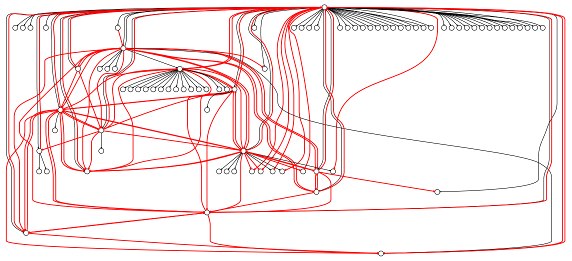
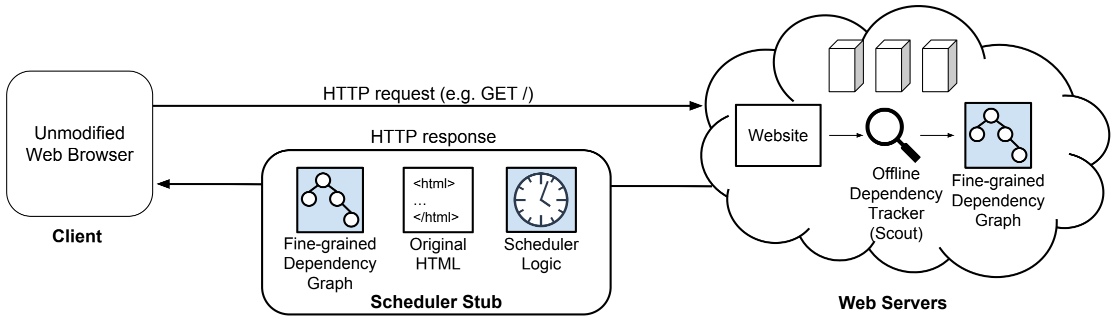

(b) The dependencies captured by Scout, which tracks fine-grained data flows. New edges are shown in red.

Figure 1: Dependency graphs for weather.com.

Unfortunately, a page's dependency graph is only partially revealed to a browser. As a result, browsers must use conservative algorithms to fetch and evaluate objects, to ensure that hidden load-before relationships are not violated. For example, consider the following snippet of HTML:

```
<script src=``http://x.com/first.js''/>
<script src=``http://y.com/second.js''/>
<img src=``http://z.com/photo.jpg''/>
```

When a browser parses this HTML and discovers the first `<script>` tag, the browser must halt the parsing and rendering of the page, since the evaluation of `first.js` may alter the downstream HTML [19]. Thus, the browser must *synchronously* fetch and evaluate `first.js`; this is true even if `first.js` does not modify the downstream HTML or define JavaScript state required by `second.js`. Synchronously loading JavaScript files guarantees correctness, but this approach is often too cautious. For example, if `first.js` and `second.js` do not modify mutually observable state, the browser should be free to download and evaluate the files in whatever order maximizes the utilization of the



HTTP request (e.g. GET /)
HTTP response
Unmodified Web Browser
Client
<html> … </html>
Fine-grained Dependency Graph
Original HTML
Scheduler Logic
Scheduler Stub
Website
Offline Dependency Tracker (Scout)
Fine-grained Dependency Graph
Web Servers

Figure 2: With Polaris, clients request web pages using standard HTTP requests. Servers return a page's HTML, as well as the Polaris scheduler (written in JavaScript) and the page's fine-grained dependency graph (generated offline by Scout). Polaris then determines the best order to fetch the external objects that are referenced by the HTML.

network and the CPU. However, pages do not expose such fine-grained dependency information to browsers. This forces browsers to make conservative assumptions about safe load orders by using coarse-grained relationships between HTML tags to guide object retrieval. As a result, pages load more slowly than necessary.

This paper makes two contributions. First, we introduce a new measurement infrastructure called Scout that automatically tracks *fine-grained data dependencies* in web pages. By rewriting JavaScript code and HTML files, Scout instruments web pages to track precise data flows between and within the JavaScript heap and the browser's internal HTML and CSS state. For example, Scout can track read/write dependencies for an individual JavaScript variable that is accessed by multiple JavaScript files. The resulting dependency graphs are more accurate than those of prior frameworks. As shown in Figure 1, our graphs also have dramatically different structures than those of previous approaches. In particular, for 81% of the 200 real-world pages that we examined, our new graphs have *different critical paths* than those of graphs from prior work (§3.5). The critical path defines the set of object evaluations which, if delayed, will always delay the end-to-end load time for a page. Thus, the fact that our new graphs look different is not just an academic observation: our graphs imply a faster way to load web pages.

Our second contribution is Polaris, a dynamic client-side scheduler which uses Scout's fine-grained dependency graphs to reduce page load times. Figure 2 provides an overview of how Polaris works. When a user makes a request for a Polaris-enabled page, the server returns a scheduler stub instead of the page's original HTML. The scheduler stub includes the Polaris JavaScript library, the page's fine-grained dependency graph (as generated by Scout), and the original HTML. The Polaris library uses the Scout graph, as well as dynamic observations about network conditions, to load objects in an order that reduces page load time.

As shown in Figure 1, our fine-grained data tracking *adds* new constraints to standard dependency graphs. However, and perhaps counterintuitively, the Polaris scheduler has *more* opportunities to reduce page load times. The reason is that, since Polaris has a priori knowledge of the true data dependencies in a page, Polaris can aggressively fetch and evaluate objects "out-of-order" with respect to lexical constraints between HTML tags. In contrast, prior scheduling frameworks lack knowledge of many dependencies, and are forced to make conservative assumptions that are derived from the lexical HTML relationships (§2.2). Those conservative assumptions guarantee the correctness of an assembled page in the face of hidden dependencies, but they often leave a browser's CPU and network connections underutilized. By using fine-grained dependency graphs, Polaris can ensure both correctness *and* high utilization of processors and network connections.

Because Polaris' scheduler is implemented in JavaScript, Polaris can reduce page load times on *unmodified commodity browsers*; this contrasts with load optimizers like Klotski [8], Amazon Silk [3], and Opera Mini [30], which require modified browsers to interact with a server-side component. Polaris is also complementary to previous load optimizers that use data compression (§6) or multiplex several HTTP requests atop a single TCP connection (§5.4).

We evaluated Polaris using 200 popular web pages and a variety of network conditions, with latencies ranging from 25 ms to 500 ms, and bandwidths ranging from 1 Mbit/s to 25 Mbits/s. Polaris reduced page load times by 34% at the median, and 59% for the 95th percentile sites.

## 2 Background

In a conventional page load, the browser first downloads the page's top-level HTML. For now, we assume that the HTML does not reference any JavaScript, CSS, or multimedia files. As the browser parses the HTML tags, it generates a data structure called the Document Object Model (DOM) tree. Each HTML tag has a corresponding node in the DOM tree; the overall structure of the DOM tree mirrors the hierarchical tag structure of the HTML. Once the HTML parse is finished and the DOM tree is complete, the browser constructs a render tree, which only contains the DOM nodes to be displayed. For example, a `<text>` node is renderable, but a `<head>` node is not. Each node in the render tree is tagged with visual attributes like background color, but render nodes do not possess on-screen positions or sizes. To calculate those geometric properties, the browser traverses the render tree and produces a layout tree, which determines the spatial location of all renderable tags. Finally, the browser traverses the layout tree and updates (or "paints") the screen. Modern browsers try to pipeline the construction of the various trees, in order to progressively display a page.

### 2.1 Loading More Complicated Pages

**JavaScript:** Using `<script>` tags, HTML can include JavaScript code. A script tag blocks the HTML parser, halting the construction of the DOM tree and the derivative data structures. Script tags block HTML parsing because JavaScript can use interfaces like `document.write()` to dynamically change the HTML after a `<script>` tag; thus, when the HTML parser encounters a `<script>` tag, the parser cannot know what the post-`<script>` HTML will look like until the JavaScript code in the tag has executed. As a result, script tags inject synchronous JavaScript execution delays into a page load. If a script tag does not contain inline source code, the browser also incurs network latencies to download the JavaScript code.

To reduce these synchronous latencies, modern browsers allow developers to mark a `<script>` tag with the `async` or `defer` attribute. An `async` script is downloaded in parallel with the HTML parse, but once it is downloaded, it will execute synchronously, in a parse-blocking manner. A `defer` script is only downloaded and executed once HTML parsing is complete.

By default, a `<script>` tag is neither `async` nor `defer`. Such scripts represent 98.3% of all JavaScript files in our test corpus of 200 popular sites (§3.5). When the HTML parser in a modern browser encounters a synchronous `<script>` tag, the parser enters *speculation mode*. The parser initiates the download of the JavaScript file, and as that download completes in the background, the parser continues to process the HTML after the script tag, fetching the associated objects and updating a speculative version of the DOM. The browser discards the speculative DOM if it is invalidated by the execution of the upstream JavaScript code. We demonstrate in Section 5 that speculative parsing is limited in its ability to resolve deep dependency chains consisting of multiple JavaScript files.

| Object Type | Median | 95th Percentile |
|---|---|---|
| HTML | 11.8% | 26.2% |
| JavaScript | 22.9% | 43.0% |
| CSS | 3.7% | 16.7% |
| Images | 44.9% | 77.4% |
| Fonts | 0.0% | 7.8% |
| JSON | 0.4% | 5.0% |
| Other | 0.0% | 7.8% |

Table 1: Per-page object distributions for 200 popular sites.

**CSS:** A page may use CSS to define the visual presentation of HTML tags. The browser represents those stylings using a CSS Object Model (CSSOM) tree. The root of the CSSOM tree contains the general styling rules that apply to all HTML tags. Different paths down the tree apply additional rules to particular types of nodes, resulting in the "cascading" aspect of cascading style sheets.

A browser defines a default set of CSS rules known as the user agent styles. A web page provides additional rules by incorporating CSS `<link>` tags. To create the render tree, the browser uses the DOM tree to enumerate a page's visible HTML tags, and the CSSOM tree to determine what those visible tags should look like.

CSS tags do not block HTML parsing, but they do block rendering, layout, and painting. The reason is that unstyled pages are visually unattractive and potentially non-interactive, so style computations should be handled promptly. Best practices encourage developers to place CSS tags at the top of pages, to ensure that the CSSOM tree is built quickly. Since JavaScript code can query the CSS properties of DOM nodes, the browser halts JavaScript execution while CSS is being processed; doing so avoids race conditions on CSS state.

**Images:** Browsers do not load `<img>` tags synchronously. Thus, a page can be completely rendered and laid out (and partially painted) even if there are outstanding image requests. However, browsers are still motivated to load images as quickly as possible, since users do not like pages with missing images.

**Other media files:** Besides images, a page can include various types of video and audio files. However, in this paper, we focus on the loading of HTML, JavaScript, CSS, and image files, which are the most common types of web objects (see Table 1). Optimizing the loading process for rich multimedia files requires complex, media-specific techniques (e.g., [11, 15]).

### 2.2 The Pitfalls of Lexical Dependencies

As described above, the traditional approach for loading a page is constrained by uncertainty. For example:

- A script tag *might* read CSS style properties from the DOM tree, so CSS evaluation must block JavaScript execution.
- A script tag *might* change downstream HTML, so when the browser encounters a script tag, either HTML parsing must block (increasing page load time), or HTML parsing must transfer to a speculative thread (a thread which, if aborted, will have wasted network and computational resources).
- In the example from Section 1, two script tags that are lexically adjacent *might* exhibit a write/read dependency on JavaScript state. Thus, current browsers must execute the script tags serially, in lexical order, even if a different order (or parallel execution) would be more efficient.

These inefficiencies arise because HTML expresses a strict tag ordering that is based on *lexical dependencies* between tags. In reality, a page's true dependency graph is a partial ordering in which edges represent *true semantic dependencies* like write/read dependencies on JavaScript state. Since HTML does not express all of the true semantic dependencies, the browser is forced to pessimistically guess those dependencies, or use optimistic speculation that may waste resources.

In Section 3, we enumerate the kinds of true semantic dependencies that pages can have, and introduce a new framework to extract them. In Section 4, we describe how developers can expose true dependencies to the browser, allowing the browser to load pages faster.

## 3 DEPENDENCY TRACKING

In a traditional dependency graph [8, 13, 18, 25, 26], a vertex represents an object like an image or a JavaScript file. An edge represents a load-before relationship that is the side-effect of parsing activity. For example, if a page incorporates an image via an `<img>` tag, the image's parent in the dependency graph will be the HTML file which contains the tag; if an image is fetched via an `XMLHttpRequest`, the image's parent will be the associated JavaScript file.

By emphasizing fetch initiation contexts, i.e., the file whose parsing causes an object to be downloaded, traditional dependency graphs mimic the lexical restrictions that constrain real browsers (§2). However, fetch initiation contexts obscure the fine-grained data flows that truly govern the order in which a page's objects must be assembled. In this section, we provide a taxonomy for those fine-grained dependencies, and describe a new measurement framework called Scout that captures those dependencies. The resulting dependency graphs have more edges than traditional graphs (because finer-grained dependencies are included). However, as we show in Section 5, fine-grained dependency graphs permit *more* aggressive load schedules, because browsers are no longer shackled by conservative assumptions about where hidden dependencies might exist.

### 3.1 Page State

Objects in a web page interact with each other via two kinds of state. The *JavaScript heap* contains the code and the data that are managed by the JavaScript runtime. This runtime interacts with the rest of the browser through the DOM interface. The DOM interface reflects internal, C++ browser state into the JavaScript runtime. However, the reflected JavaScript objects do not directly expose the rendering and layout trees. Instead, the DOM interface exposes an extended version of the DOM tree in which each node also has properties for style information and physical geometry (§2). By reading and writing this *DOM state*, JavaScript code interacts with the browser's rendering, layout, and painting mechanisms. The DOM interface also allows JavaScript code to dynamically fetch new web objects, either indirectly, by inserting new HTML tags into the DOM tree, or directly, using `XMLHttpRequests` or `WebSockets`.

### 3.2 Dependency Types

We are interested in capturing three types of data flows that involve the JavaScript heap and the DOM state belonging to HTML and CSS.

**Write/read dependencies** arise when one object produces state that another object consumes. For example, `a.js` might create a global variable in the JavaScript heap; later, `b.js` might read the variable. When optimizing the load order of the two scripts, we cannot evaluate `b.js` before `a.js` (although it is safe to *fetch* `b.js` before `a.js`).

**Read/write dependencies** occur when one object must read a piece of state before the value is updated by another object. Such dependencies often arise when JavaScript code must read a DOM value before the value is changed by the HTML parser or another JavaScript file. For example, suppose that the HTML parser encounters a JavaScript tag that lacks the `async` or `defer` attributes. The browser must synchronously execute the JavaScript file. Suppose that the JavaScript code reads the number of DOM nodes that are currently in the DOM tree. The DOM query examines a snapshot of the DOM tree at a particular moment in time; as explained in Section 2, a browser progressively updates the DOM tree as HTML is parsed. Thus, any reordering of object evaluations must ensure value equivalence for DOM queries—regardless of when a JavaScript file is executed, its DOM queries must return the same results. This guarantees deterministic JavaScript execution semantics [24] despite out-of-order evaluation.

**Write/write dependencies** arise when two objects update the same piece of state, and we must preserve the relative ordering of the writes. For example, CSS files update DOM state, changing the rules which govern a page's visual presentation. The CSS specification states that, if two files update the same rule, the last writer wins. Thus, CSS files which touch the same rule must be evaluated in their original lexical ordering in the HTML. However, the evaluation of the CSS files can be arbitrarily reordered with respect to the execution of JavaScript code that does not access DOM state.

Output devices are often involved in write/write dependencies. As described in the previous paragraph, CSS rules create a write/write dependency on a machine's display device. Write/write dependencies can also arise for local storage and the network. For example, the `localStorage` API exposes persistent storage to JavaScript using a key/value interface. If we shuffle the order in which a page evaluates JavaScript objects, we must ensure that the final value for each `localStorage` key is the same value that would result from the original execution order of the JavaScript files.

**Traditional dependencies based on HTML tag constraints** can often be eliminated if finer-grained dependencies are known. For example, once we know the DOM dependencies and JavaScript heap dependencies for a `<script>` tag, the time at which the script can be evaluated is completely decoupled from the position of the `<script>` tag in the HTML—we merely have to ensure that we evaluate the script after its fine-grained dependencies are satisfied. Similarly, we can parse and render a piece of HTML at any time, as long as we ensure that we have blocked the evaluation of downstream objects in the dependency graph.

Images do need to be placed in specific locations in the DOM tree. However, browsers already allow images to be fetched and inserted asynchronously. So, images can be fetched in arbitrary orders, regardless of the state of the DOM tree, but their insertion is dependent on the creation of the associated DOM elements. We model this using write/write dependencies on DOM elements: the HTML parser must write an initially empty `<img>` DOM node, and then the network stack must insert the fetched image bitmap into that node.

## 3.3 Capturing Dependencies with Scout

To capture the fine-grained dependencies in a real web page, we first record the content of the page using Mahimahi [28]. Next, we use a new tool called Scout to rewrite each JavaScript and HTML file in the page, adding instrumentation to log fine-grained data flows across the JavaScript heap and the DOM. Scout then loads the instrumented page in a regular browser. As the page loads, it emits a dependency log to a Scout analysis server; the server uses the log to generate the fine-grained dependency graph for the page.

**Tracking JavaScript heap dependencies:** To track dependencies in which both actors are JavaScript code, Scout leverages JavaScript proxy objects [27]. A proxy is a transparent wrapper for an underlying object, allowing custom event handlers to fire whenever external code tries to read or write the properties of the underlying object.

In JavaScript, the global namespace is explicitly nameable via the `window` object; for example, the global variable `x` is also reachable via the name `window.x`. Scout's JavaScript rewriter transforms unadorned global names like `x` to fully qualified names like `window.x`. Also, for each JavaScript file (whether inline or externally fetched), Scout wraps the file's code in a closure which defines a local alias for the `window` variable. The aliasing closures, in combination with rewritten code using fully qualified global names, forces all accesses to the global namespace to go through Scout's `window` proxy. Using that proxy, Scout logs all reads and writes to global variables.

Scout's `window` proxy also performs recursive proxying for non-primitive global values. For example, reading a global object variable `window.x` returns a logging proxy for that object. In turn, reading a non-primitive value `y` on that proxy would return a proxy for `y`. By using recursive proxying and wrapping calls to `new` in proxy generation code, Scout can log any JavaScript-issued read or write to JavaScript state. Each read or write target is logged using a fully qualified path to the `window` object, e.g., `window.x.y.z`. Log entries also record the JavaScript file that issued the operation.

Scout's proxy generation code tags each underlying object with a unique, non-enumerable integer id. The proxy code also stores a mapping between ids and the corresponding proxies. When a proxy for a particular object is requested, Scout checks whether the object already has an id. If it does, Scout returns the preexisting proxy for that object, creating proxy-level reference equalities which mirror those of the underlying objects.

Some objects lack a fully-qualified path to `window`. For example, a function may allocate a heap object and return that object to another function, such that neither function assigns the object to a variable that is recursively reachable from `window`. In these cases, Scout logs the identity of the object using the unique object id.

**Tracking DOM dependencies:** JavaScript code interacts with the DOM tree through the `window.document` object. For example, to find the DOM node with a particular id, JavaScript calls `document.getElementById(id)`. The DOM nodes that are returned by `document` provide additional interfaces for adding and removing DOM nodes, as well as changing the CSS properties of those nodes.

To track dependencies involving JavaScript code and DOM state, Scout's recursive proxy for `window.document` automatically creates proxies for all DOM nodes that are returned to JavaScript code. For example, the `DomNode` returned by `document.getElementById(id)` is wrapped in a proxy which logs reads and writes to the object via interfaces like `DomNode.height`.

Developers do not assign ids to most DOM nodes. Thus, Scout's logs identify DOM nodes by their paths in the DOM tree. For example, the DOM path `<1,5,2>` represents the DOM node that is discovered by examining the first child of the HTML tag, the fifth child of that tag, and then the second child of that tag.

A write to a single DOM path may trigger cascading updates to other paths; Scout must track all of these updates. For example, inserting a new node at a particular DOM path may shift the subtrees of its new DOM siblings to the right in the DOM tree. In this case, Scout must log writes to the rightward DOM paths, as well as to the newly inserted node. Similar bookkeeping is necessary when DOM nodes are deleted or moved to different locations.

The DOM tree can also be modified by the evaluation of CSS objects that change node styles. Scout models each CSS tag as reading all of the DOM nodes that are above it in the HTML, and then writing all of those DOM nodes with new style information. To capture the set of affected DOM nodes, Scout's HTML rewriter prepends each CSS tag with an inline JavaScript tag that logs the current state of the DOM tree (i.e., all of the live DOM paths) and then deletes itself from the DOM tree.

In Scout logs, we represent DOM operations using the `window.$$dom` pseudovariable. For example, the identifier `window.$$dom.1` represents the first child of the topmost `<html>` node. We also use the `window.$$xhr` pseudovariable to track network reads and writes via `XMLHttpRequests`. These pseudovariables allow us to use a single analysis engine to process all dependency types.

**Missing Dependencies:** To generate a page's dependency graph, Scout loads an instrumented version of the page on a server-side browser, and collects the resulting dependency information. Later, when Polaris loads the page on a client-side browser (§4), Polaris assumes that Scout's dependency graph is an accurate representation of the dependencies in the page. This might not be true if the page's JavaScript code exhibits nondeterministic behavior. For example, suppose that a page contains three JavaScript files called `a.js`, `b.js`, and `c.js`. At runtime, `a.js` may call `Math.random()`, and use the result to invoke a function in `b.js` or `c.js` (but not both). During some executions, Scout will log a dependency between `a.js` and `b.js`; during other executions, Scout will log a dependency between `a.js` and `c.js`. If there is a discrepancy between the dependency logged by Scout, and the dependency generated by the code on the client browser, then Polaris may evaluate JavaScript files in the wrong order, breaking correctness.

We have not observed such nondeterministic dependencies in our corpus. However, if a page does include such dependencies, Scout must create a dependency graph which contains the aggregate set of all possible dependencies. Such a graph overconstrains any particular load of the page, but guarantees that clients will load pages without errors. The sources of nondeterministic JavaScript events are well-understood [24], so Scout can use a variety of techniques to guarantee that nondeterministic dependencies are either tracked or eliminated. For example, Scout can rewrite pages so that calls to `Math.random()` use a deterministic seed [24], removing nondeterminism from calls to the random number generator.

For a given page, a web server may generate a different dependency graph for different clients. For example, a web server might personalize the graph in response to a user's cookie; as another example, a server might return a smaller dependency graph in response to a user agent string which indicates a mobile browser. The server-side logic must run Scout on each version of the dependency graph. We believe that this burden will be small in practice, since even customized versions of a page often share the same underlying graph structure (with different content in some of the nodes).

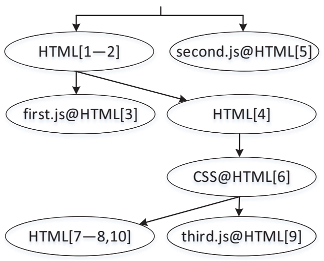
**Implementation:** To build Scout, we used Esprima [14], Estraverse [36], and Escodegen [35] to rewrite JavaScript code, and we used Beautiful Soup [32] to rewrite HTML. We loaded the instrumented pages in a commodity Firefox browser (version 40.0). Each page sent its dependency logs to a dedicated analysis server; logs were sent via an `XMLHttpRequest` that was triggered by the `onload` event.

Our implementation of Scout handles the bulk of the JavaScript language. However, our implementation does not currently support the `eval(sourceCode)` statement, which pages use to dynamically execute new JavaScript code. To support this statement, Scout would need to shim `eval()` and dynamically rewrite the `sourceCode` argument so that the rewritten code tracked dependencies.

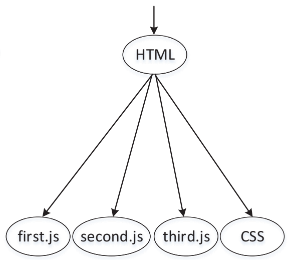
Our current implementation also does not support the `with(obj)` statement, which places `obj` at the beginning of the scope chain that the JavaScript runtime

```
1  <h1>Text</h1>
2  <p>Text</p>
3  <script src="first.js"/>
       <!--Reads <p> tag-->
4  <b>Text</b>
5  <script src="second.js"/>
       <!--Accesses no DOM nodes-->
       <!--or JS state from first.js----->
6  <link rel="stylesheet" href="...">
       <!--CSS-->
7  <span>Text</span>
8  <span>Text</span>
9  <script src="third.js"/>
       <!--Writes <b> tag-->
10 <span>Text</span>
```

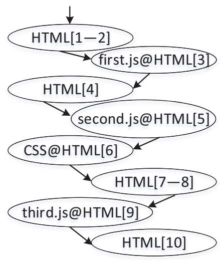
(a) The HTML for a simple page.



(b) The dependency graph generated by Scout.



(c) The dependency graph created by Klotski [8].



(d) The dependency graph created by WProf [37].

Figure 3: Comparing the order in which different tools declare that a simple page's objects must be evaluated. The notation HTML[i:j] refers to HTML lines i up to and including j. The notation obj@HTML[k] refers to the object whose corresponding tag is at HTML[k].

uses to resolve variable names. To support this statement, Scout merely needs to wrap the `obj` argument in code which checks whether `obj` is a proxy; if not, the wrapper would return one.

## 3.4 Dependency Graphs: Scout vs. Prior Tools

Figure 3(a) depicts a simple web page with two JavaScript files and one CSS file. Figures 3(b), (c), and (d) show the dependency graphs that are produced by Scout, Klotski [8], and WProf [37].

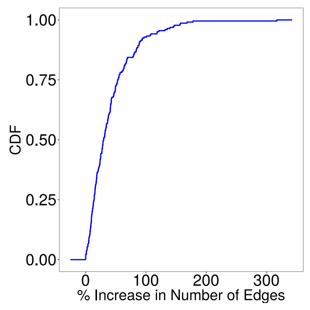
- Scout allows `second.js` and the first chunk of HTML to be evaluated in parallel, since second.js does not access DOM state or JavaScript state defined by prior JavaScript files. `first.js` does access DOM state from upstream HTML tags, but Scout allows the evaluation of `first.js` to proceed in parallel with the parsing of downstream HTML. Scout treats CSS as a read and then a write to all upstream HTML, so the CSS file must be evaluated before the evaluation of downstream HTML and downstream scripts which access DOM state.
- Klotski [8] cannot observe fine-grained data flows, so its dependency graphs are defined by lexical HTML constraints (§2). Given a dependency graph like the one shown in Figure 3(c), Klotski uses heuristics to determine which objects a server should push to the browser first. However, Klotski does not know the page's true data dependencies, so Klotski cannot guarantee that prioritized objects can actually *evaluate* ahead of schedule with respect to their evaluation times in the original page. It is only safe to evaluate an object (prioritized or not) when its ancestors in the dependency graph have been evaluated. So, Klotski's prioritized pushes can safely warm the client-side cache, but in general, it is unsafe for those pushes to synchronously trigger object evaluations.
- By instrumenting the browser, WProf observes the times at which a browser is inside the network stack or a parser for HTML, CSS, or JavaScript. Thus, WProf can track complex interactions between a browser's fetching, parsing, and evaluation mechanisms. However, this technique only allows WProf to analyze the critical path for the *lexically-defined* dependency graph. This graph does not capture true data flows, and forces conservative assumptions about evaluation order (§2.2). As shown in Figure 3(d), WProf overconstrains the order in which objects can be evaluated (although WProf may allow objects to be *fetched* out-of-lexical-order).

In summary, only Scout produces a dependency graph which captures the true constraints on the order in which objects can be evaluated. Polaris uses these fine-grained dependencies to schedule object downloads—by prioritizing objects that block the most downstream objects, Polaris reduces overall page load times (§4).

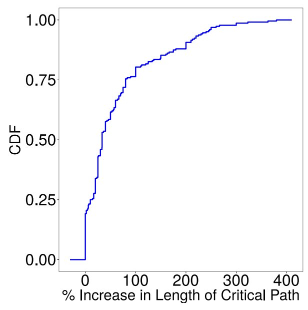
## 3.5 Results

We used Mahimahi [28], an HTTP record-and-replay tool, to record the content from 200 sites in the Alexa Top 500 list [2]. The corpus spanned a variety of page categories, including news, ecommerce, and social media. The corpus also included five mobile-optimized sites. Since our Scout prototype does not support the `eval()` and `with()` statements, we selected pages which did not use those statements.

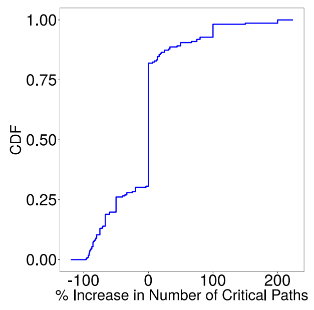
Figure 4 summarizes the differences between Scout's dependency graphs and the traditional ones that are defined by Klotski [8] and the built-in developer tools from Chrome [13], Firefox [26], and IE [25]. As shown in Figure 4(a), traditional graphs are almost always incomplete, missing many edges that can only be detected via data flow analysis. That analysis adds 29.8% additional



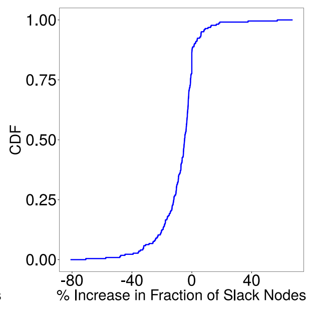
(a)



(b)

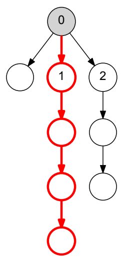


(c)

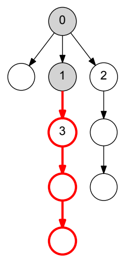


(d)

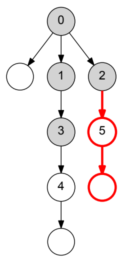
Figure 4: How traditional dependency graphs change when updated with information from fine-grained data flows. The updated graphs have additional edges which belong to previously untracked dependencies. The new edges often modify a page's critical paths. Note that a slack node is a node that is not on a critical path.



(a)



(b)



(c)

Figure 5: An example of dynamic critical paths during the load of a simple page. Dynamic critical paths are shown in red. Numbers represent the order in which Polaris requests the objects. Shaded objects have been received and evaluated; numbered but unshaded objects have been requested, but have no responses yet. We assume that all objects are from the same origin, and that only two outstanding requests per origin are allowed.

edges at the median, and 118% more edges at the 95th percentile.

Those additional edges have a dramatic impact on the characteristics of dependency graphs. For example, **adding fine-grained dependencies alters the critical path length for 80.8% of the pages in our corpus** (Figure 4(b)). The set of objects on those paths often changes, with old objects being removed and new objects being added. Furthermore, as shown in Figure 4(d), **86.6% of pages have a smaller fraction of slack nodes when fine-grained dependencies are considered.** Slack nodes are nodes that are not on a critical path. Thus, a decrease in slack nodes means that browsers have *fewer* load schedules which result in optimal page load times.

## 4 Polaris: Dynamic Client-side Scheduling

Polaris is a client-side scheduler for the loading and evaluation of a page's objects. Polaris is written completely in JavaScript, allowing it to run on unmodified commodity browsers. Polaris accepts a Scout graph as input, but also uses observations about current network conditions to determine the *dynamic critical path* for a page. The dynamic critical path, i.e., the path which *currently* has the most unresolved objects, is influenced by the order and latency with which network fetches complete; importantly, the dynamic critical path may be different than the critical path in the static dependency graph.[1] Polaris prioritizes the fetching and evaluation of objects along the dynamic critical path, trying to make parallel use of the client's CPU and network, and trying to keep the client's network pipe full, given browser constraints on the maximum number of simultaneous network requests per origin.

Figure 5 shows how a page's dynamic critical path can change over time. In Figure 5(a), Polaris has evaluated object 0, and issued requests for objects 1 and 2, because those objects are the roots for the deepest unresolved paths in the dependency graph. In Figure 5(b), Polaris has received and evaluated object 1, although object 2 is still in-flight. Polaris has one available request slot, so it requests object 3, because that object is the root of the deepest unresolved path. In Figure 5(c), Polaris has received and evaluated object 3; Polaris uses the available request slot to fetch object 4. Then, object 2 is received and evaluated. The critical path changes—the deepest chain is now beneath object 2, so Polaris requests object 5 next.

To use Polaris with a specific page, a web developer runs Scout on that page to generate a dependency graph and a *Polaris scheduler stub*. The developer then configures her web server to respond to requests for that page with the scheduler stub's HTML instead of the page's

[1]This is why a dynamic client-side scheduler is better than a static client-side scheduler that ignores current network conditions and deterministically fetches objects from a server-provided URL list.

regular HTML (see Figure 2). The stub contains four components.

- The **scheduler** itself is just inline JavaScript code.
- The **Scout dependency graph** for the page is represented as a JavaScript variable inside the scheduler.
- **DNS prefetch hints** indicate to the browser that the scheduler will be contacting certain hostnames in the near future. DNS prefetch hints are expressed using `<link>` tags of type `dns-prefetch`, e.g.,

  ```
  <link rel=''dns-prefetch''
      href=''http://domain.com''>
  ```

  DNS hints allow Polaris to pre-warm the DNS cache in the same way that the browser does during speculative HTML parsing (§2.1).
- Finally, the stub contains **the page's original HTML**, which is broken into chunks as determined by Scout's fine-grained dependency resolution (see §3.3 and Figure 3). When Scout generates the HTML chunks, it deletes all `src` attributes in HTML tags, since the external objects that are referenced by those attributes will be dynamically fetched and evaluated by Polaris.

Polaris adds few additional bytes to a page's original HTML. Across our test corpus of 200 sites, the scheduler stub was 3% (36.5 KB) larger than a page's original HTML at the median.

The scheduler uses `XMLHttpRequests` to dynamically fetch objects. To evaluate a JavaScript file, the scheduler uses the built-in `eval()` function that is provided by the JavaScript engine. To evaluate HTML, CSS, and images, Polaris leverages DOM interfaces like `document.innerHTML` to dynamically update the page's state.

In the rest of this section, we discuss a few of the subtler aspects of implementing an object scheduler as a JavaScript library instead of native C++ code inside the browser.

**Browser network constraints:** Modern browsers limit a page to at most six outstanding requests to a given origin. Thus, Polaris may encounter situations in which the next missing object on the dynamic critical path would be the seventh outstanding request to an origin. If Polaris actually generated the request, the request would be placed at the end of the browser's internal network queue, and would be issued at a time of the browser's choosing. Polaris would lose the ability to precisely control the in-flight requests at any given moment.

To avoid this dilemma, Polaris maintains per-origin priority queues. With the exception of the top-level HTML (which is included in the scheduler stub), each object in the dependency graph belongs to exactly one queue. Inside a queue, objects that are higher in the dependency tree receive a higher priority, since those objects prevent the evaluation of more downstream objects. At any given moment, the scheduler tries to fetch objects that reside in a dynamic critical path for the page load. However, if fetching the next object along a critical path would violate a per-origin network constraint, Polaris examines its queues, and fetches the highest priority object from an origin that has available request slots.[2]

**Frames:** A single page may contain multiple iframes. Scout generates a scheduler stub for each one, but the browser's per-origin request cap is a page-wide limit. Thus, the schedulers in each frame must cooperate to respect the limit and prevent network requests from getting stuck inside the browser's internal network queues.

The scheduler in the top frame coordinates the schedulers in child frames. Using `postMessage()` calls, children ask the top-most parent for permission to request particular objects. The top-most parent only authorizes a fetch if per-origin request limits would not be violated.

**URL matching:** A page's coarse-grained dependency graph has a stable structure [8]. In other words, the edges and vertices that are defined by lexical HTML constraints change slowly over time. However, the URLs for specific vertices change more rapidly. For example, if JavaScript code dynamically generates an `XMLHttpRequest` URL, that URL may embed the current date in its query string. Across multiple page loads, the associated object for the URL will have different names, even though all of the objects will reside in the same place in the dependency graph.

To handle any discrepancies between the URLs in Scout's dependency graphs and the URLs which `XMLHttpRequests` generate on the client, Polaris uses a matching heuristic to map dynamic URLs to their equivalents in the static dependency graph. Our prototype implementation uses Mahimahi's matching heuristic [28], but Polaris is easily configured to use others [8, 9, 33].

**Page-generated XHRs:** When Polaris evaluates a JavaScript file, the executed code might try to fetch an object via `XMLHttpRequest`. Assuming that a page has deterministic JavaScript code (§3.3), Scout will have included the desired object in the page's dependency graph. However, during the loading of the page in a real client browser, Polaris requires control over the order in which objects are fetched. Thus, Polaris uses an

[2]Browsers allow users to modify the constraint on the maximum number of connections per origin; Polaris can be configured to respect user-programmed values.

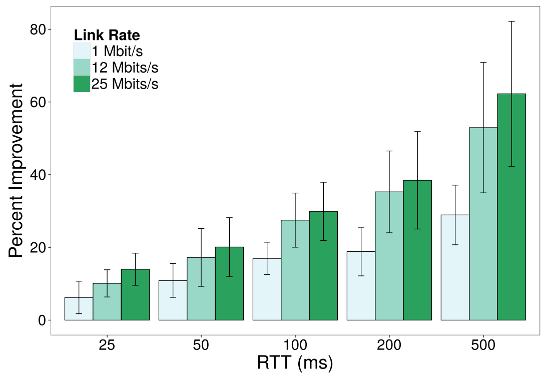


Figure 6: Polaris' average reduction in page load times, relative to baseline load times with Firefox v40.0. Each bar is the average reduction in load time across the entire 200 site corpus. Error bars span one standard deviation in each direction of the average.

| | | RTT | | |
|---|---|---|---|---|
| | | 25 ms | 100 ms | 500 ms |
| Link Rate | 1 Mbit/s | 256.3 ms | 883.9 ms | 1857.5 ms |
| | 12 Mbit/s | 309.1 ms | 1274.1 ms | 2935.0 ms |
| | 25 Mbit/s | 382.5 ms | 1385.3 ms | 3188.3 ms |

Table 2: Polaris' raw reduction in median page load times for a subset of the parameter values in Figure 6.

`XMLHttpRequest` shim [24] to suppress autonomous `XMLHttpRequests`. Polaris issues those requests using its own scheduling algorithm, and manually fires `XMLHttpRequest` event handlers when the associated data has arrived.

## 5 Results

In this section, we demonstrate that Polaris can decrease page load times across a variety of web pages and network configurations: performance improves by 34% and 59% for the median and 95th percentile sites, respectively. Polaris' benefits grow as network latencies increase, because higher RTTs increase the penalty for bad fetch schedules. Thus, Polaris is particularly valuable for clients with cellular or low-quality wired networks. However, even for networks with moderate RTTs, Polaris can often reduce load times by over 20%.

### 5.1 Methodology

We evaluated Polaris using the 200 site corpus that is described in Section 3.3. We used Mahimahi [28] to capture site content and later replay it using emulated network conditions. To build Polaris-enabled versions of each page, we post-processed the recorded web content, generating Polaris scheduler stubs for each site. We then compared the load times of the Polaris sites and the original versions of those sites. All experiments used Firefox v40.0. Unless otherwise specified, all experiments used cold browser caches and DNS caches.

A page's load time is normally defined with respect to JavaScript events like `navigationStart` and `loadEventEnd`. However, `loadEventEnd` is inaccurate for Polaris pages, since the event only indicates that the scheduler stub has been loaded; the rest of the page's objects remain to be fetched by the dynamic scheduler. So, to define the load time for a Polaris page, we first loaded the original version of the page and used tcpdump to capture the objects that were fetched between `navigationStart` and `loadEventEnd`. We then defined the load time of the Polaris page as the time needed to fetch all of those objects.

### 5.2 Reducing Page Load Times

Figure 6 demonstrates Polaris' ability to reduce load times. There are two major trends to note. First, for a given link rate, Polaris' benefits increase as network latency increases. For example, at a link rate of 12 Mbits/s, Polaris provides an average improvement of 10.1% for an RTT of 25 ms. However, as the RTT increases to 100 ms and 200 ms, Polaris' benefits increase to 27.5% and 35.3%, respectively. The reason is that, as network latencies grow, so do the penalties for not prioritizing the fetches of objects on the dynamic critical path. Polaris *does* prioritize the fetching of critical path objects. Furthermore, Polaris never has to wait for an object evaluation to reveal a downstream dependency—Polaris knows all of the dependencies at the beginning of the page load, so Polaris can always keep the network pipe full.

The second trend in Figure 6 is that, for a given RTT, Polaris' benefits increase as network bandwidth grows. This is because, if bandwidth is extremely low, transfer times dominate fetch costs. As bandwidth increases, latency becomes the dominant factor in download times. Since Polaris prioritizes the fetch orders for critical path objects (but does nothing to reduce those objects' bandwidth costs), Polaris' gains are most pronounced when latencies govern overall download costs.

Figure 6 describes Polaris' gains in relative terms. Table 2 depicts absolute gains, describing how many raw milliseconds of load time Polaris removes. Even on a fast network with 25 ms of latency, Polaris eliminates over 250 ms of load time. Those results are impressive, given that web developers strive to eliminate *tens of milliseconds* from their pages' load times [5, 6, 10].

The error bars in Figure 6 are large. The reason is that, for a given network bandwidth and latency, Polaris' benefits are determined by the exact structure of a page's dependency graph. To understand why, consider the three sites in Figure 7.

- The homepage for apple.com has a flat dependency graph, as shown in Figure 8. This means that, once the browser has the top-level HTML, the other objects can be fetched and evaluated in an arbitrary or-

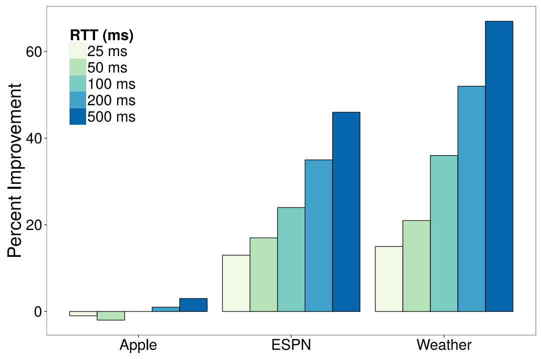


Figure 7: Polaris' average reduction in page load times, relative to baseline load times, for three sites with diverse dependency graph structures. Each experiment used a link rate of 12 Mbits/s.

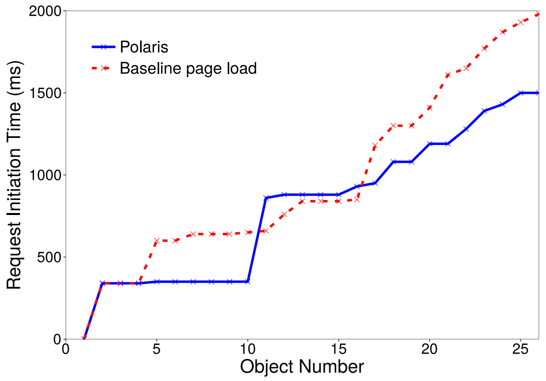


Figure 9: Request initiation times for the regular and Polaris-enabled versions of StackOverflow. These results used a 12 Mbits/s link with an RTT of 100 ms.

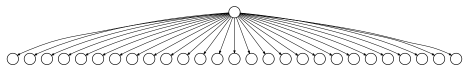


Figure 8: The dependency graph for Apple's homepage.

der; all orders will result in similar end-to-end page load times. Thus, for low RTTs, Polaris loads the apple.com homepage 1–2% *slower* than the baseline, due to computational overheads from Polaris' scheduling logic.

- In contrast, the ESPN homepage has a dependency path of length 5, and several paths of length 4. Also, 48% of the page's content is loaded from only two origins (a.espncdn.com and a1.espncdn.com), magnifying the importance of optimally scheduling the six outstanding requests for each origin (§4). In ESPN's dependency graph, many of the long paths consist of JavaScript files. However, the standard Firefox scheduler has no way of knowing this. So, when Firefox loads the standard version of the page, it initially requests a small number of JavaScript objects, and then fills the rest of its internal request queue with 32 image requests. As a result, when a JavaScript file evaluates and generates a request for another JavaScript file on the critical path, the request is often stuck behind image requests in the browser's internal network queue. In contrast, Polaris has a priori knowledge of which JavaScript files belong to deep dependency chains. Thus, Polaris prioritizes the fetching of those objects, using its knowledge of per-origin request caps to ensure that the fetches for critical path objects are never blocked.
- As shown in Figure 1(b), the weather.com homepage is even more complicated than that of ESPN. Deep, complex dependency graphs present Polaris with the most opportunities to provide gains. Thus, of the three sites in Figure 7, weather.com enjoys the largest reductions in load time.

Figure 9 depicts the order in which requests issue for the normal version of the StackOverflow site, and the Polaris version. In general, Polaris issues requests earlier; by prioritizing the fetches of objects on the dynamic critical path, Polaris minimizes the overall fetch time needed to gather *all* objects. However, as shown in Figure 9, Polaris briefly falls behind the default browser scheduler after fetching the tenth object. The reason is that, in our current Polaris implementation, HTML is rendered in large chunks. While that HTML is being rendered, Polaris cannot issue new HTML requests, because executing Polaris' JavaScript-level scheduler would block rendering (§2.1). In contrast, a native browser scheduler can issue new requests in parallel with HTML rendering. Thus, the default Firefox scheduler has a lower time-to-first paint than Polaris, and Polaris falls behind the default scheduler after the tenth object fetch. However, after Polaris renders the bulk of the HTML, Polaris quickly regains its lead and never relinquishes it. To minimize Polaris' time-to-first-paint, future versions of Polaris will render HTML in smaller increments; this will not affect Polaris' ability to optimize network utilization.

## 5.3 Browser Caching

Up to this point, our experiments have used cold browser caches. In this section, we evaluate Polaris' performance when caches are warm. To do so, we examined the HTTP headers in our recorded web pages, and, for each object that was marked as cacheable, we rewrote the caching headers to ensure that the object would remain cacheable for the duration of our experiment. Then, for each page, we cleared the browser's cache, and loaded the page twice, recording the elapsed time for the second load.

Figure 10 depicts Polaris' benefits with warm caches; the improvements are normalized with respect to Po-

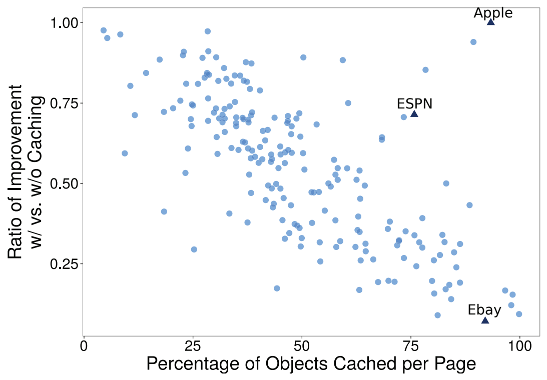


Figure 10: Polaris' benefits with warm caches, normalized with respect to Polaris' gains with cold caches. Each data point represents one of the 200 sites in our corpus. Pages were loaded over a 12 Mbits/s link with an RTT of 100 ms.

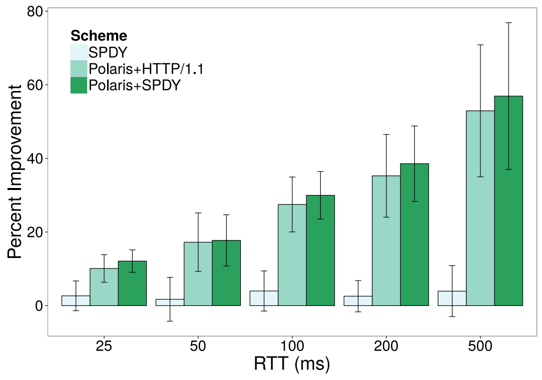


Figure 11: Average reductions in page load time using SPDY, Polaris over HTTP/1.1, and Polaris over SPDY. The performance baseline was load time using HTTP/1.1. The link rate was 12 Mbits/s.

laris' improvements when caches are cold. In general, Polaris' benefits decrease as cache hit rates increase, because there are fewer opportunities for Polaris to optimize network fetches. For example, Ebay caches 92% of all objects, including most of the JavaScript files involved in deep dependency chains; thus, Polaris provides little advantage over the standard scheduling algorithm.

That being said, there are many instances in which caching does not touch objects along a page's critical path. For example, on ESPN's site, 76% of objects are cacheable, but only one object on the deepest dependency chain is cached. Furthermore, a.espncdn.com serves many uncacheable images and JavaScript objects, leading Firefox's standard scheduler to bury critical path JavaScript files behind images that are not on the critical path (§5.2). So, even though ESPN caches 76% of its objects, Polaris still provides 71% of its cold-cache benefits.

Note that the Apple site is an outlier: it caches 93% of its objects, but Polaris provides little benefit in the cold cache case (§5.2), so Polaris provides most of that negligible benefit in the warm cache case as well.

## 5.4 SPDY

Google proposed SPDY [22], a transport protocol for HTTP messages, to remedy several problems with the HTTP/1.1 protocol. SPDY differs from HTTP/1.1 in four major ways:

- First, SPDY uses a single TCP connection to multiplex all of a browser's HTTP requests and responses involving a particular origin. This allows HTTP requests to be pipelined, and reduces the TCP and TLS handshake overhead that would be incurred if a browser opened multiple TCP connections to an origin.
- SPDY also allows a browser to prioritize the fetches of certain objects (e.g., JavaScript files which block HTML parsing). Priorities give servers hints about how to allocate limited bandwidth to multiple responses.
- SPDY compresses HTTP headers. HTTP is a text-based protocol, so compression can result in non-trivial bandwidth savings.
- Finally, SPDY allows a server to proactively push objects to a browser if the server believes that the browser will request those objects in the near future.

SPDY was a major influence on the HTTP/2 protocol [4] whose deployment is currently starting.

Mahimahi supports SPDY page loads using the mod_spdy Apache extension [20]. Thus, we could use Mahimahi to explore how SPDY interacts with Polaris. We loaded each page in our test corpus using four different schemes: HTTP/1.1 (which all of our previous experiments used), Polaris over HTTP/1.1, SPDY, and Polaris over SPDY. In our experiments, SPDY used TCP multiplexing, object prioritization, and HTTP header compression, but not server push, since few of the sites in our test corpus defined SPDY push policies.

Figure 11 compares load times using the four schemes on a 12 Mbits/s link with various RTTs; the performance baseline is the load time using HTTP/1.1. On average, load times using SPDY are 1.74%–3.98% faster than those with HTTP/1.1. Load times using Polaris over SPDY are 2.05%–4.03% faster than those with Polaris over HTTP/1.1. These results corroborate prior work [38] which found that object dependencies limit the ability of SPDY to maximize network utilization. For example, a SPDY-enabled browser may prioritize a JavaScript file in hopes of minimizing the stall time of the HTML parser. However, without Polaris, the SPDY-enabled browser is still limited by conservative lexical dependencies (§2.2), meaning that it cannot aggressively fetch objects "out-of-order" with respect to lexical constraints. In contrast, both Polaris over HTTP/1.1 and Polaris over SPDY have fine-grained dependency information. That information allows Polaris to issue out-of-lexical-order fetches which reduce page load time while respecting the page's intended data flow semantics.

In theory, SPDY-enabled web servers could use Scout's dependency graphs to guide server push policies. However, we believe that *clients*, not servers, are best qualified to make decisions about how a client's network pipe should be used. A server from origin `X` cannot see the objects being pushed by origin `Y`, so different origins may unintentionally overload a client's resource-constrained network connection. Furthermore, Scout's dependency graphs do not capture dynamic critical paths, i.e., the set of object fetches which a client should prioritize *at the current moment* (§4). Thus, a well-intentioned server may *hurt* load time by pushing objects which are not on a dynamic critical path. Polaris avoids this problem using dynamic client-side scheduling.

## 6 Related Work

Prior dependency trackers [8, 13, 25, 26, 37] deduce dependencies using lexical relationships between HTML tags. As discussed in Sections 2.2 and 3.3, those lexical relationships do not capture fine-grained data flows. As a result, load schedulers which use those dependency graphs are forced to make conservative assumptions to preserve correctness.

WebProphet [18] determines the dependencies between objects by carefully perturbing network fetch delays for individual objects; delaying a parent should delay the loads of dependent children. This technique also relies on course-grained lexical dependencies, since the perturbed browser uses those HTML dependencies to determine which objects to load.

Silo [23] uses aggressive inlining of JavaScript and CSS to fetch entire pages in one or two RTTs. However, Silo does not use the CPU and the network in parallel—all content is fetched, and then all content is evaluated. In contrast, Polaris overlaps computation with network fetches.

Compression proxies like Google FlyWheel [1] and Opera Turbo [29] transparently compress objects before transmitting them to clients. For example, FlyWheel re-encodes images into space-saving formats, and minifies JavaScript and CSS. Polaris is complementary to such techniques.

JavaScript module frameworks like RequireJS [7] and ModuleJS [16] allow developers to manually specify dependencies between JavaScript libraries. Once the dependencies are specified, the frameworks ensure that the relevant libraries are loaded in the appropriate order. Keeping manually-specified dependencies up-to-date can be challenging for a large web site. In contrast, Scout *automatically* tracks fine-grained dependencies between JavaScript files. Scout also tracks dependencies involving HTML, CSS, and images.

## 7 Conclusion

Prior approaches for loading web pages have been constrained by uncertainty. The objects in a web page can interact in complex and subtle ways; however, those subtle interactions are only partially captured by lexical relationships between HTML tags. Unfortunately, prior load schedulers have used those lexical relationships to extract dependency graphs. The resulting graphs are under-specified and omit important edges. Thus, load schedulers which use those graphs must be overly conservative, to preserve correctness in the midst of hidden dependencies. The ultimate result is that web pages load more slowly than necessary.

In this paper, we use a new tool called Scout to track the fine-grained data flows that arise during a page's load process. Compared to traditional dependency trackers, Scout detects 30% more edges for the median page, and 118% more edges for the 95th percentile page. These additional edges actually give browsers *more* opportunities to reduce load times, because they enable more aggressive fetch schedules than allowed by conservative, lexically-derived dependency graphs. We introduce a new client-side scheduler called Polaris which leverages Scout graphs to assemble a page. By prioritizing the fetches of objects along the dynamic critical path, Polaris minimizes the number of RTTs needed to load a page. Experiments with real pages and varied network conditions show that Polaris reduces load times by 34% for the median page, and 59% for the 95th percentile page.

## 8 Acknowledgements

We thank Katrina LaCurts, Amy Ousterhout, the NSDI reviewers, and our shepherd, Paul Barford, for their helpful comments and suggestions. We also thank the members of the MIT Center for Wireless Networks and Mobile Computing (Wireless@MIT) for their support. This work was supported in part by NSF grant CNS-1407470.

## 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, 2015.

[3] Amazon. Silk Web Browser. https://amazonsilk.wordpress.com/, December 16, 2014.

[4] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2. http://httpwg.org/specs/rfc7540.html, May 2015.

[5] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating User-perceived Quality into Web Server Design. In *Proceedings of World Wide Web Conference on Computer Networks: The International Journal of Computer and Telecommunications Networking*, 2000.

[6] A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the Eye of the Beholder: Meeting Users' Requirements for Internet Quality of Service. In *Proceedings of CHI*, 2000.

[7] J. Burke. RequireJS. http://requirejs.org/, 2015.

[8] 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.

[9] Chromium. web-page-replay. https://github.com/chromium/web-page-replay, 2015.

[10] 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.

[11] A. Ganjam, F. Siddiqui, J. Zhan, X. Liu, I. Stoica, J. Jiang, V. Sekar, and H. Zhang. C3: Internet-Scale Control Plane for Video Quality Optimization. In *Proceedings of NSDI*, 2015.

[12] Google. Using site speed in web search ranking. http://googlewebmastercentral.blogspot.com/2010/04/using-site-speed-in-web-search-ranking.html, April 9, 2010.

[13] Google. Chrome DevTools Overview. https://developer.chrome.com/devtools, August 2013.

[14] A. Hidayat. Esprima. http://esprima.org, 2015.

[15] J. Jiang, V. Sekar, and H. Zhang. Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE. In *Proceedings of CoNext*, 2012.

[16] L. Jung. modulejs lightweight JavaScript module system. https://larsjung.de/modulejs/, 2016.

[17] Q. Li, W. Zhou, M. Caesar, and P. B. Godfrey. ASAP: A Low-latency Transport Layer. In *Proceedings of SIGCOMM*, 2011.

[18] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. WebProphet: Automating Performance Prediction for Web Services. In *Proceedings of NSDI*, 2010.

[19] Google Developers. Remove Render-Blocking JavaScript. https://developers.google.com/speed/docs/insights/BlockingJS, April 8, 2015.

[20] Google Developers. SPDY. https://developers.google.com/speed/spdy/mod_spdy/, May 27, 2015.

[21] The Chromium Projects. QUIC, a multiplexed stream transport over UDP. https://www.chromium.org/quic, 2015.

[22] The Chromium Projects. SPDY. https://www.chromium.org/spdy, 2015.

[23] J. Mickens. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads. In *Proceedings of WebApps*, 2010.

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

[25] Microsoft. Meet the Microsoft Edge Developer Tools. https://dev.windows.com/en-us/microsoft-edge/platform/documentation/f12-devtools-guide/, 2015.

[26] Mozilla. Firefox Developer Tools. https://developer.mozilla.org/en-US/docs/Tools, 2015.

[27] Mozilla. Proxy. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy, February 16, 2016.

[28] 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.

[29] Opera. Data savings and turbo mode. http://www.opera.com/turbo, 2015.

[30] Opera. Opera Mini. http://www.opera.com/mobile/mini, 2015.

[31] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. TCP Fast Open. In *Proceedings of CoNext*, 2011.

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

[33] 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.

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

[35] Y. Suzuki. Escodegen. https://github.com/estools/escodegen, 2015.

[36] Y. Suzuki. Estraverse. https://github.com/estools/estraverse, 2016.

[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. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How Speedy is SPDY? In *Proceedings of NSDI*, 2014.