Deferrability Analysis for JavaScript

Johannes Kloos[†], Rupak Majumdar[†] and Frank McCabe[*]

Sep 4, 2017
(†) MPI-SWS and (∗) Instart Logic

Modern web browsers allow asynchronous loading of JavaScript scripts in order to speed up parsing a web page. Instead of blocking until a script has been downloaded and evaluated, the *async* and *defer* tags in a script allow the browser to download the script in a background task, and either evaluate it as soon as it is available (for async) or evaluate it in load-order at the end of parsing (for defer). While asynchronous loading can significantly speed up the time-to-render, i.e., the time that passes until the first page elements are displayed on-screen, the specification for correct loading is complex and the programmer is responsible for understanding the circumstances under which a script can be loaded asynchronously in either mode without breaking page functionality. As a result, many complex web applications do not take advantage of asynchronous loading. We present an automatic analysis of web pages which identifies which scripts may be safely deferred, that is, deferred without any observable behavior on the page. Our analysis defers a script if every other script that has a transitive read or modification dependency does not access the DOM. We approximate access and modification sets using a dynamic analysis. On a corpus of 462 professionally developed web pages from Fortune 500 companies, we show that on average, we can identify two or three scripts to defer (mean; median: 1). On 18 pages, we find at least 11 deferrable scripts. Deferring these scripts can have notable impact on time-to-render: in 49 pages, we could show that the median improvement in time-to-render was at least 100ms, with improvements up to 890ms.

1 Introduction

Modern web applications use sophisticated client-side JavaScript programs and dynamic HTML to provide a low-latency, feature-rich user experience on the browser. As the scope and complexity of these applications grow, so do the size and complexity of the client-side JavaScript used by these applications. Indeed, web applications download an

average of 24 JavaScript files with about 346KB of compressed JavaScript[1]. In network-bound settings, such as the mobile web or some international contexts, optimizing the size and download time of the web page —which is correlated with user satisfaction with the application— is a key challenge.

One particular difficulty is the loading of JavaScript. The browser standards provide a complicated specification for parsing an HTML5 page with scripts [WHATWG, 2016]. In the "normal mode," parsing the page stops while the script is downloaded, and continues again after the downloaded script has been run. With tens of scripts and thousands of lines of code, blocking and waiting on JavaScript can significantly slow down page rendering and time to rendering. In order to speed up parsing, the Web Hypertext Application Technology Working Group (WHATWG), in its HTML5 specification, added "async" and "defer" tags to scripts. A script marked async is loaded in parallel with parsing through an asynchronous background process and run as soon as it is loaded. A script marked defer is also loaded in parallel with parsing, but it is evaluated only when parsing is complete, and in the order in which it was scheduled for download among other deferred scripts. The assumption is that async scripts do not interact with the rest of the page, and deferred scripts do not interact with other (non-deferred) scripts. It is up to the page designer to ensure there is no interaction. If the result of an asynchronously loaded script is required by another script, the programmer must put in explicit synchronization.

The HTML5 specification notes that the exact processing details for script-loading attributes are non-trivial, and involve a number of aspects of HTML5. Indeed, online forums such as Stack Overflow contain many discussions on the use of defer and async tags for page performance, but most end with unchecked rules of thumb ("make sure there are no dependencies") and philosophical comments such as: "[I]t depends upon you and your scripts."

At the same time, industrial users are interested in having a simple way to use these attributes. In this paper, we define an automatic *deferring transform*, which takes a page and marks some scripts deferred without changing observable behavior. We start by defining the notion of a *safe deferrable set*, comprising a set of scripts on a given page. If all the scripts in this set are loaded using the `defer` loading mode instead of synchronously, the user visible behavior of the page does not change. To make the idea of safe deferrable sets usable, we consider the semantics of a web page in terms of event traces, as defined by Raychev et al. [2013], and characterize the safe deferrable set using event traces. In particular, we can use event traces to define a dependency order between scripts, roughly describing the ordering of scripts due to dependencies between memory operations, and the notion of DOM-accessing scripts, which have user-visible behavior. The characterization of a safe deferrable set then (roughly) states that a set of scripts on a page is a safe deferrable set iff it contains no DOM-accessing scripts and is upward-closed under the dependency order.

We further refine the characterization by showing that if the set only contains deterministic scripts, the characterization can be checked by considering a single trace. Based

[1]See `http://httparchive.org/trends.php`, as of June 2017

on this refined characterization, we describe a dynamic analysis that conservatively computes a safe deferrable set for a given page.

The dynamic analysis proceeds by tracing the execution of a page using an instrumented browser, logging accesses to mutable state (including the JavaScript heap and the page DOM), non-deterministic behavior, task creation and execution (modeling the event-based execution model of web pages) and happens-before facts between tasks. We augment this information by using an off-the-shelf race detector to detect racing tasks. From the provided information, we can calculate a safe deferral set for the page, using the characterizations. The implementation of the analysis, JSDefer, is built on top of EventRacer [Raychev et al., 2013].

We evaluate our work by applying JSDefer to a corpus of 462 websites of Fortune 500 companies. We find that 295 (64%) of these web pages contain at least one deferrable script, with 65 (14%) containing at least 6 deferrable scripts. The maximum were 38 deferrable scripts, out of 133 scripts on that page. Furthermore, we find that while race conditions and non-determinism are widespread on web pages, we can easily identify a sufficient number of scripts that do not participate in races nor have non-deterministic behavior and are thus candidates for deferral. Finally, actually deferring scripts on these pages shows reasonable improvement in time-to-render (TTR) for 59 pages, where the median improvement of time-to-render was 198.5ms. The improvement in TTR was considered significant by our industrial collaborator, a company that provides fast application delivery on the web.

We summarize the contributions of this paper.

1. We describe a deferrability analysis, which checks which scripts can be marked as deferred without changing the observable behavior on the page.

2. We provide an extensive evaluation on a large corpus of professionally developed web sites to show that a significant portion of scripts can indeed be deferred. We show the potential for improving the load performance for these pages: in our experiments, the loading time improved by 193.4 ms (mean; median: 198.5 ms) on average and 891.4 ms in the best case. The improvement is considered significant by our industrial collaborator.

The supplementary material, available after deanonymization, will contain the complete data set, as well as the JSDefer implementation.

2 Background: Loading JavaScript

We briefly recall the WHATWG specification for loading HTML5 documents by a browser. A browser parses an HTML5 page into a data structure called the *document object model* (DOM) before rendering it on the user's screen. Parsing the document may require downloading additional content, such as images or scripts, whose links are provided in the document. The browser downloads images asynchronously, while continuing to parse the document. However, scripts are handled in a different way. The browser downloads scripts synchronously by default, making the parser wait for the

download, and evaluates the script before continuing to parse the page. Of course, this puts script download and parsing on the critical path of the rendering pipeline. Since network latency can be quite high (on the order of tens or hundreds of milliseconds) and script execution time may be non-negligible, this may cause noticeable delays in page loading. To allow asynchronous loading of scripts, the WHATWG specification allows two Boolean attributes in a `script` element, *async* and *defer*. In summary, there are three loading strategies for scripts:

- Synchronous loading. When encountering a `script` tag with no special attributes, the browser suspends parsing, downloads the script synchronously, and evaluates it after download is complete. Parsing continues after the evaluation of the script.
- Asynchronous loading. When encountering a `<script src="..."async>` tag, the browser starts an asynchronous download task for the script in the background but continues parsing the page until the script has been loaded. Then, parsing is suspended and the script is evaluated before continuing with parsing.
- Deferred loading. When encountering a `<script src="..."defer>` tag, the browser starts a download task for the script background but continues parsing the page. Once parsing has finished and the script has been downloaded, it is evaluated. Moreover, scripts are evaluated in the order that their corresponding script tags were parsed in the HTML, even though a later script may have finished downloading earlier.

The precise description of the different modes can be found in section 4.12 of the WHATWG specification. It spans over ten pages, and distinguishes between five different loading modes.

While asynchronous or deferred loading is desirable from a performance perspective, it can lead to *race conditions*, that is, the output of the page may depend on the order in which scripts are executed [Raychev et al., 2013]. Consider the following example:

```
<html><body><script src="http://www.foo.com/script1.js"></script>
<script>if (!script1executed) { alert("Error!"); }</script></body></html>
```

where `script1.js` is simply `script1executed = true;`. As the script is loaded synchronously, the code has no race (yet): the `alert` function will never be called.

If we annotate the first script with the `async` tag, we introduce a race condition. Depending on how fast the script is loaded, it may get executed before or after the inline script. In case it gets executed later, an alert will pop up, noting that the external script has not been loaded yet. Changing the loading mode to defer does not cause a race, per se, but now the alert always pops up; thus deferred loading of the script changes the observable behavior from the original version.

Another kind of race condition is incurred by scripts that perform certain forms of DOM accesses. For instance, consider the following page:

```
<html><body><script src="http://www.foo.com/script2.js"></script>
            <span id="marker">Something</span></body></html>
```

where `script2.js` uses the DOM API to check if a tag with id `marker` exists. Loaded synchronously, the outcome of this check will always be negative. Asynchronous loading would make it non-deterministic, while deferred loading will remain deterministic but the check will always be positive.

Our goal is to analyze a web page and add `defer` tags to scripts, wherever possible. To ensure we can load scripts safely in a deferred way, we need to make certain that deferred loading does not introduce races through program variables or the DOM and does not change the observable behavior. Next, we make this precise.

3 Deferrability analysis

Take an HTML5 page that includes one or more JavaScript scripts, some of which are loaded synchronously. When is it safe to load one or more of the synchronously loaded scripts in defer mode instead? To answer this question, we describe criteria in terms of trace semantics, and provide a dynamic analysis that calculates an under-approximation of the set of safely deferrable scripts.

In the following, suppose we are given a web page with scripts $s_1, \ldots, s_n$ (in order of appearance). For this exposition, we assume that all the scripts are loaded synchronously; the extension to pages with mixed loading modes and inline scripts is straightforward.

On a high level, our goal is to produce a modified version of the page where some of the scripts are loaded deferred instead of synchronously, but the visible behavior of the page is the same. Concretely, when loading and displaying the page, the browser constructs a view of the page by way of building a DOM tree, containing both the visible elements of the page and the association of certain event sources (e.g., form fields or `onload` properties of images) with handler functions. Concretely, the DOM tree is the object graph reachable from `document.root` which consists of objects whose type is a subtype of `Node`; compare WHATWG [2016]. This DOM tree is built in stages, adding nodes to the tree, modifying subtrees and attaching event handlers. This can be due to parsing an element in the HTML document, receiving a resource, user interaction, or script execution.

Definition 1 A *DOM trace* consists of the sequence of DOM trees that are generated during the parsing of a page. The *DOM behavior* of a page is the set of DOM traces that executing this page may generate.

Note that even simple pages may have multiple DOM traces; for instance, if a page contains multiple images, any of these images can be loaded before the others, leading to different intermediate views.

Definition 2 For a page p with scripts $s_1, \ldots, s_n$, and a set $D \subseteq \{s_1, \ldots, s_n\}$ let p' be the page where the members of D are loaded deferred instead of synchronously. We say that D is a *safe deferral set* if the DOM behavior of p' is a subset of the DOM behavior of p.

3.1 Background: Event traces and races in web pages

We recall an event-based semantics of JavaScript [Petrov et al., 2012, Raychev et al., 2013, Adamsen et al., 2017] on which we build our analysis; we follow the simplified presentation of Adamsen et al. [2017]. For a given execution of a web page, fix a set of events E; each event models one parsing action, user interaction event or script execution (compare also the event model of HTML, WHATWG [2016]). Our semantics will be based on the following operations:

- $rd(e, x)$ and $wr(e, x)$: These operations describe that during the execution of event $e \in E$, some shared object x (which may be a global variable, a JavaScript object, or some browser object, such as a DOM node) is read from or written to.
- $post(e, e')$: This operation states that during the execution of event $e \in E$, a new event $e' \in E$ is created, to be dispatched later (e.g., by setting a timer or directly posting to an event queue).
- $begin(e)$ and $end(e)$: These operations function as brackets, describing that the execution of event $e \in E$ starts or ends.

A *trace* of an event-based program is a sequence of *event executions*. An event execution for an event e is a sequence of *operations* such that the sequence starts with a begin operation $begin(e)$, the sequence ends with an end operation $end(e)$, and otherwise consists of operations of the form $rd(e, x)$, $wr(e, x)$, and $post(e, e')$ for some event $e' \in E$. For a trace of a program consisting of event executions of events $e_1, e_2, \ldots, e_n$, by abuse of notation, we write $t = e_1 \ldots e_k$.

Furthermore, we define a *happens-before relation*, denoted hb, between the events of a trace. It is a pre-order (i.e., reflexive, transitive, and anti-symmetric) and $e_i \,\mathsf{hb}\, e_j$ holds in two cases: if there is an operation $post(e_i, e_j)$ in the trace, or if e_i and e_j are events created externally by user interaction and the interaction creating e_i happens before that for e_j.

Two events e_i and e_j are *unordered* if neither $e_i \,\mathsf{hb}\, e_j$ nor $e_j \,\mathsf{hb}\, e_i$. They have a race if they are unordered, access the same shared object, and at least one access is a write.

3.2 When is a set of scripts deferrable?

To make the deferrability criterion given above more tractable, we give a sufficient condition in terms of events. A sketch of the correctness proof is given in Section 5. We first define several notions on events, culminating in the notion of the *dependency order* and the *DOM-modifying script*. We use these two notions to give the sufficient condition.

Consider a page with scripts $s_1, \ldots, s_n$. For each script s_i, there is an event e_{s_i} which corresponds to the execution of s_i. By abuse of notation, we write s_i for e_{s_i}.

We say that e *posts* e' if $post(e, e')$ appears in the event execution of e. We say that e *transitively posts* e' if there is a sequence $e = e_1, \ldots, e_k = e'$ such that s posts e_1 and e_i posts e_{i+1}. This explicitly includes the case $e = e'$ (i.e., we take a reflexive-transitive closure).

Suppose script s transitively posts event e. We call e a *near event* if, for all scripts s', $s \,\mathsf{hb}\, s'$ implies $e \,\mathsf{hb}\, s'$. Otherwise, we call e a *far event*. We say that a script s is *DOM-accessing* iff there is a near event e such that e reads from or writes to the DOM.

Now, consider two events e_i and e_j such that $i < j$. We say that e_i *must come before* e_j iff both e_i and e_j access the same object (including variables, DOM nodes, object fields and other mutable state) and at least one of the accesses is a write. For two scripts s_i and s_j, $i < j$, we say that s_i must come before s_j iff there is a near events $e_{i'}$ of s_i and an event $e_{j'}$ such that $s_j \,\mathsf{hb}\, e_{j'}$ and $e_{i'}$ must come before $e_{j'}$.

The dependency order $s_i \preceq s_j$ is then defined as the reflexive-transitive closure of the must-come-before relation.

The proofs of the following theorems can be found in Section 5.

Theorem 1 *Let p be a page with scripts $s_1, \ldots, s_n$ and $D \subseteq \{s_1, \ldots, s_n\}$. If the following two conditions hold:*

1. *If $s_i \in D$, then script s_i is not DOM-accessing in any execution.*
2. *If $s_i \in D$ and $s_i \preceq s_j$ in any execution, then $s_j \in D$.*

then D is a safe deferral set.

The gist of the proof is that all scripts whose behavior is reflected in the DOM trace are not deferred and hence executed in the same order (even with regard to the rest of the document). Due to the second condition, each script starts in a state that it could start in during an execution of the original page, so its behavior with regard to DOM changes is reflected in the DOM behavior of the original page.

The distinction between near and far events comes from an empirical observation: when analyzing traces produced by web pages in the wild, script-posted events clearly separate in these two classes. Near events are created by the `dispatchEvent` function, or using the `setTimeout` function with a delay of less than 10 milliseconds. On the other hand, far events are event handlers for longer-term operations (e.g., XMLHttpRequest), animation frame handlers, or created using `setTimeout` with a delay of at least 100 milliseconds. There is a noticeable gap in `setTimeout` handlers, with delays between 10 and 100 milliseconds being noticeably absent.

We make use of this observation by treating a script and its near events as an essentially sequential part of the program, checking the validity of this assumption by ensuring that the near events are not involved in any races.

This allows us to formulate a final criterion, which can be checked on a single trace:

Theorem 2 *Let page p and set D be given as above, and consider a single trace executing events $e_1, \ldots, e_n$. Suppose the following holds:*

1. *If e is a near event of s and accesses the DOM, $s \notin D$.*
2. *If e is involved in a race or has non-deterministic control flow, $s \,\mathsf{hb}\, e$ and s' happens before s in program order (including $s = s'$), then $s' \notin D$.*

3. D is $\preceq$-upward closed.

Then D is a safe deferral set.

The key idea of this proof is that all scripts in D are "deterministic," so the conditions of the previous theorem collapse to checking a unique trace of the script.

In fact, the second condition of the theorem can be weakened a bit, as can be seen in the proof: it is sufficient to consider those s' who access objects that e accesses. Thus, if we have an over-approximation of the objects that e accesses, we can prune the set of scripts that must be marked non-deferrable.

3.3 JSDefer: A dynamic analysis for deferrability

The major obstacle in finding a deferrable set of scripts is the handling of actual JavaScript code, which cannot be feasibly analyzed statically. This is because of the dynamic nature of the language and its complex interactions with browsers, including the heavy use of introspection, `eval` and similar constructs, and variations in different browser implementations. For example, we found a script which takes the string representation of a function (which gives its source code) and performs regular expression matching on it; static analysis cannot hope to cope with this. In the following, we present a dynamic analysis for finding a safe deferral set that we call *JSDefer*.

Assumption: For reasons of tractability, we assume in this paper that no user interaction occurs before the page is fully loaded. This is because it is well-known that early user interaction is often not properly handled; in fact, Adamsen et al. [2017] devoted an entire paper to showing how to fix errors due to early user interaction by dropping or postponing events due to user input. Hence, we assume that early user interaction does not occur (or is pre-processed as in Adamsen et al. [2017] before being analyzed by JSDefer).

With this assumption at hand, as reasoned above, we only need to consider scripts themselves and their near events; we call this the *script initialization code*. This part of the code is run during page loading and, empirically is "almost deterministic": it does not run unbounded loops and, for the most part, only exhibits limited use of non-determinism. We provide experimental evidence for this observation below. For this reason, it would be feasible to collect all possible execution traces dynamically (by controlling all sources of non-determinism), and to derive all required information from that finite set of traces. In fact, we only collect a single trace and aggressively mark any scripts that may exhibit non-deterministic control flow.

JSDefer piggybacks on instrumented browsers. In particular, we used a modified version of the instrumented Webkit browser from the EventRacer project Raychev et al. [2013] to generate a trace, including a happens-before relation. For now, we use a simple, not entirely sound heuristic to detect non-deterministic behavior: We extended the instrumentation to also include coarse information about scripts getting data from non-deterministic and environment-dependent sources. In particular, we mark all calls to the random number generator, functions returning the current time and data, and various

properties about the page state. We used the official browser interface descriptions (given in WebIDL) and the JavaScript specification to ensure completeness of marking.

We perform deferrability analysis on the collected trace, using the following steps:

1. Perform a race analysis on the trace; our implementation uses EventRacer.
2. For each event in the trace, check whether it is a near event for some script.
3. Sequentialize the near events for each script into a single event. Call the resulting events *script events*.
4. Calculate the race sets for each script event as the union of race sets for its involved script execution event and near events.
5. Calculate read and write sets for each script event, as well as predicates that check if a script event is DOM-accessing or accesses sources of non-determinism. Additionally compute read-sets for all far events.

   Here, the read set contains all the objects that have been read by the event without a preceding write by the event, and the write set contains all objects on which a write has been performed.
6. Using the read and write sets, calculate the dependency relation between scripts. This uses the read and write sets from the previous step.
7. Compute the set of deferrable scripts using the criterion of Theorem 2.

This calculation computes a safe deferrable set, although we make no claims with regard to maximality. In a final step, we rewrite the top-level HTML file of the page to add defer annotations to all scripts in the deferrable set.

4 Evaluation

We evaluated JSDefer on the websites of the Fortune 500 companies [fortune] as a corpus. To gather deferrability information, we used an instrumented WebKit browser to generate event traces. For each website, we ran the browser with a timeout of 30s, with a 1s grace period after page loading to let outstanding events settle. In this way, we could collect traces for 462 web pages; in the other cases, either the browser could not handle the page or the page would not load. We then run JSDefer on each of the collected traces. The analysis was successful on all traces.

Rewriting the HTML files based on the safe deferral set worked on 460 pages; the other two pages contained invalid HTML that could not be handled by the tool. For 11 of the pages, we did not find JavaScript on the page; later analysis showed that at the time of trace collection, we only received an error page from these sites. Due to the large amount of data already collected, we dropped these pages from the corpus. All in all, the main part of the analysis comprises 451 web pages.

In the evaluation, we want to answer five main questions:

1. How much is deferred and asynchronous loading already used? How is it used?
2. Are our assumptions about determinism justified?
3. How many deferrable scripts can we infer?
4. What kind of scripts are deferrable?
5. Does deferring these scripts gain performance?

4.1 Tools and environment

For the experiments, we need several tools.

Instrumented browser. The instrumented browser is used to collect the trace information for the dynamic analysis. To achieve this, we extended the instrumented WebKit browser from the EventRacer project and added a simple analysis for non-determinism. The log file produced by this browser contains all the information that EventRacer uses, plus additional information on the generation of non-deterministic and environment-dependent values.

Deferrability analysis tool. This tool is the core component of JSDefer and performs actual deferrability analysis. It reads the event log from above, and produces a human-readable report detailing the following:

1. Which scripts are loaded on the page, from where, and in which way (inline synchronous, async, deferred, inserted at runtime)?
2. For each script, does it use non-deterministic or environment-dependent values? For this analysis, we take both the execution of the main body of the script and its near events into account; we call the execution of this the *script initialization*.
3. Which scripts initializations are involved in race conditions? We list the races identified by EventRacer that contain at least one script initialization event.
4. The list of scripts that can be deferred, given by location in the HTML file and script path. This list is also output in a machine-readable format.

Web page instrumentation and performance measurement. For the performance measurement, we created modified versions of the the web pages in our corpus which included the additional defer annotations. This was performed using a simple tool that reads the list of deferred scripts and modifies the top-level HTML file. We then used a proxy to (a) intercept the downloads of the main HTML files, giving the deferred and non-deferred version depending on a HTTP header, and (b) serve all resource files from the origin server.

The measurements were performed using a version of the WebPageTest tool [Viscomi et al., 2015]. We used shaping to simulate a connection over an LTE network, as to simulate a realistic usage scenario of a mobile user accessing the pages in our corpus.

Async or defer	#pages
Neither	32
Defer only	0
Async only	404
Only scripts included using standard snippets	256
Others	148
Both	15

Table 1: Number of pages in the corpus that use async or defer. The sub-classification of async scripts was done manually, with unclear cases put into "others".

Environment The trace collection and page analysis steps were performed on a machine with 48 Xeon E7-8857 CPU and 1536 GiB of memory, running Debian Linux 8 (kernel 4.4.41). The proxy set-up was run on a smaller machine (24 Xeon X5650 CPUs, 48 GiB memory) with the same OS; the WebPageTest instance was run by our industrial collaborator in their performance-test environment.

4.2 How are async and defer used so far?

As a first analysis step, we analyzed if pages were using async and defer annotations already, and in which situations this was the case. The numbers are given in Table 1.

The first observation from the numbers is that defer is very rarely used, while there is a significant numbers of users of async. Further analysis shows many of these asynchronous scripts come from advertising, tracking, web analytics, and social media integration. For instance, Google Analytics is included in this way on at least 222 websites[2]. Another common source is standard frameworks that include some of their scripts this way. In these cases, the publishers provide standard HTML snippets to load their scripts, and the standard snippets include an async annotation. On the other hand, 254 pages include some of their own scripts using async. In some pages, explicit dependency handling is used to make scripts capable of asynchronous loading, simulating a defer-style loading process.

4.3 Are our assumptions justified?

The second question is if our assumptions about non-determinism are justified. We answer it in two parts, first considering the use of non-deterministic functions, and then looking at race conditions.

Non-determinism: To perform non-determinism analysis, we used a browser that was instrumented for information flow control. This allowed us to identify scripts that actually use non-deterministic data in a way that may influence other scripts, by leaking

[2]Many common scripts are available under many aliases, so we performed a best-effort hand count.

non-deterministic data or influencing the control flow. We considered three classes of non-determinism sources:

1. `Math.random`. For most part, this function is used to generate unique identifiers, but we found a significant amount of scripts that actually use this function to simulate stochastic choice.

2. `Date.now` and related functions. These functions are included since their result depends on the environment. We found that usually, these functions are called to generate unique identifiers or time stamps, and to calculate time-outs.

   Nevertheless, we found examples for which it would not be feasible to automatically detect safety automatically. For instance, we found one page that had a busy-wait loop in the following style:

   ```
   var end = Date.now() + timeout;
   while (Date.now() < end) {}
   ```

   While it is easy to see manually that this code would not influence deferrability decisions, an automatic analysis would have to perform quite some work.

3. Functions and properties about the current browser state, including window size, window position and document name. While we treat these as a source of non-determinism, it would be better to classify them as environment dependent values; we find that in the samples we analyzed, they are not used in way that would engender non-determinism. Rather, they are used to calculate positions of windows and the like.

As it turns out, many standard libraries make at least some use of non-determinism. For instance, jQuery and Google's analytics and advertising libraries generate unique identifiers this way.

Additionally, many scripts and libraries have non-deterministic control flow. We found 1704 cases of scripts with non-deterministic control flow over all the pages we analyzed. That being said, this list contains a number of duplicates: In total, at least 546 of these scripts were used more than once[3]. They form 100 easily-identified groups, the largest of which are Google Analytics (with two script names), accounting for 187 of these scripts, various versions of jQuery from various providers (40 instances) and YouTube (20 instances).

More importantly, we analyzed how many of the scripts we identified as deferrable have non-deterministic control flow. As it turns out, there was no overlap between the two sets: Our simple heuristic of scripts calling a source of non-determinism was sufficient to rule out all non-deterministic scripts.

[3]We clustered by URL (dropping all but the last two components of the domain name and all query parameters), which misses some duplicates

Table 2: Number of deferrable scripts. This includes pages with no scripts.

Number of deferrable scripts	On how many pages?
0	167 (156 excluding pages without scripts)
1	86
2	55
3–5	89
6–10	47
more than 10	18

Race conditions: We additionally analyzed whether non-determinism due to race conditions played a role. In this case, the findings were, in fact, simple: While there are numerous race conditions, they all occur between far events. We did not encounter any race conditions that involved a script (i.e., the execution of the body of the script) or its near events.

One further aspect is that tracing pages does not exercise code in event handlers for user inputs. This may hide additional dependencies and race conditions. As reasoned above, we assume that no user interaction occurs before the page is loaded (in particular, after deferred scripts have run). The reasoning for this was given above; we plan to address this limitation in further work.

4.4 Can we derive deferrability annotations for scripts?

To evaluate the potential of inferring deferrability annotations, we used the analysis described above to classify the scripts on a given page into five broad classes:

- The script is loaded synchronously and can be deferred,
- The script is already loaded with defer or async (no annotation needs to be inferred here);
- The script is an inline script; in this case, deferring would require to make the script external, with questionable performance impact;
- The script is not deferrable since it performs DOM writes;
- The script is not deferrable because it is succeeded by a non-deferrable script in the dependency order.

The general picture is that the number of deferrable scripts highly depends on the page being analyzed. 295 of all pages contain deferrable scripts, and 209 of all pages permit deferring multiple scripts. Moreover, on 18 of the pages considered, at least 11 scripts can be deferred. Among these top pages, most have between 11 and 15 deferrable scripts (4 with 11, 2 with 12, 4 with 13, 5 with 15), while the top three pages have 16, 17 and 38 deferrable scripts on them.

Further analysis shows that some pages have been hand-optimized quite heavily, so that everything that could conceivably be deferred is already loaded with defer or async. Conversely, some pages have many scripts that can be deferred. We also find that it is quite common that some scripts will not be deferred because non-deferrable scripts depend on them. In many cases, these dependencies are hard ordering constraints: For instance, jQuery is almost never deferrable since later non-deferrable scripts will use the functionality it provides.

We also analyzed what percentage of scripts are deferrable on a given page. Discarding the pages that had no deferrable scripts on them (corresponding to a percentage of 0), we get the following picture:

Percentage of deferrable scripts	On how many pages?
$< 10\%$	180
$10 - 20\%$	56
$20 - 30\%$	37
$30 - 40\%$	14
$40 - 50\%$	6
$50 - 60\%$	1
$60 - 70\%$	1

That being said, we observe some spurious dependencies between scripts; this indicates room for improvement of the analysis. As an example, consider the jQuery library again. Among other things, it has a function for adding event handlers to events. Each of these event handlers is assigned a unique identifier by jQuery. For this, it uses a global variable `guid` that is incremented each time an event handler is added; clients treat the ID as an opaque handle. Nevertheless, if multiple scripts attach event handlers in this way, there is a an ordering constraint between them due to the reads and writes to `guid`, event though the scripts may commute with each other.

Looking at the pages with a high number of deferrable scripts, we find that there are two broad classes that cover many deferrable scripts: "Class definitions", which create or extend an existing JavaScript object with additional attributes (this would correspond to class definitions in languages such as Java), and "poor man's deferred scripts", which install an event handler for one of the events triggered at page load time (load, DOMContentLoaded and jQuery variants thereof) and only then execute their code.

4.5 Does deferring actually gain performance?

Since we found a significant number of scripts that can actually be deferred, we also measure how performance and behavior is affected by adding defer annotations. As described above under "tools and environment", we used a proxy-based setup to present versions of each web page with and without the additional defer annotations from deferrability analysis to WebPageTest. We then measured the time-to-render (i.e., the time from starting the page load to the first drawing command of the page) for each version of each page, We choose time-to-render as the relevant metric because the content delivery industry uses it as the best indicator of the user's perception of page speed. This belief

is supported by studies, e.g. Gao et al. [2017].

We took between 38 and 50 measurements for each case, with a median of 40. The measurements were taken for each page that had at least one deferrable script and could successfully be rewritten. We had to exclude six pages (amerisync, erieinsurance, kindredhealthcare, monsanto, usbank, westerunion) since at the time of the measurement, JSDefer could not yet handle these pages.

Another issue that we encountered is that many pages force a connection upgrade to SSL. In a separate analysis, we polled all 500 pages in the corpus to see how many of them force SSL upgrades; in total, 475 pages were reachable, and out of these, 209 forced an upgrade. Since our measurement setup could not deal with interposing on SSL connections, the data from measurements on these sites should be considered suspect, since some resources on these pages may not have been loaded correctly. For this reason, we threw out the data corresponding to pages that force SSL connections, or use embedded SSL links that we would have to interpose. In the end, we considered 169 pages that had deferrable scripts on them and did not force SSL upgrades or contain explicit SSL links.

The first observation to make is that the load time distribution tends to be highly non-normal and multi-modal. This can be seen in a few samples of load time distribution, as shown in Fig. 1. The violin plots in these graphs visualize an approximation of the probability distribution of the loading time for each case.

For this reason, we quantify the changes in performance by considering the *median change in time-to-render* for each page, meaning we calculate the median of all the pairwise differences in time-to-render between the modified and the unmodified version of the page. This statistic is used as a proxy for the likely change in loading time by applying JSDefer. In the following, we abbreviate the median change in time-to-render as MCTTR. We additionally use the Mann-Whitney U test to ensure that we only consider those cases where MCTTR gives us statistically significant results.

Out of the 169 considered pages, 66 had a statistically significant MCTTR.

The actual median changes are shown in Fig. 2, together with confidence intervals. The data is also given in Table 3. This table also contains the median TTR of the original page. Several things are of note here:

1. As promised in the introduction, the median improvement in TTR is 198.5ms in the examples provided, while their median load time is 3097ms.

2. Most of the pages that pass the significance test have positive MCTTR, meaning that applying JSDefer provides benefits to time-to-render: For 59 pages, JSDefer had a positive effect, versus 7 pages where it had a negative effect. (85 versus 14 including SSL pages).

3. 49 of the pages in our sample have an estimated MCTTR of at least 100ms=0.1s. This difference corresponds to clearly perceptible differences in time-to-render.

   Even when taking the lower bound of the 95% confidence interval, 32 of the pages still have this property.

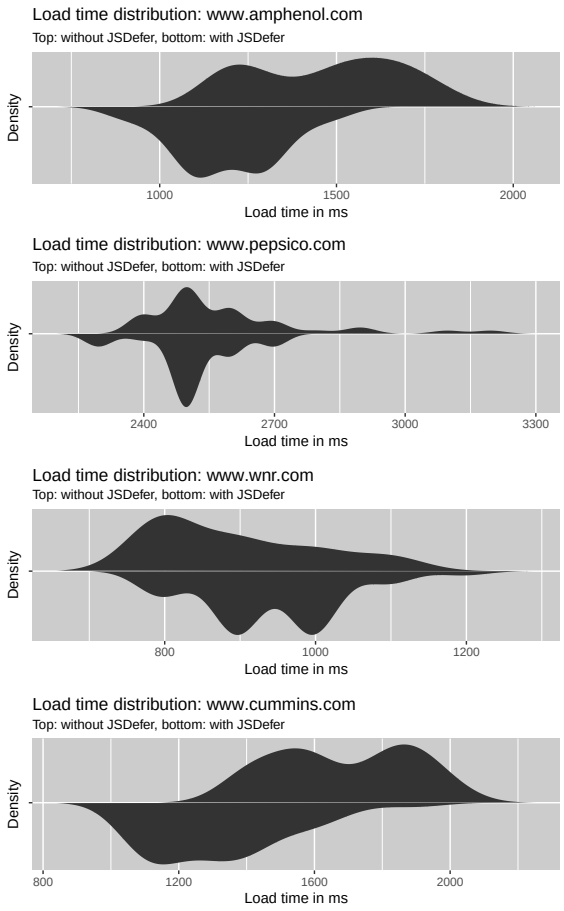

Figure 1: Violin plots of load time distributions for some pages, before and after applying JSDefer. The distribution is a smoothed representation of the (discrete) distribution of the sample.

4. For 7 pages, we get a negative MCTTR, corresponding to worse loading time. This indicates that JSDefer should not be applied blindly.

We tried to analyze the root causes for the worsening of load times. For this, we used Chrome Developer Tools to generate a time-line of the page load, as well as a waterfall diagram of resource loading times. The results were mostly inconclusive; we could observe that the request for loading some scripts on two of these pages was delayed, and conjecture that we are hitting edge cases in the browser's I/O scheduler.

Another observation that can be made by analyzing the violin plots is that JSDefer sometimes drastically changes the loading time distribution of pages, but there is no clear pattern. The interested reader may want to see for themselves by looking at the complete set of plots in the supplementary material.

An interesting factor in the analysis was the influence of *pre-loading*: For each resource (including scripts) that is encountered on a page, as soon as the reference to the script is read (which may well be quite some time before "officially" parsing the reference), a download task for that resource is started[4], so that many download tasks occur in parallel. This manifests itself in many parallel downloads, often reducing latency for downloads of scripts and resources. This eats up most of the performance we could possibly win; preliminary experiments with pre-loading switched off showed much bigger improvements. Nevertheless, even in the presence of such pre-loading, we were able to gain performance. We also performed some timing analysis of page downloads to understand how performance is gained or lost, and found that the most important factor is, indeed, the time spent waiting for scripts to become available. The time saved by executing scripts later was only a minor factor.

Finally, to judge the impact of the improvements we achieved, we discussed the results with our industrial collaborator. Instead of considering the MCTTR, they analyzed the violin plots directly, and they indicated that they consider the improvement that JSDefer can achieve to be significant.

4.6 Threats to validity

There are some threats to validity due to the set-up of the experiments.

1. External validity, questions 2–5: Websites often provide different versions of their website for different browsers, or have browser-dependent behavior. Since we use a specific version of WebKit for our instrumented browser, results for different browsers may differ.

   In practice, one would address this by providing different versions of the website as well. An efficient way of doing this is part of further work.

2. External validity, all questions: The selection of the corpus is not entirely random. That being said, the pages in the corpus were not chosen by technical criteria, and manual examination of some samples indicate that the structure and code quality is quite diverse.

[4] Glossing over the issue of connection limits

Table 3: MCTTR values for pages with significant MCTTR, sorted by ascending MCTTR. All times are given in milliseconds.

Page	MCTTR	MCTTR 95% confidence interval	Median TTR of original page
www.williams.com	-452.0	[-698.0,-201.0]	2300.0
www.visteon.com	-401.0	[-899.0,-99.0]	6996.0
www.mattel.com	-401.0	[-900.0,-1.0]	3995.0
www.statestreet.com	-299.0	[-400.0,-100.0]	2596.0
www.fnf.com	-201.6	[-500.0,-1.0]	3896.0
www.cbscorporation.com	-99.0	[-100.0,0.0]	1296.0
www.wnr.com	-98.0	[-100.0,0.0]	895.0
www.lansingtradegroup.com	98.6	[1.0,118.0]	2597.0
www.kiewit.com	99.0	[0.0,101.0]	1096.0
www.emcorgroup.com	99.0	[0.0,201.0]	1696.0
www.dovercorporation.com	99.0	[0.0,100.0]	1896.0
www.domtar.com	99.0	[1.0,100.0]	1896.0
www.eogresources.com	99.0	[0.0,100.0]	1896.0
www.johnsoncontrols.com	99.0	[0.0,101.0]	3296.0
www.altria.com	99.0	[0.0,101.0]	499.0
www.jmsmucker.com	99.0	[0.0,199.0]	996.0
www.itw.com	99.0	[1.0,100.0]	1295.0
www.walgreensbootsalliance.com	100.0	[1.0,101.0]	1096.0
www.bostonscientific.com	100.0	[1.0,101.0]	1297.0
www.apachecorp.com	100.0	[0.0,199.0]	1396.0
www.lifepointhealth.net	100.0	[99.0,100.0]	1396.0
www.marathonoil.com	100.0	[99.0,101.0]	1097.0
www.cstbrands.com	100.0	[99.0,199.0]	1897.0
www.mohawkind.com	101.0	[100.0,200.0]	1496.0
www.delekus.com	101.0	[98.0,200.0]	1795.0
www.stanleyblackanddecker.com	103.0	[100.0,199.0]	1196.0
www.fanniemae.com	112.3	[1.0,296.0]	2999.0
www.citigroup.com	114.0	[99.0,201.0]	1296.0
www.microsoft.com	130.0	[14.0,206.0]	1455.0
www.pultegroupinc.com	139.0	[93.0,219.0]	1120.0
www.mosaicco.com	196.0	[100.0,200.0]	1496.0
www.tysonfoods.com	198.0	[100.0,280.0]	1796.0
www.iheartmedia.com	198.0	[1.0,300.0]	1696.0
www.rrdonnelley.com	199.0	[104.0,201.0]	2097.0
www.raytheon.com	199.0	[0.0,401.0]	1697.0
www.navistar.com	199.6	[53.0,318.0]	2740.0
www.genesishcc.com	200.0	[1.0,399.0]	4497.0
www.chs.net	200.0	[100.0,298.0]	1796.0
www.newellbrands.com	200.0	[100.0,299.0]	1197.0
www.navient.com	200.0	[0.0,304.0]	2597.0
www.ncr.com	200.0	[96.0,300.0]	2096.0
www.sempra.com	200.0	[100.0,300.0]	1696.0
www.univar.com	200.0	[101.0,300.0]	1496.0
www.avoncompany.com	200.0	[100.0,300.0]	1596.0
www.pricelinegroup.com	200.0	[199.0,201.0]	1596.0
www.pacificlife.com	201.0	[100.0,399.0]	3296.0
www.weyerhaeuser.com	242.2	[200.0,300.0]	2497.0
www.techdata.com	298.0	[100.0,303.0]	2296.0
www.tenneco.com	299.0	[200.0,300.0]	1896.0
www.dana.com	299.0	[200.0,300.0]	1496.0
www.cablevision.com	299.0	[298.0,300.0]	2196.0
www.amphenol.com	300.0	[200.0,400.0]	1496.0
www.calpine.com	300.0	[201.0,302.0]	2098.0
www.nov.com	300.0	[103.0,498.0]	3396.0
www.harman.com	303.0	[300.0,400.0]	2195.0
www.burlingtonstores.com	395.0	[200.0,501.0]	4179.0
www.centene.com	398.0	[308.0,412.0]	2306.0
www.cummins.com	398.9	[299.0,496.0]	1695.0
www.markelcorp.com	500.0	[498.0,501.0]	1596.0
www.spectraenergy.com	501.0	[499.0,600.0]	2395.0
www.spiritaero.com	598.0	[499.0,601.0]	1797.0
www.wholefoodsmarket.com	611.7	[412.0,790.0]	2138.0
www.deanfoods.com	700.0	[401.0,3900.0]	3796.0
www.mutualofomaha.com	702.0	[700.0,800.0]	2396.0
www.lkqcorp.com	800.0	[700.0,900.0]	3301.0
www.ppg.com	891.4	[514.0,1299.0]	5096.0

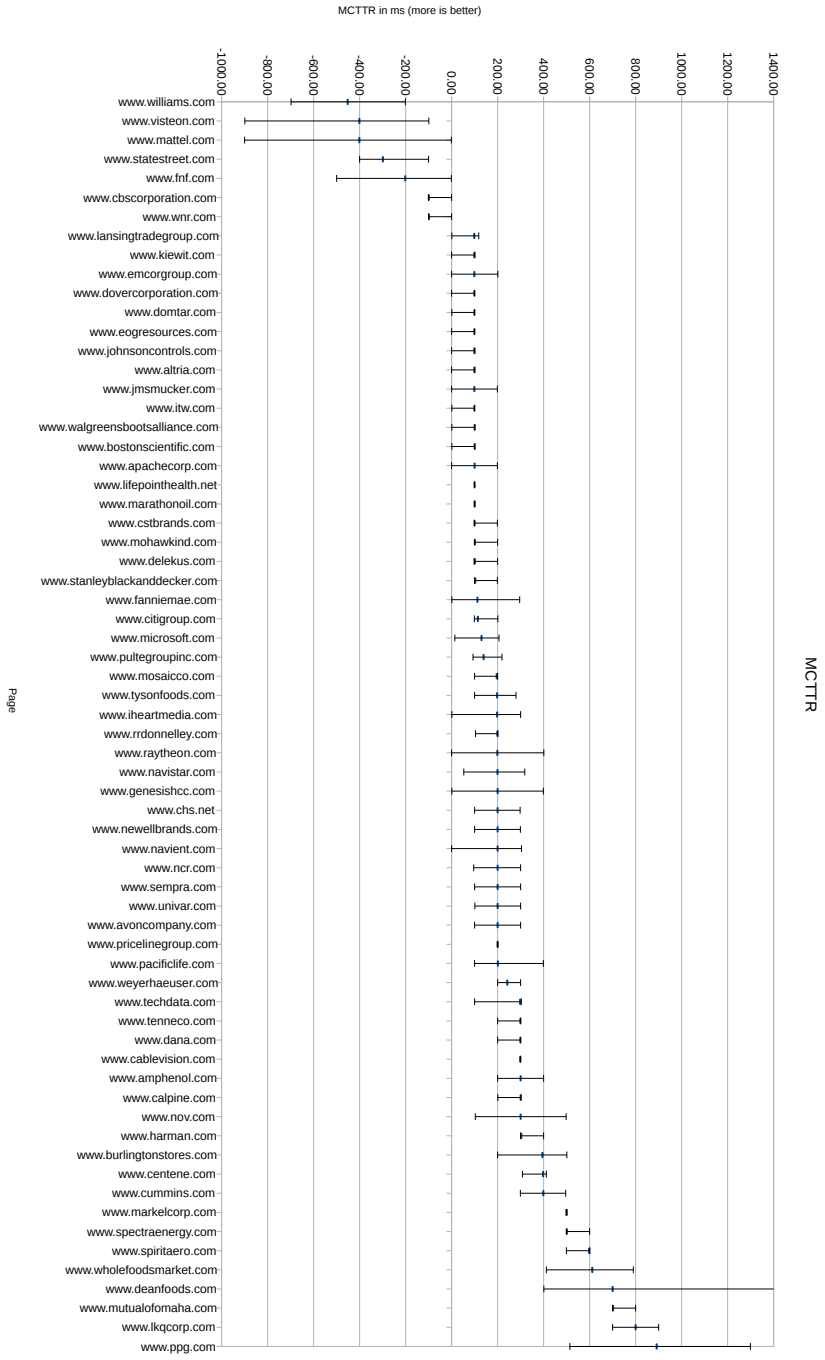

Figure 2: MCTTR values for pages with significant MCTTR. This is a visualization of Table 3.

3. Internal validity, question 5: We could not completely control for network delays in the testing set-up.

4. Internal validity, question 2: Due to the set-up of the analysis, we could not ensure that the pages did not change between analysis steps. Thus, in the non-determinism matching step, we may have missed cases. We did cross-check on a few samples, but could not do so exhaustively.

5. Internal validity, question 5: The SSL issue already described above. To ensure validity, we reported data from a known-good subset as well as the complete data.

5 Soundness of the analysis

In this section, we sketch the proofs of theorems 1 and 2. We deliberately leave out details of the semantics of web pages, deferring to browser implementations. It can be made more precise by giving a formal specification of the semantics of JavaScript and HTML, as executed by a specific browser. We omit such a specification since it is out of scope for this paper, and would engender a research project in its own right. For some progress towards such semantics, compare Bodin et al. [2014], Bohannon and Pierce [2010], Petrov et al. [2012], Guha et al. [2010] et cetera.

The main ingredient will be two notions. One is commutativity, as defined in Rinard and Diniz [1996]: "Two operations *commute* if they generate the same final result regardless of the order in which they execute". The other is the notion of observational equivalence.

Definition 3 (DOM traces, DOM behavior, observational equivalence and commutativity) Let a web page be given.

1. The *DOM state* is defined as above.

2. The *page state* is the part of the browser state at a given point in time that contains all the state accessible by a JavaScript script on the page. The page state includes the DOM state as well as the JavaScript environment of the page.

3. A *DOM trace* is a sequence of DOM states that can be constructed by executing the page and appending the current DOM state to the sequence whenever it differs from the last state in the sequence (compare Definition 1).

   The DOM trace of an event is defined similarly by considering only the execution of the event; the DOM trace of a sequence of events is defined in the same way.

4. The *DOM behavior* of a page is the collection of all possible DOM traces for that page. The DOM behavior of events and sequences of events is defined analogously.

5. A page p_1 *observationally refines* a page p_2 iff the DOM behavior from p_1 is included in the DOM behavior of p_2. The same holds for events sequences.

   p_1 and p_2 are *observationally equivalent* iff they observationally refine each other.

6. Two events e_1 and e_2 *commute* if the following two conditions holds starting from any page state s: First, starting from state s, $e_1; e_2$ and $e_2; e_1$ have the same DOM behavior. Second, starting from s, executing $e_1; e_2$ can produce state s' iff executing $e_2; e_1$ can produce state s'.

Also, let p be a page and D a set of scripts on p. Let p_D be a version of the page with the scripts in D loaded `defer`. Then D is a safe deferral set if p_D observationally refines p.

With these definitions, we can sketch the proofs of these theorems.

Theorem 3 (Restatement of 1) *Let p and D be given such that*

1. *If $s_i \in D$, then s_i is not DOM-accessing in any execution.*
2. *D is $\preceq$-upward closed.*

Then p_D observationally refines p.

Proof (Proof sketch) By induction over $|D|$. The case $|D| = 0$ is trivial.

Let $e_1, \ldots, e_k$ be a trace of p_D, $s \in D$ the first script in page order that is contained in D, and e_i the event corresponding to the execution of s.

By the induction hypothesis, we known that $p_{D \setminus \{s\}}$ observationally refines p, so it is sufficient to show that p_D observationally refines $p_{D \setminus \{s\}}$. Without loss of generality, assume that $D = \{s\}$. Then $p_{D \setminus \{s\}} = p$.

Let $e_1, \ldots, e_m$ be the maximal prefix of $e_1, \ldots, e_k$ such that $e_1, \ldots, e_m, e'$ would be a valid execution prefix of p, and e' correspond to the execution of s. We have to show that $e_i\ e_1, \ldots, e_m, e_i, e_{m+1}, \ldots, e_{i-1}$ is a valid execution prefix of p. We again proceed by induction over the length of this sub-sequence. The case where $m + 1 > i - 1$ is trivial. Otherwise, it suffices to show that e_i and e_{i-1} can be exchanged, since the induction hypothesis implies the rest.

Suppose first of all that e_i and e_{i-1} race. Then they can be exchanged, since (by the race condition) $e_1, \ldots, e_{i-2}, e_i, e_{i-1}$ must be a valid execution of p.

Thus, we may assume that e_i and e_{i-1} do not race. We show that they commute, so two conditions must be satisfied:

1. $e_{i-1}; e_i$ and $e_i; e_{i-1}$ must have the same DOM behavior. But since e_i does not access the DOM, its DOM behavior is empty, and the DOM behaviors of both sequences reduce to the DOM behavior of e_{i-1}.
2. Starting from a state s, $e_{i-1}; e_i$ and $e_i; e_{i-1}$ must achieve the same states.

   Suppose e_i and e_{i-1} access the same object, and at least one access is a write. We consider the possible reasons e_{i-1} has been posted:

   - e_{i-1} is an event due to user interaction.

     By the assumptions above, user interaction only takes place after the `DocumentContentLoaded` handler has been executed. But by the semantics of `defer`, e_i is executed before that handler. So e_{i-1} cannot come before e_i.

- There is some script s' such that e_{i-1} is transitively posted by s'.

  Suppose first that s' comes after s in the page. Then, by definition of the must-come-before relation, $s \preceq s'$. Thus, $s' \in D$, and by the semantics of `defer`, $e_i \,\mathsf{hb}\, e_{i-1}$ — contradiction.

  Clearly, $s' \neq s$ (because $s = s'$ implies $e_i \,\mathsf{hb}\, e_{i-1}$), so s' comes before s in the page. If e_{i-1} was a near event of s', then s would have to come before s' by the definition of near events, contradiction. Thus, e_{i-1} is a far event of s'. But this implies that e_{i-1} and e_i are racing — contradiction.

- e_{i-1} is not posted due to user interaction or transitively from one script. The only kind of event left at this point are events corresponding to browser-internal behavior; all of these events only access the DOM and browser-internal state that is not accessible to JavaScript.

  Since e_i and e_{i-1} must access the same object, this implies that they both access a DOM object — contradiction, e_i is not DOM-accessing.

Thus, e_i and e_{i-1} do not access any shared object, where one of the accesses is a write. At this point, we can apply the Bernstein criteria [Bernstein, 1966] to show commutativity.

In a full proof, one would have to also account for the near events of all scripts; this doesn't pose any major challenges, but complicates the argument with technicalities.

Theorem 4 (Restatement of 2) *Let p and D be given, and $e_1, \ldots, e_n$ be a trace of p.*

1. *If e is a near event of s and accesses the DOM, $s \notin D$.*
2. *If e is involved in a race or has non-deterministic control flow, $s \,\mathsf{hb}\, e$ and s' happens before s in program order (including $s = s'$), $s' \notin D$.*
3. *D is $\preceq$-upward closed.*

Then p_D observationally refines p.

Proof (Proof sketch) We reduce to Theorem 1. It suffices to show that that if $s \in D$, then it is not DOM-accessing in any execution. For this, we show that if a near event e of s is DOM-accessing, then $e = e_i$ for some i with $s \,\mathsf{hb}\, e$ in the given trace.

Using condition (2), we find that all events that are transitively posted by s are deterministic, in the sense that they have only one trace. Since they are not involved in race conditions, we can treat them independently of other events. Furthermore, the "no non-deterministic control flow" condition also implies that their execution is independent of any earlier non-determinism in the execution. Thus, we can assume without loss of generality that there is exactly one execution trace for the near events of s on p. So, if e is DOM-accessing in any trace, it is DOM-accessing in all traces, in particular in $e_1, \ldots, e_n$, and hence $e = e_i$ for some i as above.

The second half of (2) is used to ensure that D is $\preceq$-upward closed for any execution, using a similar determinism argument.

6 Related work

Accelerating web page loading One key ingredient of website performance is front-end performance: How long does it take to load and display the page, and how responsive is it? One factor that influences front-end performance is script loading time [Souders, 2008]. Much has been written about optimizing loading and display times for web pages; we point out Google's guidelines on improving display times [google], which, among other things, recommends using `async` and `defer` to speed up page loading. In fact, scripts that provide external services (such as analytics, advertising or social media integration) are often loaded asynchronously.

The question of asynchronous JavaScript loading and improving page loading times in general has lead to various patents, e.g., Lipton et al. [2010], Kuhn et al. [2014], FAINBERG et al. [2011]; these patents describe specific techniques for "do-it-yourself" asynchronous script loading. Only one of them describes a technique for selecting scripts to load asynchronously, which boils down to loading *all* scripts this way.

Apart from asynchronous loading, another technique to improve script loading times is to make the scripts themselves smaller. Besides compression (including compiler techniques to optimize the code for size, e.g. Google, Inc. [2016]), one may "page out" functions from scripts by replacing function bodies with stubs that, if called, download the function implementation from the network [Livshits and Kiciman, 2008]. Asynchronous loading complements these techniques.

Many additional techniques exist that optimize web page loading times, working on many different levels if the stack. Since these techniques are not specific to JavaScript and can be used complementarily, we omit the details here.

Parallelisation and commutativity The deferring transform can be seen as a close relative of transformations employed by parallelizing compilers. In particular, we can phrase the question of deferrability in terms of *commutativity*: A script is deferrable if it does not access the DOM and commutes with all (later) non-deferrable scripts.

Parallelizing compilers use two classes of criteria to figure out which parts of the code can be run in parallel. The more traditional approach, which is often used for the parallelisation of array-processing loops, is based around the *Bernstein criteria* Bernstein [1966], which state that two blocks A and B of code are parallelizable if A neither reads nor writes memory cells that B writes, and vice versa. The dependency order defined above is then a variant of the dependency graph used in loop parallelisation. We omit discussion the many papers dealing with details of loop parallelisation, since the techniques described there, while interesting, are not directly relevant to this work.

A variant of this approach can be used for less-structured data. Tao Pingali et al. [2011] analyzes algorithms structurally, identifying sets of *active nodes* and the operations performed on them, and parallelises code sections that do not interfere based on this information.

The other approach to parallelisation, introduced by Rinard and Diniz [1996], analyzes parts of the program for *commutativity*. Two functions A and B commute if, starting from the same program state, executing A and then B gives the same state that exe-

cuting B and then A would give. This form of analysis is used for parallelizing code that operates processing data in a less-structures way. An extension of commutativity analysis [Aleen and Clark, 2009] replaces the concrete state used in the analysis with an abstracted version, which allows showing commutativity for functions where the concrete behavior is slightly different, but the observable behavior is the same.

Semantics of JavaScript and HTML The semantics for JavaScript and HTML are well-known to be quite complex and somewhat unusual. The main specification for JavaScript is the EcmaScript standard ES6; it essentially describes how to implement a JavaScript interpreter. For HTML, there are two versions of the specification: the W3C standard and the "living standard" published by WHATWG [2016].

As mentioned in the previous section, there are various formalizations that describe aspects of the web stack. The semantics of the basic JavaScript language have been modeled various times, including a functional core calculus [Guha et al., 2010], direct small-step semantics [Maffeis et al., 2008] and big-step semantics mechanized in Coq Bodin et al. [2014].

Formalizations of HTML and browser behavior usually concentrate on specific aspects and browser behaviors. For instance, Featherweight Firefox [Bohannon and Pierce, 2010] focuses on modeling the event model. Other work focuses on modeling all possible information flows, with the application of security of data flows in mind [Bichhawat et al., 2014]. The EventRacer project [Petrov et al., 2012, Raychev et al., 2013] has formalized the task model of HTML pages, together with memory accesses, to perform race detection.

Analysis of JavaScript and web pages The analysis of JavaScript code and, to a lesser degree, web pages, has become a subject of considerable interest. Apart from the aforementioned EventRacer and browsers equipped for information flow control analysis, there are several other projects to provide static or dynamic analyses for JavaScript.

Outside of specific targeted analyses like AJAX race analysis [Zheng et al., 2011], there is a general dynamic analysis framework for JavaScript called Jalangi [Sen et al., 2013]. In contrast to the approach taken in many other dynamic analysis, which are build on instrumented browsers, Jalangi works by performing a source-to-source translation, making it independent of the execution environment. At the same time, it provides no easy access to the interaction with the underlying browser; for this reason, we could not use it to implement it JSDefer.

Additionally, there are various static analysis tools for JavaScript. In particular, we are aware of three projects to perform type analysis. TAJS [Jensen et al., 2009, 2011, 2012] and JSAI [Kashyap et al., 2014] have made quite some progress towards completely automatic analysis of general JavaScript code, but both have scaling issues for large pages. The flow tool [Facebook, Inc., 2016], in contrast, trades completeness for scalability, working more like a traditional type checker/type inference engine.

References

Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. Repairing event race errors by controlling nondeterminism. In *Proceedings of ICSE 2017*, 2017.

Farhana Aleen and Nathan Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, pages 241–252, 2009. doi: 10.1145/1508244.1508273. URL `http://doi.acm.org/10.1145/1508244.1508273`.

Arthur J Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, (5):757–763, 1966.

Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. Information flow control in webkit's javascript bytecode. In *Principles of Security and Trust - Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 159–178, 2014. doi: 10.1007/978-3-642-54792-8_9. URL `http://dx.doi.org/10.1007/978-3-642-54792-8_9`.

Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised javasript specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 87–100, 2014. doi: 10.1145/2535838.2535876. URL `http://doi.acm.org/10.1145/2535838.2535876`.

Aaron Bohannon and Benjamin C. Pierce. Featherweight firefox: Formalizing the core of a web browser. In *USENIX Conference on Web Application Development, WebApps'10, Boston, Massachusetts, USA, June 23-24, 2010*, 2010. URL `https://www.usenix.org/conference/webapps-10/featherweight-firefox-formalizing-core-web-browser`.

ES6. *ECMAScript 2015 Language Specification – ECMA-262 6th Edition.* ECMA International, Rue du Rhone 114, CH-1204 Geneva, 2015.

Facebook, Inc. flow – a static type checker for javascript, 2016. URL `https://flowtype.org`.

L. FAINBERG, O. Ehrlich, G. Shai, O. Gadish, A. DOBO, and O. Berger. Systems and methods for acceleration and optimization of web pages access by changing the order of resource loading, February 3 2011. URL `https://www.google.com/patents/US20110029899`. US Patent App. 12/848,559.

fortune. Fortune 500, 2016. URL http://beta.fortune.com/fortune500/.

Qingzhu Gao, Prasenjit Dey, and Parvez Ahammad. Perceived performance of webpages in the wild: Insights from large-scale crowdsourcing of above-the-fold qoe, 2017.

google. Remove Render-Blocking JavaScript, Apr 2015. URL https://developers.google.com/speed/docs/insights/BlockingJS.

Google, Inc. Closure tools, 2016. URL https://developers.google.com/closure/.

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The Essence of JavaScript. In *ECOOP 2010. Proceedings*, pages 126–150, 2010. URL http://dx.doi.org/10.1007/978-3-642-14107-2_7. updated version at http://arxiv.org/abs/1510.00925.

Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, pages 238–255, 2009.

Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of javascript web applications. In *FSE'11 and ESEC'11*, pages 59–69, 2011.

Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *ISSTA 2012*, pages 34–44, 2012.

Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. JSAI: a static analysis platform for javascript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 121–132, 2014. doi: 10.1145/2635868.2635904. URL http://doi.acm.org/10.1145/2635868.2635904.

B. Kuhn, K. Marifet, and J. Wogulis. Asynchronous loading of scripts in web pages, April 29 2014. URL https://www.google.com/patents/US8713424. US Patent 8,713,424.

E.J. Lipton, B.C.L. Roy, S. Calvert, M.E. Gibbs, N. Kothari, M.J. Harder, and D.V. Reed. Dynamically loading scripts, March 30 2010. URL https://www.google.com/patents/US7689665. US Patent 7,689,665.

V. Benjamin Livshits and Emre Kiciman. Doloto: code splitting for network-bound web 2.0 applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*, pages 350–360, 2008. doi: 10.1145/1453101.1453151. URL http://doi.acm.org/10.1145/1453101.1453151.

Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for javascript. In *Programming Languages and Systems, 6th Asian Symposium,*

APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings, pages 307–325, 2008. doi: 10.1007/978-3-540-89330-1_22. URL http://dx.doi.org/10.1007/978-3-540-89330-1_22.

Boris Petrov, Martin T. Vechev, Manu Sridharan, and Julian Dolby. Race detection for web applications. In *PLDI 2012*, pages 251–262, 2012. doi: 10.1145/2254064.2254095. URL http://doi.acm.org/10.1145/2254064.2254095.

Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, Muhammad Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 12–25, 2011. doi: 10.1145/1993498.1993501. URL http://doi.acm.org/10.1145/1993498.1993501.

Veselin Raychev, Martin T. Vechev, and Manu Sridharan. Effective race detection for event-driven programs. In *OOPSLA 2013*, pages 151–166, 2013. doi: 10.1145/2509136.2509538. URL http://doi.acm.org/10.1145/2509136.2509538.

Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, May 21-24, 1996*, pages 54–67, 1996. doi: 10.1145/231379.231390. URL http://doi.acm.org/10.1145/231379.231390.

Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for javascript. In *ESEC/FSE'13*, pages 488–498, 2013. URL http://doi.acm.org/10.1145/2491411.2491447.

Steve Souders. High-performance web sites. *Commun. ACM*, 51(12):36–41, December 2008. ISSN 0001-0782. doi: 10.1145/1409360.1409374. URL http://doi.acm.org/10.1145/1409360.1409374.

Rick Viscomi, Andy Davies, and Marcel Duran. *Using WebPageTest: Web Performance Testing for Novices and Power Users*. O'Reilly Media, Inc., 1st edition, 2015. ISBN 1491902590, 9781491902592.

WHATWG. HTML – Living Standard, Sep 2016. URL https://html.spec.whatwg.org/multipage/.

Yunhui Zheng, Tao Bao, and Xiangyu Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011*, pages 805–814, 2011. doi: 10.1145/1963405.1963517. URL http://doi.acm.org/10.1145/1963405.1963517.