Deferrability Analysis for JavaScript

Johannes Kloos[1], Rupak Majumdar[1], and Frank McCabe[2]

[1] MPI-SWS
[2] Instart Logic

Abstract. 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 full 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[3]. 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— 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 [28].

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

O. Strichman and R. Tzoref-Brill (Eds.): HVC 2017, LNCS 10629, pp. 35–50, 2017.
https://doi.org/10.1007/978-3-319-70389-3_3

Normally, 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, this can significantly slow down page rendering. To address this, HTML5 added "async" and "defer" loading modes. A script marked async is loaded in parallel with parsing and run as soon as it is loaded. Scripts marked defer are also loaded in parallel with parsing, but are evaluated only when parsing is complete, in the order in which they were scheduled for download.

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, the user visible behavior of the page does not change. To make the idea of safe deferrable sets usable, we characterize the safe deferrable set using event traces [23]. In particular, we can use event traces to define a dependency order between scripts, and the notion of DOM-accessing scripts, which have user-visible behavior. A safe deferral set is contains no DOM-accessing scripts and is upward-closed under the dependency order. We also show that if a set contains only deterministic scripts, it is sufficient to check a single trace to characterize a safe deferral set, and describe a dynamic analysis based on this criterion.

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. 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, where the median load time of a page is 3097ms.

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 be deferred.We show the potential for improving the load performance for these pages: in our experiments, the median loading time improvement was 198.5 ms.

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, linked in the document. The browser downloads images asynchronously, while continuing to parse the document. In contrast, it downloads scripts synchronously by default, making the parser wait for the download, and evaluates the script before continuing to parse the page. This puts script download and parsing on the critical path. 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 ([28], sec. 4.12) 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="...&quot;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="...&quot;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.

While asynchronous or deferred loading is desirable from a performance perspective, it can lead to *race conditions*, i.e., the output of the page may depend on the order in which scripts are executed [23]. 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, 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

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 [28]. 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 [22,23,1] on which we build our analysis; we follow the simplified presentation from [1]. 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 in [28]). 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.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 $e = e_1, \ldots, e_k = e'$, $k \geq 1$, such that for all $1 \leq i < k$, e_i posts e_{i+1}; i.e., we take the 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.

Theorem 1. *Let p be a page with scripts $s_1, \ldots, s_n$ and $D \subseteq \{s_1, \ldots, s_n\}$. D is a safe deferral set 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 $s_j \in D$.*

The proof can be found in the technical report [17]. 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 $e_1, \ldots, e_n$. D is a safe deferral set if 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.*

The proof can be found in [17]. The key idea of this proof is that all scripts in D are "deterministic enough," so the conditions of the previous theorem collapse to checking a unique trace.

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. 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; compare [1]. Hence, we assume that early user interaction either does not occur or is handled as in [1].

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. We use the second criterion in the previous section above, aggressively marking potentially non-deterministic scripts.

JSDefer use an instrumented browser from the EventRacer project [23] 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, marking three sources: The random number generator, the current time, and various bits of browser state. In JSDefer, we check if a given script accesses any of these sources of non-determinism. We leave the integration of a proper taint-tracking based non-determinism check (e.g., building on [4]) as future work.

We perform deferrability analysis on the collected trace using Theorem 2. This calculation computes a safe deferrable set. We then rewrite the top-level HTML file of the page to add defer attributes to all scripts in the deferrable set.

4 Evaluation

We evaluated JSDefer on the websites of the Fortune 500 companies [10] as a corpus. To gather deferrability information, we used an instrumented WebKit browser from the EventRacer project [23] to generate event traces. Out of these 500 pages, we could successfully collect 451 pages; 38 websites timed out, 11 websites returned an error and 2 contained invalid HTML.

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

1. How much and in what way is defer and async already 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?

Async or defer	#pages
Neither	32
Defer only	0
Async only	404
Both	15

Async only: Only standard scripts?	#pages
Only standard scripts and snippets	256
Other	148

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".

4.1 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[4]. 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.

4.2 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 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 relatives. 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.

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

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) {}
```

Automatically detecting that such code can be deferred seems quite difficult.

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 one more than one page[5]. They form 100 easily-identified groups, the largest of which are Google Analytics (187 instances), jQuery (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.

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 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.

4.3 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,

[5] 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 and percentage of deferrable scripts. The number of deferrable scripts includes pages with no scripts; for the percentage, we only consider pages with at least one deferrable script.

# deferrable scripts	# pages	% deferrable scripts	# pages
no scripts	11	$< 10\%$	180
0	156	$10 - 20\%$	56
1	86	$20 - 30\%$	37
2	55	$30 - 40\%$	14
3–5	89	$40 - 50\%$	6
6–10	47	$50 - 60\%$	1
more than 10	18	$60 - 70\%$	1

- 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; see the left column of Tab. 2. We also analyzed what percentage of scripts are deferrable on a given page; discarding the pages that had no deferrable scripts on them, we get the picture in the right column of Tab. 2.

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.

Many scripts are marked as non-deferrable because of dependencies. 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. 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 defini-

tions", 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.

4.4 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. 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 [27]. 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. [11].

Since our setup did not allow us to interpose on SSL connections, we had to drop pages that force an upgrade to SSL. In total, out of the 500 pages considered, 209 force an SSL upgrade. Taking the intersection of the sets of pages that have deferrable scripts and don't force an SSL upgrade, we were left with 169 pages.

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.

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. These violin plots visualize an approximation of the probability distribution of the loading time for each case.

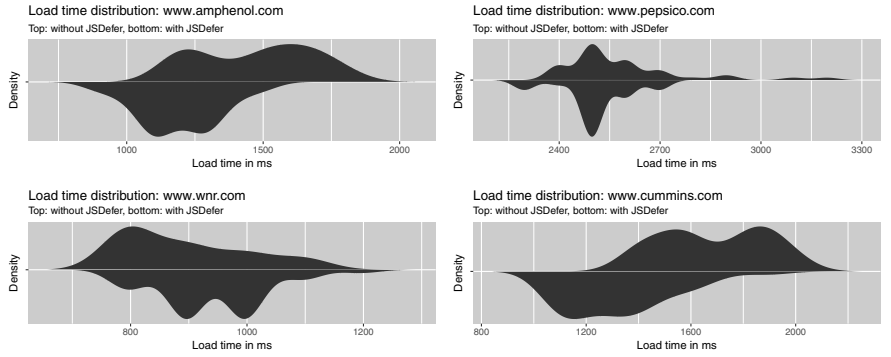

Fig. 1. Violin plots of load time distributions for some pages, before and after applying JSDefer. The graphs show a smoothed representation of the sample distribution.

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.
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[6], 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.

[6] 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

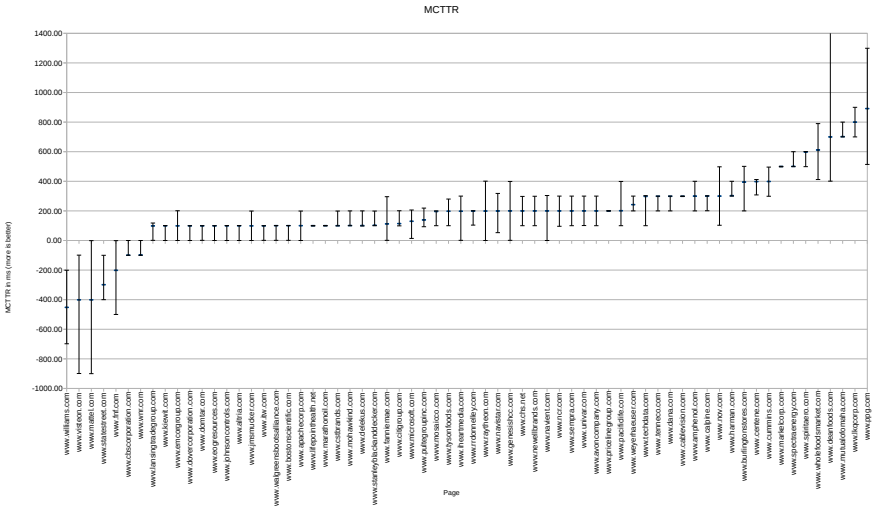

Fig. 2. MCTTR values for pages with significant MCTTR; Visualization of Table 3.

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.

4.5 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.
   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. Internal validity, question 5: We could not completely control for network delays in the testing set-up.
3. 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 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 is script loading time [26]. Google's guidelines [12] recommend using `async` and `defer` to speed up page loading.

The question of asynchronous JavaScript loading and improving page loading times in general has lead to various patents, e.g., [19,18,9]; they 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. [13]), one may "page out" functions from scripts by replacing function bodies with stubs that, if called, download the function implementation from the network [20]. Asynchronous loading complements these techniques, as well as the many other techniques to improve load time.

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*[24,2]: In Rinard et al.'s work, two functions A and B commute if executing A and then B gives the same results that executing B and then A gives. In our setting, a script is deferrable if it does not access the DOM and commutes with all (later) non-deferrable scripts.

The Bernstein Criteria [3] describe that two program blocks A and B are parallelizable if A neither reads nor writes memory cells that B writes, and vice versa. This is used to define the dependency graph that identifies parallelizable parts of a program; our dependency order is constructed in a similar way.

Semantics analysis of JavaScript and web pages: The semantics of JavaScript and HTML are complex and unusual; natural-language descriptions can be found in [7] (JavaScript) and [28] (HTML). There are various formalizations of JavaScript [14,21,5], and formalizations of fragments of browser behavior, considering the event mode [6], information flow control [4] and race detection [22,23].

Additional analysis tools exist for JavaScript, including Jalangi2 [25], which performs a dynamic analysis using source-to-source-translation, and various static analysis like TAJS [15], JSAI [16] and the type inference engine flow [8].

References

1. Adamsen, C.Q., Møller, A., Karim, R., Sridharan, M., Tip, F., Sen, K.: Repairing event race errors by controlling nondeterminism. In: ICSE 2017 (2017)
2. Aleen, F., Clark, N.: Commutativity analysis for software parallelization: letting program transformations see the big picture. In: ASPLOS '09 (2009)
3. Bernstein, A.J.: Analysis of programs for parallel processing. IEEE Trans. Elec. Comp. (5), 757–763 (1966)

4. Bichhawat, A., Rajani, V., Garg, D., Hammer, C.: Information flow control in webkit's javascript bytecode. In: POST 2014 (2014)
5. Bodin, M., Charguéraud, A., Filaretti, D., Gardner, P., Maffeis, S., Naudziuniene, D., Schmitt, A., Smith, G.: A trusted mechanised javasript specification. In: POPL '14 (2014)
6. Bohannon, A., Pierce, B.C.: Featherweight firefox: Formalizing the core of a web browser. In: WebApps'10 (2010)
7. ECMA International: ECMAScript 2015 Language Specification (2015)
8. Facebook, Inc.: flow: a static type checker for JavaScript, `https://flowtype.org`
9. FAINBERG, L., Ehrlich, O., Shai, G., Gadish, O., DOBO, A., Berger, O.: Systems and methods for acceleration and optimization of web pages access by changing the order of resource loading (Feb 3 2011), `https://www.google.com/patents/US20110029899`, US Patent App. 12/848,559
10. Fortune 500 (2016), `http://beta.fortune.com/fortune500/`
11. Gao, Q., Dey, P., Ahammad, P.: Perceived performance of webpages in the wild: Insights from large-scale crowdsourcing of above-the-fold QoE (2017), arXiv:1704.01220
12. Google, Inc.: Remove Render-Blocking JavaScript (Apr 2015), `https://developers.google.com/speed/docs/insights/BlockingJS`
13. Google, Inc.: Closure tools (2016), `https://developers.google.com/closure/`
14. Guha, A., Saftoiu, C., Krishnamurthi, S.: The Essence of JavaScript. In: ECOOP 2010. See also `http://arxiv.org/abs/1510.00925`
15. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for javascript. In: SAS 09
16. Kashyap, V., Dewey, K., Kuefner, E.A., Wagner, J., Gibbons, K., Sarracino, J., Wiedermann, B., Hardekopf, B.: JSAI: a static analysis platform for javascript. In: FSE-22 (2014)
17. Kloos, J., Majumdar, R., McCabe, F.: Deferrability analysis for JavaScript. Tech. rep., MPI-SWS (2017), see `http://www.mpi-sws.org/~jkloos/jsdefer-tr.pdf`
18. Kuhn, B., Marifet, K., Wogulis, J.: Asynchronous loading of scripts in web pages (Apr 29 2014), `https://www.google.com/patents/US8713424`
19. Lipton, E., Roy, B., Calvert, S., Gibbs, M., Kothari, N., Harder, M., Reed, D.: Dynamically loading scripts (Mar 30 2010), `https://www.google.com/patents/US7689665`, US Patent 7,689,665
20. Livshits, V.B., Kiciman, E.: Doloto: code splitting for network-bound web 2.0 applications. In: FSE '08 (2008)
21. Maffeis, S., Mitchell, J.C., Taly, A.: An operational semantics for javascript. In: APLAS 2008 (2008)
22. Petrov, B., Vechev, M.T., Sridharan, M., Dolby, J.: Race detection for web applications. In: PLDI 2012 (2012)
23. Raychev, V., Vechev, M.T., Sridharan, M.: Effective race detection for event-driven programs. In: OOPSLA 2013 (2013)
24. Rinard, M.C., Diniz, P.C.: Commutativity analysis: A new analysis framework for parallelizing compilers. In: PLDI '96 (1996)
25. Sen, K., Kalasapur, S., Brutch, T.G., Gibbs, S.: Jalangi: a selective record-replay and dynamic analysis framework for javascript. In: ESEC/FSE'13 (2013)
26. Souders, S.: High-performance web sites. Commun. ACM 51(12), 36–41 (Dec 2008)
27. Viscomi, R., Davies, A., Duran, M.: Using WebPageTest: Web Performance Testing for Novices and Power Users. O'Reilly Media, Inc., 1st edn. (2015)
28. WHATWG: HTML – Living Standard (Sep 2016), `https://html.spec.whatwg.org/multipage/`