Blocking the load of cross-origin, parser-blocking scripts inserted via document.write

Shivani Sharma, Bryan McQuade, Josh Karlin
loading-dev@chromium.org
[public document]

Summary

For users on high latency connections, parser-blocking scripts loaded via document.write can cause significant delays in user perceived page load latency.
Historically, third party content such as ads and analytics trackers used document.write to load script resources. Over the past five+ years, non-blocking alternatives, such as async scripts, have become available. Today, nearly all third party snippets support and recommend asynchronous loading. Despite widespread support for asynchronous loading, roughly ten percent of page loads use document.write to load third party script resources in a way that harms performance.

For a detailed description of the performance issues introduced by scripts loaded via document.write, and potential approaches to address them, read "Reducing the impact of document.written scripts on user perceived page load latency" . One of the potential approaches mentioned there is the following:

- For users on very slow connections such as 2G, the performance penalty from third-party scripts loaded via document.write is often so severe as to delay display of main page content for tens of seconds. In some cases, the load of these scripts can cause pages to fail to load altogether. Blocking the load of third-party scripts loaded via document.write yields dramatic improvements in page load performance for affected pages, reducing time to meaningful paint by up to 30 seconds (example 1, 2). Of 200 analyzed URLs, we did not find any pages whose main content was broken by blocking scripts loaded via document.write.

This document highlights the design, metrics and performance impact of the above change (blocking third-party scripts introduced by document.write when on slow connections).

Design

The tracking bug for this feature is: 599875

The following conditions are checked on a script inserted via document.write before deciding if the script should be blocked. Note, that blocking here only applies if it is a HTTP cache miss.

- **Main Frame:** The blocking of scripts is restricted to scripts in the main frame.

- **Parser-blocking:** Since asynchronous scripts are not parser blocking, we do not block those scripts. This will encourage developers currently dependent on load of scripts via document.write to move to using asynchronous loading alternatives, which allow third party content to load without blocking the display of content later in the document and have been available for 5+ years. The majority of third party snippet providers have offered asynchronous loading options for years.

- **Cross Origin:** Only cross origin scripts are blocked. Some pages use document.write to load a script for rendering some part of the main page. We do not want to break those pages and intend to reduce such breaks with the condition that if the script is loaded from the same domain as the main document then it should not be blocked. Cross origin scripts inserted via document.write, on the other hand, are likely to be for third-party scripts. (During ~200 URL lab A/B tests, all pages with cross-origin scripts written with document.write that showed an improvement in SpeedIndex had cross-origin scripts being used for third-party content like ads and analytics.)

- **Slow connections:** Since the performance penalty of these scripts leads to a worst case page load experience in slow connections, we intend to block scripts only if all the above conditions are true and if the connection is slow. Currently slow connection is restricted to 2G. In the future this might be expanded using the Network Quality Estimator to include page loads when the network performance is 2G-like. Having said that, for testing we would also have a blink setting to enable this feature independent of connection types.

- **Not a refresh:** To further reduce the possibility of a page break by blocking a document.written script, there will be a check that if the user is reloading the same page then do not consider the document.written scripts on this page for blocking.

Implementation

The above design is implemented while determining the cache policy of the resource in the renderer process in blink. When the cache policy for a resource is decided in FrameFetchContext::resourceRequestCachePolicy(), if all of the conditions given in the design section are true then the cache policy for that script is marked as ReturnCacheDataDontLoad implying that if the script is in the cache then return it. If it is not in the cache do not load it from the network. The current implementation returns the script from the cache even if it is stale and we are working on whether we should have a stale-while-revalidate implementation for such scenarios.

Performance impact measured in the lab

On 2G, blocking the load of scripts inserted via document.write significantly improves both worst case and mean loading performance for roughly 10% of sampled pages.

Among a sample of ~200 randomly sampled pages, using WebPagetest to simulate a 2G connection, we observe that roughly 10% of these pages load scripts via document.write in a way that significantly impacts their loading performance. Typically, these pages are among the slowest loading pages.

For half of the pages that show improvement as a result of blocking the load of scripts inserted via document.write, we observe 50% speedups in SpeedIndex, with improvements ranging from 10 to 30 seconds. For the other half of pages that show improvement, we observe 25% speedups, with improvements ranging from 5 to 10 seconds.

On average, blocking the load of scripts inserted via document.write results in a 2 second, or 8%, improvement in SpeedIndex on 2G.

Below are a few video examples that show the speedup impact from disabling load of scripts inserted via document.write on 2G:

1. https://youtu.be/Xie7QWHEsn0 (http://m.authorstream.com/...)
   First paint reduced from 15s to 4s.
   Time to main content rendered reduced from 29s to 11s.

2. https://youtu.be/99CedQ-7DHs (http://www.ticedu.ca/)
   First paint, as well as time to main content rendered, reduced from 46s to 17s.

Metrics and Field Trial

The goal of the field trial is to measure improvement in first meaningful page loads as perceived by the user. To be able to do that effectively, we would like to observe and compare metrics from pages that had all the conditions satisfied for blocking of a document.written script as mentioned in the “Design” section. More on this is in the next section “Metrics specific to triggering of feature”.

Metrics would be observed to answer the following questions:

- Motivation for this optimization - Is it really an impactful change?
- Is this change leading to page breaks?
- Finally, is this change leading to user perceived page loading improvement?

Motivation for this optimization - Is it really an impactful change?
To be able to answer this question we need to answer what percentage of the pages is this feature targeting to improve. The metrics to be observed for that include:

- PageLoad.Timing2.ParseBlockedOnScriptLoadFromDocumentWrite: Percentage of pages in slow networks that are blocked on third-party scripts invoked via document.write.

Is this change leading to page breaks?
To be able to answer this question the metrics that will be an indicator of possible page breaks is:

- PageLoad.Clients.DocWrite.Block.ReloadCount: Percentage of pages that are blocked on third-party scripts invoked via document.write and hit reload at least once.
- The expectation is that this metric should not show a significant increase.
- As mentioned above this change will not consider scripts for blocking if it is a page reload to help in reducing any page breaks.

Is this change leading to user perceived page loading improvement?
For observing this, the following metrics will be observed:

- ParseDuration: Measures the time that the HTML parser was active, for main frame documents that finished parsing. This metric is expected to decrease significantly for pages where scripts were blocked.

- ParseStartToFirstContentfulPaint: Measures the time from when the HTML parser started, to when the page first paints content. This metric is expected to decrease for pages where scripts were blocked.

- ParseBlockedOnScriptLoadFromDocumentWrite: Measures the time that the HTML parser spent blocked on the load of scripts inserted from document.write, for main-frame documents that started parsing. This metric is recently introduced for this and related projects. Ideally the metric ParseBlockedOnScriptLoad would have been sufficient but it's possible that that we may get a doc.written parser blocking script out of the way only to get delayed on a different script in the experiment population that we wouldn't have reached otherwise due to e.g. user abandonment. To take care of such scenarios, this metric will more directly measure the impact of our changes with a UMA focused on document.write blocking.

- Parser blocking time before first contentful paint (To be implemented): This metric will give a meaningful metric to see the impact of this change on parser blocking time not just in total but before first “contentful” paint.

Metrics specific to triggering of feature

As mentioned above, we would like to compare metrics between experiment and control groups from pages that had all the conditions satisfied for a document.write script to be blocked. For this we would need this information to propagate from blink where the decision to block a script or not is taken, to the browser process's page load metrics framework which logs the histograms.

To achieve this Charles Harrison has enhanced the PageLoadTiming IPC structure between renderer and browser process to add metadata to convey which features were triggered.

Using this new meta data there would be this feature specific versions of the metrics given above that will be logged for experiment and control groups:
- PageLoad.Clients.DocWrite.Block.Timing2.ParseDuration
- PageLoad.Clients.DocWrite.Block.Timing2.ParseStartToFirstContentfulPaint
- PageLoad.Clients.DocWrite.Block.Timing2.ParseBlockedOnScriptLoad
- PageLoad.Clients.DocWrite.Block.Timing2.ParseBlockedOnScriptLoadFromDocumentWrite
- PageLoad.Clients.DocWrite.Block.ReloadCount

Using these new histograms help in ignoring the metrics from scenarios where :
- There was no main frame, parser-blocking script to be blocked.
- There was a script to be blocked but the connection was not “slow”.
- There was a script to be blocked but the script was from the same domain as the main resource.

Field trial study results

Summarized below are results from Android Dev channel over 7 days of aggregated data. Counts of control and experiment groups were 1022495 and 1088034 respectively.

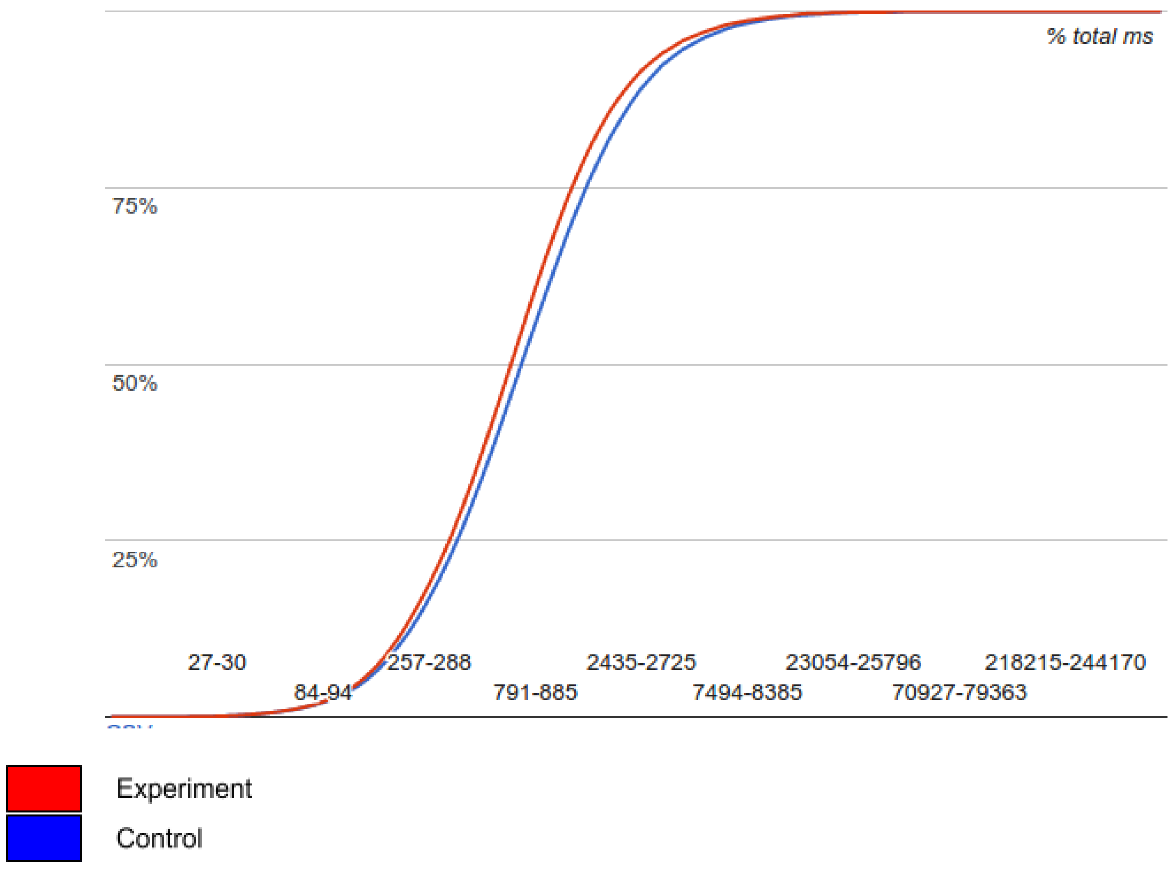
Metrics from pages that have a cross-origin, parser-blocking script

- Parse Start to First Contentful Paint Duration improved by the following percentages (statistically significant):

Mean: 11%

50th percentile: 10%

90th percentile: 15%

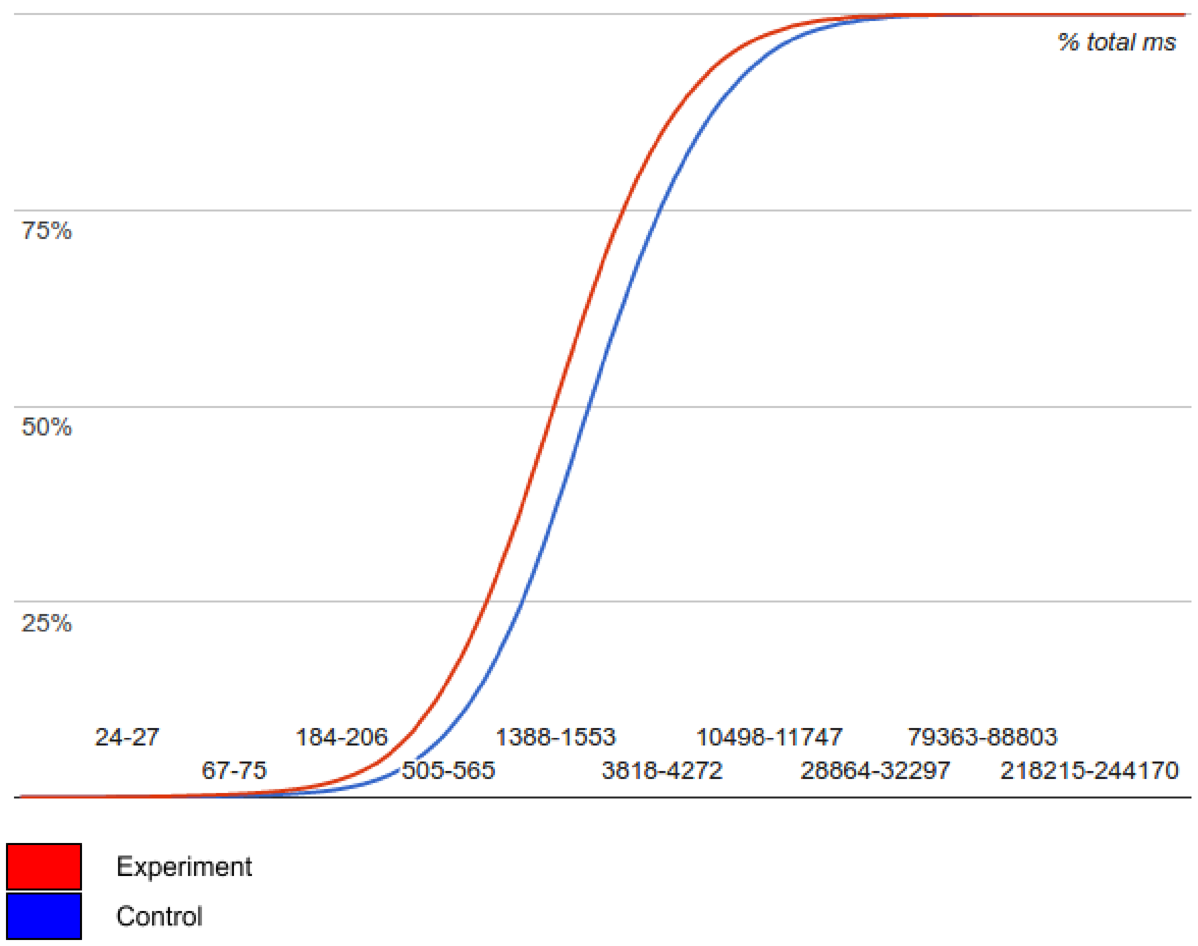


- Parse Duration improved by the following percentages (statistically significant):

Mean: 29%

50th percentile: 28%

90th percentile: 30%



Note that out of these affected pages, some will have the script in cache so the behaviour on those pages will be unaffected.

In addition, number of page reloads, which can be indicative of broken pages, are not statistically significantly different between control and experiment groups.