Reducing the impact of document.written scripts on user perceived page load latency

Charles Harrison, Shivani Sharma, Bryan McQuade
loading-dev@chromium.org
March 8, 2016
[public document]

For users on high latency connections, parser-blocking scripts loaded via document.write cause significant delays in user perceived page load latency. Most scripts inserted via document.write are for third party content such as ads and trackers. This document gives a high-level overview of the performance issues introduced by scripts loaded via document.write, and explores potential approaches to address them.

This is a long document. Most readers can focus on the Summary, Background, and Third Party Content sections.

Summary

Findings

Parser-blocking scripts loaded via document.write significantly impact worst case and mean loading performance. Roughly 10% of page loads are adversely affected by document.write script loading. For these page loads, SpeedIndex increases 25% to 50% on 2G and 3G (we did not analyze 4G or WiFi). In absolute terms, as a result of loading parser-blocking scripts via document.write, these page loads are delayed by an additional one to ten seconds on 3G, and 5 to 30 seconds on 2G (video examples on 2G: 1, 2).

Recommendations

Developer outreach

- To increase awareness of the performance effect of scripts loaded via document.write, and make developers aware of readily available alternatives that aren't harmful to page load performance, we can reach out to web developers with affected pages via notifications in Chrome Developer Tools for pages that load scripts via document.write. Paul Irish indicated that Chrome Developer Tools would be interested in providing such a warning.

Chrome loading improvements

Evaluated but decided not to pursue further:

- We evaluated loading scripts inserted via document.write asynchronously, which takes these scripts out of the critical path of the page load. We found that due to script ordering dependencies, scripts loaded via document.write often fail to execute correctly when loaded asynchronously. We determined that this is not a viable option to mitigate the performance impact of scripts inserted via document.write. [more details: 1, 2]
- Given that scripts loaded via document.write are primarily for third party content that is not needed to render the main page content, we considered disabling document.write altogether. However, since pages use document.write for a variety of reasons, many of them not harmful to loading performance, we decided that the relative benefits of removing document.write do not outweigh the frustration that such a change could cause among developers. [more details]

Evaluated and plan to pursue further:

- 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. Shivani Sharma is exploring running a finch trial to disable the load of scripts inserted via document.write for users on slow connections such as 2G, and will share a design doc with more details soon. [more details]

- For users on faster connections such as 3G, where the impact of these scripts is significant but less severe than on 2G, we evaluated ways to mitigate the performance impact of scripts loaded via document.write rather than blocking the load of these scripts. For these users, we estimate that roughly half of the performance penalty incurred by scripts loaded via document.write can be mitigated through a combination of improved preload scanning and third party resource prediction. Charles Harrison is exploring adding these features to Chrome. [more details: preloading, prediction]

Background

When the HTML parser encounters a parser-blocking script, it must stop and execute that script before continuing to parse HTML later in the document. If the script is an external script, the parser must also wait for the fetch to complete, which may incur multiple network round trips (DNS, TCP, TLS, request). For users on high latency connections, these network round trips significantly increase parser blocking time, and in many cases, user perceived page load latency.

To mitigate the effect of these parser blocking network round trips, modern browsers include a preload scanner that runs when the parser is blocked and looks ahead in the document to fetch scripts and other resources that are likely to be needed soon. This allows the browser to begin fetching critical resources sooner, which helps to minimize parser blocking time and in turn user wait time.

However, scripts inserted by document.write are not currently discovered by the preload scanner. Thus, the HTML parser blocks on the network fetch of scripts inserted by document.write, which can lead to high parser blocking time, and, in cases where the script is included before the contents of the document, significant increases in time before page content is displayed on the screen.

Third party content

Nearly all script loads via document.write are for third party content.

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. These pages are often the slowest pages to load and represent worst case page load experience.

Pages continue to use synchronous document.write-based loading of third party content for a variety of reasons. In some cases, those pages were authored before asynchronous loading techniques were available, and have not been updated. In other cases, the page author copy/pasted a synchronous loading snippet into their pages without understanding the performance implications. In a few cases, the snippet provider doesn't yet support asynchronous loading, forcing the page to use a synchronous script.

The user-perceived performance cost of loading third party content via document.write rather than asynchronous loading is often dramatic. This example video shows the difference in loading performance for a page on a typical 3G connection. On the right, third party ad content is loaded using a blocking document.write-based technique, while on the left, the same third party content is loaded asynchronously. Use of the blocking loading technique, which prevents the main page content from being parsed and rendered until the third party ad content is fully loaded, more than doubles time to first meaningful paint, from 4.5 seconds to 12 seconds. On 2G, where network round trip times are even higher, the delay is even more dramatic.

Improving performance of scripts loaded via document.write

To reduce the user-perceived performance impact of scripts loaded via document.write, we propose a combination of outreach to developers of pages that currently load scripts via document.write, as well as making performance improvements to Chrome to reduce the performance impact of document.written scripts.

Developer outreach

In many cases, web developers use document.write to load third party script resources synchronously, without fully understanding the performance implications, and when asynchronous alternatives are readily available. To increase awareness and drive adoption of asynchronous script loading, we can reach out to web developers to encourage migration from sync to async loading.

Chrome Developer Tools enables us to reach developers with performance problems in a scalable way. Paul Irish has indicated that Chrome Developer Tools and Lighthouse could include an audit for parser-blocking scripts inserted via document.write.

The example video shows the dramatic speedup achieved for one page as a result of converting that page to use asynchronous rather than synchronous third party script loading. In this example, converting from sync to async loading required just 30 minutes of work, to

update the HTML markup of the page to use the asynchronous rather than synchronous GPT snippet, as documented here.

While we expect outreach to drive some web developers to migrate to asynchronous loading, outreach is unlikely to get all developers to update their content. Pages authored before asynchronous loading techniques were available are often unmaintained, and are unlikely to be updated, for example. Thus, in addition to outreach, we explored mitigating the performance impact of scripts loaded via document.write by making changes to Chrome. We explore these changes below.

Optimizing Chrome

Below we evaluate Chrome changes to mitigate the performance impact of scripts loaded via document.write.

Load scripts inserted via document.write asynchronously

One way to avoid the performance overhead of document.written scripts is to load them asynchronously, without blocking the parser. For example, if a page contains the following inline script:

```
<script>
  var useSSL = 'https:' == document.location.protocol;
  var src = (useSSL ? 'https:' : 'http:') +
      '//www.googletagservices.com/tag/js/gpt.js';
  document.write('<scr' + 'ipt src="' + src + '"></scr' + 'ipt>');
</script>
```

In order to prevent the script load from blocking the parser, Chrome could load this script as if it were an async script, which allows the parser to continue processing the document without yielding until the script has loaded and executed.

Unfortunately, this causes scripts to execute in a different order than they would have normally. Consider this common pattern observed on many pages:

```
<script>
  var useSSL = 'https:' == document.location.protocol;
  var src = (useSSL ? 'https:' : 'http:') +
      '//www.googletagservices.com/tag/js/gpt.js';
  document.write('<scr' + 'ipt src="' + src + '"></scr' + 'ipt>');
</script>
<script>
  // The googletag object is provided by the gpt.js script, loaded above.
  googletag.display('div-gpt-ad-12345');
</script>
```

In the above example, the successful execution of the second script block depends on availability of the googletag object, which is provided by the gpt.js script loaded in the first block. If the document.written script inserted in the first script block were loaded asynchronously, the googletag object used in the second script is not guaranteed to be defined at the time the second block executes, causing that block's execution to fail, and in turn the ad loaded by the googletag.display() call to fail.

The above pattern, where a script inserted by document.write is needed by an inline script block that comes shortly later in the HTML markup, is common, so loading document.written scripts asynchronously turns out not to be a promising option.

Decision:
It was determined that loading document.written scripts asynchronously often causes the content associated with those scripts to fail to load, due to changes in ordering of script execution. Thus, we have decided not to pursue this approach.

Load all scripts asynchronously but in order, without blocking the parser

A more aggressive form of the above would be to execute all scripts in the document asynchronously, but in order. In this approach, the HTML parser would execute without blocking on scripts, instead executing scripts in order, but as they finish loading. However, scripts often depend on executing at their declared location in the document, so this approach has a number of issues that cause pages to break in ways that are difficult to reason about and debug. For example:

- A script that performs a document.write assumes that the tokens it writes will be inserted into the document in the location immediately following the script. It is technically possible to address this case by remembering the insertion point for the script, but this adds significant complexity to the parser.
- Many scripts inspect the state of the document at the time they execute, so to guarantee correct execution, the document presented to the script should include only DOM nodes up to the point where the script appears in the document. This too is possible, but adds significant complexity.
- Executing scripts out of order with the parser can cause pages to render incorrectly, leading to flashes of unstyled content or other unexpected rendering artifacts that may result in a worse experience.

Decision:
It was determined that loading all scripts asynchronously but in order often causes the content associated with those scripts to fail to load, due to changes in ordering of script execution. Kentaro Hara did a similar investigation in the past, and came to the same conclusion. Thus, we have decided not to pursue this approach.

Disable document.write

There is general consensus that document.write is an undesirable feature of the web platform. Nonetheless, it is used for a variety of reasons on a variety of pages, loading third party scripts being just one of them. We briefly explored whether it would be feasible to disable document.write altogether, but decided that the developer pain this would cause, and in turn the frustration and lost trust of web developers who depend on this feature, outweighs the benefits of removing document.write.

We would like to identify a longer term path to removing document.write from the web platform. One that Ojan Vafai suggested is to add a mode to Content Performance Policy that allows page authors to disallow document.write, and advocate to web developers that they enable this mode. Then, in a future revision of HTML, we could consider disabling document.write. Any pages that opted in to using that future revision of HTML (via e.g. a new doctype) would also be opting in to disabling document.write. This is a possible path to removing document.write from the platform, however it is clearly a very long term path (5+ years, if at all). The rest of this document explores more near term options to reducing the performance penalty of scripts inserted via document.write in Chrome.

Decision:
While document.write is widely considered to be an undesirable feature of the web platform, the developer pain and frustration that would result from removing document.write was determined to be too high a cost relative to the benefits of removing the feature. We see removing document.write as a potential long term web platform effort.

Disable loading of scripts inserted by document.write, for users on slow connections

Third party scripts loaded via document.write significantly delay the display of main page content. On slow connections such as 2G, these delays can often be in the tens of seconds (example videos: 1, 2). These substantial delays give users the impression that the page may not load successfully, which can lead users to abandon the page loads. In some cases, the load of these scripts caused pages to fail to load altogether, due to the scripts being a single point of failure.

To mitigate the impact of these significant delays in page load time on 2G, we evaluated the impact of blocking the load of document.written scripts for users on slow connections. Note that, if the script is already available in the browser cache, we continue to allow it to execute, since no network fetch must be performed. In contrast to 'Disable document.write' above, we also continue to allow other uses of document.write that are not harmful to loading performance. For example, https://netbanking.hdfcbank.com/netbanking/ uses

document.write to dynamically create HTML in its frame https://netbanking.hdfcbank.com/netbanking/RSLogin.html?v=2. Example HTML for this page:

```
<script>
if(parent.daemon=="NETBANKING"){
   document.write(
    '<table width="1000" border="0" cellspacing="0" cellpadding="0" >');
   document.write('<tbody>');
   ...
} else{
   document.write(
   '<table width="770" border="0" cellspacing="0" cellpadding="0">');
   document.write('<tr>');
   ...
}
</script>
```

This use of document.write is not harmful to loading performance. Thus, we propose to continue to allow pages that use document.write in ways like above to work correctly, while blocking only the load of scripts inserted via document.write in the top-level frame, which is very harmful to performance. Note that scripts inserted via document.write in child frames (such as iframes) are not blocked; only scripts inserted via document.write in the top-level frame are blocked from loading.

Note that if a script is blocked because it is loaded via document.write, references to objects it exposes to other scripts later in the document (via the window object, for example) will fail in a similar way as they might fail if Chrome tried to automatically load the script asynchronously, as described in the section on making document.written scripts async. But trying to make scripts load asynchronously can cause them to fail in non-deterministic ways that are difficult to reason about and debug. Additionally, in some cases, loading these scripts asynchronously may cause them to change the document in unintended ways, while blocking them gives a more consistent and predictable user experience.

Also, blocking these scripts encourages developers to move to asynchronous alternatives and thus in the long term the performance benefit will be on all browsers.

Note that blocking the load of scripts loaded via document.write is different from integrating an ad blocker, which blocks all third party content. Blocking all third party content prevents loading of poorly performing third party content, such as content that loads scripts via document.write, as well as third party content that does not adversely impact user perceived load performance. Selectively blocking the load of scripts loaded via document.write targets only the worst performing cases that use the performance antipattern of loading scripts via document.write, while allowing performant third party content to continue to load successfully.

We do not intend to provide developers with a mechanism to opt out of blocking the load of scripts inserted via document.write. Developers currently dependent on load of scripts via document.write should move to using script tags in HTML markup, or 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, and remaining third party snippet providers that do not yet support asynchronous loading have had ample time to introduce asynchronous alternatives.

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

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.

At the mean, 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.

Document.written scripts used for main page content
Of the 200 analyzed URLs, we did not find any pages whose main content was clearly broken by blocking scripts loaded via document.write.

4.5% (9 of ~200) of analyzed pages loaded scripts that were not obviously for third party content via document.write.

6 of 9 pages loaded these scripts on the same top-level domain as the main document URL (example: https://www.maharashtra.gov.in/site/common/governmentresolutions.aspx). We may wish to allow parser-blocking scripts on the same TLD as the main document to execute. Of the 6 pages analyzed, it was determined that these pages do not require document.write to load these script, and could migrate to loading these scripts using script tags in HTML markup, which does not incur the same performance penalty. Additionally, these pages appear to continue to be fully functional when load of these same-TLD scripts inserted via document.write is blocked.

Of the remaining pages that load scripts that may be used for main page content via document.write, 2 of the 3 pages use the Google Loader to load visualization library scripts. Despite loading these visualization scripts, these pages do not appear to actually make use of the visualization libraries, and appear to be fully functional when the load of these visualization libraries is blocked (example page: http://www.railyatri.in/platform-locator). Since some pages may depend on the Google Loader, we may wish to include a whitelist of scripts, such as the Google Loader, that should not be blocked when loaded via document.write. Note that the Google Loader does appear to provide an asynchronous loading option, so developers can migrate to using the Google Loader without loading via document.write. Chris Bentzel notes that whitelists are undesirable, so we may wish to forego a whitelist and accept a small percentage (less than 1%) of broken pages that use the Google Loader to load libraries via document.write.

The remaining one page that used document.write to load a script that may be needed for main page content, https://m.ramada.com/hotels/india/powai-mumbai/ramada-powai-hotel-and-convention-centre/hotel-overview, also appears to be fully functional when the load of its document.written script is blocked.

Based on the above, we can mitigate risk of breaking main page content by allowing the load of document.written scripts on the same TLD as the main document, and optionally maintaining a whitelist of document.written scripts that may be needed for main page content. That said, it is unclear that allowing same TLD scripts to load or adding a whitelist results in a significant user benefit, so we may also wish to take the simpler approach of blocking all scripts loaded via document.write until there is clear evidence that exceptions are needed.

This work aims to minimize the number of pages that are broken by blocking the load of scripts inserted via document.write. Our analysis shows that blocking the load of scripts inserted via document.write does not appear to affect main page content for any of the 200 analyzed pages. We expect that a small percentage of page loads, under 0.5%, may be broken by this optimization. This amount of breakage is acceptable given the very substantial performance improvements that it brings.

Decision:

Blocking the load of scripts loaded via document.write for users on 2G connections substantially speeds up roughly 10% of page loads, which are typically among the slowest loading pages. Of 200 analyzed URLs, we did not find any pages whose main content was clearly broken by blocking scripts loaded via document.write. Shivani Sharma will be exploring running a Finch trial in India on 2G in an upcoming Chrome release. We will share additional details about this trial in a separate design document.

Open Questions:

- Should we block scripts loaded by document.write if they are on the same origin as the main document? Currently disinclined, as there isn't evidence that doing so results in any user visible improvement (we have yet to find a page where this is the case; the percentage of pages for which this is the case is likely very small).
- Should we include a whitelist for scripts like those loaded by the Google Loader, which can be loaded via document.write and may be used for main page content? Again, inclined not to do this initially, since there is no evidence that doing so results in a user visible improvement for more than a small handful of pages. It may be better to forego a whitelist and block all scripts, which is simpler overall, and results in breaking less than 1% of pages.

Improve preload scanning to enable discovery of scripts inserted via document.write

Parser-blocking scripts inserted via document.write are not currently discovered by the preload scanner. Thus, the HTML parser blocks on the network fetch of scripts inserted by document.write, which can lead to high parser blocking time, and, in cases where the script is included before the contents of the document, significant increases in time before page content is displayed on the screen.

Many scripts inserted via document.write are inserted via inline script blocks. Many of these script blocks are provided by third-party ad or analytics trackers, and are designed to run in isolation, without depending on other script blocks in the document. Below are a few examples:

```
<script type="text/javascript">
  var useSSL = 'https:' == document.location.protocol;
  var src = (useSSL ? 'https:' : 'http:') +
      '//www.googletagservices.com/tag/js/gpt.js';
  document.write('<scr' + 'ipt src="' + src + '"></scr' + 'ipt>');
</script>

<script type="text/javascript">
  document.write(unescape(
    "%3Cscript src=%27http://s10.histats.com/js15.js%27
type=%27text/javascript%27%3E%3C/script%3E"));
```

```
</script>

<script type="text/javascript">
 var sc_project=NNNNN;
 var sc_invisible=0;
 var sc_security=".....";
 var scJsHost = (("https:" == document.location.protocol) ?
    "https://secure." : "http://www.");
 document.write("<sc"+"ript type='text/javascript' src='" +
    scJsHost+
    "statcounter.com/counter/counter.js'></"+"script>");
</script>
```

Though not trivially parseable by the tokenizer, many of these script blocks are amenable to isolated execution, as they depend on minimal global state. Using a new V8 context, document.write functions can bubble up to blink where we can issue a preload. Note that this technique will not work for a subset of scripts. For instance, nondeterministic URLs, such as those that include a random number or a timestamp, are still not possible to preload. Resources loaded from external scripts (as opposed to inline script blocks) are also not amenable to this technique (see resource prediction).

On pages where we don't elect to block the loading of a script (slow/2G connections), we propose augmenting the preload scanner to mitigate the impact of scripts loaded via document.write on faster connections such as 3G.

Results:
All of the evaluation happens during document parsing. Thus, our key metric was document parse time (domInteractive - domLoading). We also looked at Speed Index, de-noised to subtract out the time it took to receive the first byte of the main payload.
Our test setup used WebPageTest on an emulated 3G connection on a Moto G. We logged traces when the feature was used in any way, so the data could be filtered further.

Initial results are promising for reducing both document parse time and Speed Index. Parse time improvements are the most pronounced. For pages with preloadable scripts loaded via document.write, we see a 13% median improvement and a 15% mean improvement in parse time on 3G. This ends up being around a 1.5 second and 1.8 second improvement at the median and mean, respectively.

Speed Index (denoised) is a noisier metric, but for pages that use the feature, we see a 4.9% improvement at the median and a 3.7% improvement at the mean. This translates to about a 0.95 second and 1.05 second improvements at the median and mean, respectively.

Decision:

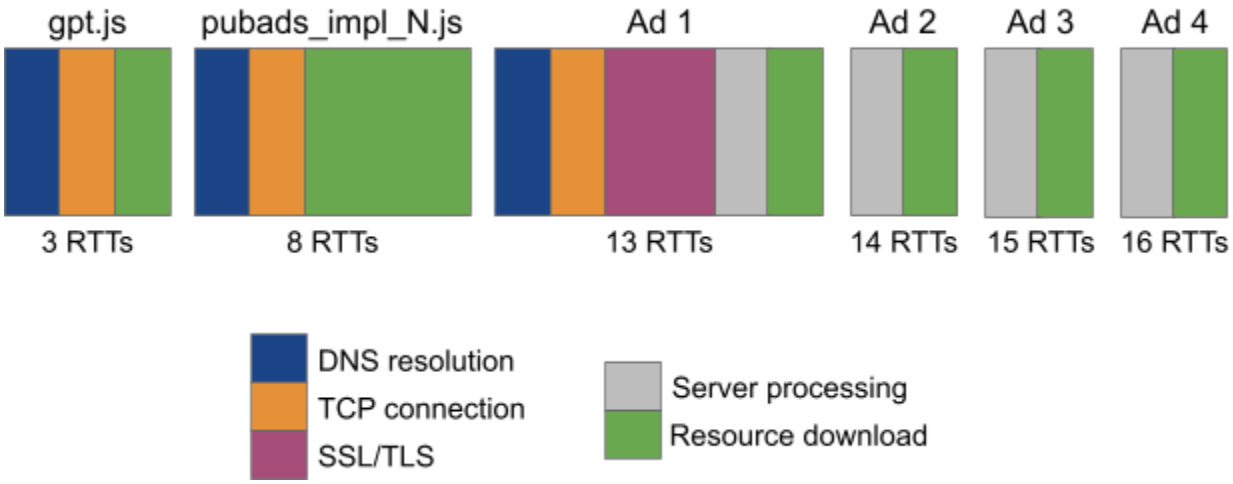
The results from the lab test look good. We have sent out an intent to implement which has received support from other blink developers. Combined with resource prediction, below, we can substantially speed up page loads on 3G and faster connections.

Add resource prediction to fetch scripts that will be written by document.write

The preload scanning process can be optimized to discover scripts that will block the parser via document.write. This is detailed in the above section. However, another large class of scripts inserted via document.write occur in external scripts that were themselves document.written. This "chain" of script insertions effectively force the browser to fetch the desired resources synchronously and serially. We propose adding a resource predictor to preload these chains of external scripts. The predictor should be robust enough to detect when we should preconnect and when we should preload.

We estimate that preload scanning combined with third party resource prediction can address roughly 50% of the existing performance penalty associated with document.write.

For instance, www.googletagservices.com/tag/js/gpt.js, when used synchronously, loads its impl script into the page using document.write. In turn, individual ads are then loaded synchronously on the securepubads.g.doubleclick.net domain. Oftentimes, a page will load multiple ads from the securepubads.g.doubleclick.net domain. In the normal case, each of these loading steps is performed synchronously. As an example, on http://www.india.com/indian-premier-league-2015/chennai-super-kings-vs-delhi-daredevils-ipl-2015-watch-free-live-streaming-and-telecast-of-csk-vs-dd-on-star-sports-online-345865/, we observe the following sequence of parser-blocking activity associated with gpt.js:



The load of pubads_impl_N incurs additional round trips during download due to its larger download size. This sequence of events incurs 16 parser blocking network round trips due to ads loading. At 300ms per round trip, as is typical on 3G, this adds up to 4.5 seconds of parser-blocking latency. Additional delays in the hundreds of milliseconds per resource are

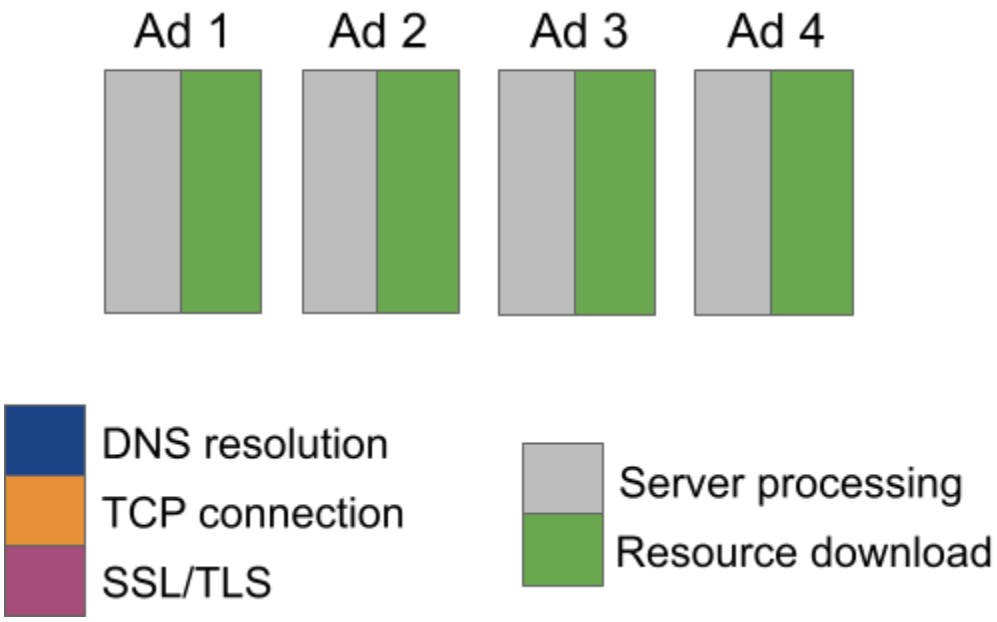
incurred due to per-ad server processing time, resulting in roughly an additional second of delay.

Improved preload scanning, as described previously, can eliminate the parser blocking round trip times for gpt.js. However, the remaining 13 round trips remain. Third party resource prediction can be used to reduce the cost of these round trips.

First, third party resource prediction can learn that whenever gpt.js is loaded, pubads_impl_N.js will be needed soon, and we can prefetch pubads_impl_N.js in parallel with gpt.js, eliminating its parser-blocking round trips.

Next, prediction can learn that when gpt.js or pubads_impl_n are loaded, we are likely to need a connection to the securepubads.g.doubleclick.net domain, which is used to load Ad 1 through Ad 4. Note that, since the URLs for these ads vary from load to load, it is not possible to predict the full URL of the resource. Nonetheless, we can remove the connection overhead for these ads from the critical path of the page load.

The combination of predictively loading pubads_impl_N.js and preconnecting to the securepubads.g.doubleclick.net domain reduces the critical path overhead of ad loading substantially. With these improvements, we reduce the critical path to:



Or 4 network round trips to load the individual ads, plus server processing time for each of the 4 ads. On a connection with a 300ms RTT, we expect this to take roughly 1.2 seconds plus server processing time. This results in a reduction of 3.6 seconds of critical path page load

time. 0.9 seconds of improvement come from improved preload scanning, and the remaining 2.7 seconds come from improved resource prediction.

Other pages exhibit similar patterns. Below are a few examples:

1. http://www.vegrecipesofindia.com/shahi-paneer/ loads the script http://tags.t.tailtarget.com/tag/TT-10518-0/TailTarget/ via a script tag in the HTML markup, which in turn document.writes two parser-blocking scripts: http://d.tailtarget.com/profiles.js and http://partner.googleadservices.com/gampad/google_service.js.
   These scripts are written from an external script, so they are not discoverable by the preload scanner. These scripts have a short cache lifetime, so they are unlikely to be fresh in the cache on a repeat visit after a few hours have passed. Here, prediction can be used to load these scripts in parallel with http://tags.t.tailtarget.com/tag/TT-10518-0/TailTarget/, removing them from the critical path of the page load.
2. http://webmusic.name/songs/1/All_Bollywood_Songs/ loads the script http://soma.smaato.net/oapi/js/smaatoAdDisplay.js via a script tag in the HTML markup, which in turn document.writes the parser-blocking scripts http://soma.smaato.net/oapi/js/htmlParser.js and http://soma.smaato.net/oapi/js/postscribe.js. These scripts are served without an explicit cache lifetime, and so may be heuristically cached based on 10% of their Last-Modified dates, or a few hours.

In the above two example cases, resource prediction can be used to get these scripts out of the critical path of the page load.

Open Issues:

Though pubads_impl_N.js is in the critical path of the page load, it has a very long cache lifetime, and thus, theoretically, we expect it to have a high browser cache hitrate. In practice, developers report that they observe lower cache hit rates for resources than the long cache lifetimes that they expect. Thus, the benefit of predictively loading pubads_impl_N.js in the wild is unclear. To get better insight, Charles Harrison is working on running a Finch experiment. Details of the Finch experiment will be covered in a separate design document for this feature.

Chrome includes several existing predictors. These generally operate at a higher level than subresource dependencies. For instance, the dns prefetch / preconnect predictor in net/ is keyed off navigations. This means when navigating to a new url that has the same pattern of gpt.js → pubads_impl_n.js, the initial load will fail to preconnect.

There is also the autocomplete predictor, which predicts and prerenders based on input from the omnibox. This predictor is orthogonal to preload/preconnect predictors which can work in tandem with prerender.

Another, experimental predictor is Chrome's speculative prefetch. Unlike the predictor in net/, this system does perform the actual fetch for a resource (as opposed to a preconnect). It too is keyed of the main navigation. The feature is still in the experimental phase and is not currently in active development.

Results:
Experiment results are pending.

Decision:
Decisions regarding this feature are pending experiment results.