Don't fight the browser preload scanner

- What's a preload scanner?
- How to tell when the preload scanner is working
- Injected async scripts
- Lazy loading with JavaScript
- CSS background images
- Inlining too many resources
- Rendering markup with client-side JavaScript
- Help the preload scanner help you
- Resources

One overlooked aspect of optimizing page speed involves knowing a bit about browser internals. Browsers make certain optimizations to improve performance in ways that we as developers can't—but only so long as those optimizations aren't unintentionally thwarted.

One internal browser optimization to understand is the browser preload scanner. This post will cover how the preload scanner works—and more importantly, how you can avoid getting in its way.

What's a preload scanner?

Every browser has a primary HTML parser that tokenizes raw markup and processes it into an object model. This all merrily goes on until the parser pauses when it finds a blocking resource, such as a stylesheet loaded with a `<link>` element, or script loaded with a `<script>` element without an `async` or `defer` attribute.

HTML parser diagram.

Fig. 1: A diagram of how the browser's primary HTML parser can be blocked. In this case, the parser runs into a `<link>` element for an external CSS file, which blocks the

browser from parsing the rest of the document—or even rendering any of it—until the CSS is downloaded and parsed.

In the case of CSS files, both parsing and rendering are blocked in order to prevent a flash of unstyled content (FOUC), which is when an unstyled version of a page can be seen briefly before styles are applied to it.

The web.dev home page in an unstyled state (left) and the styled state (right).

Fig. 2: A simulated example of FOUC. At left is the front page of web.dev without styles. At right is the same page with styles applied. The unstyled state can occur in a flash if the browser doesn't block rendering while a stylesheet is being downloaded and processed.

The browser also blocks parsing and rendering of the page when it encounters `<script>` elements without a `defer` or `async` attribute.

Scripts loaded from `<script>` elements with a `type=module` attribute are deferred by default.

The reason for this is that the browser can't know for sure if any given script will modify the DOM while the primary HTML parser is still doing its job. This is why it's been a common practice to load your JavaScript at the end of the document so that the effects of blocked parsing and rendering become marginal.

These are good reasons for why the browser *should* block both parsing and rendering. Yet, blocking either of these important steps is undesirable, as they can hold up the show by delaying the discovery of other important resources. Thankfully, browsers do their best to mitigate these problems by way of a secondary HTML parser called a *preload scanner*.

A diagram of both the primary HTML parser (left) and the preload scanner (right), which is the secondary HTML parser.

Fig. 3: A diagram depicting how the preload scanner works in parallel with the primary HTML parser to speculatively load assets. Here, the primary HTML parser is blocked as it loads and processes CSS before it can begin processing image markup in the `<body>` element, but the preload scanner can look ahead in the raw markup to find that image resource and begin loading it before the primary HTML parser is unblocked.

A preload scanner's role is speculative, meaning that it examines raw markup in order to find resources to opportunistically fetch before the primary HTML parser would otherwise discover them.

How to tell when the preload scanner is working

The preload scanner exists *because* of blocked rendering and parsing. If these two performance issues never existed, the preload scanner wouldn't be very useful. The key to figuring out whether a web page benefits from the preload scanner depends on these blocking phenomena. To do that, you can introduce an artificial delay for requests to find out where the preload scanner is working.

Take this page of basic text and images with a stylesheet as an example. Because CSS files block both rendering and parsing, you introduce an artificial delay of two seconds for the stylesheet through a proxy service. This delay makes it easier to see in the network waterfall where the preload scanner is working.

> *The WebPageTest network waterfall chart illustrates an artificial delay of 2 seconds imposed on the styleesheet.*
>
> **Fig. 4:** A WebPageTest network waterfall chart of a web page run on Chrome on a mobile device over a simulated 3G connection. Even though the stylesheet is artificially delayed through a proxy by two seconds before it begins to load, the image located later in the markup payload is discovered by the preload scanner.

As you can see in the waterfall, the preload scanner discovers the `<img>` element *even while rendering and document parsing is blocked*. Without this optimization, the browser can't fetch things opportunistically during the blocking period, and more resource requests would be consecutive rather than concurrent.

With that toy example out of the way, let's take a look at some real-world patterns where the preload scanner can be defeated—and what can be done to fix them.

This smattering of patterns isn't an exhaustive list, just some common ones.

Injected `async` scripts

Let's say you've got HTML in your `<head>` that includes some inline JavaScript like this:

```
<script>
  const scriptEl = document.createElement('script');
  scriptEl.src = '/yall.min.js';
```

```
  document.head.appendChild(scriptEl);
</script>
```

Injected scripts are `async` by default, so when this script is injected, it will behave as if the `async` attribute was applied to it. That means it will run as soon as possible and not block rendering. Sounds optimal, right? Yet, if you presume that this inline `<script>` comes after a `<link>` element that loads an external CSS file, you'll get a suboptimal result:

This WebPageTest chart shows the preload scan defeated when a script is injected.

Fig. 5: A WebPageTest network waterfall chart of a web page run on Chrome on a mobile device over a simulated 3G connection. The page contains a single stylesheet and an injected `async` script. The preload scanner can't discover the script during the render blocking phase, because it's injected on the client.

Let's break down what happened here:

1. At 0 seconds, the main document is requested.
2. At 1.4 seconds, the first byte of the navigation request arrives.
3. At 2.0 seconds, the CSS and image are requested.
4. Because the parser is blocked loading the stylesheet and the inline JavaScript that injects the `async` script comes *after* that stylesheet at 2.6 seconds, the functionality that script provides isn't available as soon as it could be.

This is suboptimal because the request for the script occurs only after the stylesheet has finished downloading. This delays the script from running as soon as possible. This could have the potential to affect a page's Time to Interactive (TTI). By contrast, because the `<img>` element is discoverable in the server-provided markup, it's discovered by the preload scanner.

So, what happens if you use a regular `<script>` tag with the `async` attribute as opposed to injecting the script into the DOM?

```
<script src="/yall.min.js" async></script>
```

This is the result:

A WebPageTest network waterfall depicting how an async script loaded by using the HTML

Fig. 6: A WebPageTest network waterfall chart of a web page run on Chrome on a mobile device over a simulated 3G connection. The page contains a single stylesheet and a single `async` `<script>` element. The preload scanner discovers the script during the render blocking phase, and loads it concurrently with the CSS.

There may be some temptation to suggest that these issues could be remedied by using `rel=preload`. This would certainly work, but it may carry some side effects. After all, why use `rel=preload` to fix a problem that can be avoided by *not* injecting a `<script>` element into the DOM?

> *A WebPageTest waterfall showing how the rel=preload resource hint is used to promote discovery of an async injected script—albeit in a way that may have unintended side-effects.*
>
> **Fig. 7:** A WebPageTest network waterfall chart of a web page run on Chrome on a mobile device over a simulated 3G connection. The page contains a single stylesheet and an injected `async` script, but the `async` script is preloaded to ensure it is discovered sooner.

Preloading "fixes" the problem here, but it introduces a new problem: the `async` script in the first two demos—despite being loaded in the `<head>`—are loaded at "Low" priority, whereas the stylesheet is loaded at "Highest" priority. In the last demo where the `async` script is preloaded, the stylesheet is still loaded at "Highest" priority, but the script's priority has been promoted to "High".

Resource priority can be discovered in the network tab in modern browsers. For Chrome DevTools in particular, you can right click on the column headers to ensure the priority column is visible. Be sure to test in multiple browsers, as resource priority varies by browser and other factors.

When a resource's priority is raised, the browser allocates more bandwidth to it. This means that—even though the stylesheet has the highest priority—the script's raised priority may cause bandwidth contention. That could be a factor on slow connections, or in cases where resources are quite large.

The answer here is straightforward: if a script is needed during startup, don't defeat the preload scanner by injecting it into the DOM. Experiment as needed with `<script>` element placement, as well as with attributes such as `defer` and `async`.

Ilya Grigorik has written an informative post that goes into great detail about injected `async` scripts. Give it a read if you want a deep dive into this topic.

Lazy loading with JavaScript

Lazy loading is a great method of conserving data, one that's often applied to images. However, sometimes lazy loading is incorrectly applied to images that are "above the fold", so to speak.

This introduces potential issues with resource discoverability where the preload scanner is concerned, and can unnecessarily delay how long it takes to discover a reference to an image, download it, decode it, and present it. Let's take this image markup for example:

```
<img data-src="/sand-wasp.jpg" alt="Sand Wasp" width="384" height="255">
```

The use of a `data-` prefix is a common pattern in JavaScript-powered lazy loaders. When the image is scrolled into the viewport, the lazy loader strips the `data-` prefix, meaning that in the preceding example, `data-src` becomes `src`. This update prompts the browser to fetch the resource.

This pattern isn't problematic until it's applied to images that are in the viewport during startup. Because the preload scanner doesn't read the `data-src` attribute in the same way that it would an `src` (or `srcset`) attribute, the image reference isn't discovered earlier. Worse, the image is delayed from loading until *after* the lazy loader JavaScript downloads, compiles, and executes.

> *A WebPageTest network waterfall chart showing how a lazily-loaded image that is in the viewport during startup is necessarily delayed because the browser preload scanner can't find the image resource, and only loads when the JavaScript required for lazy loading to work loads. The image is discovered far later than it should be.*
>
> **Fig. 8:** A WebPageTest network waterfall chart of a web page run on Chrome on a mobile device over a simulated 3G connection. The image resource is unnecessarily lazy-loaded, even though it is visible in the viewport during startup. This defeats the preload scanner and causes an unnecessary delay.

Depending on the size of the image—which may depend on the size of the viewport—it may be a candidate element for Largest Contentful Paint (LCP). When the preload scanner cannot speculatively fetch the image resource ahead of time—possibly during the point at which the page's stylesheet(s) block rendering—LCP suffers.

For more information on optimizing LCP beyond the scope of this article, read Optimizing Largest Contentful Paint.

The solution is to change the image markup:

```
<img src="/sand-wasp.jpg" alt="Sand Wasp" width="384" height="255">
```

This is the optimal pattern for images that are in the viewport during startup, as the preload scanner will discover and fetch the image resource more quickly.

> *A WebPageTest network waterfall chart depicting an loading scenario for an image in the viewport during startup. The image is not lazily loaded, which means it is not dependent on the script to load, meaning the preload scanner can discover it sooner.*
>
> **Fig. 9:** A WebPageTest network waterfall chart of a web page run on Chrome on a mobile device over a simulated 3G connection. The preload scanner discovers the image resource before the CSS and JavaScript begin loading, which gives the browser a head start on loading it.

The result in this simplified example is a 100-millisecond improvement in LCP on a slow connection. This may not seem like a huge improvement, but it is when you consider that the solution is a quick markup fix, and that most web pages are more complex than this set of examples. That means that LCP candidates may have to contend for bandwidth with many other resources, so optimizations like this become increasingly important.

Images aren't the only resource type that can suffer from suboptimal lazy loading patterns. The `<iframe>` element can also be affected, and since `<iframe>` elements can load many subresources, the impact of performance could be substantially worse.

CSS background images

Remember that the browser preload scanner scans *markup*. It doesn't scan other resource types, such as CSS which may involve fetches for images referenced by the `background-image` property.

Like HTML, browsers process CSS into its own object model, known as the CSSOM. If external resources are discovered as the CSSOM is constructed, those resources are requested at the time of discovery, and not by the preload scanner.

Let's say your page's LCP candidate is an element with a CSS `background-image` property. The following is what happens as resources load:

> *A WebPageTest network waterfall chart depicting a page with an LCP candidate loaded from CSS using the background-image property. Because the LCP candidate image is in a resource type that the browser preload scanner can't examine, the resource is delayed from loading until the CSS is downloaded and processed, delaying the LCP candidate's paint time.*
>
> **Fig. 10:** A WebPageTest network waterfall chart of a web page run on Chrome on a mobile device over a simulated 3G connection. The page's LCP candidate is an element with a CSS `background-image` property (row 3). The image it requests doesn't begin fetching until the CSS parser finds it.

In this case, the preload scanner isn't so much defeated as it is uninvolved. Even so, if an LCP candidate on the page is from a `background-image` CSS property, you're going to want to preload that image:

```
<!-- Make sure this is in the <head> below any
     stylesheets, so as not to block them from loading -->
<link rel="preload" as="image" href="lcp-image.jpg">
```

If your LCP candidate is from a `background-image` CSS property, but that image varies based on the viewport size, you'll need to specify the `imagesrcset` attribute on the `<link>` element.

That `rel=preload` hint is small, but it helps the browser discover the image sooner than it otherwise would:

> *A WebPageTest network waterfall chart showing a CSS background image (which is the LCP candidate) loading much sooner due to the use of a rel=preload hint. The LCP time improves by roughly 250 milliseconds.*
>
> **Fig. 11:** A WebPageTest network waterfall chart of a web page run on Chrome on a mobile device over a simulated 3G connection. The page's LCP candidate is an element with a CSS `background-image` property (row 3). The `rel=preload` hint helps the browser to discover the image around 250 milliseconds sooner than without the hint.

With the `rel=preload` hint, the LCP candidate is discovered sooner, lowering the LCP time. While that hint helps fix this issue, the better option may be to assess whether or not your image LCP candidate *has* to be loaded from CSS. With an `<img>` tag, you'll have more control over loading an image that's appropriate for the viewport while allowing the preload scanner to discover it.

Inlining too many resources

Inlining is a practice that places a resource inside of the HTML. You can inline stylesheets in `<style>` elements, scripts in `<script>` elements, and virtually any other resource using base64 encoding.

Inlining resources can be faster than downloading them because a separate request isn't issued for the resource. It's right in the document, and loads instantly. However, there are significant drawbacks:

- If you're not caching your HTML—and you just can't if the HTML response is dynamic—the inlined resources are never cached. This affects performance because the inlined resources aren't reusable.
- Even if you can cache HTML, inlined resources aren't shared between documents. This reduces caching efficiency compared to external files that can be cached and reused across an entire origin.
- If you inline too much, you delay the preload scanner from discovering resources later in the document, because downloading that extra, inlined, content takes longer.

Take this page as an example. In certain conditions the LCP candidate is the image at the top of the page, and the CSS is in a separate file loaded by a `<link>` element. The page also uses four web fonts which are requested as separate files from the CSS resource.

A WebPageTest network waterfall chart of page with an external CSS file with four fonts referenced in it. The LCP candidate image is discovered by the preload scanner in due course.

Fig. 12: A WebPageTest network waterfall chart of a web page run on Chrome on a mobile device over a simulated 3G connection. The page's LCP candidate is an image loaded from an `<img>` element, but is discovered by the preload scanner because the CSS and the fonts required for the page load in separate resources, which doesn't delay the preload scanner from doing its job.

Now what happens if the CSS *and* all the fonts are inlined as base64 resources?

A WebPageTest network waterfall chart of page with an external CSS file with four fonts referenced in it. The preload scanner is delayed significantly from discovering the LCP image .

Fig. 13: A WebPageTest network waterfall chart of a web page run on Chrome on a mobile device over a simulated 3G connection. The page's LCP candidate is an image loaded from an `<img>` element, but the inlining of the CSS and its four font resources in the `` delays the preload scanner from discovering the image until those resources are fully downloaded.

The impact of inlining yields negative consequences for LCP in this example—and for performance in general. The version of the page that doesn't inline anything paints the LCP image in about 3.5 seconds. The page that inlines everything doesn't paint the LCP image until just over 7 seconds.

There's more at play here than just the preload scanner. Inlining fonts is not a great strategy because base64 is an inefficient format for binary resources. Another factor at play is that external font resources aren't downloaded unless they're determined necessary by the CSSOM. When those fonts are inlined as base64, they're downloaded whether they're needed for the current page or not.

Could a preload improve things here? Sure. You *could* preload the LCP image and reduce LCP time, but bloating your potentially uncacheable HTML with inlined resources has other negative performance consequences. First Contentful Paint (FCP) is also affected by this pattern. In the version of the page where nothing is inlined, FCP is roughly 2.7 seconds. In the version where everything is inlined, FCP is roughly 5.8 seconds.

Be very careful with inlining stuff into HTML, especially base64-encoded resources. In general it is not recommended, except for very small resources. Inline as little as possible, because inlining too much is playing with fire.

Rendering markup with client-side JavaScript

There's no doubt about it: JavaScript definitely affects page speed. Not only do developers depend on it to provide interactivity, but there has also been a tendency to rely on it to deliver content itself. This leads to a better developer experience in some ways; but benefits for developers don't always translate into benefits for users.

One pattern that can defeat the preload scanner is rendering markup with client-side JavaScript:

> *A WebPageTest network waterfall showing a basic page with images and text rendered completely on the client in JavaScript. Because the markup is contained within JavaScript, the preload scanner can't detect any of the resources. All resources are additionally delayed due to the extra network and processing time that JavaScript frameworks require.*
>
> **Fig. 14:** A WebPageTest network waterfall chart of a client-rendered web page run on Chrome on a mobile device over a simulated 3G connection. Because the content is contained in JavaScript and relies on a framework to render, the image resource in the client-rendered markup is hidden from the preload scanner. The equivalent server-rendered experience is depicted in Fig. 9.

When markup payloads are contained in and rendered entirely by JavaScript in the browser, any resources in that markup are effectively invisible to the preload scanner. This delays the discovery of important resources, which certainly affects LCP. In the case of these examples, the request for the LCP image is *significantly* delayed when compared to the equivalent server-rendered experience that doesn't require JavaScript to appear.

This veers a bit from the focus of this article, but the effects of rendering markup on the client go far beyond defeating the preload scanner. For one, introducing JavaScript to power an experience that doesn't require it introduces unnecessary processing time that can affect Interaction to Next Paint (INP).

Additionally, rendering extremely large amounts of markup on the client is more likely to generate long tasks compared to the same amount of markup being sent by the server. The reason for this—aside from the extra processing that JavaScript involves—is that browsers stream markup from the server and chunk up rendering in such a way that avoids long tasks. Client-rendered markup, on the other hand, is handled as a single, monolithic task, which may affect page responsiveness metrics such as Total Blocking Time (TBT) or First Input Delay (FID) in addition to INP.

The remedy for this scenario depends on the answer to this question: **Is there a reason why your page's markup can't be provided by the server as opposed to being rendered on the client?** If the answer to this is "no", server-side rendering (SSR) or statically generated markup should be considered where possible, as it will help the preload scanner to discover and opportunistically fetch important resources ahead of time.

If your page *does* need JavaScript to attach functionality to some parts of your page markup, you can still do so with SSR, either with vanilla JavaScript, or hydration to get the best of both worlds.

Help the preload scanner help you

The preload scanner is a highly effective browser optimization that helps pages load faster during startup. By avoiding patterns which defeat its ability to discover important resources ahead of time, you're not just making development simpler for yourself, you're creating better user experiences that will deliver better results in many metrics, including some web vitals.

To recap, here's the following things you'll want to take away from this post:

- The browser preload scanner is a secondary HTML parser that scans ahead of the primary one if it's blocked to opportunistically discover resources it can fetch sooner.
- Resources that aren't present in markup provided by the server on the initial navigation request can't be discovered by the preload scanner. Ways the preload scanner can be defeated may include (but are not limited to):
  - Injecting resources into the DOM with JavaScript, be they scripts, images, stylesheets, or anything else that would be better off in the initial markup payload from the server.
  - Lazy loading above-the-fold images or iframes using a JavaScript solution.
  - Rendering markup on the client that may contain references to document subresources using JavaScript.
- The preload scanner only scans HTML. It does not examine the contents of other resources—particularly CSS—that may include references to important assets, including LCP candidates.

If, for whatever reason, you *can't* avoid a pattern that negatively affects the preload scanner's ability to speed up loading performance, consider the `rel=preload` resource hint. If you *do* use `rel=preload`, test in lab tools to ensure that it's giving you the desired effect. Finally, don't preload too many resources, because when you prioritize everything, nothing will be.

Resources

- Script-injected "async scripts" considered harmful

- How the Browser Pre-loader Makes Pages Load Faster
- Preload critical assets to improve loading speed
- Establish network connections early to improve perceived page speed
- Optimizing Largest Contentful Paint

Hero image from Unsplash, by Mohammad Rahmani .

□□□ #3
□ □□□ □□□ 2026-10-31 22:10:35 UTC
□ cocobook-sync □□□ 2026-11-04 17:37:49 UTC