Critical CSS Rules — Decreasing time to first render by inlining CSS rules for over-the-fold elements

Gorjan Jovanovski
University of Amsterdam
Amsterdam, Netherlands
jovanovski@gorjan.info

Vadim Zaytsev
Raincode
Brussels, Belgium
vadim@grammarware.net

1 Problem Statement

In this project we explored the scenarios that lead to the blocking of a browser's rendering process during loading of a web page. This is caused by multiple factors, from network speed, to server, client and browser speed, but also by CSS files. CSS, like HTML, are considered render blocking resources, which, until fully downloaded and parsed, do not allow the browser to start painting the web page on the screen. This is understandable for HTML since the main content of the site resides there, but CSS files often have more information than is needed for the rendering process to initiate. This information consists of rules and media queries that apply to elements under the scroll line which are not initially visible on the screen. By testing a list of 1000 most popular sites from Alexa [Ale16], we determined that there is an average of 1.96 seconds added to the time to first render of a page when loading non-inlined CSS files. Further research done on this topic revealed that unnecessarily loading of CSS rules for elements not visible on the first render of a site contributes greatly to this issue, especially if additional GET requests are made, and more rules are parsed into the CSSOM tree. To see if indeed there are sites with content under the scroll line, we used page height data from the top 1000 websites, and compared that info to the most popular screen resolutions obtained from the W3 Consortium, concluding that 82.2% of the tested sites do not fit on 72.2% of the most popular screen resolutions. This goes to show that there are elements that are initially not visible on the screen, yet CSS rules that apply to them need to be parsed into the CSSOM tree before rendering can being.
An often cited solution to this problem in research [WBKW13], by top industry engineers [KO16] and companies [Dev16], is the use of critical (over-the-fold) CSS. Critical CSS refers to rules in the CSS of a page that affect initially visible elements above the scroll line (also refered to as over-the-fold elements), and disregard rules that exclusively affect elements not initially visible. The goal of this project is to analyze the effects that inlining critical CSS rules in a web page, while asynchronously loading the rest, has on the time to first render, and provide a tool to automate the detection, extraction and injection of CSS rules that apply to over-the-fold elements.

2 Research Questions

Using this knowledge, we formulated a hypothesis: **Inlining critical CSS rules in web pages and loading non-critical ones asynchronously, creates a significant decrease in the time to first render**.

In order to successfully test our hypothesis, we first had to answer the following important questions that will direct our research in a right way research:

- **RQ1: Do requests to external CSS files make a significant negative impact on the time to first render?**
- **RQ2: What methods do existing tools use for detection, extraction and inlining of critical CSS rules?**
- **RQ3: How can critical CSS inlining be automated for server-side dynamic web pages?**

2.1 Time to first render

The time to first render is defined as the time until the first non-white content is painted to the browser display. It is impacted by network speed (speed of connection to server), server processing speed, and initial page parsing and rendering. Since the server and connection response time can be improved with upgrades to the hardware components, the best way a web developer to improve the time to first render is to help the browser speed up the processing of objects in the head tag.

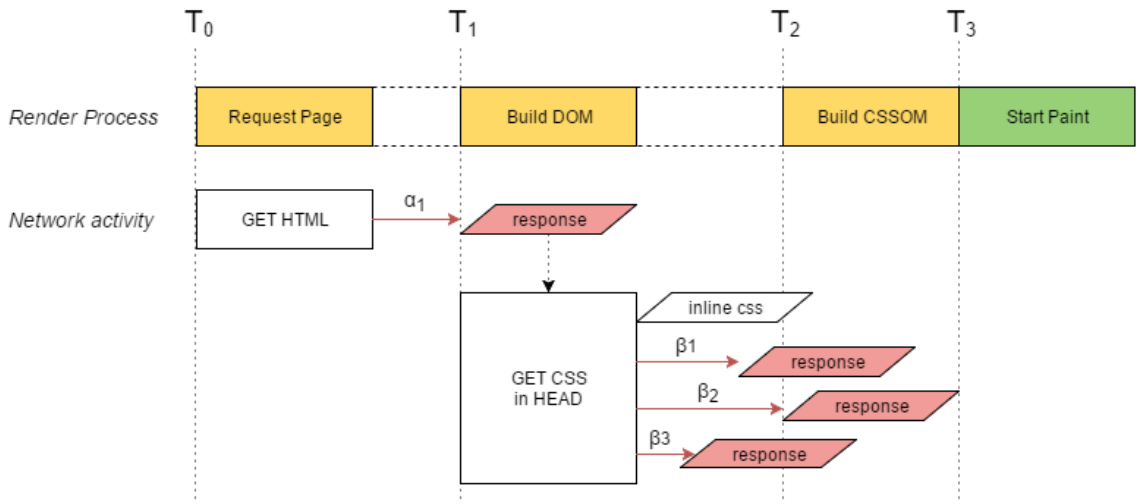

Figure 1: The render process used by all major browsers

As seen in figure 1, the time to first render depends on multiple factors: α_1 — the time that it takes from the beginning of the request, until the full HTML of the document is received; the time to parse the result and the time to receive and parse all requested external CSS files. In the example figure, there are three different CSS files, which the browser starts downloading at the same time. Unlike DOM elements, which can be added to the tree as they are received, CSS rules can override each other, hence the browser can't start building the CSSOM until all CSS files have been downloaded and parsed, including the inlined rules in the head. The time until the CSSOM building process starts, T_2, is equal to the CSS file that takes longest to fetch from the server, in this example, β_2. The time it takes to build the CSSOM, $T_3 - T_2$, is proportional to the sum of the sizes of all CSS files being parsed, meaning more CSS rules processed during the initial loading of the page contribute to a longer time to first render.

$$T_1 = \alpha_1$$

$$T_2 = T_1 + time\_to\_parse\_html() + max(\beta_1, \beta_2, \beta_3)$$

$$T_3 = T_2 + time\_to\_parse\_all\_css()$$

Based on this, we can see that requests for external CSS files cause a bottleneck in the rendering process, as well as parsing more CSS rules, and improving those areas can contribute to decreasing of time to first render.

3 Research

3.1 Related Academic Research

Researchers from the University of British Columbia in Canada worked on a tool called Cilla that is capable of detecting unmatched and ineffective selectors and properties as well as undefined class values [MM12]. Other research has been performed between the Concordia University and the University of British Columbia where refactoring opportunities for CSS code in terms of duplication and size reduction have been analysed [MTM14]. We consider both as important steps for general optimization of CSS code which will keep it clean and up-to-date. However, they do little to help speed up the initial rendering of the page content, something that we focus on in this project.

3.2 Impact of external CSS files on time to first render

We used the top 1000 most popular sites from Alexa, recording the difference in time to first render on every site between having requests for external CSS files blocked and allowed. From the final results that contained 909 successfully processed sites, we deduced that, on average, the time to first render happen around **1.96 seconds** sooner when blocking for requests of external CSS files was enabled. This means that the sum of T_2 and T_3 is reduced on average by 1.96 seconds. The reduction could possibly be even higher since we were testing only the landing pages of websites, where some news-like sites may have more extensive content requiring additional CSS files on inner pages of the site.

We also set a task to gather the height of the analysed web pages, in order to determine how much of the content is hidden under the fold of the most common screen resolutions. WebPageTest.org, the tool used in this research, reported that 80.4% of the pages in our test group have a height over 1100px, which was to be expected, since websites today focus on putting more content on one page in order to let the user have a seamless scrolling experience.

For us to make use of the height data that was gathered, we turned to two separate sources in order to determine the most common screen resolutions in use today. Statistics from the W3 Consortium show that almost 50% of users browse the web on resolutions that have a height $\leq$ 800px. That data corresponds with in-house analytics from TheNextWeb.com, where screen resolutions $\leq$ 800px are shared among 60% of the 67 million visitors of the site from April 2015 to April 2016.

4 Existing tools

We analysed current methodologies and tools that aim to overcome this problem, and by exploring their public Git issues lists and manual inspection of their source code, were able to determine faults and missing features in all implementations. We focused on Node.js implemented tools, that are able to run on any environment and setup.

4.0.1 Penthouse

Penthouse requires the developer to manually specify what CSS files need to be tested against an HTML file and would fail on invalid CSS. It does not handle relative URLs and does no injection of critical CSS in the HTML, but simply outputs it to a file.

4.0.2 CriticalCSS

CriticalCSS has problems while parsing multiple types of pseudo elements and classes in selectors, and is known to invalidate them if they do not strictly adhere to the CSS convention. It does scan the HTML code provided for included CSS files, but can only work on local HTML documents, and does not fetch remotely (developers must use additional libraries to accomplish this). It also does not inject critical CSS in the HTML code.

4.0.3 Critical

Critical is the most robust of all the three Node.js packages that we examined. It does allow inlining of critical CSS in the resulting HTML (and minifies that as well), finds CSS files in HTML, and allows developers to add certain CSS rules to be ignored while parsing CSS files. All of this is accomplished because Critical is a wrapper around Penthouse, adding the extra functionality mentioned to the exiting package, but lacks any option to integrate in dynamic web sites, and incorrectly parses URLs in CSS files.

5 Focusr

Using this knowledge, we present a Node.js tool, named Focusr, that can perform three main functions:

- Detect and extract CSS files linked in input HTML documents.
- Determine and extract CSS rules that apply only to elements inside a specific viewport resolution — critical above-the-fold CSS.
- Remove externally linked CSS files from the resulting HTML page, inline the critical CSS rules and use Javascript to asynchronously load the remainder of the CSS rules.

The main part, detection of over-the-fold elements, and their accompanying critical CSS rules is accomplished by rendering the initial site (local or remote) in a headless browser named PhantomJS. Once rendering is complete, we pass along a list of all CSS selectors found withing linked CSS files in the head of the page, and look for elements that they refer to. The absolute location of each found element on the page is then compared to a predefined viewport by the user for which optimization is being done. If an element, or part of it, resides within this viewport, the accompanying CSS selector that is was found with, is marked as critical. Once all rules are processed, the critical ones are embedded in a style tag in the head of the page, while the original links to the CSS files are loaded via Javascript asynchronously during loading.

For static sites, this is accomplished pre-deployment, generating a new HTML file with inlined critical CSS rules and Javascript code. This allows Focusr to be added as a step to an already existing deployment script or routine (such as Grunt or Gulp), or run independently as a separate tool.

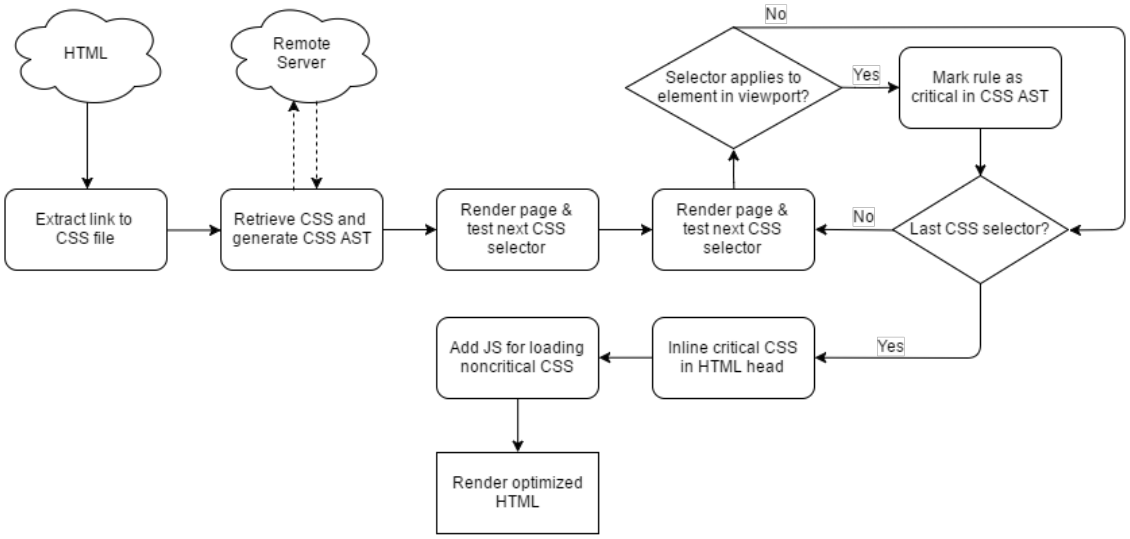

Figure 2: A solution outline of Focusr

5.1 Dynamic sites

In order to show that the same improvements can be accomplished on dynamic sites as well, we created a proof-of-concept plugin for Wordpress, chosen for having over 60% of the global market share of content management system. It uses the generated critical CSS rules and Javascript files from Focusr, to hook into the action system of Wordpress and deliver them in the header and footer accordingly, all while using regular expressions to remove any existing link tags that are present. The extraction of critical CSS happens once pre-deployment (like in static pages), and the removal and inlining stages happen every time a page is requested, allowing Focusr to work independent of any theme/plugin setup in Wordpress.

6 Scenarios

Focusr works best on sites that have CSS rules for elements that are not initially visible on screen (hidden, or under the scroll line). If there is no mismatch between the viewport resolution, and the website resolution, it still may provide an advantage by hiding CSS for elements like modal boxes and menus, which are hidden during load. One could argue that inlining all the CSS rules is also a solution, but since more and more sites use CSS libraries like Bootstrap, Kickstart, Skeleton or Google/Adobe fonts, inlining all of them would still pose a strain on the creation of the CSSOM tree, thing reducing the time saved to a minimum.

7 Validation

We validated our work by running the same test on the initial list of 1000 websites, but this time comparing the time difference between the original and Focusr optimised versions of the sites. An average 1.3 second reduction in the sum of T_2 and T_3 was observed, corresponding to our initial research results, taking into account that

this time around there was actual data to be added to the CSSOM. The same improvement in the time to first render was observed with the Wordpress plugin deployed on TheNextWeb.com, where 9 consecutive tests on a 5 Mbps connection showed a 2.3 second speedup, where a 3.22 second speedup was observed on lower, 3G 1.6 Mbps connection. As future work, the Wordpress plugin could be made independent from the main Focusr tool, allowing for analysis and extraction of critical CSS rules by its self as a specialized Wordpress tool.

References

[Ale16] Alexa. Alexa Top Sites. Web, 2016.

[Dev16] Google Developers. Critical rendering path. Web, 2016.

[KO16] Paul Kinlan and Addy Osmani. Detecting Critical CSS For Above-the-Fold Content. Web, 2016.

[MM12] Ali Mesbah and Shabnam Mirshokraie. Automated analysis of css rules to support style maintenance. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 408–418. IEEE, 2012.

[MTM14] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. Discovering refactoring opportunities in cascading style sheets. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 496–506. ACM, 2014.

[WBKW13] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying page load performance with wprof. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 473–485, 2013.