International Journal of Computer (IJC)
ISSN 2307-4523 (Print & Online)
https://ijcjournal.org/index.php/InternationalJournalOfComputer/index

# Methods of Automated CSS Refactoring for Web Application Performance Optimization

Evgeny Klimenchenko*

*Software Engineer, JPMorgan, London, UK*
*Email: me@evgeny.dev*

**Abstract**

This paper investigates the effectiveness of automated CSS refactoring techniques in optimizing web application performance. Focusing on two key methods - removal of unused CSS and implementation of scoped CSS - the study conducts experiments on both dynamic and static web pages. Performance metrics such as First Contentful Paint (FCP) and Largest Contentful Paint (LCP) are used to measure the impact of these techniques. The results reveal that removing unused CSS consistently improves performance, with a 4.77% decrease in loading time for dynamic pages and a 3.58% decrease for static pages. Surprisingly, the implementation of scoped CSS led to slight performance degradations in the test environment. This research provides insights into the relative effectiveness of these automated CSS optimization strategies and highlights the need for context-specific testing in web development practices. The findings contribute to the ongoing discussion on best practices for CSS performance optimization in modern web applications.

***Keywords:*** CSS Optimization; Web Performance; Automated Refactoring.

## 1. Introduction

In this paper, we will look at Cascading Style Sheets (CSS). CSS is important for controlling the visual presentation of web pages, playing a major role in modern web development. CSS allows developers to design complex layouts and deliver a seamless user experience. However, as web applications grow, so does the complexity and size of their CSS codebase. This often leads to inefficient CSS, so it directly affects the performance of web pages by increasing load times and rendering delays. If CSS is poorly optimized it can lead to what is known as "render-blocking," where a webpage's content cannot be displayed until the CSS is fully loaded and CSS Object Model (CSSOM) is constructed [1].

---

*Received: 7/27/2024*
*Accepted: 9/27/2024*
*Published: 10/7/2024*

---

* Corresponding author.

Therefore, CSS optimization becomes critical for improving both user experience and search engine rankings.Some studies have shown that an increase in a web page's load time from 1 to 10 seconds correlates with a 123% increase in bounce rates. This might prove the importance of performance optimization for user retention that leads to business success [2].To bring more clarity we need to understand what it means to refactor CSS. Refactoring CSS means cleaning up and optimizing the code to make it more efficient without altering its external behavior. In other words all the predefined CSS functionality should stay the same after we refactor it for performance. In large-scale projects, manual CSS refactoring is labor-intensive, time-consuming, and error-prone. Automated CSS refactoring approaches enable developers to incorporate refactoring into their development pipelines, ensuring more efficient enforcement of CSS best practices across teams. There are many different methods to make your CSS more performant such as Modular CSS (breaking CSS into small/reusable components), use of utility classes (creating small, single-purpose utility classes for common styling needs, e.g. popular library – Tailwind[3]), avoiding over-specificity, removing of unused classes etc. However, with rapidly growing CSS codebase developers would want to look towards automating these optimizations for the sake of time saving. The objective of this paper is to present automated CSS refactoring techniques and assess their impact on performance metrics. This study aims to show which optimizations are more critical than others.

The motivation behind this research is to explore and document the most effective methods for automatic CSS performance optimization, particularly in a real-world context where the impact of automated tools remains underexplored. In this paper I will use evaluation techniques that will measure load times, file size reduction, and rendering efficiency. Additionally, I will investigate how these tools can be integrated into development workflows for long-term scalability and maintainability. This research aims to enhance understanding of the ways in which automated CSS refactoring can improve both user experience and developer efficiency. This will also improve my understanding of the topic as I have a great interest in the modern web development.

This paper begins with a literature review on the role of CSS in web performance. Next I will present analysis of modern tools and methodologies for CSS refactoring. I will conduct an experiment to measure the performance improvements brought by automated refactoring tools. At the end I will discuss the results to highlight the strengths, limitations, and future potential of these technologies.

## 2. Literature Review

### *2.1. Introduction to CSS Performance Issues*

#### *2.1.1.Overview of CSS's Role in Web Performance*

There are different ways that CSS can impact the performance of the application. I will go over each of those. First is the file size. When you have a non-modular CSS it can grow into a huge file and when a user tries to open a page this file needs to be loaded first, before user can see anything. As I discussed earlier CSS needs to load and then build CSSOM in order to show anything on the page and the size of the CSS file will affect that. Second is specificity. Specificity in CSS is the calculation of the priority of CSS rules, the bigger the specificity the longer it will take for the CSS engine to make all the calculations [4]. The last one is rendering delays. This happens when the browser tries to apply all the CSS styles on the page. This is often caused by inefficient styles that force the browser to repaint the layout multiple times before content can be shown.

In modern web development all of these issues can be addressed by using different tools and techniques. Developers are moving towards modular CSS techniques, where CSS is split into smaller, reusable components. This helps to reduce the file sizes and improves maintainability of the CSS codebase. Also, by using utility first CSS frameworks, such as Tailwind CSS [3], we can use small, single-purpose classes, helping reduce specificity and avoid deep nesting. These strategies help optimize performance by reducing the overall complexity of the CSS file and ensuring faster rendering times.

In conclusion, CSS performance is affected by file size, specificity, and rendering delays. Optimizing these factors improves load times and enhances the user experience. By understanding how CSS impacts performance, developers can make informed choices to create efficient, scalable, and maintainable stylesheets that are less prone to the issues mentioned above.

The key studies that go further about CSS performance issues:

- Improving Web Performance by Optimizing Cascading Style Sheets (CSS): Literature Review and Empirical Findings [5] - The article provides an overview of CSS optimization techniques. However, it does not provide good information about different automatic CSS performance techniques.
- Study: Web Users Prefer Speed Over Customization [6] - A well-designed study demonstrates that the speed of the user interface is the most crucial factor among other elements, such as adaptive behaviour and content density.
- Optimizing CSS for performance Chapter 10 of AdvancED CSS book by Joseph R. Lewis and Meitar Moscovitz [7] - The book explains CSS performance issues and how to fix them. This book is a bit old, but It is still useful even though technology is always changing.

While several studies, such as those by Kuparinen (2023) [5], have explored the optimization of CSS performance, few have delved into automated methods. Kuparinen's work emphasizes manual optimization techniques, which, while effective, are labor-intensive and prone to error, especially in large codebases. This gap highlights the need for more research on automated solutions, such as the ones explored in this paper. Moreover, the work of Lewis and Moscovitz (2009) [7] on CSS performance, though foundational, does not account for advancements in tooling like PurgeCSS or modern utility-first frameworks like Tailwind CSS. This paper aims to extend these earlier findings by focusing on automation, addressing a critical need in modern web development workflows where scalability and maintainability are key concerns.

#### *2.1.2. Key Metrics Affected by CSS*

In the context of web performance, we are interested in certain metrics that will help us in measuring and comparing our automated techniques. The following key metrics are commonly used to measure the impact of CSS on the overall performance of a website:

- ***First Contentful Paint (FCP)***

FCP measures how long it takes for the first piece of content to be rendered on the screen. This content could

be text, an image, or a non-white background. CSS affects FCP because the browser must fully load and apply CSS before rendering any styled content. Large or inefficient CSS files can delay this process, causing slower FCP times. More on this can be found at Google Web.dev on FCP [8].

- ***Largest Contentful Paint (LCP)***

LCP measures the time it takes for the largest visible content element (like a hero image or a large text block) to be rendered on the screen. LCP is especially important for user experience, as users perceive slower LCP times as slow-loading pages. CSS impacts LCP in similar ways to FCP, as the browser must apply styles before rendering larger elements. If the CSS is bloated or inefficient, LCP times will increase.Learn more about LCP at

- Google Web.dev on LCP [9].

***Cumulative Layout Shift (CLS)***

CLS measures how much visible content shifts while the page is loading. This can happen if elements on the page (like images or text) move unexpectedly as other resources (such as CSS) are loaded and applied. Poorly optimized CSS, especially in combination with large stylesheets or delayed font loading, can cause layout shifts, leading to a higher CLS score. Further reading on CLS is available at Google Web. Dev on CLS [10].
***Time to Interactive (TTI)***

TTI measures how long it takes for a page to become fully interactive. In other words, this is the time it takes from when a user first opens the page until it can respond to user inputs (clicks, scrolling, etc.). CSS can influence TTI by delaying the time it takes for the page to render, particularly if the CSS blocks rendering or has high specificity.
Explore more about TTI at Google Web.dev on TTI [11].

### *2.2. CSS Performance optimization techniques*

#### *2.2.1.Class Scoping*

When CSS is not scoped, the browser has more rules to check. This increases rendering delays. Also, there is a possibility that CSS rules will leak to an element that wasn't meant to be styled with the rule. So what is CSS scoping? Scoped styled allow you to contain a set of styles within a single component or a set of components on the page. You can do this many different ways.

The easiest of them all is to name every single element with a unique class name. Some libraries use a string of generated tokens to add to your class names to make them unique. This technique has a very high potential for automating.

The second option is Shadow DOM. The shadow DOM enables complete encapsulation of styles within web components. It isolates the styles applied to a specific component, preventing them from affecting other parts of the page. Shadow DOM is mostly used in development of web components and provides a great isolation of

styles. This technique might be difficult to apply using automation than class naming. It is still possible to achieve, but will need the automation script to be aware of component context [12].

The last way of doing class scoping is the use of the CSS @scope rule. It is currently under development and not widely supported yet. As of the time of writing, Chrome (128) and Safari (17.5) do support it, but Firefox (130) doesn't [13]. This technique can be difficult to automate. Same issues as Shadow DOM. The script must know the component's context to scope it.

The honorable mention is the 'scoped' attribute on the <style> tag. This paper will not mention it since it has been deprecated.

Scoping classes through these methods ensure that the CSSOM size is small and speed up the browser's ability to match styles to elements, improving overall page rendering performance.

The key resources to read about Class Scoping are:

- Improving Web Performance by Optimizing Cascading Style Sheets (CSS): Literature Review and Empirical Findings [5]
- "Using Shadow DOM," MDN Web Docs [12]
- "Styled Scoped" by C. Coyier [14]
- "@scope," MDN Web Docs [15]

#### *2.2.2. Class Scoping*

Unused CSS refers to the styles that are included in the code but are not applied to any elements in the current page view. These unused styles get left out as the project grows. The first downside of unused CSS is that it increases the CSS file size. Thus, it increases download times and negatively impacts performance. Second, when a browser downloads CSS, it must process the entire file, even if only a small part of the styles is actually applied to the page. This creates processing overhead, as the browser spends time parsing and loading unused styles. Removing unused CSS not only reduces file size but also speeds up the CSSOM construction process, leading to faster page rendering.

It is a very difficult problem to solve. If you already have some unused CSS, you will need to find a way to remove it. Automation might not be an answer. To understand which CSS selectors are unused, you should check coverage on every single page of your entire site. You should check it while running all JavaScript. Test all combinations of states and media queries [16].

I will analyse different ways of removing Unused CSS and see which automatic removal provides the best results. Also, I want to understand what the performance implications are when we have unused CSS.

The key resources to read about Unused CSS:

- "How Do You Remove Unused CSS From a Site?" by C. Coyier [16] "CSS performance optimization" by *MDN Web Docs* [17]
- "Optimizing for Production" TailwindCSS documentation [18]

## 3. Methodology

In this section, I will try to outline the approach to analysing the impact of automatic CSS refactoring. The two CSS performance issues that I will explore, discussed in previous section, are class scoping and unused CSS. They are identifies as key areas for optimization in large-scale applications [5].

### *3.1. Experiment Setup*

To evaluation the effect of class scoping and unused CSS optimization, I will use several key performance metrics (such as First Contentful Paint (FCP), Largest Contentful Paint (LCP), and Time to Interactive (TTI)). Those key metrics will be used before and after I apply automated refactoring techniques.

The steps for these experiments will include:

- ***Selecting Web Pages for Analysis:***

I will have a set of web pages selected for this analysis. There will be static and dynamic page types so we can see the performance implications on both. Some pages will have unscoped CSS with long queries. Others will have a lot of unused styles. For each of the test web pages, we will have baseline webpages with optimized CSS. The pages will go through a performance audit tool (e.g., Google Lighthouse).

- ***Tools for Measuring Performance:***

I will measure the performance of each page using tools like:

  - *Google Lighthouse:* It provides metrics on page performance, including FCP, LCP, and TTI.
  - *Chrome DevTools:* The Coverage tool in DevTools will be used to identify the amount of unused CSS on each page. This tool is not very reliable in identifying the correct percentage of unused CSS. Thus, I will use it with caution.
  - *Custom Python Script:* A Python script will be developed to automate the performance measurements. This script will extract metrics from Lighthouse reports. It will show the performance differences before and after refactoring.

### *3.2. Analysis of Class Scoping*

I will analyse class scoping by comparing performance metrics from two versions of the same web pages: one using global CSS and the other using scoped CSS. I will use CSS Modules to generate unique class names for each component automatically. CSS Modules will be integrated into the build process. This will ensure that styles are scoped automatically.

I will use performance metrics such as FCP and LCP to compare between the globally scoped and class-scoped

versions. I expect that scoping will lead to reduced CSSOM size and faster rendering times.

### *3.3. Analysis of Unused CSS*

Same will go for the unused CSS. I will assess the impact of removing unused CSS by running the baseline version first. Then I will remove all unused CSS using tools such PurgeCSS [19]. The same performance metrics (FCP, LCP, and TTI) will be used to understand performance impact of original and optimized versions of the web pages.

### *3.4. Reporting*

The performance differences between the original and refactored pages will be visualized using graphs. The custom Python script mentioned earlier will:

- Pull the Lighthouse reports for each version of the pages.
- Extract the relevant metrics (FCP, LCP, TTI).
- Plot graphs comparing performance before and after applying class scoping and unused CSS removal techniques.

These graphs will illustrate the impact of automated CSS refactoring, showing improvements in load times, rendering performance, and overall page interactivity. The results will be analysed to determine which technique (class scoping or unused CSS removal) has the most significant impact on performance, and the findings will be discussed in the next section of the paper.

## 4. Experiment and Analysis

For this experiment, I created two simple webpages. One had dynamic rendering, where I called a JS function that attaches some HTML to the root component. The second one was a static page. Both of them had 500+ unused styles, some deeply nested styles, and some complicated queries. You can find the repository with everything I used for the analysis on GitHub [20].

I used the *serve* CLI command to run my webpage on an HTTP server [21]. Then I ran the CLI version of Lighthouse with some throttling configurations.

```
lighthouse http://localhost:62129/html_dynamic_unoptimized --throttling-
method=devtools \
    --throttling.cpuSlowdownMultiplier=4 \
    --throttling.downloadThroughputKbps=1500 \
    --throttling.uploadThroughputKbps=750 \
    --throttling.latencyMs=150
```

I chose throttling to simulate a slower, real-world environment. The web pages I tested are small, so without throttling, the performance differences might not be noticeable. By slowing down the CPU and network speed, I

created conditions similar to those on mobile devices or slower networks. This helps show how the optimizations would perform in a more realistic scenario. Without throttling, the results could be too fast to reveal meaningful differences.For all the performance tests through Lighthouse, I did three independent cache-free runs. Then I took the average numbers of key metrics of interest, such as FCP and LCP.

The code for the Python script to get the averages from Lighthouse JSON is below.

```
import json
import os

# Function to load JSON data from a file
def load_json(file_path):
	with open(file_path, 'r') as f:
		return json.load(f)

# Function to extract relevant metrics (FCP, LCP)
def extract_metrics(data):
	fcp = []
	lcp = []

	# Loop through each run and extract metrics
	for run in data:
		fcp.append(run['audits']['first-contentful-paint']['numericValue'])
		lcp.append(run['audits']['largest-contentful-
paint']['numericValue'])

	return {
		'fcp': fcp,
		'lcp': lcp,
	}

# Function to calculate average of a list
def calculate_average(metrics):
	return sum(metrics) / len(metrics)

# Load unoptimized and optimized results from files
unoptimized_file = os.path.join(os.path.dirname(__file__),
'static_unoptimized_results.json')
optimized_file = os.path.join(os.path.dirname(__file__), 'scoped_css',
'static_optimized_results.json')

unoptimized_data = load_json(unoptimized_file)
optimized_data = load_json(optimized_file)

# Extract metrics from both datasets
unoptimized_metrics = extract_metrics(unoptimized_data)
optimized_metrics = extract_metrics(optimized_data)

# Example: Calculate and print the average for each metric (Unoptimized)
```

```
print("Unoptimized Metrics Averages:")
print(f"FCP: {calculate_average(unoptimized_metrics['fcp']):.2f} ms")
print(f"LCP: {calculate_average(unoptimized_metrics['lcp']):.2f} ms")

# Example: Calculate and print the average for each metric (Optimized)
print("\nOptimized Metrics Averages:")
print(f"FCP: {calculate_average(optimized_metrics['fcp']):.2f} ms")
print(f"LCP: {calculate_average(optimized_metrics['lcp']):.2f} ms")
```

Lighthouse results for dynamic unoptimized page were like this:

```
Unoptimized Metrics Averages:
FCP: 1267.93 ms
LCP: 1267.93 ms
```

Results for static unoptimized page:

```
Unoptimized Metrics Averages:
FCP: 1275.03 ms
LCP: 1275.03 ms
```

### *4.1. Unused CSS*

For my first analysis of automated CSS refactoring, I wanted to show how unused CSS impacts the performance of my webpage. I used PostCSS [22] with the PurgeCSS plugin applied [19].

PostCSS configuration file `postcss.config.js`:

```
import purgecss from '@fullhuman/postcss-purgecss'

export default (ctx) => ({
  plugins: [
    purgecss({
          content: ['./**/*.html']
    })
  ]
})
```

I used the PostCSS CLI command to automatically clean the CSS of unused queries.

```
npx postcss ./static_unoptimized.css -o ./unused_css/static_optimized.css
```

Then I executed the lighthouse performance test on a newly created page with optimized CSS. The results were pretty good, considering that the webpage was small.

Optimized dynamic web page results:

```
Optimized Metrics Averages:
FCP: 1207.40 ms
LCP: 1207.40 ms
```

Optimized static web page results:

```
Optimized Metrics Averages:
FCP: 1229.40 ms
LCP: 1229.40 ms
```

### *4.2. Scoped CSS*

To apply scoped CSS optimization to our CSS, I used PostCSS too. Yet, this time I used the `postcss-modules` plugin [23]. The CLI command to optimize the CSS was the same as for section 4.1. The script changed all CSS queries, including global queries, to this format `[name]__[local]___[hash:base64:5]`. The nesting of global queries persisted and had to be fixed manually.

The results of the lighthouse performance analysis after CSS optimization are as follows.

Optimized dynamic web page results:

```
Optimized Metrics Averages:
FCP: 1266.81 ms
LCP: 1266.81 ms
```

Optimized static web page results:

```
Optimized Metrics Averages:
FCP: 1285.35 ms
LCP: 1285.35 ms
```

### *4.3. Analysis*

For easy visualization I have used a python script to plot the differences of FCP and LCP.

```
import matplotlib.pyplot as plt
import numpy as np

# Data
categories = ['Dynamic', 'Static']
optimizations = ['Removing Unused CSS', 'Making Scoped CSS']
```

```
# Metrics for Removing Unused CSS
removing_unused_css = {
    'Dynamic': {'Unoptimized': 1267.93, 'Optimized': 1207.40},
    'Static': {'Unoptimized': 1275.03, 'Optimized': 1229.40}
}

# Metrics for Making Scoped CSS
making_scoped_css = {
    'Dynamic': {'Unoptimized': 1251.32, 'Optimized': 1266.81},
    'Static': {'Unoptimized': 1275.03, 'Optimized': 1285.35}
}

# Prepare the plot
fig, axes = plt.subplots(1, 2, figsize=(14, 6), sharey=True)
fig.suptitle('Average FCP and LCP Times Before and After Optimization',
fontsize=16)

# Plot settings
bar_width = 0.35
opacity = 0.8
index = np.arange(len(categories))

# Function to create bar charts
def create_bar_chart(ax, data, title):
    unoptimized = [data[cat]['Unoptimized'] for cat in categories]
    optimized = [data[cat]['Optimized'] for cat in categories]

    ax.bar(index, unoptimized, bar_width, alpha=opacity, label='Unoptimized',
color='skyblue')
    ax.bar(index + bar_width, optimized, bar_width, alpha=opacity,
label='Optimized', color='salmon')

    ax.set_xlabel('Content Type')
    ax.set_ylabel('Time (ms)')
    ax.set_title(title)
    ax.set_xticks(index + bar_width / 2)
    ax.set_xticklabels(categories)
    ax.legend()
    ax.grid(True, linestyle='--', alpha=0.7)

# Create bar charts for each optimization strategy
create_bar_chart(axes[0], removing_unused_css, 'Removing Unused CSS')
create_bar_chart(axes[1], making_scoped_css, 'Making Scoped CSS')

plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

**5. Results**

The experiments conducted on both dynamic and static web pages showed some insights into the effectiveness

of automated CSS refactoring techniques. The two main techniques analyzed were the removal of unused CSS and the implementation of scoped CSS.

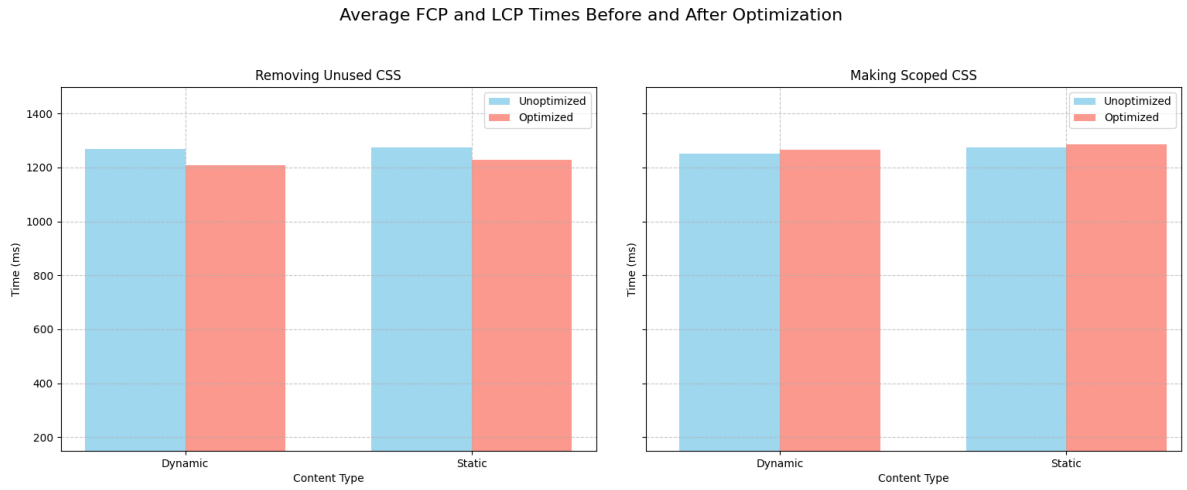


**Figure 1:** The results of analysis

### *5.1. Removal of Unused CSS*

For the dynamic web page:

- Unoptimized FCP and LCP: 1267.93 ms
- Optimized FCP and LCP: 1207.40 ms
- Improvement: 60.53 ms (4.77% decrease)

For the static web page:

- Unoptimized FCP and LCP: 1275.03 ms
- Optimized FCP and LCP: 1229.40 ms
- Improvement: 45.63 ms (3.58% decrease)

### *5.2. Implementation of Scoped CSS*

For the dynamic web page:

- Unoptimized FCP and LCP: 1251.32 ms
- Optimized FCP and LCP: 1266.81 ms
- Degradation: 15.49 ms (1.24% increase)

For the static web page:

- Unoptimized FCP and LCP: 1275.03 ms

- Optimized FCP and LCP: 1285.35 ms
- Degradation: 10.32 ms (0.81% increase)

## 6. Discussion

The results of our experiments reveal several interesting insights into the effectiveness of automated CSS refactoring techniques.

- Removal of Unused CSS:
  - This technique showed positive results for both dynamic and static web pages.
  - The improvement was more pronounced in the dynamic web page (4.77% decrease in loading time) compared to the static web page (3.58% decrease).
  - These results suggest that removing unused CSS can lead to modest but meaningful performance improvements, particularly for dynamic content.
- Implementation of Scoped CSS:
  - Contrary to expectations, the implementation of scoped CSS led to a slight degradation in performance for both dynamic and static pages.
  - The dynamic page saw a 1.24% increase in loading time, while the static page experienced a 0.81% increase.
  - This unexpected result warrants further investigation. Possible explanations is the benefits of scoped CSS might be more apparent in larger, more complex applications where selector conflicts are more likely to occur.
- Comparison of Techniques:
  - Removal of unused CSS consistently outperformed the implementation of scoped CSS in our experiments.
  - This suggests that, for immediate performance gains, focusing on eliminating unused styles might be more beneficial than implementing scoped CSS.
- Limitations:
  - The scope of this study is limited to relatively simple web applications, which may not fully reflect the performance implications of automated CSS refactoring techniques in more complex systems. Additionally, the experiments were conducted with a single set of web pages, and the diversity of page structures and CSS architectures in real-world applications might lead to varying results. The chosen metrics, such as FCP, LCP, and TTI, offer valuable insights, but further studies incorporating user-perceived performance and scalability metrics could provide a more comprehensive understanding. Finally, while tools like PurgeCSS and PostCSS Modules proved effective, their applicability to legacy systems and compatibility with various CSS frameworks remains underexplored.

The results from the removal of unused CSS and the implementation of scoped CSS, though subtle, offer critical insights into the nuances of performance optimization in modern web applications. The reduction in load times through the elimination of unused CSS highlights the impact of removing unnecessary elements, particularly in dynamic content-heavy environments. However, the slight degradation in performance metrics for scoped CSS

might be due to the relatively simple test environment. In larger, more complex web applications, scoped CSS is likely to offer more substantial improvements by mitigating selector conflicts and preventing style leakage across components. Future work should focus on testing scoped CSS in environments with more intricate CSS architectures to reveal its true potential.

## 7. Conclusion

This study showed methods of automated CSS refactoring and their performance impact on the web application. Removing of the unused CSS proved to be an effective optimization technique, consistently improving performance across both dynamic and static web pages. The implementation of scoped CSS, contrary to expectations, led to slight performance degradations in our specific test environment. However, the scoped CSS performance degradation could be explained by the small test environment that couldn't show the real benefit that it might lead to in large applications.

Further investigation into performance implications of scoped CSS in large, more complex applications can be done in the future. A more comprehensive study involving a wider range of web applications and CSS optimization techniques would provide a broader understanding of their relative effectiveness.

In conclusion, while automated CSS refactoring techniques show promise for improving web performance, their effectiveness can vary based on the specific technique and application context. Web developers should carefully consider and test these techniques in their unique environments to ensure they achieve the desired performance improvements. As web technologies continue to evolve, ongoing research and experimentation in this area will be crucial for developing more effective and reliable CSS optimization strategies.

## References

[1] Google, "Optimize resource loading," *web.dev*. [Online]. Available: https://web.dev/learn/performance/optimize-resource-loading. [Accessed: 05-Sep-2024].

[2] Google/SOASTA Research, 2017. [Online]. Available: https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/page-load-time-statistics/. [Accessed: 05-Sep-2024].

[3] Tailwindcss Documentation [Online]. Available https://tailwindcss.com/docs/installation. [Accessed: 05- Sep-2024]

[4] E. J. Etemad and T. Atkins Jr., Selectors Level 4. *W3C*, Mar. 2024. [Online]. Available: https://drafts.csswg.org/selectors/#specificity-rules [Accessed: 06-Sep-2024]

[5] S. Kuparinen, Improving Web Performance by Optimizing Cascading Style Sheets (CSS): Literature Review and Empirical Findings, Master's thesis, *Faculty of Science*, University of Helsinki, Helsinki, Finland, May 2023. [PDF]. Available: https://helda.helsinki.fi/server/api/core/bitstreams/694695cf-1bdf-4432-b255- f68d6bdb4b76/content

[6] "Study: Web Users Prefer Speed Over Customization," *Website Optimization*, Aug. 12, 2018. [Online]. Available: https://www.websiteoptimization.com/speed/tweak/design-factors/ [Accessed: 06-Sep-2024]

[7] J. R. Lewis and M. Moscovitz, "Optimizing CSS for Performance," in AdvancED CSS, *Apress*, 2009, pp. 275–289. [Online]. Available: https://doi.org/10.1007/978-1-4302-1933-0_10 [Accessed: 06-Sep-2024]

[8] P. Walton, "First Contentful Paint (FCP)," *web.dev*, Dec. 6, 2023. [Online]. Available: https://web.dev/articles/fcp [Accessed: 06-Sep-2024]

[9] P. Walton, "Largest Contentful Paint (LCP)," *web.dev*, Feb. 19, 2024. [Online]. Available: https://web.dev/articles/lcp [Accessed: 06-Sep-2024]

[10] P. Walton and M. Mihajlija, "Cumulative Layout Shift (CLS)," *web.dev*, Apr. 12, 2023. [Online]. Available: https://web.dev/articles/cls [Accessed: 06-Sep-2024]

[11] P. Walton, "Time to Interactive (TTI)," *web.dev*, Nov. 17, 2023. [Online]. Available: https://web.dev/articles/tti [Accessed: 06-Sep-2024]

[12] "Using Shadow DOM," *MDN Web Docs*, Aug. 30, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_shadow_DOM [Accessed: 09- Sep-2024]

[13] "@scope," *Can I use*, Aug. 2024. [Online]. Available: https://caniuse.com/?search=@scope [Accessed: 09-Sep-2024]

[14] C. Coyier, "Style Scoped," *chriscoyier.net*, Oct. 19, 2023. [Online]. Available: https://chriscoyier.net/2023/10/19/style-scoped/ [Accessed: 09-Sep-2024]

[15] "@scope," *MDN Web Docs*, Sep. 9, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/CSS/@scope [Accessed: 09-Sep-2024]

[16] C. Coyier, "How Do You Remove Unused CSS From a Site?" *CSS-Tricks*, Nov. 29, 2019. [Online]. Available: https://css-tricks.com/how-do-you-remove-unused-css-from-a-site/ [Accessed: 11-Sep-2024]

[17] "CSS Performance Optimization," *MDN Web Docs*, Sep. 9, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Learn/Performance/CSS [Accessed: 11-Sep-2024]

[18] "Optimizing for Production," *Tailwind CSS v2 Documentation*, Aug. 2024. [Online]. Available: https://v2.tailwindcss.com/docs/optimizing-for-production#basic-usage [Accessed: 11-Sep-2024]

[19] "PurgeCSS - Documentation" *PurgeCSS*, Aug. 2024. [Online]. Available:

https://purgecss.com/introduction.html [Accessed: 11-Sep-2024]

[20] Dzheky, CSS Refactoring Test, *GitHub*. [Online]. Available: https://github.com/Dzheky/css_refactoring_test [Accessed: 13-Sep-2024]

[21] Vercel, Serve, *GitHub*. [Online]. Available: https://github.com/vercel/serve [Accessed: 14-Sep-2024]

[22] "PostCSS - A Tool for Transforming CSS with JavaScript," *PostCSS*, Aug. 2024. [Online]. Available: https://postcss.org/ [Accessed: 14-Sep-2024]

[23] Alexander Madyankin, postcss-modules, *GitHub*. [Online]. Available: https://github.com/madyankin/postcss-modules