The Journal of Systems & Software 198 (2023) 111593

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Optimize along the way: An industrial case study on web performance☆

Jasper van Riet [a], Ivano Malavolta [a,*], Taher A. Ghaleb [b]

[a] Vrije Universiteit Amsterdam, The Netherlands
[b] School of Electrical Engineering and Computer Science, University of Ottawa, Canada

ARTICLE INFO

Article history:
Received 3 May 2022
Received in revised form 30 October 2022
Accepted 21 December 2022
Available online 28 December 2022

Dataset link: https://doi.org/10.5281/zenodo.7256902

Keywords:
Performance
Web browsers
Industrial case study

ABSTRACT

Context: Fast loading web apps can be a key success factor in terms of user experience. However, improving the performance of a web app is not trivial, since it requires a deep understanding of both the browser engine and the specific usage scenarios of the web app under consideration.
Aims: In this paper, we present an industrial case study at 30 MHz, an agricultural technology company, in which we target a large web-based dashboard, where its performance was improved via 13 distinct interventions over a four-month period. Moreover, we conduct a user study to analyse whether web performance metrics correlate with the user perceived page load time in optimization scenarios.
Methods: First, we design a replicable performance engineering plan, where the technical realization of each intervention is reported in detail along with its development effort. Second, we develop a benchmarking tool that supports 11 widely used web performance metrics. Finally, we use the benchmarking tool to quantitatively evaluate the performance of the target web app and measure the effect of 13 interventions on both desktop and mobile devices. For the user study, we record six videos of different page loads and ask participants about their opinion about the time a web page is considered ready. We calculate the correlation of the user perceived data with each web performance metric.
Results: We observe a considerable performance improvement over the course of the 13 interventions. Among others, we achieve 98.37% and 97.56% time reductions on desktop and mobile, respectively, for the First Contentful Paint metric. In addition, we achieve 48.25% and 19.85% improvements for the Speed Index (SI) metric on desktop and mobile, respectively. Our user study shows that the Lowest Time to Widget metric, a product-specific web performance metric, is perfectly correlated with perceived performance during the optimization process.
Conclusion: This study shows the importance of a continuous focus on performance engineering in the context of large-scale web apps to improve user browsing experience. We recommend developers to carefully plan their performance engineering activities, since different interventions might require different efforts and can have different effects on the overall performance of the web application.

1. Introduction

By 2025, 1.8 billion *more* people will be using mobile internet.[1] One of the key enablers of this massive trend is the standardization and compatibility of Web technologies, primarily lead by the efforts of foundations like W3C and several companies. With standard geolocation APIs, push notifications, accessing the camera, etc., the Web is becoming a fully-fledged software platform and capable of providing richer experiences to end users (Malavolta, 2016). The *performance* of a web application is key to its success. In experiments by Google Search, a slowdown of its search results page by 200 milliseconds (ms) resulted in 0.22% fewer searches during the following three weeks. Worse, in the three weeks following, the number of searches performed by users decreased further: decreasing by 0.36% compared to their behaviour before the experiment.[2] In a more positive example, a 2.2 s page speed reduction resulted in the yearly downloads of Mozilla's browser

☆ Editor: Heiko Koziolek.

* Correspondence to: Vrije Universiteit, Faculty of Sciences, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands.
E-mail addresses: jvriet@hey.com (J. van Riet), i.malavolta@vu.nl (I. Malavolta), taher.ghaleb@queensu.ca (T.A. Ghaleb).

[1] https://www.gsma.com/mobileeconomy.

[2] https://ai.googleblog.com/2009/06/speed-matters.html.

https://doi.org/10.1016/j.jss.2022.111593

Firefox to increase by approximately 60 million.[3] When YouGov surveyed consumers on what they found more frustrating, a slow website or one that is down, 80% of participants voiced the opinion that the slow website would frustrate them more.[4]

Despite the aforementioned benefits, improving the performance of web applications is not an easy task as it requires very demanding engineering efforts. Developers need to have a deep understanding of a variety of optimization techniques defined at different abstraction levels and requiring different technical backgrounds (e.g., caching, prefetching, image optimization, code bundling) (Wagner, 2016). In addition, the pervasive web performance metrics in both research and industry, the page load time (PLT) reported by browsers and the Speed Index (SI) pioneered by WebPageTest,[5] are both unable to accurately reflect the human perception of the loading process of a web page (Netravali et al., 2018; Gao et al., 2017; Hoßfeld et al., 2018). While prior research presents metrics they find more accurate to the human perception (Netravali et al., 2018; Bocchi et al., 2016), these newly introduced metrics rarely gain traction (van Riet and Malavolta, 2019). In terms of optimization, other scientific studies tend to focus on improving the performance of web pages in general, as opposed to one specific case. From the perspective of a company or individual, there is no scientifically validated method of web performance improvement.

This paper presents a case study of improving the performance of a Web dashboard at 30 MHz, a technology company in the agricultural sector that provides growers insight into their data. The main product of 30 MHz is a Web dashboard for providing insights for growers into geographically distributed data produced by sensors of humidity, microclimate, wind speed, etc. With most of the data being produced in real-time by a multitude of networked sensors, the performance of the dashboard is key for the overall experience delivered to end users; and thus a major factor for the success of the business proposition of 30 MHz. In our case study, we analyse the extent to which we can improve the performance of a single-page Web application (SPA) for both desktop and mobile devices over a four-month period. To do this, we design a ***performance engineering plan*** (PEP) for the 30 MHz dashboard, where the technical realization of 13 interventions is reported in detail and with replicability in mind. Then, we develop a ***benchmarking tool*** supporting 11 widely-used web performance metrics, such as First Contentful Paint, Median Time to Widget, etc. The benchmarking tool is based on Google Lighthouse,[6] an open-source audit tool widely used both in academia and industry. Over a period of four months we implement each intervention of the PEP and apply the benchmarking tool for a ***quantitative evaluation*** of the performance of the 30 MHz dashboard according to the 11 supported metrics. Moreover, we conduct a ***user study*** to understand how quantitative metrics compare to the perceived performance by real users.

Using a structured system of incremental interventions, we achieve a 48.25% improvement of the SI on desktop devices and a 19.85% improvement on mobile devices. We also achieve 98.37% and 97.56% time reductions on desktop and mobile, respectively, for the First Contentful Paint metric. As a result, this paper provides insight into the web performance optimization process for both researchers and practitioners. In addition, this paper highlights the value of continuous performance work to product owners.

This paper addresses two **research questions**: *how can the performance of a web application be improved iteratively?* (**RQ1**) and *to what extent do quantitative performance metrics correlate with user-perceived performance improvements of web apps?* (**RQ2**). The main ***contributions*** of this study are: (1) a description of a detailed performance engineering plan for web applications; (2) a carefully executed real-world case study of web page performance optimization, closely adhering to the scientific principle of reproducibility; (3) a quantitative evaluation of the impact of each intervention of the plan on the performance of the 30 MHz web dashboard; (4) a discussion of the obtained results and their implications; (5) an exploratory user study, comparing the correlation between web performance metrics and the user perceived page load time in optimization scenarios; and (6) a replication package of the study containing its results, raw data and data analysis scripts.[7]

In van Riet et al. (2020) we reported on a case study for improving mobile web performance. In this paper, we extend the contributions of that paper by (a) expanding the measurements and analysis to desktop machines, (b) performing additional statistical analyses to verify how performance metrics differ between desktop and mobile devices, and (c) conducting a user study to understand how quantitative metrics compare to the performance perceived by real users.

The ***target audience*** of this paper includes researchers and practitioners interested in measuring and improving the performance of web apps on mobile devices. Our case study also provides an overview on the impact that different optimization techniques have on the overall performance of a real industrial product, and hence help practitioners better understand which techniques can be applied in their own projects. Finally, our results provide evidence about the value of continuous performance engineering to product owners.

The remainder of this paper is organized as follows. Section 2 provides an overview of 30 MHz, the company in which we perform our case study. Section 3 illustrates the design of the cases study. Section 4 presents the results and discussions for our research questions. Section 5 discusses our main findings. Section 6 discusses the validity threats to our study. Section 7 reviews the related work to our study. Finally, Section 8 concludes the paper and suggests the possible future work.

2. Context: The 30 MHz platform

The context for this paper is the company of 30 MHz. This section gives an overview of the company and its product that we use as object for this case study.

2.1. The 30 MHz company

30 MHz is a technology company in the agricultural sector. It offers a data platform for growers and everyone else involved in the growing process, by providing sensors that provide up to the minute data of produce, along with a platform to view this data. With a target audience that relies on this platform to be able to perform their jobs, there is no tolerance for a slow application. In addition, due to the nature of the environment in which such an application is often used, crop fields and greenhouses, network connections are often sub-optimal. This means optimal usage of the available network resources is important.

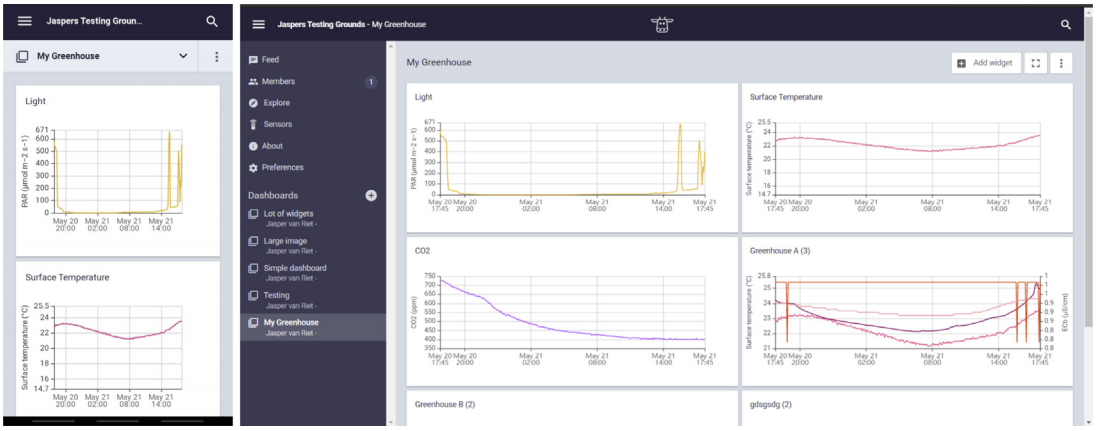
[3] https://blog.mozilla.org/metrics/2010/04/05/firefox-page-load-speed-part-ii.

[4] https://blog.eggplantsoftware.com/Surviving-Black-Friday-Avoiding-Outages-Is-No-Longer-Enough.

[5] https://sites.google.com/a/webpagetest.org/docs/usingwebpagetest/metrics/speed-indexgaintraction.

[6] https://developers.google.com/web/tools/lighthouse.

[7] https://github.com/S2-group/JSS-2022-web-performance-rep-pkg.



(a) Mobile dashboard (b) Desktop dashboard

Fig. 1. The mobile (a) and desktop (b) dashboards of the 30 MHz platform.

2.2. The 30 MHz product

The core product of 30 MHz is called the "30 MHz platform", a data platform for growers and everyone else involved in the growing process; various sensors provide up to the minute data of produce, which is then viewed and analysed via a cloud-based backend. With a target audience that relies on this platform to be able to perform their jobs, there is no tolerance for a slow application. Moreover, due to the nature of the environment in which such an application is often used (i.e., crop fields and greenhouses), network connections are often sub-optimal. This means that optimal usage of the available network resources is fundamental for the platform.

On the client side, users interact with the platform via a dedicated web application, which follows the principle of responsive design, i.e., its layout adapts to the size and capabilities of the device running it (smartphone, tablet, etc.). The web application is developed as a Single Page Application (SPA) and it is maintained by a team of three engineers. As of starting the case study, the web application is based on the Angular 7 framework, it is hosted on Amazon AWS[8] infrastructure and uses Amazon's CloudFront[9] service as edge nodes. Around 70% of users access the web application using a desktop device, whereas the remaining 30% of users access the web application from a smartphone or tablet. The specific focus of the case study is on the dashboards available inside the platform. These dashboards, along with the rest of the platform, follow web responsive design principles, adjusting layouts and more. An example of these dashboards can be seen in Fig. 1(a) for the mobile application and Fig. 1(b) for the desktop application. As shown in Fig. 2, the 30 MHz dashboard consists of a scrollable list of widgets. Users can configure their dashboards to show various widgets, in whichever order is desired. Each of these widgets is updated at a specified interval, ranging from near real-time to every minute, once a day, etc. The 30 MHz dashboard supports four types of widgets, as follows.

- Single value widgets (A) show a single value from a data source, updated regularly; these provide an immediate insight into the state of various sensors at a glance.
- Interactive charts (B) show the value of a data source over time; charts allow putting the current data into context, thus allowing the promptly detection of anomalies.
- Gauge widgets (C) show a value within a fixed range, allowing for a quick visual confirmation whether a certain metric is within a desired range.
- Image widgets (D) display values from data sources as an overlay on top of an image uploaded by the user (e.g., the plan of a greenhouse).

The widgets provide vital information about the state of plants and flowers, shaping decisions made on a daily basis. Due to the environment in which these dashboards are used, as well as the frequency with this they are returned to, performance is critical.

3. Study design

3.1. Goal and research questions

The ***goal*** of this case study is to implement and analyse performance-oriented interventions to characterize their individual and combined impact on web performance as seen from the viewpoint of web developers, in the context of the 30 MHz platform. To ensure high standards of objectivity and reproducibility, we follow established guidelines on experimentation in software engineering (Wohlin et al., 2012). The goal leads to the following **research questions**:

> **RQ1: How can the performance of a web application be improved iteratively?**
> This case study aims to iteratively improve the web performance of the 30 MHz web application, as opposed to finding and resolving a single bottleneck. This requires an iterative benchmarking mechanism to measure the significance of each improvement, as well as repeated analysis of new releases.
> By answering this question, we provide software engineers working on web applications with insights on how a web performance can be improved by applying different types of interventions.
> **RQ2: To what extent do quantitative performance metrics correlate with user-perceived performance improvements of web apps?**
> Despite the quantitative performance improvement, the real measure of success is user satisfaction; i.e., having a quantitatively improved loading performance should be aligned with user perception of that improvement. We evaluation web performance metrics to investigate which metrics better reflect, i.e., correlate with, user-perceived performance. We also investigate whether the browser loading indicator

[8] https://aws.amazon.com.
[9] https://aws.amazon.com/cloudfront.

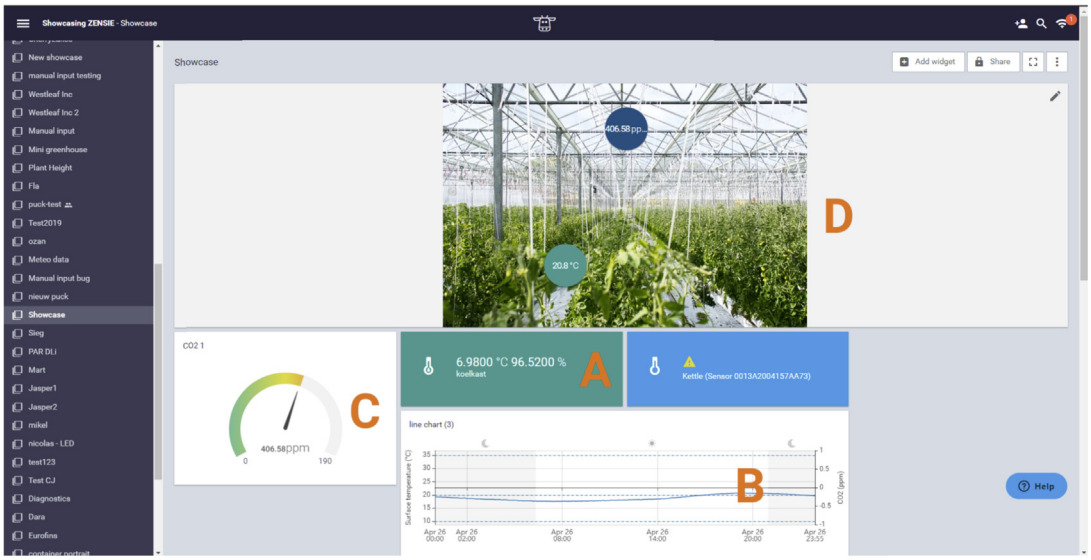


Fig. 2. Examples of widgets in the 30 MHz dashboard (the desktop version).

impacts the user perceived page load time. To address this question, we perform a multi-object variation user study. Unlike previous user studies (Netravali et al., 2018; Gao et al., 2017), this user study focuses on relative values as opposed to absolute values.

By answering this question, we provide software engineers with a practical assessment of the value of various web performance metrics in the context of web optimization. We also provide insights to the web community on the importance of animations used as indicators for page load.

3.2. Quantitative performance measurements (RQ1)

The quantitative analysis of this case study comprises the interventions we implement and their effects. In our experiments, we used the same page, shown in Fig. 2, on both desktop and mobile devices.

3.2.1. Variables selection and experiment plan

The unit of analysis of this study is the 30 MHz dashboard (see Section 2) and the various interventions of the performance engineering plan (PEP) are directly applied to it in sequence (see Section 4.1.3). Accordingly, the ***independent variable*** of this study represents the intervention applied to the 30 MHz dashboard. This variable has 13 levels, each of which corresponds to each intervention of the PEP. We described these interventions in the context of our reported results in Section 4. We should note that there were no other factors involved during the experiment, neither before nor after and not even during applying the interventions and calculating the metrics. In particular, we ensured a homogeneous environment throughout the experiment as there were no changes to the operating system, server, dependencies, updates, network conditions, or the code base. Moreover, there were no users using the 30 MHz platform other than the one who was conducting the experiments, who had complete control over all variables that could interfere and thus invalidate the measurements. In addition, performing 30 runs of the web page by each benchmark helped in minimizing edge cases.

The ***dependent variables*** are shown in Table 1. These variables correspond to the 11 web performance metrics we collect on the 30 MHz dashboard. Those metrics serve as the core source of truth for evaluating the impact of each intervention on the performance of the 30 MHz dashboard. We select these 11 metrics based on both academic consensus and industry standards (Wagner, 2016) (see Section 7 for more details). It is important to note that these metrics do not have the same goal but rather complement each other, focusing on four aspects of web applications, namely *Rendering*, *Computation*, *Networking* and *General*. This allows us to perform a comprehensive assessment of the performance of the 30 MHz dashboard from different complementary perspectives. The calculation of these metrics was performed via our benchmarking tool and not collected by Google Lighthouse. We describe below the metrics of each of the above aspects.

Rendering metrics. Rendering a web page is defined as taking HTML, CSS and JavaScript code as input and producing a visual representation, as described by the code, as output. Metrics in this category measure the state of this process, or reflect properties about the end result.

- The *First Contentful Paint* (FCP) is the time in milliseconds (ms) till the first DOM content, the tree of elements that describes a web page, is rendered. When loading a web page, a browser starts off by showing a white screen. The FCP equals the time till the first element is drawn over this screen.
- The *First Meaningful Paint* (FMP) is the time till the most significant change in content is rendered above-the-fold. Specifically, it reflects the time till the next paint after the biggest layout change,[10] counted using the `LayoutAnalayzer`[11] in Chromium.
- The *DOM Size* (DOM) is a simple reflection of the number of DOM elements in the DOM tree. Large DOM trees are associated with bad performance,[12] due to high memory cost and significant performance cost to recompute positioning and styling of elements.

Computation metrics. Computational metrics represent the amount of computational complexity of an application, as well as the extent to which this computational complexity is perceivable by users. A key part of all metrics in this category is the time

10 https://docs.google.com/document/d.

11 https://source.chromium.org/chromium/chromium/src/+/master:third_party/blink/renderer/core/layout/layout_analyzer.h.

12 https://web.dev/dom-size.

Table 1
Performance metrics (dependent variables) considered in this case study.

Name	ID	Focus	Description
First Contentful Paint	FCP	Rendering	Time (ms) till first DOM element is rendered
First Meaningful Paint	FMP	Rendering	Time (ms) till largest DOM element is rendered
Speed Index	SI	General	Integral of visual loading progress
Total Blocking Time	TBT	Computation	Time (ms) during which page is blocked from interaction
Estimated Input Latency	EIL	Computation	Time (ms) needed to respond to interaction during load
First CPU Idle	FCI	Computation	Time (ms) till page is ready to accept any interaction
Time to Interactive	TTI	Computation	Time (ms) till page is fully responsive to interaction
Network Requests	NR	Network	Number of network requests
DOM Size	DOM	Rendering	Number of DOM elements
Lowest Time to Widget	LTTW	General	Time (ms) till first widget is rendered
Median Time to Widget	MTTW	General	Median of all visible Time to Widget values (ms)

needed to respond to user interactions. If an interaction is not responded to within 50 ms, users may experience a delay.[13] Note that we considered only one user at a time, given that we are focusing on client-side performance. Yet, we expect the results to be consistent for multiple users, since measurements in our experiments were conducted on staging, in which we used the same server and architecture as in production.

- *First CPU Idle* (FCI), also known as, First Interactive, is the point at a page can be considered minimally interactive: most page elements are interactive and responsive within a reasonable amount of time. The precise definition[14] is as follows.
  A task T is defined as *lonely* if there exists a window E of size at most 250 ms, such that:
  - T is completely contained in E
  - E.start is at least 5 s away from FMP
  - The windows [E.start - 1 s, E.start] and [E.end, E.end + 1 s] overlaps no tasks longer than 50 ms
  - E, however, is allowed to contain as many long tasks as it can fit

  To get the First Interactive Candidate, find the first window W after FMP such that:
  - If W overlaps a task T, either duration of T is less than 50 ms, or T is lonely
  - W.duration $\leq$ f(W.start - FMP)

  The FCI equals the maximum value of the First Interactive Candidate and the `DOMContentLoadedEventEnd` provided by the browser, which triggers when all DOM content has been rendered.
- *Time to Interactive* (TTI) is a similar metric to the FCI. The TTI can also be referred to as First Consistently Interactive. The TTI is the time till a web page is able to consistently respond quickly to interaction. It is defined as follows.
  Find the first 5 s window W after FMP such that:
  - W overlaps no tasks longer than 50 ms
  - For all timestamp t in W, the number of resource requests in flight at t is no more than 2

  To get the Consistently Interactive Candidate, find the last long task L before W:
  - Consistently Interactive Candidate is the end of L
  - If there is no long task before L, Consistently Interactive Candidate = FMP

  TTI then equals the maximum of Consistently Interactive Candidate and the `DOMContentLoadedEventEnd`.
- *Total Blocking Time* (TBT) is the sum of time between the FCP and the TTI in which any task that runs on the main thread takes longer than 50 ms. This is considered blocking because an in-progress task cannot be interrupted. If such a task exists, it is thus likely not possible to respond within 50 ms. This is significant enough a delay that the user might notice it.[15]
- *Estimated Input Latency* (EIL) is an estimation of the time needed to respond to user interactions within the busiest five-second window of the page load. Ideally, this time should be below 50 ms.

Network metrics.
Network Requests (NR) is a simple count of the number of network requests performed during the page load. This includes network requests made for the purposes of retrieving code, such as HTML or JavaScript code, as well as API calls needed to retrieve data.

General metrics. General web performance metrics do not measure a specific subset of the page load process. These metrics also do not represent the amount of computation that is executed. Instead, general metrics focus on the time till users feel a web page is ready.

- The *Speed Index* (SI), originally introduced by WebPageTest[16] and in Google Lighthouse implemented via Speedline,[17] aims to measure the time till the page visually appears to be ready for use, see Section 7.1. Speedline makes use of the PSI as opposed to the original definition of the SI.
- *Time to Widget* (TTW). The TTW is inspired by Twitter's *Time to First Tweet*, which Twitter defines as: "the amount of time it takes from navigation (clicking the link) to viewing the first Tweet on each page's timeline" (Firtman, 2018). In the case of 30 MHz, the base unit of the dashboard is a widget, thus the TTW measures the time from starting the load of a page till seeing a widget rendered. In particular, the chart and image widgets of a dashboard tend to load last. Moreover, chart and image widgets tend to be bigger than their other widget counterparts, thereby drawing more attention. The time at which these two widget types load is thus a good indication of the user-perceived page load time. Both widget types register a TTW timestamp when rendered. We do this by using hooks in both the charting APIs and the image APIs that trigger upon successful rendering. Since each widget registers its own timestamp and a dashboard may have multiple widgets, we aggregate these for our metrics to simplify the interpretation of the

13 https://web.dev/estimated-input-latency.
14 https://docs.google.com/document/d/1GGiI9-7KeY3TPqS3YT271upUVimo-XiL5mwWorDUD4c/edit.
15 https://web.dev/tbt.
16 https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.
17 https://github.com/paulirish/speedline.

Table 2
Desktop and mobile devices used for testing.

Specification/Device	Desktop	Mobile
Model	–	Samsung Galaxy S10E (World)
CPU	Intel i7-8550U	Samsung Exynos 9820
Processor speed	1,8 GHz	speed 2,73 GHz
Total cores	4	8
RAM	16 GB DDR4	6 GB LPDDR5
Storage	Samsung EVO 960 m.2 (r: 3200 MB/s, w: 1900 MB/s)	Samsung 128 GB eUFS 2.0 (r: 350 MB/s, w: 150 MB/s)
OS	Windows 10 (various versions)	Android 10

obtained results. One aggregation equals the time till any widget is loaded, the *Lowest Time to Widget* (LTTW) and the second equals the median TTW across the loaded widgets, the *Median Time to Widget* (MTTW).

3.2.2. The benchmarking tool and measurement infrastructure

Our benchmarking tool uses the programmatic interface to the Google Lighthouse,[18] along with the User Timing API.[19] We use Google Lighthouse because of prevalence in the industry, as well as its extensibility and open-source nature.[20] The benchmarking tool measures the metrics shown in Table 1. To compute metrics, the benchmark tool performs 30 runs of the web page and collects metrics from each run. We do so to reduce the influence of a single anomalous run. We perform the benchmarking on both desktop and mobile devices. The technical specifications for the devices used for this testing can be found in Table 2.[21,22] On both devices, we use Lighthouse's throttling[23] to emulate a slow 4G network speed as well as slowing down the device CPU by a factor of four. We do so to avoid biasing the case study towards the high-performance devices used. We disable the storage reset option in Lighthouse, since a complete reset of cookies and local cache is not representative of the user experience. A user does not login and do not clear their browser cache every time they visit the web application. During the case study, we perform testing on different versions of Google Chrome desktop and Chrome Android browsers, namely 78, 79 and 80. We chose Google Chrome for this study since (i) currently it is the most used browser across all platforms,[24] including desktops, laptops, tablets, and smartphones and (ii) 30 MHz actively advises their customers to use Google Chrome when running their web dashboard. In this paper we consider the results on Google Chrome 80 only since we wanted to avoid having an additional confounding factor due to the specific version of Google Chrome.

We perform Android testing using the Android ADB Tool[25] in addition to Chrome's remote devtools functionality.[26] We use the devices shown in Table 2 to test performance improvements throughout the case study and to repeat all tests at the end of the study, to ensure consistency between tests.

To make sure that background processes do not interfere with test results, we terminate all non-essential processes on the desktop device. We also restart the mobile device before each benchmarking session. The browsers we used are "clean" in all cases in the sense that they have no extensions installed. We collect all test results at the end of the case study, using Chrome 80 (on Windows and Android) on the 30 MHz release staging instance, which uses the same configuration options as the production instance (including CloudFront edge nodes). Finally, to ensure consistency, we use the same dashboard for every benchmark. This dashboard is shown in Fig. 2.

During the case study, we use a number of different tools to make the process of finding performance issues easier. In particular, we use the Chrome devtools very often, specifically for the network[27] and performance[28] views. The network view shows the requests being made by the web page. For example, this can be used to find and inspect the number of API calls made during a page call, including their contents. In addition, network logs could be exported to `.HAR` files (Enghardt et al., 2019), which could then be processed using scripts. The performance view is a timeline tool, showing a flame chart of the code being executed and the amount of computational time taken up by objects and functions in the code. Finally, Google Lighthouse offers pointers via its performance view,[29] which also offers a good starting point for optimization.

3.2.3. Iterative process

Once there is a full picture of how computation resources are being used by the web application, it is time to start designing solutions. For example, if the application is making a large number of API calls, it would be beneficial to reduce this number. Once solutions are found and implemented, the benchmarking tool described in Section 3.2.2 is used to create a *before* and *after* performance comparison.

The proposed interventions are applied cumulatively, starting with Intervention 1, followed by Interventions 1 + 2, then Interventions 1 + 2 + 3, and so on. The process of judging whether an intervention is worth implementing is rather complicated. As explained in Section 3.2, not all metrics and interventions have the same goal. However, changes in some metrics can sometimes be linked, such as an improvement in the FCP metric and a reduction in the SI metric, because computation can start earlier. In addition, metrics can behave independently from one another. Thus, judging interventions should be done on a case-by-case basis, depending on what the goals of the optimization were. As we will see in the next sections, this is further complicated due to differences between devices and increased variability in benchmarking results when dealing with lesser performing web pages.

3.2.4. Data exploration

We give an overview of all measures collected throughout the case study using a set of boxenplots (Hofmann et al., 2017). Such a visualization provides a bird's-eye view of the obtained measures to those readers who are more interested in the technical aspects of the study (e.g., which interventions lead to stronger improvements), rather than on the statistical analysis.

18 https://developers.google.com/web/tools/lighthouse.
19 https://developer.mozilla.org/en-US/docs/Web/API/User_Timing_API.
20 https://github.com/GoogleChrome/lighthouse.
21 https://www.anandtech.com/show/13918/samsung-starts-production-of-1-tb-eufs-21-storage-for-smartphones.
22 https://www.gsmarena.com/samsung_galaxy_s10e-9537.php.
23 https://github.com/GoogleChrome/lighthouse/blob/master/docs/throttling.md.
24 https://gs.statcounter.com/browser-market-share.
25 https://developer.android.com/studio/command-line/adb.
26 https://developers.google.com/web/tools/chrome-devtools/remote-debugging.
27 https://developers.google.com/web/tools/chrome-devtools/network.
28 https://developers.google.com/web/tools/chrome-devtools/evaluate-performance.
29 https://developers.google.com/web/updates/2018/05/lighthouse.

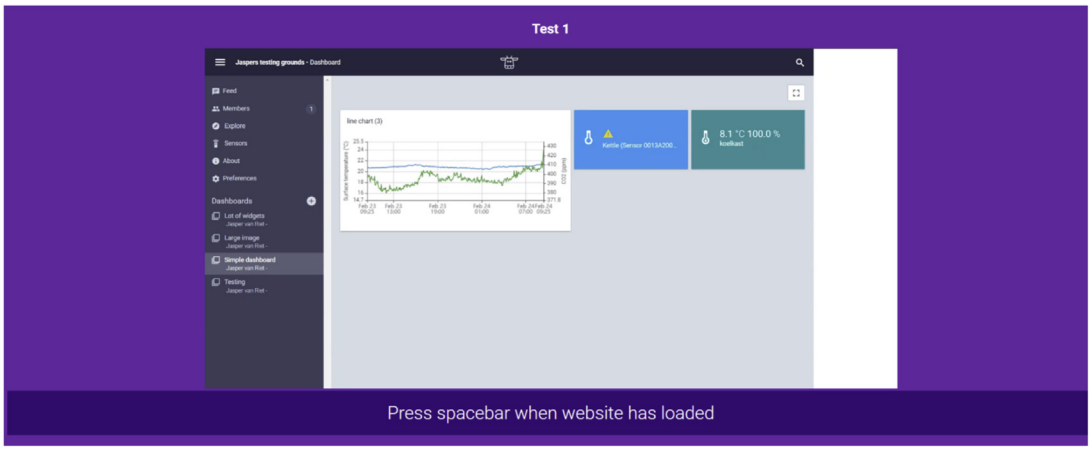


Fig. 3. An example web page for the user study.

3.2.5. Statistical analysis

Once we collect the measurements for every intervention, we apply statistical analysis to interpret the results. We use the median of the collected measurements for each metric to allow for a quick comparison. The median is preferred over the mean to minimize the impact of outliers. In addition, we compute the Cliff's Delta effect size (Cliff, 1993) for each intervention, comparing it to the previous state of the web application. The Cliff's Delta is a non-parametric effect size estimator, comparing two distributions, resulting in an indication as to the frequency with which the values in one distribution are larger than the values in the second distribution. We then interpret these values using the guidelines by Grissom and Kim (2005) and translate them into quantity sizes: *negligible*, *small*, *medium* and *large*. Finally, to test for statistical significance, we apply the Mann–Whitney U test, followed by a Benjami/Hochberg correction, with an α of 0.05.

3.3. User-perceived performance (RQ2)

We perform a user study to test the correlation of web performance metrics with user-perceived performance. Our goal here is to assess the extent to which the observed effects of the interventions performed in RQ1 reflect user perception.

3.3.1. Experiment

Unlike in RQ1, we record the videos of the loading of three different pages, each with different characteristics. We do so to avoid bias in our experiment (Gao et al., 2017), given that it involves human subjects whose perceived page load time could strongly impact the results of the experiment if we consider a single web page (Egger et al., 2012). Each of the three pages contains a different dashboard of the 30 MHz platform. For each page load, we record two videos: one with the browser loading indicator and the other without. We collect page load metrics for each of these page loads. We later display the videos on a website to each user involved in our user study (see an example web page in Fig. 3). A video does not start until a user presses the spacebar. We ask users to press the spacebar again when they think the website has finished loading. The resulting time is the perceived page load time for that specific user.

Before conducting the experiment, we give users an experiment guide detailing what we expect from them (the guide can be found in the replication package). We ask users explicitly to keep their hand on the spacebar and treat the experiment as a reflex game. We also show users a demonstration video for each pair of videos to mitigate any possible confusion. After users are done with all six videos, we collect the time measurements and ask them to fill in a questionnaire to collect demographics data.

3.3.2. Variables selection and experiment plan

The independent variables in this experiment are the metrics obtained by measuring the page loads, in addition to the size of the dashboards that were shown. The resulting dependent variable is the user perceived page load time (uPLT). The uPLT is defined as the difference, in milliseconds, between the two moments at which the spacebar is pressed, for all users, using the high resolution millisecond timestamps provided by `performance.now()`,[30] which returns the time since the current web page was first requested to load by the user. The median of these values equals to the uPLT. The metrics used for this user study match the ones featured in Table 1. All measurements are collected using a test bench that utilizes Google Chrome in combination with Google Lighthouse. Since each video needs to be captured in the same run, these metrics are captured in each run.

The objects of the user study are the three pages shown in Table 3 having different characteristics and performance from each other. The first page, shown in Fig. 4(a), contains one chart and two smaller widgets. The second page, shown in Fig. 4(b) (while loading) and Fig. 4(c) (after loading), contains a large image. The third page, shown in Fig. 4(d), contains three charts, and eight smaller widgets. The subjects for the user study are 19 employees at 30 MHz, 11 of them are from the engineering department and two work on the 30 MHz platform web application. 16 of the participants used the 30 MHz platform on most or more days of the week. The majority of participants are between the ages of 40 to 49 (see Table 4).

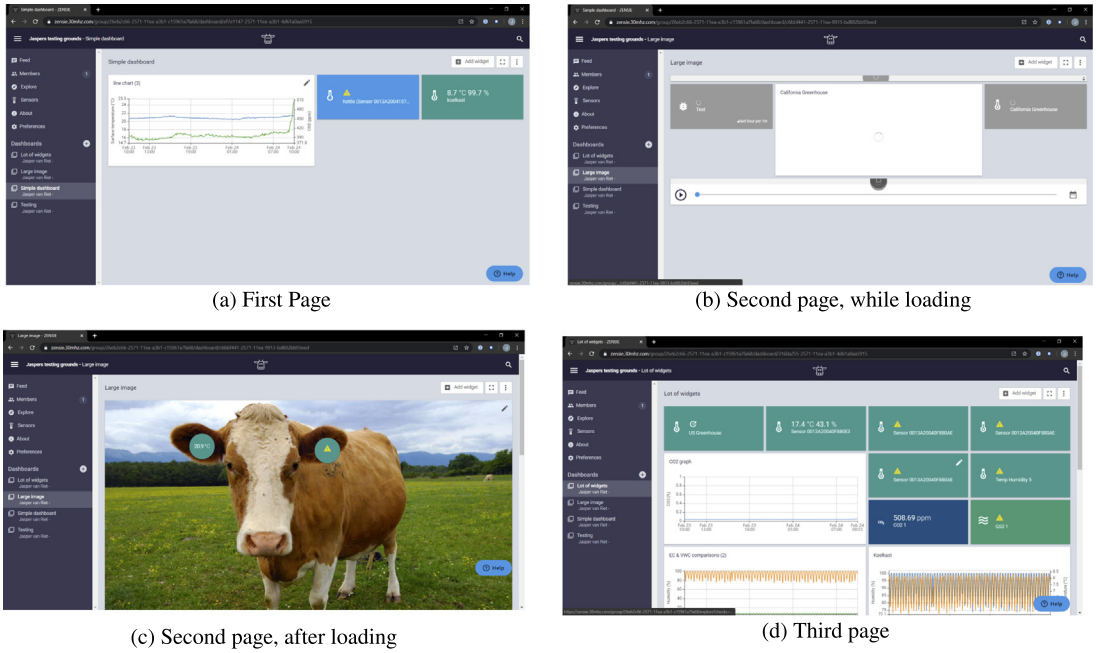
3.3.3. Statistical analysis

For each web performance metric, we calculate the Kendall's τ correlation coefficient (Newson, 2002) by comparing the results of each metric with uPLT. The uPLT metric is calculated by taking the median of user results for a particular video. Besides the correlation coefficient, we also calculate the *p-value* for each result and correct it using the Holm–Bonferroni correction (Bonferroni, 1936).

3.4. Study replicability

To foster independent verification and replication of this case study, a full replication package is publicly available.[31] The replication package includes (i) expanded definitions of all performance metrics, (ii) the implementation of the benchmarking tool, (iii) all collected raw data and (iv) the Python scripts to visualize and analyse the data.

[30] https://developer.mozilla.org/en-US/docs/Web/API/Performance/now.
[31] https://doi.org/10.5281/zenodo.7256902.



(a) First Page (b) Second page, while loading

(c) Second page, after loading (d) Third page

Fig. 4. The pages shown during the user study.

Table 3
Objects used for the user study.

Identifier	Description
LightBase LightLoader	One chart and two smaller widgets (see Fig. 4(a)).
ImageBase ImageLoader	Image-heavy dashboard. Due to the image loading method of the 30 MHz dashboard, the image takes no space while loading, but then takes up the whole screen when fully loaded (see Figs. 4(b) and 4(c)).
HeavyBase HeavyLoader	Dashboard with several widgets. It consists of three charts and eight smaller widgets (see Fig. 4(d)).

Table 4
User study demographics.

Age	Number of participants
20–29	3
30–39	4
40–49	10
50 or more	2

4. Case study results

This section presents the results of the iterative performance improvements along with the user study results.

4.1. Performance interventions

4.1.1. Familiarization with the product and first measurements

The first step of the case study consisted in having one of the academic researchers join the 30 MHz team to (i) have access and get familiar with the code base of the 30 MHz dashboard, (ii) identify the performance metrics of interest for the team (including the development of a first prototype of the benchmarking tool presented in Section 3.2.2) and (iii) collect the first performance measures as baseline for the whole case study. This phase took approximately 2 weeks full-time. During this phase, the interaction with the industrial partners was continuous and involved several clarifications about the technical aspects of the 30 MHz dashboard.

A co-product of this phase was the identification of a set of performance issues of the 30 MHz dashboard, which eventually were used as input for the definition of the PEP. Performance issues we identified using various profiling and inspection tools. Firstly, we made heavy use of the Chrome devtools, in particular its network and performance views.[32] The network view shows the requests being made by the web page. The performance view shows a flame chart of the JavaScript code being executed and the amount of computational time consumed by each object and function within the whole web app. Finally, we used Google Lighthouse to identify a number of additional performance issues.

To keep the obtained results comparable and replicable, in each run of the benchmark the backend always provides 20 distinct data sources to the 30 MHz dashboard. As shown in the first column of Figs. 6 and 7, the performance of the baseline version of the dashboard was a problem for both desktop and mobile devices. For example, the median of the FCP metric for the mobile device was 6.2s, whereas the median for real web apps reported by the HTTP Archive[33] is 5.7 s.

Prior to any intervention, the 30 MHz web app is an Angular 7 SPA hosted on AWS and using CloudFront for edge nodes, which is used as a baseline. The web app has a bundle size of 5.91MB. When the web app is loaded, a list of widgets is retrieved by issuing an HTTP request to the `/dashboard` REST endpoint in the backend. Still, at this point, the user still see the web page as shown in Fig. 5a. Then, each widget is populated via other HTTP requests: single value, gauge and image widgets make a single HTTP request to the `/stats` endpoint, which displays the page as shown in Fig. 5b. The chart widget, however, makes up to 6 HTTP requests, including one call to the `/stats` endpoint, which displays the page as shown in Fig. 5c. As a result, in the median case, the dashboard makes 153 network requests. In addition to potentially exhausting the maximum concurrent requests

32 https://developers.google.com/web/tools/chrome-devtools.
33 https://httparchive.org/reports/loading-speed#fcp.

Fig. 5. The dashboard loading process.

Table 5
Taxonomy of interventions implemented.

Category	Identifier	Description	Interventions	Effort	Count
Bug fixes	BF1	Interaction with platform APIs	I2	Within one day	1
	BF2	Unnecessary execution of code	I1	Within one day	1
	BF3	Leaking of code	I4	Within one day	1
Optimizations	UD1	Update dependencies	I6, I11	Multiple days	2
	OD1	Data transfer: Connect/fetch in advance	I10, I12	Within one day	2
	OD2	Data transfer: Avoid unneeded network requests	I3, I13	Multiple weeks	2
	OC1	Complexity/Algorithm: Optimize computation to more opportune times	I7	Multiple days	1
	OA1	Interaction with platform: Leverage optimization opportunities in platform APIs	I9	Within one day	1
Architecture changes	AP1	Migrate to a Progressive Web Application	I5	Multiple weeks	1
	AT1	Switch to a more optimized technology	I8	Multiple weeks	1

enabled by browsers (Wijnants et al., 2018), which can be as low as 100, depending on the browser, these requests are not all executed in parallel, with some calls being blocked by another call. As a result, in some cases a single network request could have up to 20.23 ms as round trip time, which is significantly high.

Once all data are retrieved by the widgets, it has to be displayed. This process is negligible for both the single value, gauge and image widgets, with the time taken up being minimal. However, chart widgets are considerably expensive from a computational perspective. Chart widgets make use of NVD3,[34] an SVG-based charting framework last updated in August 2017. The runtime impact of NVD3 is significant, with the 30 MHz dashboard generally having a frame rate of only 5–10 frames per second. This has a significant impact on metrics measuring interactivity. Moreover, careful tracing in the devtools performance view shows NVD3 using a significant amount of computation time during the loading process when drawing in charts.

4.1.2. Definition of the performance engineering plan

To gain insight into the types of improvements being implemented, a taxonomy of interventions is created. This taxonomy is shown in Table 5 (Selakovic and Pradel, 2016). Multiple interventions can belong to the same identifier. Interventions were divided into three core categories: bug fixes, optimizations and architecture changes. These have the following definitions:

- Bug fix: a (unintentional) flaw in the code. For example, a computationally intensive function that is run at all times, even if it is only applicable to a certain edge case.
- Optimization: a more efficient method achieving the same functionality, such as caching the result of a certain API call so that it can be re-used later.
- Architecture change: a significant restructuring of the application. For example, changing the technology used in the application.

The sequence of the interventions in the PEP was defined based on a combination of priorities given by 30 MHz developers and the expected improvements (based on the Google LightHouse documentation and other sources on Web performance). Specifically, at 30 MHz the decision to accept an intervention was based on the following points: (i) key metrics related to the user experience (i.e., SI, MTTW, FMP) should improve, (ii) metrics deemed as secondary (i.e., LTTW and FCP) should improve as well, but a worsening is still acceptable; (iii) all the other metrics are considered case by case depending on the involved trade-offs with respect to the key and secondary metrics.

Moreover, the process in which targets of interventions were found was based on the analysis of possible bottlenecks. Specifically, the 30 MHz development team based their interventions on their knowledge of whether each considered bottleneck was deemed resolvable; for example, they did not consider bottlenecks related to product features since they could not be changed for business reasons.

Not all interventions required the same amount of engineering effort, nor were all interventions as immediately apparent to the user. Interventions can be a refactoring, wherein no functionality is changed, or feature improvements, which do change functionality. A number of interventions did substantially change certain features of the dashboards, however, every intervention is motivated primarily by the goal of achieving better page load performance. An estimation of the effort required to design, implement and test an intervention is described for every leaf of the taxonomy.

The overall performance of the 30 MHz platform after the interventions, which we will further discuss in Section 4.1.3, is significantly improved.[35] This improvement is visualized in a timeline of interventions, see Figs. 6 and 7 for desktop and mobile devices, respectively (a side-by-side version of the two figures can be found in our replication package, so to allow for easier comparison between desktop and mobile results for each intervention). Across the board, major improvements are visible in page load metrics. While the FCP and FMP particularly catch the eye, with instant to almost instant rendering on the screen depending on the device, general and computation metrics have also undergone significant improvement. Desktop and mobile devices rarely match completely in response to an intervention. In some cases, the two devices had opposite reactions. This behaviour is visible in both timelines.

[34] http://nvd3.org.

[35] It is important to note that the *aggregated* performance score produced by Google Lighthouse on the baseline is 19, which can be considered as a *poor* score in terms of user experience.

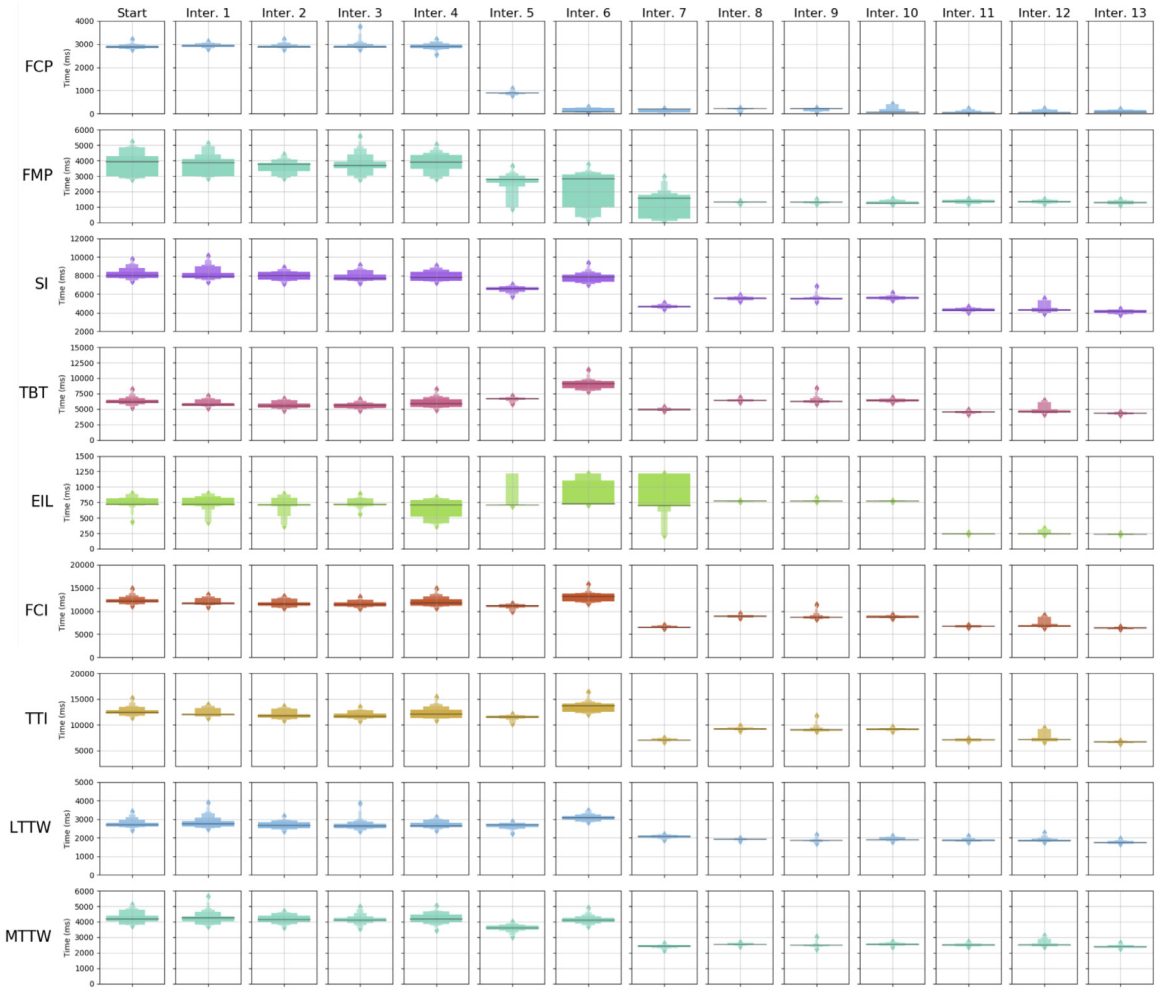


Fig. 6. Timeline of metrics during case study using boxenplots on the desktop device.

From the timeline, it can be observed that the variation between measurements within the same benchmark on desktop decreases the more interventions are implemented. After Intervention 8, we observe that consistency has increased considerably. This intervention changed charting frameworks, as we will discuss in Section 4.1.3. From this we take that NVD3, the old charting framework, was introducing a significant amount of variability to the performance of the web application. On mobile, this cannot be observed to the same degree. Such inconsistent behaviour in the performance can make it much harder to measure the effect of interventions accurately, thus underlining the need for a large number of runs on the benchmarking tool.

Table 6
Baseline performance results.

Metric	Desktop	Mobile
FCP	2864.07	6290.52
FMP	3941.31	7646.05
SI	8011.19	15,310.98
TBT	6167.60	10,708.86
EIL	715.29	1701.13
FCI	12,139.54	19,928.40
TTI	12,392.29	21,018.35
NR	269	153
DOM	13 249	13 219
LTTW	2702.93	4796.37
MTTW	4174.95	7280.71

4.1.3. Execution of the interventions

We implement the interventions in an iterative fashion and, thus, can be presented in timeline form, from the baseline situation shown in Section 4.1.1 to the end result. These interventions can, therefore, be identified sequentially. The baseline performance results can be found in Table 6. All metric values used in this case study consist of the median of the 30 runs. In the remainder of this section, we go through the whole web engineering process and discuss the technical details and impact of each intervention on both mobile and desktop devices. For each intervention, the performance difference per metric is presented, as well as the Cliff's Delta effect size (ES) and the corrected *p-values*. If the effect size is negligible, we show its value as a hyphen.

Intervention 1: Prevent unneeded calculations

A number of elements are drawn into each chart in the 30 MHz dashboards. One of these is a notification line: a line drawn in the chart that can be set by a user to an arbitrary value. If the chart values cross that line, the user receives a notification via email. An ad-hoc calculation of the Y axis is needed to make sure that notification lines fit within the range of the chart. After

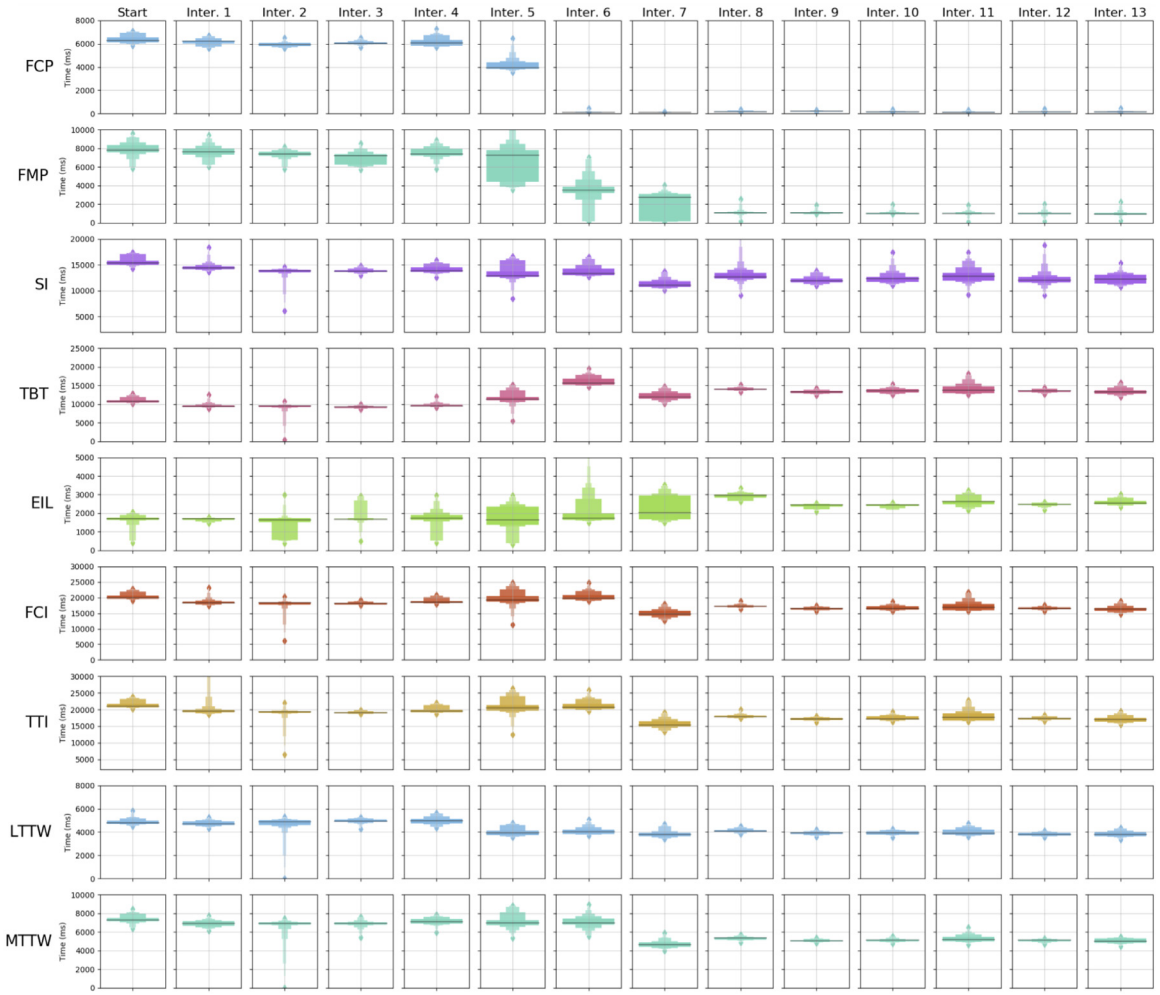


Fig. 7. Timeline of metrics during case study using boxenplots on the mobile device.

tracing the application with the devtools performance view, this calculation proves to be expensive and it becomes apparent that it was executed in all cases, even when no notification line was present. Having this calculation only be executed in the case where it is actually needed is a simple optimization and should reduce computation time.

This intervention reduced computational metrics significantly on desktop. The TBT is reduced by 7.24%, while the FCI and TTI are reduced by 3.86% and 3.52% respectively, see Table 7. As described in Section 7.2, computation is the biggest bottleneck on mobile, whereas desktop devices have network being the major bottleneck. This difference is apparent in the results of this intervention, as not only do we observe a reduction of the TBT, FCI and TTI by 12.92%, 7.80% and 7.44%, respectively, general metrics such as the SI also undergoes a reduction by 6.14% on the mobile device. Thus, on mobile several categories of metrics are reduced by shortening the page load, whereas on desktop only the computational metrics experienced an increase in performance.

Intervention 2: URL bar on mobile triggers *onResize*

When the browser window resizes, the web application resizes its layout. This is a good thing, it allows the application to adjust the number of widgets on screen, along with their width and height, to the current viewport. There is one problematic edge case for this behaviour, visible in Chrome on Android: when the URL bar animates, the resize handler is fired. The URL bar hides itself when scrolling down and reappears when changing direction. As a result, every time the user changes scrolling detection, scrolling performance is reduced to mere singular frames per second.

This is fixed by only resizing the layout when the width of the window has changed and, thus, no longer triggering on every resize. While this intervention focused primarily on runtime performance, a 3.65% reduction in SI is observed on mobile, see Table 8. A further improvement in FCP is visible on both the desktop and mobile device, with a reduction by 2.76% on desktop and a 4.40% reduction on mobile. A possible hypothesis for this improvement could be a reduction in the number of triggered reflows[36] during the page load, although this could not be verified using performance tracing.

Intervention 3: reducing API calls

As described in Section 4.1.1, a single chart widget can increase the total number of network requests by up to 6 API calls. One of these API calls is the call to /organization/users. Users can comment on charts, and these comments are then shown inside the chart. To look up the names of users, the user directory for an organization needs to be retrieved. This call is duplicated among all chart widgets. This is unneeded. An intervention was implemented to cache the result of this call, and have new callers attach themselves to the current call if one exists, instead of making a new request. As a result, this API call is now made once per dashboard, instead of once per widget.

[36] https://developers.google.com/speed/docs/insights/browser-reflow.

Table 7
Intervention 1 (BF2): Prevent unneeded calculations.

Metric	Desktop				Mobile			
	Δ	Δ %	ES	*p*-value	Δ	Δ %	ES	*p*-value
FCP	62.50	2.18	M	0.0107	-85.27	-1.36	S	0.0475
FMP	-69.32	-1.76	-	0.4588	-171.69	-2.20	S	0.0613
SI	-63.36	-0.79	-	0.4336	-940.55	-6.14	L	1.92e-7
TBT	-446.36	-7.24	L	0.0070	-1,383.06	-12.92	L	3.21e-9
EIL	2.56	0.36	-	0.4336	1.81	0.11	-	0.3341
FCI	-469.19	-3.86	M	0.0107	-1,554.06	-7.80	L	3.69e-9
TTI	-435.90	-3.52	M	0.0107	-1,564.50	-7.44	L	1.35e-8
NR	-1.50	-0.56	-	0.4336	0	0	-	-
DOM	0	0	-	0.4336	-3	-0.02	M	0.0044
LTTW	53.48	1.98	-	0.4336	-102.51	-2.14	S	0.0390
MTTW	59.33	1.42	-	0.4336	-375.70	-5.16	L	3.48e-5

Table 8
Intervention 2 (BF1): URL bar on mobile triggers *onResize*.

Metric	Desktop				Mobile			
	Δ	Δ %	ES	*p*-value	Δ	Δ %	ES	*p*-value
FCP	-80.71	-2.76	S	0.0610	-273.22	-4.40	L	0.0003
FMP	111.28	2.87	-	0.2648	-253.86	-3.32	S	0.0454
SI	-289.66	-3.64	-	0.3076	-524.57	-3.65	L	9.15e-7
TBT	-122.99	-2.15	S	0.0701	135.59	1.45	-	0.2679
EIL	-7.80	-1.09	S	0.2020	-71.68	-4.21	S	0.0816
FCI	-135.71	-1.16	S	0.0952	-60.53	-0.33	-	0.2679
TTI	-131.08	-1.10	S	0.0864	-86.41	-0.44	S	0.1629
NR	1.50	0.56	L	1.30e-10	3	1.96	L	7.02e-10
DOM	5	0.04	L	3.06e-8	14	0.11	L	6.03e-10
LTTW	-186.65	-6.77	S	0.1518	207.99	4.43	S	0.0816
MTTW	-216.78	-5.12	-	0.3076	-12.61	-0.18	-	0.3894

Logically, this should impact computational metrics. Indeed, on mobile a 3.53% reduction is visible in the TBT, see Table 9. Moreover, the FMP undergoes a reduction by 2.32%. It is possible this particular network call may have been acting as a blocker for the primary content on mobile. On desktop, no statistically significant changes were observed. The combination of a FMP and TBT reduction on mobile indicates these API calls acted as blockers for the mobile device. The fact these reductions are not apparent on the desktop device hints towards the different critical paths the two devices find themselves on.

Intervention 4: Stop refreshing when inactive

Intervention 4 is not an intervention made for initial page load reasons. Instead, the change is made to increase the responsiveness when users switch back to the tab which hosts the 30 MHz platform. The widgets in the 30 MHz platform update at a set interval, sometimes as often as every minute. When the platform occupied an inactive tab, it would stop rendering these updates, queuing them up instead. When switching back to this tab, each one of these updates would execute one by one. If the user had been away for a particularly long time, this queue would be very large.

Resolving this issue is simple. Tabbing away from the web application suspends the updates, and switching back to the web application forces an update. This intervention adds computational logic, thus complexity and, as a result, computational metrics suffer. On desktop, the TBT is increased by 4.33%, a 241.70 ms difference. On mobile, larger changes in metrics are observable. Not only is the TBT increased by 3.60%, the FMP, FCI, TTI and MTTW are all increased by more than 2%, see Table 10. While these same metrics also increase on desktop, the results are not statistically significant.

Intervention 5: Migrate to a PWA

Intervention 5 is a significant intervention. It migrates the 30 MHz platform to a Progressive Web Application (PWA).[37] A PWA offers significant opportunities for performance improvement, due to its caching technology. A PWA stores a copy of the resources needed to render a web page on the device itself. As a result, the HTML, CSS, JS and even image files no longer need to be retrieved from a server, these are simply retrieved from the device's storage. This allows for significant improvement of rendering metrics. In addition, the service worker, the core technology that makes up a PWA, has the ability to cache network requests. This allows for caching certain API URLs. In the case of the 30 MHz application, these could not be used, due to the real-time nature of the application. The results of the migration to a PWA can be seen in Table 11.

Significant performance increases are visible across almost all metrics, yet most notable in the results is the decrease in the FCP, losing more than 2 s for both devices. The desktop device undergoes its FCP and FMP decrease, percentually, significantly more than the mobile device. While the desktop undergoes a

[37] https://web.dev/what-are-pwas.

Table 9
Intervention 3 (OD2): Deduplicate an API call.

Metric	Desktop				Mobile			
	Δ	Δ %	ES	p-value	Δ	Δ %	ES	p-value
FCP	30.81	1.08	-	0.4823	96.60	1.63	M	0.0068
FMP	-290.58	-7.29	-	0.4823	-171.13	-2.32	M	0.0271
SI	0.55	0.01	S	0.4303	-130.60	-0.94	-	0.3384
TBT	-11.51	-0.21	-	0.4519	-333.70	-3.53	M	0.0037
EIL	2.82	0.40	-	0.4519	52.43	3.21	S	0.0670
FCI	-201.49	-1.75	-	0.4303	-239.21	-1.31	-	0.2149
TTI	-218.40	-1.85	-	0.4303	-318.44	-1.64	S	0.0452
NR	-24.50	-9.11	-	0.4823	-12	-7.69	L	2.64e-10
DOM	6	0.05	S	0.4303	0	0	S	0.0037
LTTW	52.07	2.03	-	0.4303	48.39	0.99	-	0.0672
MTTW	99.09	2.47	-	0.4519	-0.17	0	M	0.4705

Table 10
Intervention 4 (BF3): Stop refreshing when inactive.

Metric	Desktop				Mobile			
	Δ	Δ %	ES	p-value	Δ	Δ %	ES	p-value
FCP	11.48	0.40	-	0.4647	40.28	0.67	-	0.4588
FMP	202.62	5.49	S	0.2632	196.70	2.72	M	0.0125
SI	143.40	1.87	-	0.3091	170.68	1.24	S	0.1473
TBT	241.70	4.33	M	0.0434	328.70	3.60	L	3.55e-5
EIL	-5.25	-0.74	-	0.3557	31.87	1.89	-	0.3208
FCI	458.32	4.04	S	0.1532	456.58	2.53	L	6.58e-5
TTI	460.82	3.97	S	0.1532	542.94	2.85	L	4.61e-5
NR	4	1.64	L	0.0005	2	1.39	L	5.91e-6
DOM	58	0.44	L	3.06e-11	20	0.15	L	0.0004
LTTW	22.48	0.86	-	0.3010	3.85	0.08	-	0.4283
MTTW	63.87	1.55	-	0.3039	209.66	3.04	M	0.0033

69.29% reduction in FCP and a 28.11% reduction in FMP, the mobile device's FCP is reduced by 34.43%, with the FMP not showing a statistically significant difference. The primary bottleneck for the desktop device is the network, which this intervention specifically addresses and, thus, this makes sense. This is also apparent from a 15.39% reduction in the SI on desktop, as opposed to a 6.68% reduction on mobile. While both are significant increases in performance, the desktop device gains significantly more from this change. Curiously, the LTTW did not decrease on desktop, yet the MTTW did. This indicates users see the first chart being rendered with the same timing as before, yet the median number of charts load in faster. This effect is reversed on mobile, where the LTTW is reduced by a second, yet the MTTW did not decrease. This goes against our intuition, as this is the exact opposite of the results indicated by the FMP, which only undergoes a large improvement on desktop devices.

Intervention 6: Upgrade to Angular 8

Developers tend to use outdated libraries in their web apps (Lauinger et al., 2017). Therefore, though upgrading the Angular's version requires minimal effort from a development perspective, it is considered a conscious development choice. As per our suggestion to developers, Angular 7 was upgraded to Angular 8 , thus making such an upgrade a worth intervention to investigate.[38]

Due to dependency conflicts and the need to support Internet Explorer 11, notable features like differential loading and the Ivy compiler were unable to be used. No other performance changes affecting the 30 MHz web platform were found in the changelog. Nonetheless, significant performance differences were achieved by the upgrade, particularly in the FCP on both devices. The FCP on Desktop is reduced by 91.76%, while on mobile it is reduced by 96.85%, a reduction of 3853.87 ms, see Table 12.

In addition, on mobile the FMP is reduced by a similarly large amount, 51.78%. It is not entirely clear what changes on the Angular side caused these differences in particular. Moreover, clear discrepancies are visible between the two devices. For example, the FMP on mobile is reduced from 7245.56 ms to 3494.09 ms, while the FMP on the desktop device did not undergo a significant change. Finally, both devices undergo a large increase in the TBT (by 35.62% for desktop TBT and by 40.02% for mobile TBT).

Intervention 7: Intelligently optimize the number of widgets rendered

As described in Section 4.1.1, 20 widgets are loaded upon first loading the dashboard. Scrolling down and hitting a certain boundary loads 20 more, with this process continuing until all widgets have been loaded. The NVD3 based charting framework is computationally heavy. Ideally, the number of charts rendered is kept to a minimum. Chart widgets are large, with 6 chart widgets fitting above the fold in a full-screen window on a 1080p monitor, yet, 20 widgets are loaded. This is suboptimal. Ideally, only 7

[38] https://blog.angular.io/version-8-of-angular-smaller-bundles-cli-apis/-and-alignment-with-the-ecosystem-af0261112a27.

Table 11
Intervention 5 (AP1): Migrate to a PWA.

Metric	Desktop				Mobile			
	Δ	Δ %	ES	p-value	Δ	Δ %	ES	p-value
FCP	-2,001.08	-69.29	L	5.27e-11	-2,089.61	-34.43	L	4.36e-9
FMP	-1,094.98	-28.11	L	5.26e-9	-172.19	-2.32	S	0.1044
SI	-1,201.12	-15.39	L	5.27e-11	-927.50	-6.68	L	0.0017
TBT	873.80	14.99	L	0.0001	1,684.37	17.81	L	2.97e-8
EIL	-1.79	-0.25	-	0.5000	-79.17	-4.61	S	0.1918
FCI	-647.91	-5.49	M	0.0019	549.38	2.96	S	0.0119
TTI	-528.66	-4.38	M	0.0124	659.63	3.37	S	0.0088
NR	-94.50	-38.03	L	5.09e-11	1	0.68	L	5.43e-5
DOM	0	0	S	0.1270	-19	-19	M	0.0017
LTTW	41.52	1.57	-	0.5000	-1,006.70	-20.32	L	4.74e-9
MTTW	-563.37	-13.48	L	4.60e-9	-106.97	-1.51	S	0.1674

Table 12
Intervention 6 (UD1): Update to Angular 8.

Metric	Desktop				Mobile			
	Δ	Δ %	ES	p-value	Δ	Δ %	ES	p-value
FCP	-814.00	-91.76	L	3.93e-11	-3,853.87	-96.85	L	3.16e-10
FMP	26.25	0.94	-	0.2940	-3,751.50	-51.78	L	7.02e-8
SI	1,254.18	19	L	3.93e-11	325.51	2.51	S	0.0562
TBT	2,387.25	35.62	L	3.93e-11	4,458.00	40.02	L	5.41e-10
EIL	18.56	2.63	L	6.48e-8	80.93	4.95	S	0.0562
FCI	2,064.50	18.53	L	3.93e-11	688.51	3.61	S	0.0562
TTI	2,120.75	18.38	L	3.93e-11	470.51	2.32	S	0.1352
NR	0	0	-	0.3434	0	0	-	-
DOM	-5	-0.04	L	3.31e-10	-8	-0.06	L	5.41e-10
LTTW	404.45	15.06	L	3.93e-11	67.62	1.71	S	0.1130
MTTW	480.99	13.30	L	3.31e-10	-20.03	-0.29	-	0.4628

chart widgets would be loaded: enough to fill the viewport, in addition to a widget below the fold to allow scrolling. There are a number of ways to calculate this. One such way is to calculate the height of the viewport, along with the cumulative height of the widgets, and halt the loading of new widgets as soon as we have assembled enough height. This does not work for the 30 MHz application since image widgets have variable heights, depending on the image.

Instead, a scoring based system is implemented. This scoring system is based entirely around chart widgets. If a number of chart widgets exist in the batch of widgets, we make sure we render at least 10 widgets, and from then on check as we loop through the widgets if the number of chart widgets exceeds 4. If so, we limit the number of widgets rendered right there. If this dashboard is just chart widgets, we render just 7 widgets. The resulting behaviour fills the viewport in all cases and dramatically reduces the number of chart widgets being rendered.

This change drastically reduces the computation needed to load the page, but only after the FCP, and consequentially should result in a performance increase for both computational and general metrics. While the scoring based algorithm does require a small amount of upfront computational time, this should be offset by the number of widgets prevented from rendering.

This intervention causes a 41.22% reduction in the SI on desktop devices. Other metrics, such as the FMP, TBT, FCI, TTI, LTTW and MTTW are all reduced by more than 30%, see Table 13. Again of note are the differences between the two devices. The desktop device is able to achieve larger performance increases on all compared metrics. The SI on the mobile device is reduced by 17.19%. One would expect a computational optimization to affect mobile devices the most due to the computational bottleneck present on those devices (see Section 7.2), yet this is not the case here. While computational metrics undergo a large improvement on mobile, such as the TTI with a reduction of 25.61%, these improvements are relatively smaller than what was undergone on the desktop device. Nonetheless, on both devices the performance increase according to the metrics is significant. From this, we can deduct that this intervention has a significant positive impact on user-perceived performance.

Intervention 8: Migrate to Apache Echarts

While optimizing the number of widgets being rendered, as described in Intervention 7 achieved significant gains, more performance improvements could still be made in the area of charts. Together with product owners, it was decided to change charting frameworks: from NVD3 to Apache ECharts.[39] ECharts (Li et al., 2018) is a charting framework with a significant focus on achieving high performance, and this became immediately apparent in the 30 MHz application. Note that in this intervention, we migrated the widgets to Apache Echarts and then benchmarked the web app, while benchmarking involving NVD3 was rather done separately across all the previous interventions (i.e., from the baseline to Intervention 7). The 5–10 frames per second mentioned in Section 7.2 is able to be improved to 50 frames per second, in the median case, while interacting with charts. This change required a large amount of engineering work, adding

[39] https://echarts.apache.org.

Table 13
Intervention 7 (OC1): Intelligently optimize the number of widgets rendered.

Metric	Desktop				Mobile			
	Δ	Δ %	ES	p-value	Δ	Δ %	ES	p-value
FCP	125.50	171.74	-	0.3286	2.07	1.65	-	0.4882
FMP	-1,266.17	-44.80	L	9.66e-6	-749.00	-21.44	L	6.77e-6
SI	-3,237.66	-41.22	L	1.97e-11	-2,283.20	-17.19	L	9.66e-10
TBT	-4,176.25	-45.49	L	1.97e-11	-3,704.59	-23.75	L	3.22e-11
EIL	-22.36	-3.09	S	0.0511	310.60	18.09	S	0.0418
FCI	-6,689.00	-50.64	L	1.97e-11	-4,978.50	-25.18	L	3.22e-11
TTI	-6,658.75	-48.75	L	1.97e-11	-5,307.25	-25.61	L	3.22e-11
NR	-44	-28.57	L	3.46e-12	-44	-29.93	L	8.67e-14
DOM	-5,425	-40.75	L	1.74e-12	-5,371	-40.62	L	1.99e-12
LTTW	-1,027.20	-33.24	L	1.97e-11	-254.81	-6.35	L	0.0002
MTTW	-1,674.77	-40.87	L	1.97e-11	-2,316.91	-33.22	L	3.22e-11

Table 14
Intervention 8 (AT1): Migrate from NVD3 to Apache ECharts.

Metric	Desktop				Mobile			
	Δ	Δ %	ES	p-value	Δ	Δ %	ES	p-value
FCP	22	11.08	L	1.02e-8	54.58	42.81	L	1.02e-9
FMP	-231.74	-14.85	-	0.3287	-1,694.50	-61.73	S	0.0157
SI	911.79	19.75	L	2.63e-11	1,682.29	15.29	L	2.22e-7
TBT	1,491.25	30.35	L	2.63e-11	2,055.34	17.28	L	3.78e-8
EIL	65.57	9.34	-	0.3287	936.86	46.20	M	0.0010
FCI	2,362.00	36.23	L	2.63e-11	2,426.50	16.41	L	1.71e-8
TTI	2,197	31.38	L	2.63e-11	2,451.25	15.90	L	1.71e-8
NR	10	9.09	L	7.10e-12	10	9.71	L	6.18e-13
DOM	-7,373	-93.47	L	8.76e-14	-7,337	-93.44	L	1.44e-13
LTTW	-152.92	-7.41	L	2.19e-9	324.31	8.62	L	3.07e-6
MTTW	110.78	4.57	L	1.40e-8	689.95	14.81	L	2.41e-8

and changing a large number of features and, thus, is not solely a refactor. Bundle size increased by 200 KB. The results of the migration to ECharts are negative for the page load performance, see Table 14.

A performance reduction of 19.75% and 15.29% is visible for the SI of desktop and mobile devices respectively. This intervention can best be seen as a trade-off between page load performance and runtime performance. Runtime performance is increased to an acceptable level, yet page load performance has decreased. ECharts is by default a charting framework that uses canvas as opposed to NVD3's SVG, so we observe the DOM size reduced to just 515 elements on desktop, instead of 7888 elements before. Common consensus is that such a large reduction of DOM size should lead to better page load performance,[40] which this case study is unable to replicate. Changing ECharts configuration options to use SVG did not result in an improvement.

Due to the complicated nature of this intervention and the dependencies involved, ascribing performance differences is difficult. Observable is a large reduction in the FMP on mobile, reducing by 61.73%. This is the only metric for which performance is observed to increase on mobile. While a decrease is also measured on desktop, this result is not statistically significant. The desktop device does show improvement in another metric, the LTTW. The LTTW is reduced by 7.41%. Logically, this would lead to the conclusion that, compared to NVD3, ECharts requires more computation, but at a later point in the page load lifecycle. Since the desktop device has more computational power available, the first chart loading in is not penalized by the increased computational demands as much as it is on mobile, where the LTTW increases by 8.62%.

Intervention 9: Move third-party JavaScript code to body

The 30 MHz platform uses the Zendesk[41] platform to provide customer support. This comes in the form of a button that is loaded onto the page via a third-party script. This JavaScript code had been placed in the <head> section of the HTML page. While in the <head>, scripts act as blockers for further parsing of the DOM (Wang et al., 2013). The script is not necessary during the page load, thus could be moved to the end of the <body> section, where it would be parsed after the HTML of the page had been rendered. This should lead to a reduction in the computation, but also the page load in general, due to the blocking nature.

This assertion largely proves correct. Observable in the results is an enlarged effect on the mobile device. Notably, the SI is not reduced on desktop, whereas it is reduced by almost a second on mobile, see Table 15. This could be caused by a different critical path on the two devices, with the critical path on mobile being blocked for a longer time. While both devices have to process the code, if the desktop device is not waiting on this code to execute and is instead executing tasks in parallel, no significant benefit would be apparent in general metrics. Conversely, it is also possible the two devices are on the same critical path, yet the larger amount of time needed to process the code on mobile (due to the lesser computational power) leads to a longer blocking time.

40 https://developers.google.com/web/tools/lighthouse/audits/dom-size.

41 https://www.zendesk.com.

Table 15
Intervention 9 (OA1): Move third-party script to body.

Metric	Desktop				Mobile			
	Δ	Δ %	ES	p-value	Δ	Δ %	ES	p-value
FCP	-7.00	-3.17	S	0.1071	3.00	1.65	-	0.2572
FMP	-14.77	-1.11	S	0.0657	-7.00	-0.67	-	0.2572
SI	19.89	0.36	-	0.4441	-805.12	-6.35	L	0.0001
TBT	-163.25	-2.55	L	0.0004	-736.75	-5.28	L	2.05e-7
EIL	-0.50	-0.07	-	0.4441	-526.80	-17.77	L	1.17e-10
FCI	-230.50	-2.60	L	0.0001	-684.50	-3.98	L	2.32e-7
TTI	-179.25	-1.95	L	0.0001	-761.50	-4.26	L	2.57e-7
NR	0	0	-	0.3338	0	0	-	0.2529
DOM	1	0.19	L	8.67e-15	1	0.19	L	1.44e-13
LTTW	-46.67	-2.44	L	6.32e-6	-174.82	-4.28	L	8.04e-6
MTTW	-55.07	-2.17	L	6.32e-6	-326.75	-6.11	L	4.07e-7

Table 16
Intervention 10 (OD1): Preload fonts.

Metric	Desktop				Mobile			
	Δ	Δ %	ES	p-value	Δ	Δ %	ES	p-value
FCP	-157.50	-73.74	L	3.82e-7	-39.00	-21.07	L	0.0009
FMP	-79.51	-6.05	S	0.0275	-30.74	-2.95	S	0.2092
SI	17.31	0.31	S	0.1788	449.32	3.78	S	0.1661
TBT	156.52	2.51	M	0.0130	327.75	2.48	S	0.1661
EIL	-0.10	-0.01	-	0.3448	-4.00	-0.16	-	0.4845
FCI	46.50	0.54	S	0.0630	225.50	1.36	S	0.2092
TTI	111.50	1.24	S	0.0274	269.75	1.58	S	0.1661
NR	1	0.83	L	4.45e-7	1	0.88	L	1.26e-11
DOM	0	0	-	-	0	0	-	-
LTTW	25.01	1.34	M	0.0008	-23.10	-0.59	-	0.4198
MTTW	52.80	2.13	L	0.001	37.67	0.75	S	0.2092

Intervention 10: Preload fonts

An analysis of the page render process found one very specific blocker: a font required on average 150 ms, every page load, to be retrieved. This can be optimized by loading this font earlier on in the page lifecycle using preload.[42] As a result, we observe both FCP and FMP increase in performance, see Table 16. On desktop, the FCP is now 68.08 ms, after a reduction of 157.50 ms. When transitions happen in the range of 0 to 100 ms, users perceive these to be instant according to Google Web Fundamentals guidelines,[43] thus, as of this milestone, users will perceive the 30 MHz platform to paint to screen instantly upon navigating to the page. Meanwhile, on mobile the FCP equals 146.08 ms at this point, which falls in Google's range of 100 to 300 ms: "users experience a slight perceptible delay".

Intervention 11: Remove unused third party JavaScript code

Intervention 11 is not a performance optimization per se. Instead, third party analytics JavaScript code from Hotjar,[44] an analytics platform allowing for profiling of the behaviour of users, was removed. This was discussed with product owners after noticing particularly large computation times being required by the script. This intervention has very different effects on both devices. While on desktop this change reduces the SI by a large amount (23.11%), there was no difference in SI on mobile, see Table 17.

The improvements were undergone by the desktop device are much larger than one would expect from this change. Aside from the improvement in SI, all computational metrics are improved significantly, indicating the computational time required by the web application is reduced significantly. Moreover, the large loss in TBT, with a 1842.88 ms reduction, signals that the Hotjar script was part of a long-running task. However, we observe no real performance change for the mobile device. Aside from a performance reduction in the EIL, by 7.28%, no metrics undergoes a statistically significant change. This indicates the critical path is different on mobile, with no performance increase from the removal of the code as a result.

Intervention 12: Preconnect to origins

Pages connecting to third-party origins during the page load can opt to use the `preconnect`[45] keyword to have the browser open a connection to these origins as soon as possible. The 30 MHz platform connects to a number of third party origins for analytics and icons, thus, in a similar vein to Intervention 11, these origins were preconnected to. This could not be done for all third party scripts, since using `preconnect` uses up CPU time per connection.[46] If an opened connection is not used soon enough

42 https://developer.mozilla.org/en-US/docs/Web/HTML/Preloading_content.
43 https://developers.google.com/web/fundamentals/performance/rail.
44 https://www.hotjar.com.
45 https://www.w3.org/TR/resource-hints/#preconnect.
46 https://www.igvita.com/2015/08/17/eliminating-roundtrips-with-preconnect.

Table 17
Intervention 11 (UD1): Remove analytics script.

Metric	Desktop				Mobile			
	Δ	Δ %	ES	*p*-value	Δ	Δ %	ES	*p*-value
FCP	-17.51	-31.22	M	0.0140	-11.37	-7.78	-	0.4555
FMP	120.10	9.73	M	0.0102	-3.62	-0.36	-	0.4555
SI	-1,286.40	-23.11	L	2.26e-11	491.46	3.99	S	0.1514
TBT	-1,842.88	-28.80	L	2.26e-11	105.50	0.78	S	0.1958
EIL	-528.87	-68.95	L	2.26e-11	177.20	7.28	L	0.0002
FCI	-1,959.71	-22.53	L	2.26e-11	127.99	0.76	S	0.1958
TTI	-2,036.01	-22.30	L	2.26e-11	276.74	1.59	S	0.1958
NR	-4	-3.31	L	3.89e-12	-3	-2.63	L	5.07e-13
DOM	-1	-0.19	L	8.67e-14	-1	-0.19	L	3.99e-13
LTTW	-38.30	-2.03	M	0.0025	-23.79	-0.61	-	0.4555
MTTW	-38.07	-1.50	S	0.0432	119.12	2.35	S	0.1205

Table 18
Intervention 12 (OD1): Preconnect to origins.

Metric	Desktop				Mobile			
	Δ	Δ %	ES	*p*-value	Δ	Δ %	ES	*p*-value
FCP	4.01	10.39	M	0.0309	10.99	8.07	S	0.1767
FMP	-15.95	-1.18	-	0.4015	-0.25	-0.02	S	0.1723
SI	10.93	0.26	-	0.4823	-730.33	-5.73	M	0.0610
TBT	3.40	0.07	-	0.4015	-213.75	-1.56	S	0.1153
EIL	-6.35	-2.67	-	0.4015	-98.40	-3.78	L	0.0069
FCI	45.01	0.67	S	0.3393	-344.99	-2.04	S	0.1039
TTI	-13.34	-0.19	-	0.4015	-401.75	-2.28	S	0.1124
NR	4	3.42	L	1.74e-12	5	4.50	L	1.71e-11
DOM	0	0	-	-	0	0	-	-
LTTW	-8.89	-0.48	-	0.4015	-62.29	-1.62	S	0.0672
MTTW	3.55	0.14	-	0.4015	-78.32	-1.51	S	0.0672

within the load process, the benefit gained from `preconnect` is not worth the tradeoff.

The resulting performance difference from this intervention is minimal. The EIL is reduced on mobile, by 3.78%, see Table 18. While the measurements also indicate improvements in the SI and computational metrics on mobile, these differences are not statistically significant.

Intervention 13: Reduce API calls made by charts

In a continuation of the work done in Intervention 3, more work was done to reduce the number of API calls. One call, similar to Intervention 3, was entirely unneeded. The other, a call to the `/comments` endpoint, was able to be consolidated into the earlier mentioned API call to `/dashboard`. This `/comments` call was a blocker, comments would only be drawn over the chart when it completed. It is thus expected to improve both computational and general metrics. The measurements indicate this is only the case for the desktop device. The desktop SI is reduced by 3.38% and the MTTW is reduced by 5.05%. The TBT, FCI and TTI are reduced by 5.26%, 6.10% and 5.49% respectively, see Table 19. On mobile, this performance increase is not visible.

4.1.4. Summary of results for RQ1

The overall performance of the 30 MHz platform after the interventions shown in Section 4.1.3 has improved significantly. On desktop, we observe an SI reduction of 48.25%, FCP reduction of 98.37% and TTI reduction of 46.00% (see Table 20). On mobile, we observe improvements in the majority of the metrics, such as SI's improvement by 19.85%. The most notable gains are in the category of rendering metrics, such as FCP improves by 97.56% and FMP by 75.01%. All measurements achieve a high statistical significance. As observed throughout the interventions, desktop and mobile devices rarely match entirely in their response to an intervention. For example, Intervention 11 is responsible for a 23.11% SI reduction on desktop, whereas no statistically significant difference observed on mobile.

In general, on mobile, we are unable to match the improvements observed on desktop. Overall, however, both devices have undergone performance improvements. Often, these performance improvements were not immediately visible from one intervention to the next. To further illustrate this, the Cliff's Delta effect size of the SI was calculated for all combinations of interventions, see Fig. 8a and b. The small, medium and large effect sizes are denoted by `S`, `M` and `L`, respectively, with a hyphen representing a negligible effect size. This gives an overview of the effect size of the interventions. For example, on desktop, Intervention 6 can be observed to have a large negative effect size compared to Intervention 5, yet Intervention 7 is able to turn the tide and achieve a large effect size compared to previous interventions, again. On mobile, we observe the lack of improvement in the SI from the later interventions. The effect size of every intervention following Intervention 7 is both large and negative. This juxtaposition of successes highlights just how different the two platforms are in handling the 30 MHz platform. It can be observed that the difference in performance after Intervention 7 can in large

Table 19
Intervention 13 (OD2): Reduce API calls.

Metric	Desktop				Mobile			
	Δ	Δ %	ES	p-value	Δ	Δ %	ES	p-value
FCP	4.00	9.39	-	0.2893	4.75	3.20	-	0.2554
FMP	-46.29	-3.46	M	0.0056	-44.03	-4.41	S	0.1844
SI	-145.09	-3.38	L	0.0004	197.95	1.64	-	0.4261
TBT	-239.65	-5.26	L	1.32e-7	-279.25	-2.07	S	0.1844
EIL	-6.35	-2.67	L	0.0002	42.40	1.69	M	0.0564
FCI	-413.50	-6.10	L	2.23e-9	-257.50	-1.55	S	0.1017
TTI	-388.90	-5.49	L	9.75e-10	-304.99	-1.77	S	0.0822
NR	-26	-21.49	L	2.45e-13	-26	-22.41	L	1.39e-10
DOM	-8	-1.55	L	8.67e-14	-8	-1.55	L	1.83e-12
LTTW	-101.03	-5.49	L	8.36e-7	14.81	0.39	-	0.4551
MTTW	-126.13	-5.05	L	1.52e-7	-90.38	-1.77	-	0.2947

Table 20
End of case study performance results.

Metric	Desktop				Mobile			
	Value	Δ %	ES	p-value	Value	Δ %	ES	p-value
FCP	46.58	-98.37%	L	1.44e-11	153.33	-97.56%	L	2.59e-11
FMP	1,292.25	-67.21%	L	1.44e-11	953.33	-75.01%	L	2.59e-11
SI	4,145.96	-48.25%	L	1.44e-11	12,271.07	-19.85%	L	1.16e-10
TBT	4,318.65	-29.98%	L	1.44e-11	13,180.75	23.08%	L	1.42e-10
EIL	230.97	-67.71%	L	1.44e-11	2,549.60	49.88%	L	2.59e-11
FCI	6,368.88	-47.54%	L	1.44e-11	16,344.58	-17.98%	L	2.59e-11
TTI	6,692.08	-46.00%	L	1.44e-11	16,958.08	-19.32%	L	2.59e-11
NR	95	-64.68%	L	4.34e-12	90	-41.18%	L	1.97e-12
DOM	507	-96.17%	L	1.74e-12	507	-96.16%	L	1.97e-12
LTTW	1,740.23	-35.62%	L	1.44e-11	3,798.72	-20.80%	L	2.59e-11
MTTW	2,371.17	-43.20%	L	1.44e-11	5,006.26	-31.23%	L	2.59e-11

part be attributed to the difference resulted from Intervention 10: preload fonts. While this intervention was a sizeable success on the desktop device, the mobile device not only undergoes no statistically significant results across most metrics, but the effect size is also negative.

While some of the interventions are specific to the 30 MHz platform, many other interventions can be applied in other contexts as well. In particular, optimizations based on techniques such as preloading and minimize complex calculations can easily be abstracted and applied to other web applications. However, specific technology-linked interventions, such as Intervention 8, migration to ECharts, are harder to apply in a generalized context.

4.2. Summary of results of RQ2

Correlation plots for all metrics are shown in Fig. 9. As we observe, LTTW, MTTW, SI and TBT follow a similar rate of change as uPLT. Only for the LTTW can this claim be statistically confirmed, with a *p-value* of 0.0306 (see Table 21). The LTTW achieves a perfect correlation with the uPLT.

For each pair of videos, the browser loading indicator appeared for the first video but not the second video. We found no correlation between the cases with and without indicators. This is likely related to the fact that the loading indicator finishes a few seconds before loading the charts. The high correlation between LTTW and uPLT suggests that users are using the charts as an indication of the loading status.

Table 21
Correlation between web performance metrics and user perceived performance.

Metric	Kendall's τ coefficient	p-value
FCP	−0.467	0.5444
FMP	0.600	0.4083
SI	0.867	0.1667
TBT	0.867	0.1667
EIL	−0.333	0.5444
FCI	0.733	0.2222
TTI	0.867	0.1667
NR	0.828	0.1667
DOM	0.867	0.1667
LTTW	1.000	**0.0306**
MTTW	0.867	0.1667

5. Discussion

This case study shows the importance of performance engineering in web development. Through iterative improvement, we achieved significant performance improvements. These improvements are often not attributable to a single intervention. Instead, the overall performance improvement is the result of a sequence of interventions over time. For developers, this highlights the **need for continuous focus on performance**, as opposed

#	B	1	2	3	4	5	6	7	8	9	10	11	12	13
B														
1	-													
2	S	-												
3	M	M	S											
4	S	S	-	-										
5	L	L	L	L	L									
6	S	S	S	-	S	L								
7	L	L	L	L	L	L	L							
8	L	L	L	L	L	L	L	L						
9	L	L	L	L	L	L	L	L	-					
10	L	L	L	L	L	L	L	L	-	S				
11	L	L	L	L	L	L	L	L	L	L	L			
12	L	L	L	L	L	L	L	L	L	L	L	-		
13	L	L	L	L	L	L	L	L	L	L	L	L	L	

(a) Desktop interventions

#	B	1	2	3	4	5	6	7	8	9	10	11	12	13
B														
1	L													
2	L	L												
3	L	L	-											
4	L	M	-	S										
5	L	L	M	M	L									
6	L	L	S	S	M	S								
7	L	L	L	L	L	L	L							
8	L	L	L	L	L	S	L	L						
9	L	L	L	L	L	L	L	L	L					
10	L	L	L	L	L	L	L	L	M	S				
11	L	L	L	L	L	S	M	L	-	M	S			
12	L	L	L	L	L	L	L	L	M	-	-	M		
13	L	L	L	L	L	L	L	L	M	-	-	S	-	

(b) Mobile interventions

Fig. 8. Cliff's Delta effect size of the SI of combinations of desktop interventions (a) and mobile interventions (b). Green cells indicate improvement, red cells indicate a decrease in performance. B denotes the baseline situation. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

to dedicated sprints specifically for performance. This finding is supported by industry professionals, such as Addy Osmani.[47]

During the case study we also experienced a strong reduction of the variability of the collected measures as the performance of the 30 MHz dashboard was improving. At the beginning of this work, the high variability of various measures made it very difficult for us to evaluate the impact of the interventions we were applying. Such variability meant that simple performance measuring strategies, such as making a subjective judgement based on the "feel" of the application, were entirely infeasible. In this context, two main factors helped us: (i) **continuously using a stable benchmarking tool**, which allowed us to fully trust the collected measures and control to some extent the potential confounding factor of the measurement infrastructure and (ii) **statistically analysing the impact of each intervention**, so to have objective evidence of the correlation between the performed intervention and the change in the distribution of the collected measures. We suggest developers to include these two factors in their web performance engineering activities to have robust and objective data on which better informed decisions can be taken. As a first step towards this practice, we make our benchmarking tool publicly available as an open-source project under the MIT license. Moreover, with this study we also exemplify how statistical analysis can play a key role in objectively comparing the performance between different versions of the same web app, thus strongly supporting the decision process of product owners throughout all web engineering activities. To streamline the statistical analysis, it can be embedded into a dedicated Continuous Integration (CI) pipeline; in this way the CI pipeline (a) masks the complexity of the statistical analysis to the development team and (b) shows only the metrics requiring special attention and the candidate points of intervention in the targeted web app.

While some of these interventions are specific to the 30 MHz platform, many can be applied generally. Optimizations based on techniques such as preloading and minimizing complex calculations can easily be abstracted and applied to other web apps. Technology-specific interventions, such as Intervention 8 (Migration to ECharts), are harder to apply in a generalized case, aside from acting as an indication of the improvements that can be accomplished by **paying attention to the performance impact of external dependencies**.

As described in Section 4.1.2, the process of deciding which interventions to accept/reject depended on the analysis of bottlenecks (and their related trade-offs). For example, if there was a certain bottleneck with a product feature or a browser feature, then the focus was on how to alleviate such a bottleneck to improve the user experience of such a feature. In our case study, we relied on a combination of priorities given by 30 MHz developers and the expected improvements (based on the documentation of Google LightHouse and other sources on Web performance). Accepting or rejecting an intervention depends on whether certain metrics improve or not. Based on the feedback from 30 MHz developers, an intervention is considered successful if key metrics (such as FMP, SI and MTTW) improve. Also, while other metrics, such as FCP and LTTW, should ideally be improved, a slight worsening can still be acceptable. The remaining metrics should also have some improvement, but it is acceptable if they worsen, while general or FMP/FCP metrics improve in return. For example, looking at the results for Intervention 6 (Table 12), which was indicated above, we observe that FCP and FMP metrics significantly improved on mobile, thus leading to the acceptance of such intervention, even though the remaining metrics became worse since these metrics are strongly beneficial to the perception of page loads. On desktop, however, FMP became slightly worse. Yet given that FCP significantly improved, this intervention was still accepted since showing users content as quickly as possible is worth it and cannot be ignored, even if the overall load time increased.

Our results show that metrics in some interventions might improve on one device (e.g., desktop) but worsen on the other (e.g., mobile). The difference in performance between desktop and mobile devices was mainly attributed to computation or network bottleneck (Nejati and Balasubramanian, 2016), depending on the metrics as they measure different aspects. While some metrics are general, other metrics specifically measure computational load or network performance. **Computation is the primary bottleneck on mobile devices**, as using a mobile version of a page can reduce the size of objects but does not really simplify the computation required to load the page. Also, **browsers on desktop devices have more stable network conditions**, unlike mobile devices, where network is known to be relatively slow and unreliable (Wang et al., 2016; Sivakumar et al., 2014). Therefore, it is expected to see higher improvement or lower worsening on desktop than on mobile for metrics associated with page load. However, for other metrics, such as those related to rendering, metrics to perform similarly on both devices, with some interventions as exceptions, which could be due to the type of pages we used (Nejati and Balasubramanian, 2016). Still, future research should further investigate the actual causes behind metric improvement or worsening across different devices.

47 https://dev.to/ben/addy-osmanis-18-point-web-performance-checklist-2e1.

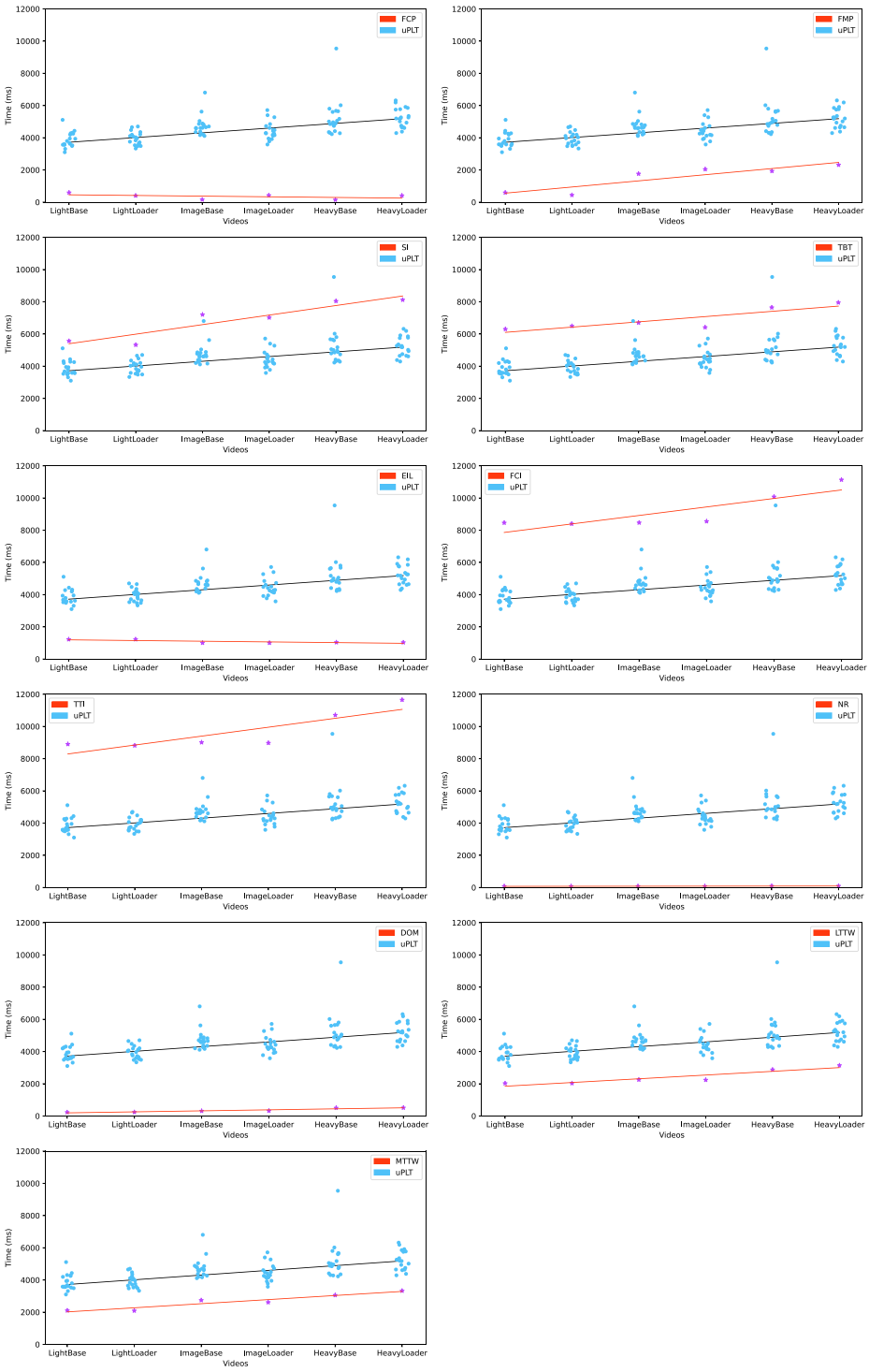


Fig. 9. Correlation plots of the metrics with uPLT.

Performance metrics highlight different aspects of a web app. Therefore, **there is no single metric that can be considered the best representative of web performance**. Therefore, in this study we use multiple performance metrics in order to build on their characteristics. Still, we believe that more data and empirical studies like this one are needed to better understand the correlation between quantitative performance metrics and user-perceived performance. In addition, performance metrics and tools for the Web do evolve over time, as researchers and industrial parties frequently update their performance metrics for the sake of finding better alternatives. As a representative example, the weights of performance metrics of Google Lighthouse change across different versions.[48] Overall, the lessons learned are: (i) web performance engineering is a process where **design decisions must be taken with the overall web application in mind and with an openness towards trade-offs**, and (ii) measuring **multiple performance metrics is fundamental** since a single

48 https://developer.chrome.com/docs/lighthouse/performance/performance-scoring.

metric may fail in capturing the entire picture. It is important to stress these are linked. It is not possible to judge the value of trade-offs without seeing the full picture, nor is it feasible to expect all performance metrics to always return a positive answer. The user study performed as part of this paper was unable to determine an influence by the browser loading indicator. In the case of the 30 MHz application, the browser loading indicator stops spinning multiple seconds before elements such as the graphs load in. Considering the accuracy of the LTTW in terms of predicting relative performance differences, the lack of influence leads to the conclusion that the charts and image widgets are critical to the user perceived page load performance. Without these, developers may risk overlooking decisions that improve the overall state of their application, due to negative results in a limited area. It is recommended to maintain a performance budget per performance metric, that allows for certain variation when the net result is positive.

6. Threats to validity

Construct validity. The nature of performance optimization makes it difficult to conclude when version A of a web application is better than version B. With a careful analysis of the performance metrics, along with theorizing beforehand what the expected results should be, it was possible to determine the best path to take. This approach was further corroborated by the user study. The benchmarking tool is based on Google Lighthouse, a widely used tool for the purpose of measuring performance of web pages. All metrics used as part of this case study are commonly used in both industry and academia except for the Time to Widget, due to its specialized nature.

External validity. Angular is one of the larger web frameworks in use today.[49] However, its specific opinionated methodology does fundamentally influence the engineering work performed with it. Nonetheless, none of the work done during this case study is Angular-specific and, thus, the results of this case study are relatively easy to generalize to other web applications. While some of the work done at 30 MHz has a more specialized nature, for example, not all web pages or even Angular applications have this much of a focus on charts, many of the interventions can be abstracted to alternative scenarios. In addition, the methodology applied as part of this case study, using an iterative optimization approach backed by a robust benchmarking tool, is sound and easily applied to other web performance optimization scenarios. Moreover, we found a strong divergence between the two test devices in their response towards interventions, with observations that are applicable to the entire web platform. Finally, we chose Google Chrome as it is currently the most commonly used browser across all platforms, and the 30 MHz customers are actively advised to use it. Yet, our findings might not be generalizable to all browsers, hence future work should further investigate the impact of our proposed interventions on other browsers.

Internal validity. It is possible for benchmarking performed during this case study to have been affected by external factors. For example, browsers have received updates, network conditions might have changed or external programs may have been interfering. To mitigate the effects of these possible factors, all results reported in Section 4 are based on the measurements collected at the end of the case study in a homogeneous environment. This minimizes the number of possible variables. In addition, performing 30 runs of the web page by each benchmark helps minimize edge cases, such as a slower load than usual. Such a case could be caused by a large number of reasons, from network hiccups to a sudden clock change in the CPU. For the purposes of this case study, we are interested in the overall picture of performance, as opposed to one specific edge case. To mitigate the chances of misunderstandings during the user study, each participant was supervised and given the chance to ask questions beforehand. Moreover, videos were prepended with demo videos, after observing participants in pilot runs did not immediately understand what is being asked.

Conclusion validity. To mitigate the chance of drawing incorrect conclusions from the user study, statistical significance was calculated and only those metrics for which the null hypothesis could be rejected are the basis of conclusion. Similar work was performed for measurements retrieved for interventions. All p-values used in this case study were corrected. Assumptions underlying the method of correction were always tested, and when these assumptions could not be met, different methods were used that could be applied. Still, we believe that more data is needed to show how statistically significant our calculated metrics are correlated with user perception. In particular, future research should further expand our analyses using more pages to investigate whether our observations hold in terms of correlations and their statistical significance.

Failed interventions. A number of wanted interventions were unable to succeed, while these would have likely positively affected the 30 MHz platform. Brotli is a more optimized compression algorithm than gzip in most cases[50,51] (OD3). It was unable to be implemented due to limitations within Amazon's CloudFront service. Similar limitations prevented this case study from implementing WebP image compression (OD3). WebP images are 26% more efficient than PNGs and 25% to 34% more efficient than JPEGs, while maintaining quality.[52] Finally, legacy support prevented utilizing Angular's Ivy compiler, as well as differential loading. The Ivy compiler (AT1) is able to produce smaller bundle sizes,[53] while differential loading (OT1) allows splitting the legacy resources delivered to browsers, so that modern browsers do not have to receive resources, such as polyfills, that are not needed.[54]

7. Related work

7.1. Web performance metrics

Performance optimization is not possible without a good target to optimize for. This makes performance metrics important. Most common is the use of the `Page Load Time` (PLT). The PLT is the time from `navigationStart` to the JavaScript `onLoad` event. The `onLoad` event is triggered when the page has fully finished loading, including HTML, CSS, script files and images. This takes into account all content on the page, even content that is not visible. As a result, content above the fold, defined as content that is inside the user's viewport, may have finished loading long before the onLoad event fires. To account for this,

[49] https://w3techs.com/technologies/comparison/js-angularjs,js-react.

[50] https://hacks.mozilla.org/2015/11/better-than-gzip-compression-with-brotli.

[51] https://engineering.linkedin.com/blog/2017/05/boosting-site-speed-using-brotli-compression.

[52] https://developers.google.com/speed/webp.

[53] https://blog.angular.io/version-9-of-angular-now-available-project-ivy-has-arrived-23c97b63cfa3.

[54] https://blog.angular.io/version-8-of-angular-smaller-bundles-cli-apis-and-alignment-with-the-ecosystem-af0261112a27.

the *Speed Index* (SI) metric[55] was introduced. The SI is an integral function over the visual loading progress, captured using a video of the page loading process. Thus, the SI is a representation of how quickly the page, yet also how smoothly, the page loads.

Gao et al. (2017) performed a user study comparing these metrics with the human perception. Participants were tasked to watch two side-by-side videos of web pages loading, and then asked which one loaded fastest. The authors found that in only 55% of the cases, the PLT chose the same answer as the participants. The SI performed worse, matching the majority vote of the participants in just 53% of the answers. To combat this inaccuracy, the authors presented an improved version of SI: the *Perceptual Speed Index* (PSI). The PSI uses a different comparison technique between frames; Structural Similarity instead of mean pixel-histogram difference. The result is a metric that is capable of answering the question whether a web page is able to load quickly without any noticeable jitter.

Nonetheless, the PSI performs worst of all. The PSI matched just 43% of the answers. By default both the SI and PSI compute the time till Visual Complete, the point at which no more visual processing is occurring above the fold. By changing this endpoint to onLoad, the authors show that the SI and PSI can achieve results of 81% and 80% respectively, thus outperforming the PLT significantly. A similar user study was performed by Netravali et al. (2018). The paper provides evidence that the PLT and SI metrics are lacking due to missing the ability to detect the state of interactivity of a page: a page may be visually complete while not being functional. The paper finds that the PLT overestimates the time till a page is interactive, while the SI delivers an underestimation. To measure this state of interactivity, Google proposed the Time to Interactive (TTI).[56] The TTI does so by defining a page as interactive when:

- The *First Contentful Paint* (FCP) has occurred. The FCP registers when the first DOM element is registered in the tree of the page.
- Event handlers have been registered for most visual page elements.
- On interacting with the page, it responds within 50 ms.

Similar in goal is the Total Blocking Time (TBT). The TBT is the "sum of all time periods between FCP and Time to Interactive, when task length exceeded 50ms".[57] Finally, in the First Meaningful paint (FMP) metric, Google proposes yet another approach: the time till the primary content of the page is rendered. [58] This is defined as the time at which the biggest change in the layout has occurred.

This case study uses the metrics defined here to measure the current performance of the 30 MHz platform. In addition, we make the case for product-specific metrics, as opposed to general metrics applicable to any web application, for the purpose of optimization.

7.2. Web performance optimization

There are a plenty of ways to improve the performance of web pages. Prior research generally attempts to improve the performance of web pages in general, as opposed to one particular web page (van Riet and Malavolta, 2019). This approach starts with the fundamental building blocks of the modern internet: the protocols. First shown in 2013, the QUIC (Langley et al., 2017) protocol is shown by Wolsing et al. to be more performant than an ageing TCP (Wolsing et al., 2019). However, its real-world usage is low, barely being used outside of Google servers (Wolsing et al., 2019; Rajiullah et al., 2019). In similar fashion, while HTTP/2 may outperform HTTP/1.1 significantly, it delivers 63% of resources, a number that went up just 4 percentage points from 2018 to 2019. Moreover, HTTP/2 prioritization schemes in browsers are still in their infancy, and sometimes less efficient than naive algorithms (Wijnants et al., 2018).

The dependencies being pushed using such prioritization schemes can be discovered using a tool such as WProf (Wang et al., 2013), and its mobile counterpart WProf-M (Nejati and Balasubramanian, 2016). With these tools, we are able to explain the current problems in the page load process, as well as the differences between desktop and mobile devices. On desktop devices, the papers find that the network acts as the bottleneck. On mobile devices, often with much less powerful CPUs, the bottleneck can be found to be the computational part of the page load. Moreover, only 20% of the critical path of resources between desktop and mobile devices is the same. The critical path is the shortest time in which the steps a browser needs to perform to render a web page to screen are executed. Due to variances between devices, some steps may take longer on one device than on the other, thus changing the shortest possible path. As a result, performance optimizations deployed for one set of devices may have an entirely different effect on the other set.

To combat the computational bottleneck of mobile devices, Wang et al. (2016) propose a preprocessing system wherein the critical state of a web page is found and then transferred over to the device, with the rest of the resources being transferred afterwards. In the median case, this reduced the PLT by 60%. This approach requires a customized browser. An iteration of this concept is provided by Netravali and Mickens (2018), who present the Prophecy system. Prophecy does not require a custom browser and achieves a PLT reduction of 53%.

At 30 MHz, the web application is a SPA, which brings its own challenges. Stępniak and Nowak (2017) show the largest improvements to a SPA can be achieved by minimizing the bundle size, the size of the content that is delivered upon requesting the web page. Further of highlight is the impact the removal of CSS rules can have. As shown by the dependency tracing in WProf Wang et al. (2013), CSS parsing blocks any JavaScript from executing at that time. Removing CSS rules thus does not only save on transferred data size, it also reduces the blocking time.

A final important consideration is the effect of optimizations on the user experience. Kelton et al. (2017) found evidence of web pages having areas of "high collective fixation". By improving the timing at which these areas load in, the user perceived performance is shown to increase. In this paper, such thinking is used to develop a product-specific web performance metric, focused on the objects most likely to draw attention. Moreover, as opposed to improving the web performance of many apps, often using one method, this case study focuses on just one application, iteratively improving it over time using a variety of techniques.

8. Conclusion

The work done as part of the case study at 30 MHz is a showcase of the potential value of iterative performance optimization. While further work still is possible, a large performance improvement has been achieved during this case study. The structured approach shown in this case study is able to achieve a 48.25% reduction of the SI on desktop devices and a 19.85% reduction on mobile devices. The FCP was reduced by 98.37% and 97.56% on

55 https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

56 https://web.dev/interactive.

57 https://web.dev/lighthouse-total-blocking-time.

58 4https://developers.google.com/web/tools/lighthouse/audits/first-meaningful-paint.

desktop and mobile devices respectively. For the target audience of professionals in the software development sector, this sums up to a significant amount of time on a weekly basis. Using a user study, the LTTW showed a perfect correlation with the user perceived load time in the context of an optimization process, thus proving the value of such product-specific metrics.

For future work, we aim to explore the impact of both programming languages and frameworks to ensure a good performance by default, without such specialized performance engineering. Additionally, a more in-depth analysis into the critical rendering paths of devices would lessen the difficulty of reasoning about performance optimizations. Further, more work still is needed on providing a truly objective web performance metric, closely correlated to the actual user-perceived performance, particularly when attempting to predict the absolute value of the current load time of a web page. While this paper is able to demonstrate the value of metrics such as the LTTW, more work is needed to both compose new metrics and compare currently available metrics. Finally, more such industrial case studies would allow for even more generalized findings, as common themes among Web performance issues and optimizations could be found.

CRediT authorship contribution statement

Jasper van Riet: Investigation, Data curation, Visualization, Writing – original draft. **Ivano Malavolta:** Conceptualization, Methodology, Investigation, Writing – review & editing, Supervision. **Taher A. Ghaleb:** Investigation, Writing – original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

https://doi.org/10.5281/zenodo.7256902

References

Bocchi, Enrico, De Cicco, Luca, Rossi, Dario, 2016. Measuring the quality of experience of web users. ACM SIGCOMM Comput. Commun. Rev. 46 (4), 8–13.

Bonferroni, Carlo, 1936. Teoria statistica delle classi e calcolo delle probabilita. In: Pubblicazioni Del R Istituto Superiore Di Scienze Economiche E Commericiali Di Firenze, Vol. 8. pp. 3–62.

Cliff, Norman, 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. Psychol. Bull. 114 (3), 494.

Egger, Sebastian, Reichl, Peter, Hoßfeld, Tobias, Schatz, Raimund, 2012. "Time is bandwidth"? Narrowing the gap between subjective time perception and quality of experience. In: 2012 IEEE International Conference on Communications (ICC). IEEE, pp. 1325–1330.

Enghardt, Theresa, Zinner, Thomas, Feldmann, Anja, 2019. Web performance pitfalls. In: International Conference on Passive and Active Network Measurement. Springer, pp. 286–303.

Firtman, Maximiliano, 2018. Hacking Web Performance. O'Reilly Media, Inc., Sebastopol, CA, USA.

Gao, Qingzhu, Dey, Prasenjit, Ahammad, Parvez, 2017. Perceived performance of top retail webpages in the wild: Insights from large-scale crowdsourcing of above-the-fold qoe. In: Proceedings of the Workshop on QoE-Based Analysis and Management of Data Communication Networks. pp. 13–18.

Grissom, Robert J., Kim, John J., 2005. Effect Sizes for Research: A Broad Practical Approach. Lawrence Erlbaum Associates Publishers.

Hofmann, Heike, Wickham, Hadley, Kafadar, Karen, 2017. Value plots: Boxplots for large data. J. Comput. Graph. Statist. 26 (3), 469–477.

Hoßfeld, Tobias, Metzger, Florian, Rossi, Dario, 2018. Speed index: Relating the industrial standard for user perceived web performance to web qoe. In: 2018 Tenth International Conference on Quality of Multimedia Experience (QoMEX). IEEE, pp. 1–6.

Kelton, Conor, Ryoo, Jihoon, Balasubramanian, Aruna, Das, Samir R, 2017. Improving user perceived page load times using gaze. In: 14th {*USENIX*} Symposium on Networked Systems Design and Implementation ({*NSDI*} 17). pp. 545–559.

Langley, Adam, Riddoch, Alistair, Wilk, Alyssa, Vicente, Antonio, Krasic, Charles, Zhang, Dan, Yang, Fan, Kouranov, Fedor, Swett, Ian, Iyengar, Janardhan, et al., 2017. The quic transport protocol: Design and internet-scale deployment. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication. pp. 183–196.

Lauinger, Tobias, Chaabane, Abdelberi, Arshad, Sajjad, Robertson, William, Wilson, Christo, Kirda, Engin, 2017. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web. In: Proceedings of the 24th Annual Network and Distributed System Security Symposium. Internet Society, pp. 1–15.

Li, Deqing, Mei, Honghui, Shen, Yi, Su, Shuang, Zhang, Wenli, Wang, Junting, Zu, Ming, Chen, Wei, 2018. ECharts: a declarative framework for rapid construction of web-based visualization. Vis. Inf. 2 (2), 136–146.

Malavolta, Ivano, 2016. Beyond native apps: web technologies to the rescue!(keynote). In: Proceedings of the 1st International Workshop on Mobile Development. pp. 1–2.

Nejati, Javad, Balasubramanian, Aruna, 2016. An in-depth study of mobile browser performance. In: Proceedings of the 25th International Conference on World Wide Web. pp. 1305–1315.

Netravali, Ravi, Mickens, James, 2018. Prophecy: Accelerating mobile page loads using final-state write logs. In: 15th {*USENIX*} Symposium on Networked Systems Design and Implementation ({*NSDI*} 18). pp. 249–266.

Netravali, Ravi, Nathan, Vikram, Mickens, James, Balakrishnan, Hari, 2018. Vesper: Measuring time-to-interactivity for modern web pages. In: Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation, NSDI. USENIX Association.

Newson, Roger, 2002. Parameters behind "nonparametric" statistics: Kendall's tau, Somers' D and median differences. Stata J. 2 (1), 45–64.

Rajiullah, Mohammad, Lutu, Andra, Khatouni, Ali Safari, Fida, Mah-Rukh, Mellia, Marco, Brunstrom, Anna, Alay, Ozgu, Alfredsson, Stefan, Mancuso, Vincenzo, 2019. Web experience in mobile networks: Lessons from two million page visits. In: The World Wide Web Conference. pp. 1532–1543.

Selakovic, Marija, Pradel, Michael, 2016. Performance issues and optimizations in javascript: an empirical study. In: Proceedings of the 38th International Conference on Software Engineering. pp. 61–72.

Sivakumar, Ashiwan, Gopalakrishnan, Vijay, Lee, Seungjoon, Rao, Sanjay, Sen, Subhabrata, Spatscheck, Oliver, 2014. Cloud is not a silver bullet: A case study of cloud-based mobile browsing. In: Proceedings of the 15th Workshop on Mobile Computing Systems and Applications. pp. 1–6.

Stępniak, Wojciech, Nowak, Ziemowit, 2017. Performance analysis of SPA web systems. In: Information Systems Architecture and Technology: Proceedings of 37th International Conference on Information Systems Architecture and Technology–ISAT 2016–Part I. Springer, pp. 235–247.

van Riet, Jasper, Malavolta, Ivano, 2019. Client-side performance of web-based applications: the state of the art. [Online; accessed 10. Apr. 2020] https://jaspervanriet.nl/assets/literature_study.pdf.

van Riet, Jasper, Paganelli, Flavia, Malavolta, Ivano, 2020. From 6.2 to 0.15 seconds–an industrial case study on mobile web performance. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 746–755.

Wagner, Jeremy, 2016. Web Performance in Action: Building Fast Web Pages. Simon and Schuster.

Wang, Xiao Sophia, Balasubramanian, Aruna, Krishnamurthy, Arvind, Wetherall, David, 2013. Demystifying page load performance with WProf. In: 10th {*USENIX*} Symposium on Networked Systems Design and Implementation ({*NSDI*} 13). pp. 473–485.

Wang, Xiao Sophia, Krishnamurthy, Arvind, Wetherall, David, 2016. Speeding up web page loads with shandian. In: 13th {*USENIX*} Symposium on Networked Systems Design and Implementation ({*NSDI*} 16). pp. 109–122.

Wijnants, Maarten, Marx, Robin, Quax, Peter, Lamotte, Wim, 2018. HTTP/2 prioritization and its impact on web performance. In: Proceedings of the 2018 World Wide Web Conference. pp. 1755–1764.

Wohlin, Claes, Runeson, Per, Höst, Martin, Ohlsson, Magnus C, Regnell, Björn, Wesslén, Anders, 2012. Experimentation in software engineering. Springer Science & Business Media.

Wolsing, Konrad, Rüth, Jan, Wehrle, Klaus, Hohlfeld, Oliver, 2019. A performance perspective on web optimized protocol stacks: Tcp+ tls+ http/2 vs. quic. In: Proceedings of the Applied Networking Research Workshop. pp. 1–7.

Jasper van Riet Senior Mobile Developer at Team Liquid, Utrecht, The Netherlands. In 2020 he received a Master degree in Computer Science (with specialization on Computer Systems Security) from the Vrije Universiteit Amsterdam, The Netherlands. His research interests include software engineering and mobile apps development.

Ivano Malavolta Associate professor and Director of the Network Institute at the Vrije Universiteit Amsterdam (The Netherlands). His research focuses on data-driven software engineering, with a special emphasis on software architecture, mobile software development, robotics software. He applies empirical methods to assess practices and trends in the field of software engineering. He authored several scientific articles in international journals and peer-reviewed international conferences proceedings. He is program committee member and reviewer of international conferences and journals in the software engineering field. He received a PhD in computer science from the University of L'Aquila in 2012. He is a member of ACM, IEEE, VERSEN, and Amsterdam Data Science.

Taher A. Ghaleb Postdoctoral research fellow at the School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Ontario, K1N 6N5, Canada. Some of his research interests include continuous integration, software testing, and mining software repositories. Ghaleb received his Ph.D. in computing from Queen's University, Canada.