From 6.2 to 0.15 seconds – an Industrial Case Study on Mobile Web Performance

Jasper van Riet
Vrije Universiteit Amsterdam
The Netherlands
j.c.a.van.riet@student.vu.nl

Flavia Paganelli
30MHz
The Netherlands
flavia@30mhz.com

Ivano Malavolta
Vrije Universiteit Amsterdam
The Netherlands
i.malavolta@.vu.nl

***Abstract*—*Background*. A fast loading web app can be a key success factor in terms of user experience. However, improving the performance of a web app is not trivial and requires a deep understanding of both the browser engine and the specific usage scenarios of the web app under consideration.**
***Aims*. This paper presents an industrial case study targeting a large web-based dashboard, where its performance was improved via 13 distinct interventions over a four-month period.**
***Method*. Firstly, we design a replicable performance engineering plan, where the technical realization of each intervention is reported in details together with its development effort. Secondly, we develop a benchmarking tool which supports 11 widely-used web performance metrics. Finally, we use the benchmarking tool to quantitatively evaluate the performance of the target web app.**
***Results*. We observe a considerable performance improvement over the course of the 13 interventions. Among others, we achieve a 97.56% reduction of the time for the First Contentful Paint (*i.e.,* from 6.29 to 0.15 seconds) and a 19.85% improvement of the Speed Index metric (*i.e.,* from 15.31 to 12.27 seconds).**
***Conclusions*. This case study shows the value of a continuous focus on performance engineering in the context of large-scale web apps. Moreover, we recommend developers to carefully plan their performance engineering activities since different interventions require different efforts and can have very different effects on the overall performance of the system.**

***Index Terms*—Performance, Web Apps, Industrial Case Study.**

I. Introduction

By 2025, 1.2 billion *more* people will be using mobile internet[1]. One of the key enablers of this massive trend is the standardization and the compatibility of Web technologies, primarily lead by the efforts of foundations like W3C and several companies. Indeed, with standard APIs for geolocation, push notifications, accessing the camera, etc., the Web is becoming a fully-fledged software platform and capable of providing richer experiences to end users [3]. The *performance* of a mobile web app is vital towards its success, specially on mobile devices where hardware and connectivity are constrained. For instance, the BBC lost an additional 10% of users for every additional second their web app took to load[2].

However, even though the value of fast web apps is without doubts, improving the performance of mobile web apps is still a very demanding engineering effort. 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) [9]. Assessing how state-of-the-art performance techniques are applied in real industrial projects is fundamental for better understanding their characteristics, gains, and the effort required for their concrete application.

This paper presents an industrial case study performed at 30MHz, a technology company in the agricultural sector. 30MHz is a scale-up of 35 employees that service over 300 customers across 30 countries. The main product of 30MHz is a web dashboard for providing growers insights 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 30MHz.

The case study is composed of three main phases. In the first phase we design a **performance engineering plan** (PEP) for the 30MHz dashboard, where the technical realization of 13 interventions is reported in detail and with replicability in mind. Secondly, 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[3], an open-source audit tool widely used both in academia and industry. Thirdly, over a period of 4 months we (i) implement each intervention of the performance engineering plan and (ii) apply the benchmarking tool for a **quantitative evaluation** of the performance of the 30MHz dashboard according to the 11 supported metrics.

Thanks to the incremental interventions of the PEP, we achieved considerable performance improvements across many performance metrics. Moreover, we provide an estimation of the engineering effort required for implementing each intervention of the PEP in the context of the 30MHz dashboard.

The **main contributions** of this study are: (i) the description of a detailed *performance engineering plan* for web apps; (ii) a *quantitative evaluation* of the impact of each intervention of the plan on the performance of the 30MHz web dashboard; (iii) a *discussion* of the obtained results and their implications; (v) a *replication package* of the study containing its results, raw data, and data analysis scripts.

[1]https://www.gsma.com/mobileeconomy
[2]https://link.medium.com/OXxPF5BqN6
[3]https://developers.google.com/web/tools/lighthouse

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 techniques have on the overall performance of a real industrial product, and hence help practitioners in better reasoning about which techniques can be applied in their own projects. Finally, our results provide objective evidence about the value of *continuous performance engineering* to product owners.

II. Context: the 30MHz Platform

The core product of 30MHz is 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. Furthermore, 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 app, 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 app 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 app is based on the Angular 7 framework, it is hosted on Amazon AWS[4] infrastructure and uses Amazon's CloudFront[5] service as edge nodes.

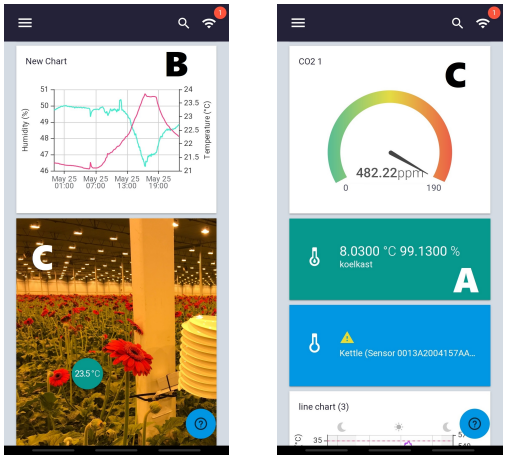

Fig. 1: Examples of widgets in the 30MHz dashboard.

In concertation with the product owner of the 30MHz web app, we decided to focus the case study on the *interactive dashboard* of the web app in order to (i) keep our efforts within reasonable limits and with measurable outcomes and, more importantly, (ii) direct our attention to the most important and frequently accessed component of the whole platform. Indeed, the dashboard is the first screen of the web app accessed by end users and provides vital information about the state of plants and flowers, shaping decisions made on a daily basis.

As shown in Figure 1, the 30MHz 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 30MHz dashboard supports four types of widgets:

- *Single value widgets* (item A in Figure 1) 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).

In this paper we focus on the performance of the 30MHz dashboard *on mobile devices* since (i) about 30% of all 30MHz users access their dashboards via a mobile device, and this number is expected to grow in the future, (ii) it enables us to keep the study well scoped, and (iii) it enables us to build a controllable measurement infrastructure (see Section III-C).

III. Study Design

A. Goal and Research Question

We formulate the **goal** of the case by following the Goal-Question-Metric approach [13], specifically: "Evaluate *the execution of a performance engineering plan* for the purpose of *characterising its impact* on the *performance* as seen from the viewpoint of the *web developers and researchers*, in the context of the *30MHz dashboard*".

The goal leads to the following **research question**: *What are the activities, improvements, and technical challenges faced in practice when engineering the performance of a mobile web app?* Differently from other works in the literature, this case study aims to analyse and iteratively improve the web performance of a mobile web app, as opposed to finding and resolving a single bottleneck. This requires a benchmarking mechanism to measure the actual impact of each improvement, as well as repeated analyses of new releases. By answering this question, web developers will be provided with objective data on how a web app can be improved by applying different types of performance-oriented interventions. Moreover, by answering this question we provide a comprehensive overview of the challenges faced in practice when aiming to optimise the performance of a mobile web app.

B. Variables Selection and Experiment Plan

The unit of analysis of this study is the 30MHz dashboard (see Section II) and the various interventions of the PEP

[4]https://aws.amazon.com
[5]https://aws.amazon.com/cloudfront

are directly applied to it in sequence (see Section IV-B). Accordingly, the **independent variable** of this study represent the intervention applied to the 30MHz dashboard; the levels of this variable are 13 and correspond to each single intervention of the PEP. For the sake of space, interventions are described contextually to our results in Section IV.

The **dependent variables** are shown in Table I and correspond to the 11 web performance metrics we will collect on the 30MHz dashboard. Those metrics serve as the objective source of truth for evaluating the impact of each intervention on the performance of the 30MHz dashboard. The choice of those 11 metrics are based on both academic consensus and industry standards [9] (see also Section VII). It is important to note that metrics chosen complement each other and focus on different aspects of the web app, specifically: Rendering (R), Computation (C), Networking (N), and General (G). This decision allows us to carry out a comprehensive assessment of the performance of the 30MHz dashboard from different complementary perspectives.

TABLE I: Performance metrics considered in this study

Name	ID	Focus	Description
First Contentful Paint	FCP	R	Time (ms) till the first DOM element is rendered
First Meaningful Paint	FMP	R	Time (ms) till the largest DOM element is rendered
Speed Index	SI	G	Integral of visual loading progress[6]
Total Blocking Time	TBT	C	Time (ms) during which the web app is blocked from interaction
Estimated Input Latency	EIL	C	Time (ms) needed to respond to interaction during load
First CPU Idle	FCI	C	Time (ms) till the web app is ready to accept any interaction
Time to Interactive	TTI	C	Time (ms) till the web app is fully responsive to interaction
Fetch Requests	FR	N	Number of requests to fetch (remote) resources
DOM Size	DS	R	Number of DOM elements
Lowest Time to Widget	LTTW	G	Time (ms) till the first widget is rendered
Median Time to Widget	MTTW	G	Median of all visible Time to Widget values (ms)

The **plan** of the experiment is organized as follows. Firstly, we consider the initial version V_0 of the 30MHz dashboard and measure it to collect the values of the 11 performance metrics. Then, we sequentially apply each intervention I_x according to the PEP (where $1 \leq x \geq 13$). For each I_x we (i) keep a record of the technical activities performed during I_x, together with their corresponding effort in man/hours and subsequently (ii) measure the 30MHz dashboard again. In this way, we obtain the values of all the performance metrics both *before* and *after* the application of each single intervention; this enables us to precisely reconstruct the impact of each single I_x on the overall performance of the 30MHz dashboard.

C. The Benchmarking Tool and Measurement Infrastructure

Our benchmarking tool is implemented as a JavaScript module. Given a web app, the tool automatically runs it for a number of times and collects the metrics shown in Table I. In order to take into account the intrinsic fluctuations when dealing with time-based metrics like FCP and FMP, the tool repeats its measurement for 30 independent runs and we considered all the collected measures in our statistical analysis.

The benchmarking tool orchestrates the runs of the 30MHz dashboard via the Android Debug Bridge (ADB[7]) along with the remote debugging API provided by Google Chrome[8].

In this study, the benchmarking tool runs on a laptop connected to a mobile device via USB. At every run, the value of each performance metric is collected via the programmatic interface of the Google Lighthouse audit tool [9](version 5), along with the User Timing standard API[10]. We decided to use the Google Lighthouse tool because it is actively maintained, open-source (thus facilitating the replicability of our study), extensible, and can be easily integrated into a third-party software pipeline, like our benchmarking tool.

In this study we use a Samsung Galaxy S10E running Android 10 as operating system. The device is equipped with a 2.7 GHz Octa-Core processor, 6Gb of RAM, and an Adreno 640 GPU (Exynos system on chip). To make sure background processes are not interfering with the measures, the device is restarted before each run. All runs are executed on the Google Chrome Android app (version 80) installed on the device; the app does not have any extensions installed and is reset at every run. The device communicates with the 30MHz dashboard via a Wi-fi network; to ensure that the Wi-Fi conditions do not alter our results, the device is always placed within the same network throughout the whole case study duration. In order to maintain a realistic and more representative environment in our runs, we use the *throttling* function of Google Lighthouse, which emulates a slow 4G network speed and slows down the device CPU by a factor of four.

Finally, to ensure consistency, the 30MHz dashbord always shows the same widgets and the same data for all the runs. The 30MHz dashboard is served via the 30MHz release staging instance, which uses the same configuration options as the production instance (including CloudFront edge nodes).

D. Data analysis

Exploration. We give an *overview* of all the measures collected during the whole case study through a set of boxen plots. The idea is to provide 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.
Statistical analysis. *For each single intervention*, we (i) report the delta of the median value of each performance metric before and after the intervention (the median is preferred over other measures of central tendency to minimise the impact of outliers) and (ii) check the statistical significance

[6]https://web.dev/speed-index

[7]https://developer.android.com/studio/command-line/adb
[8]https://developers.google.com/web/tools/chrome-devtools
[9]https://github.com/GoogleChrome/lighthouse
[10]https://www.w3.org/TR/user-timing-3/

of the performance improvement *for each metric* via the non-parametric Mann-Whitney U test ($\alpha = 0.05$). In order to avoid bias, we perform the Mann-Whitney U test as a two-sided test, *i.e.,* we make no assumptions about whether the intervention actually improves or deteriorates the performance of the 30MHz dashboard; the null hypothesis of our Mann-Whitney U tests states that the performance metrics before and after the intervention belong to the same population (*i.e.,* they have the same median). We chose the Mann-Whitney U test because it does not make any assumptions about the distributions being compared. Since we are applying the statistical test multiple times (*i.e.,* once for each metric-intervention pair), in order to reduce the chances of Type-I errors we correct the obtained p-values via the Benjamini/Hochberg procedure [8].

E. Study Replicability

In order to foster independent verification and replication of this case study, a full replication package is publicly available[11]. 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.

IV. Case Study Results

A. Familiarization with the product and First Measurements

The first step of the case study consisted in having one of the academic researchers join the 30MHz team in order to (i) have access and get familiar with the code base of the 30MHz dashboard, (ii) identify the performance metrics of interest for the team (including the development of a first prototype of the benchmarking tool shown in Section III-C), 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 30MHz dashboard.

A co-product of this phase was the identification of a set of performance issues of the 30MHz 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[12]. 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.

In order to keep the obtained results comparable and replicable, in each run of the benchmark the backend always provides 20 distinct data sources to the 30MHz dashboard. As shown in the first column of Figure 2, the performance of the baseline version of the dashboard was a problem. For example, the median of the FCP metric was 6.2s, where the median for real web apps reported by the HTTP Archive[13] is 5.7s. Prior to any intervention, the 30MHz web app is an Angular 7 SPA hosted on AWS and using CloudFront for edge nodes. The web app has a bundle size of 5.91MB. When the web app is loaded, the list of widgets to show to the user is obtained by issuing an HTTP request to the $/\mathrm{dashboard}$ REST endpoint in the backend. Then, each widget is populated via other HTTP requests: single value, gauge, and image widgets make a single HTTP request to the $/\mathrm{stats}$ endpoint, whereas the chart widget makes up to 6 HTTP requests, including one call to the $/\mathrm{stats}$ endpoint. As a result, in the median case, the dashboard makes 153 network requests. In addition to potentially exhausting the maximum concurrent requests enabled by browsers [12], which depending on the browser can be as low as 100, 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 20.23ms as round trip time, which is significantly high.

Once all data has been 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[14], an SVG-based charting framework last updated in August 2017. The runtime impact of NVD3 is significant, with the 30MHz dashboard generally having a frame rate of only 5 to 10 frames per second.

B. Definition of the Performance Engineering Plan

With a clear picture of the internals and potential performance issues of the 30MHz dashboard, it was possible to start reasoning about the interventions for solving them. For example, the web app was making a large number of network requests when dealing with chart widgets, so we thought it would be beneficial to investigate on techniques tailored to reduce such requests (*e.g.,* caching). However, in general the process of judging whether an intervention is worth implementing is difficult. As we explained in Section III-Cs, not all metrics have the same goal, and similarly, not all interventions have the same effect. Moreover, some metrics are not independent, *e.g.,* an improvement in the FCP can also lead to a reduction of the SI because computation is able to start earlier. Interventions thus had to be defined on a case-by-case basis, depending on the specific characteristics of each performance issue to be fixed. Table II shows the PEP resulting from this process; they will be detailed in Section IV-C.

The order of execution of the interventions was primarily due to priorities internally defined within the team and by customer requests. For each single intervention I_x we (i) implemented it in the 30MHz dashboard, (ii) collected the new performance metrics via our benchmarking tool, and (iii) inspected again the 30MHz dashboard to verify that

[11] https://github.com/S2-group/ICSME-2020-replication-package

[12] https://developers.google.com/web/tools/chrome-devtools

[13] https://httparchive.org/reports/loading-speed#fcp

[14] http://nvd3.org

TABLE II: The Performance Engineering Plan (PEP)

ID	Description	Category	Hours
I_1	Prevent expensive computations when they are not necessary	Computation	6
I_2	Minimize the number of reflow operations	UI	8
I_3	Reuse data instead of making redundant requests to remote APIs	Networking	8
I_4	Stop refreshing the UI when the web app becomes inactive	UI	8
I_5	Cache static resources via service workers	Networking	160
I_6	Upgrade the used web framework to its most recent release	Third-party components	30
I_7	Render only the UI elements that are visible above the fold at load time	UI	25
I_8	Adopt alternative libraries optimized for high performance	Third-party components	230
I_9	Defer as much as possible the execution of non-critical JavaScript code	Computation	2
I_{10}	Preload fonts	Networking	2
I_{11}	Remove unused third-party JavaScript code	Third-party components	1
I_{12}	Preconnect to critical third-party origins	Networking	2
I_{13}	Design the interact with remote APIs around the (most recurrent) usage scenarios	Networking	20

the performance issues targeted by I_x disappeared. Not all interventions required the same amount of engineering effort. Interventions could be a simple refactoring (*e.g.,* I_10), wherein no functionality is changed, or a fully-fledged feature improvement, which do even impact the UI of the web app (*e.g.,* I_8). Indeed, a number of interventions did substantially change certain features of the 30MHz dashboard; however, every intervention was always primarily motivated by the goal of achieving a better performance when loading the 30MHz dashboard. An estimation of the effort required to design, implement and verify each intervention is reported in the *hours* column in Table II.

C. Execution of the Interventions

Figure 2 gives an overview of all the data we collected throughout the 4 months of the case study; rows represent performance metrics and columns represent interventions. Each plot shows the distribution of the corresponding performance metric (*e.g.,* FCP) after an intervention (*e.g.,* I_1) during the execution of the 30 runs of our benchmarking tool.

Across the board, major improvements are visible in various performance metrics. While the FCP and FMP metrics particularly catch the eye, with instant to almost instant rendering of contents on the screen, other metrics have also seen significant improvements (*e.g.,* FR, DOM). Interestingly, some of the metrics did not improve (*i.e.,* TBT, EIL), confirming the facts that (i) web performance engineering is a matter of trade-offs and (ii) considering multiple metrics is fundamental for clearly understanding the impact of single interventions.

In the remainder of this section we go through the whole web engineering process and discuss the technical details and impact of each intervention.

Intervention $\mathbf{I_1}$ – A number of elements are drawn into each chart in the 30MHz dashboard. One of these is a notification line: a line drawn into 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 was needed to make sure notification lines fit within the range of the chart. After tracing the app with the devtools performance view, this calculation proved to be computationally expensive and it was executed in all cases, even when no notification line was present. With this intervention, the 30MHz dashboard executes the computation of the notification line only when it is set by the user. Among others, this intervention lead to a reduction of the TBT, FCI, TTI, and SI metrics by 12.92%, 7.80% and 7.44%, and 6.14%, respectively.

Intervention $\mathbf{I_2}$ – When the browser window is resized, the web app resizes its layout accordingly. Generally, this allows the web app to adjust the number of widgets on screen, along with their width and height, to the current viewport. However, there was one problematic edge case for this behaviour. By default, the Chrome engine hides the URL bar when the user scrolls down the page and shows it again when the user scrolls in the other direction. As a result, every time the user changes scrolling direction, a complete *reflow* operation is performed by the browser engine[15], triggering the recalculation of the geometries of all widgets. With this intervention the layout of the 30MHz dashboard is recomputed only when the *width* of the window has changed, and thus no longer triggering on every scroll. While this intervention focused primarily on runtime performance, we observed a reduction of 4.40% and 3.65% of the FCP and SI metrics.

Intervention $\mathbf{I_3}$ – In the 30MHz dashboard, users can place comments on chart widgets. To look up the names of users able to comment, chart widgets make an additional network call to the $/\text{organization}/\text{users}$ endpoint. This call was duplicated among *all* chart widgets. This intervention caches the results of the first call to the users endpoint so that subsequent callers use the locally available data, instead of making new network requests. As a result, this API call is now made once per dashboard, instead of once per widget. This particular network call may have been acting as a blocker for the primary content of the web app. Indeed, a 3.53% reduction is visible in the TBT metric. Moreover, the FMP sees a reduction by 2.32%.

Intervention $\mathbf{I_4}$ – This intervention was not conceived for improving the performance of the initial load of the web app. Instead, the change was meant to increase the responsiveness when users switch back to the tab running the 30MHz dashboard. The widgets in the 30MHz dashboard update themselves at a set interval, sometimes as often as every minute. When the dashboard runs in 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. With this intervention we ensured that tabbing away from the 30MHz dashboard suspended the updates, and then switching back to its tab forced an update. This intervention adds computational logic, thus complexity, and as a result

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

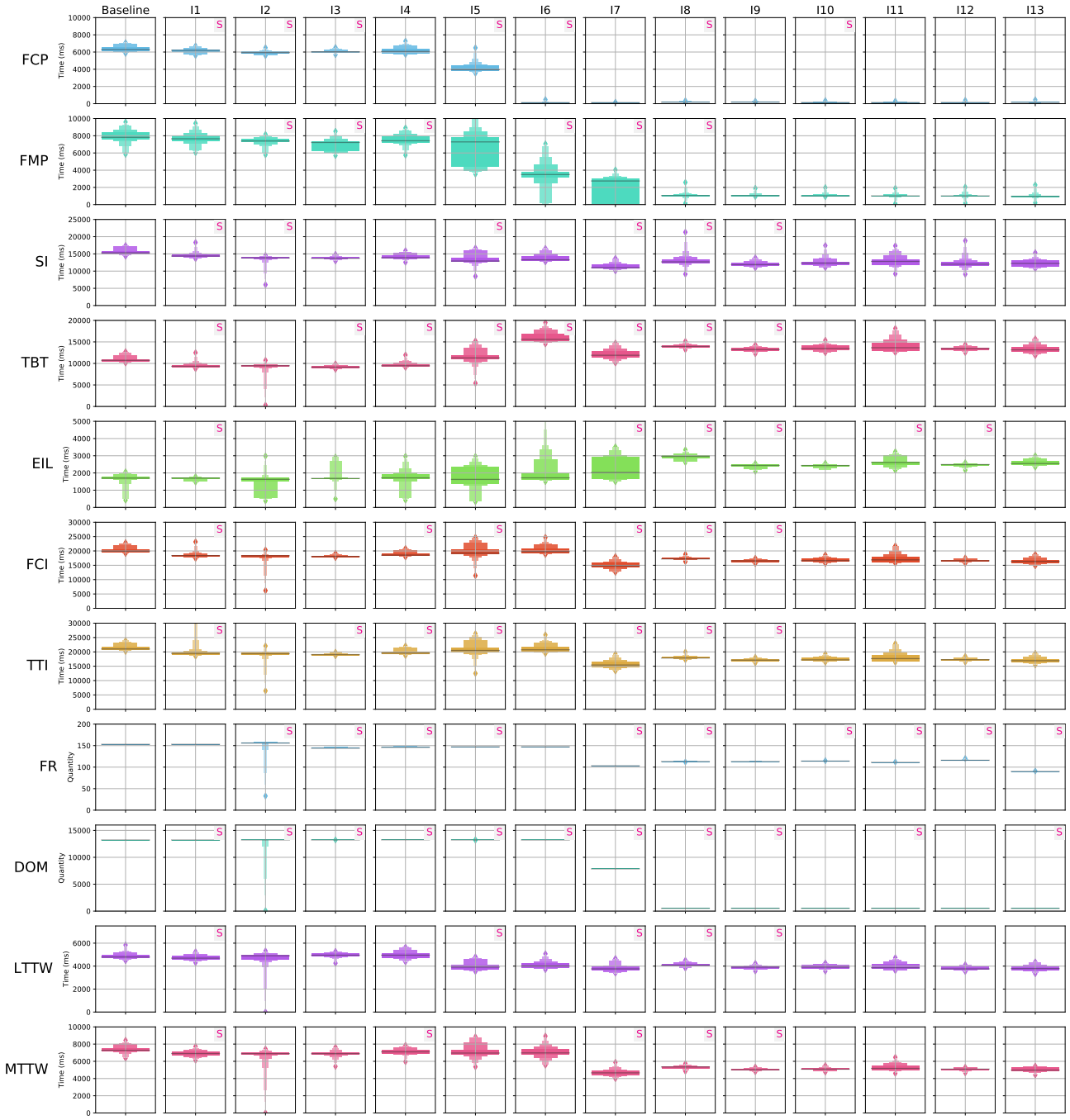

Fig. 2: Overview of the performance metrics across all interventions (S = statistically significant change after the intervention)

computationally-oriented metrics suffered. For example, not only is the the TBT increased by 3.60%, but also the FMP, FCI, TTI, and MTTW increased by more than 2%.

Intervention I_5 – This intervention involved a significant engineering effort (160 hours – see Table II) and involved the usage of *service workers*. In this context, service workers offer significant opportunities for performance improvement due to their caching capabilities. Among other features, service workers allow the web app to locally store a copy of its static resources (*e.g.,* HTML, CSS, and JavaScript files, images, etc.) on the mobile device, instead of fetching them from the network. Significant performance increases are visible across almost all metrics, yet most notable in the results is the 34.43% decrease of FCP, which is equivalent to about 2 seconds. Notably, we also observe a 20.32% and 6.68% reduction of the LTTW and SI metrics. It is important to note that the FR metric remains stable because it is related to the *fetch* operation, which can either hit the cache or the network; even though studying in detail the networking activity of the 30MHz dashboard is within the scope for this study, after intervention

I_5 we manually inspected some of the runs and we noticed that indeed the cache was playing a remarkable role in I_5.

Intervention $\mathbf{I_6}$ – In this intervention we updated the web app to Angular 8. Due to dependency conflicts and the need to support Internet Explorer 11 on desktop browsers, in this intervention we did not use differential loading nor the Ivy compiler, while no other performance changes affecting the 30MHz dashboard were found in the changelog. Nonetheless, very large performance differences were achieved, particularly in the FCP and FMP metrics, which had a 96.85% and 51.78% reduction, respectively. We do not have hard evidence about which upgrade in the Angular 8 framework caused these differences. Nevertheless, performance is a strong concern also for framework maintainers, so we suggest developers to always keep their web frameworks updated to the latest versions, so to take advantage of the performance improvements carried out by framework maintainers.

Intervention $\mathbf{I_7}$ – The NVD3-based charting framework is computationally heavy. Ideally, the number of charts rendered at load time should be as low as possible. The 30MHz dashboard was always loading 20 widgets, even if the user could see only a subset of them while loading the web app. Indeed, chart widgets tend to be large, with at most 3 of them fitting above the fold on most mobile devices. There are a number of techniques to calculate which widgets are visible in a given viewport. The most basic technique 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. Unfortunately, this technique is not applicable to the 30MHz dashboard since image widgets have variable heights, depending on the shown image. Instead, we implemented an ad-hoc scoring algorithm which enumerates the widgets to be shown and renders only up to 4 chart widgets, independently of the other widgets of the dashboard. The scoring system fills the viewport in all cases and dramatically reduces the number of chart widgets being rendered. This is also reflected in the metrics directly related to the number of elements being rendered and the widgets; indeed, the metrics with the most considerable improvement are related to the number of elements in the DOM (DOM, -40.62%) and the mean time to widget (MTTW, -33.22%). Amongst the others, this intervention leads to a 23.75% reduction of the TBT metric, 25.18% reduction of the FCI metric, and a 25.61% reduction of the TTI metric. Interestingly, the EIL metric is worse since the scoring algorithm required a small amount of upfront computational time, eventually leading to an additional latency with respect to the interactivity of the web app. In summary, this intervention shows that the user experience of a web app can be improved by reasoning on product-specific characteristics and on the physical devices on which the web app will run.

Intervention $\mathbf{I_8}$ – Together with product owners, it was decided to change the framework for rendering charts: we went from NVD3 to Apache ECharts[16]. ECharts [2] is a charting framework with a significant focus on high performance, and this became immediately apparent in the 30MHz dashboard. Indeed, we measure the frames frequency of the web app *while interacting with charts* and we could observe that *before* I_8 the web app was running at less than 10 frames per second, whereas *after* the intervention it was running at 50 frames per second. This change required a large amount of engineering work, adding and updating a large number of features, and is thus not solely a refactor. As an indication, bundle size increased by 200 KB. Overall, when looking at the 30MHz dashboard at load time, we noticed a general worsening of its performance. Only the FMP (-61.73%) and DOM (93.44%) metrics had an improvement. The FMP metric improved because NVD3 was doing a large amount of computation during the earlier phases of the page load, the DOM tree was constantly being updated, along with a large number of reflow operations (further updating the massive DOM with more than 13k nodes). The positive impact on the DOM metric is due to the fact that ECharts uses the canvas HTML element to render charts, as opposed to NVD3's SVG. Overall, this intervention can best be considered 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. This intervention was received very positively within 30MHz due to how much it affected one of the core features of the 30MHz platform: the charts; the results were immediately noticeable.

Intervention $\mathbf{I_9}$ – 30MHz uses a third-party platform to provide customer support features. This comes in the form of a button that is loaded onto the page via third-party JavaScript code. This JavaScript code was loaded directly in the $<\mathrm{head}>$ of the HTML page. When loaded in the $<\mathrm{head}>$, JavaScript code act as blockers for further parsing of the DOM [10]. The script was not necessary during the page load, thus we decided to move it to the end of the $<\mathrm{body}>$ section, where it would be parsed *after* the HTML of the page had been rendered. This intuition proved successful. Notably, the SI is reduced by almost a second and the EIL metric had a -17.77% reduction.

Intervention $\mathbf{I_{10}}$ – An analysis of the page render process found one very specific blocker: a font required on average 150ms to be retrieved for every page load. We solved this bottleneck by loading this font earlier on in the page lifecycle using the $\mathrm{preload}$ keyword [17]. Overall, this intervention did not lead to strong performance improvement. We observe that most of the metrics remain stable, whereas we see that both FCP and FMP metrics have been impacted positively.

Intervention $\mathbf{I_{11}}$ – In this intervention we removed the JavaScript code of Hotjar, a third-party analytics platform for users profiling. This was discussed with product owners after noticing particularly large computation times being required by such component. This intervention did not impact the performance of the 30MHz dashboard, neither positively nor negatively. Aside from a performance reduction in EIL (-7.28%), no other metric had a statistically significant change. This may

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

[17]https://www.w3.org/TR/preload

be an indication that the execution of the Hotjar JavaScript code was not in the critical path of the 30MHz dashboard.

Intervention $\mathbf{I_{12}}$ – Pages connecting to third-party origins during the page load can opt to use the preconnect[18] keyword to have the browser open a connection to these origins as soon as possible. The 30MHz platform connects to a number of third-party origins for analytics and icons, thus, in a similar vein to I_{11}, these origins were preconnected to. This could not be done for all third-party JavaScript components since preconnections use considerable CPU time per connection, meaning that the benefits gained from the preconnections may be nullified if an opened connection is not used soon enough within the page load process. Similarly to $I_1 1$, the performance impact of this intervention is minimal, with only the EIL metric having a 3.78% reduction.

Intervention $\mathbf{I_{13}}$ – 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 since comments can be overlaid to the chart only when it is rendered. It is thus expected to improve both computational and general metrics. However, also in this case we did not observe relevant performance changes in the web app.

V. Wrap up and Lessons Learned

This case study shows the value of performance engineering work in the field of web development. Through careful iterative improvement, we achieved significant performance improvements. Indeed. if we consider the 30MHz dashboard at the baseline and after I_{13}, its overall performance has improved considerably. As shown in Table III, we observe improvements in most metrics, yet the most notable gains are made in the category of the metrics related to rendering: the FCP went from more than 6 seconds to 0.15 seconds and the FMP went from more than 8 seconds down to about 1 second. Also the number of DOM nodes was reduced drastically and we know that this was due to the usage of the canvas element for rendering charts instead of SVG in intervention I_8.

TABLE III: Final results of the case study

Metric	Baseline	After $\mathbf{I_{13}}$	Δ %	p-value
FCP	6,290.52	153.33	-97.56%	2.59e-11
FMP	7,646.05	953.33	-75.01%	2.59e-11
SI	15,310.98	12,271.07	-19.85%	1.16e-10
TBT	10,708.86	13,180.75	23.08%	1.42e-10
EIL	1,701.13	2,549.60	49.88%	2.59e-11
FCI	19,928.40	16,344.58	-17.98%	2.59e-11
TTI	21,018.35	16,958.08	-19.32%	2.59e-11
NR	153	153	-41.18%	1.97e-12
DOM	13,219	507	-96.16%	1.97e-12
LTTW	4,796.3	3,798.72	-20.80%	2.59e-11
MTTW	7,280.71	5,006.26	-31.23%	2.59e-11

Interestingly, we observe two metrics which are impacted negatively by our interventions: TBT (23.08%) and EIL (49.88%). TBT (*i.e.,* the Total Blocking Time) is the time while the main thread of the web app is blocked from user interaction and we can notice that it worsened after interactions I_5, I_6, and I_8. In order to understand this phenomenon we need to see it in relation with the FCP. Indeed, TBT is technically defined as the difference between the FCP and the TTI. In our case study, the TTI remained quite stable over across interactions, whereas the FCP was strongly improved exactly in interventions I_5, I_6, and I_8 (less visible in Figure 2 due to the low values in the scale of the plot). When looking at EIL, it is defined as the time needed to react to user input while loading and it is basically tailored at ensuring a smooth user experience at load time. We can notice that it was most impacted by interventions I_7 (computing visible widgets and rendering only them) and I_8 (usage of the EChart library). We conjecture that in both interventions we paid the price of executing more JavaScript code at load time in order to achieve a smoother experience *after* the page is fully loaded. Overall, the lessons learned for both TBT and EIL are: (i) web performance engineering is a process where **design decisions must be taken with the overall web app in mind and with an openness towards trade-offs**, and (ii) measuring **multiple performance metrics is fundamental** since one single 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. Without the both of these, developers may scare away from making 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.

We quantitatively assess the *magnitude* of the performance improvement of each metric by calculating the Cliff's Delta effect size measure between its baseline and the I_{13} intervention. We obtained a *large* effect size for each single performance metric considered in this study. As shown in Figure 2, often performance improvements were not immediately visible from one intervention to the next. This phenomenon is statistically confirmed by the effect size measures obtained between each pair of interventions, which mostly ranged between *negligible* and *small* effect sizes. For the sake of space we do not report the values of the effect size measures in this paper, they are fully available in our replication package. In summary, the obtained performance improvements are often not attributable to a single intervention. Instead, the overall performance improvement is the result of a combination of interventions over time. For developers, this highlights the **need for continuous focus on performance**, as opposed to dedicated sprints specifically-dedicated for performance. This finding is also informally confirmed by professionals in the field[19]. With this case study we contribute with objective evidence of this phenomenon.

During the case study we also experienced a strong re-

[18]https://www.w3.org/TR/resource-hints/#preconnect

[19]http://dev.to/ben/addy-osmanis-18-point-web-performance-checklist-2e1

duction of the variability of the collected measures as the performance of the 30MHz dashboard was improving. At the beginning of our 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 analyzing 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 in order 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. In order to streamline the statistical analysis, it can be embedded into a dedicated Continuous Integration (CI) pipeline; in this way the CI pipeline (i) masks the complexity of the statistical analysis to the development team and (ii) shows only the metrics requiring special attention and the candidate points of intervention in the targeted web app.

Finally, while some of these interventions are specific to the 30MHz platform, many can be applied generally. Optimisations based on techniques such as preloading and minimising complex calculations can easily be abstracted and applied to other web apps. Specific technology-specific interventions, such as I_8 (*i.e.,* the migration to ECharts), are harder to apply in a generalised case, aside from acting as an indication of the improvements that can be accomplished by paying attention to the performance impact of external dependencies.

VI. Threats to validity

External validity. The subject of this case study relies on the Angular web framework. The specific opinionated methodology of Angular influenced the engineering work performed in our study. Nonetheless, 12 out of 13 interventions are not Angular-specific; this makes us reasonably confident that the results of this case study can be generalized to other web apps and projects. Moreover, while some of the activities done at 30MHz have a more specialised nature (*e.g.,* not all web apps have a strong focus on charts like the 30MHz dashboard), the majority of the reported interventions can be mapped to alternative and more general scenarios.

Internal validity. The measures collected during the case study may have been affected by external factors. For example, browsers have received updates, network conditions might have changed, or external programs may have been interfering. In order to minimise those risks and to report consistent measures, we repeated all the measurements at the end of the 4-months period and we reported them in Section V. Also, we set up a dedicated benchmarking tool and make it publicly available in our replication package; the benchmarking tool can be used for replicating this study on different subjects and under different conditions. In order to reduce the influence of anomalous runs, the measurement of the 30MHz dashboard is repeated for 30 independent runs and we considered all the collected measures in our statistical analysis. Finally, the correctness of all interventions has been checked via several rounds of code reviews performed by the team members in 30MHz; as a further confirmation, all interventions of this case study are currently deployed in production at 30MHz.

Construct validity. We avoided ambiguities in the design of the case study by defining a *a priori* all its details like used metrics, measurement tool, interventions, etc. We mitigated the mono-method and mono-operation biases by collecting multiple metrics from the 30MHz dashboard and performing 13 distinct interventions, respectively. We are reasonably confident about the reliability of the implementation of the interventions because their source code was developed collaboratively by the team responsible for the 30MHz dashboard and underwent several code reviews. A number of wanted interventions were unable to succeed, while these would have likely improved the 30MHz dashboard. Brotli is a more optimised compression algorithm than gzip in most cases, but we could not use it due to limitations within Amazon's CloudFront service. Similar limitations prevented this case study from using the WebP image compression (OD3), which are more efficient than the PNG and JPEG algorithms. Finally, legacy support prevented using the Ivy compiler for Angular and differential loading. Our benchmarking tool is based on Google Lighthouse, a widely-used tool for the purpose of measuring the performance of web pages. All metrics used as part of this case study are commonly used in both industry and academia, with the exception of Time to Widget, due to its focus on the widgets of the 30MHz dashboard.

Conclusion validity. To minimise the chances of drawing incorrect conclusions from the case study, the assumptions of the used statistical tests have been taken into account. To minimize the error rate of the results, all p-values reported in this study have been corrected by following the Benjamini-Hochberg procedure. Moreover, our discussion of the results is based only on those metrics for which the null hypothesis could be rejected. Finally, a complete replication package is publicly avail- able for independent verification and inspection of each reported intervention.

VII. Related Work

In the literature, web performance has been targeted from different angles and at different levels of abstraction. Wolsing *et al.* showed that the QUIC protocol performs better than the TCP protocol [14]. However, its real-world usage is low, with few cases outside of Google servers [14], [6]. In a similar fash-

ion, 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 prioritisation schemes in browsers are still in their infancy, and in some cases less efficient than naive algorithms [12].

The dependencies being pushed using prioritisation schemes can be discovered using tools like WProf [10] and its mobile counterpart WProf-M [4]. With these tools, researchers are able to investigate and measure 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 less powerful CPUs, the bottleneck can be found to be the computational part of the page. Furthermore, only 20% of the critical path of resources between desktop and mobile devices is the same. The critical path is the sequence of steps to convert the HTML, CSS, and JavaScript into rendered pixels in the browser. Due to variances between devices, some steps may take longer on one device than on the other. As a result, performance optimisations 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.* 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 [11]. In the median case, this reduced the PLT by 60%. This approach requires a customised browser. An iteration of this concept is provided by Netravali *et al.*, who present the Prophecy system [5]. Prophecy does not require a custom browser and achieves a PLT reduction of 53%.

At 30MHz, the web app is a SPA, which brings its own challenges. Stępniak *et al.* report that the largest improvements to a SPA can be achieved by minimising the bundle size, *i.e.,* the size of the content that is delivered upon requesting the web page [7]. Removing CSS rules thus does not only save on transferred data size, but it also reduces the blocking time of JavaScript code.

A final important consideration is the effect of optimisations on the user experience. Kelton *et al.* found evidence of web pages having areas of "high collective fixation" [1]. By improving the timing at which these areas load, the user perceived performance is shown to increase. The authors developed a product-specific web performance metric, focused on the objects most likely to draw attention.

While this body of research provides valuable insights on web performance, our study goes a step further by investigating the web performance engineering process in a *concrete industrial context*. Moreover, the scope of our study is completely different since it does not focus on a single technique or metric, but it targets *the whole performance engineering process*, where a full plan composed of several interventions has been executed and evaluated in practice.

Performance Engineering Plan (PEP) and executing it on a real software product. While further work still is possible, a *large* performance improvement has been achieved during this case study for all considered performance metrics. The structured approach shown in this case study is able to achieve a 19.85% reduction of the SI metric on a real mobile device. The FCP was reduced by 97.56% on desktop and mobile devices respectively. For the target audience of professionals in the agricultural sector, this sums up to a significant amount of time on a weekly basis. All interventions in the PEP have been deployed in production and are currently being used by hundreds of customers worldwide.

VIII. Conclusions and Future Work

The work done as part of the case study at 30MHz is a showcase of the potential value of designing a precise

As future work, we plan to (i) expand the measurements and analysis to desktop machines, (ii) deepen our statistical analysis, specially by verifying how performance metrics differ between desktop and mobile devices, and (iii) conduct a user study to understand how quantitative metrics compare to the perceive performance experienced by real users.

References

[1] C. Kelton, J. Ryoo, A. Balasubramanian, and S. R. Das. Improving user perceived page load times using gaze. In *14th {USENIX} Symposium on Networked Systems Design and Implementation*, pages 545–559, 2017.

[2] D. Li, H. Mei, Y. Shen, S. Su, W. Zhang, J. Wang, M. Zu, and W. Chen. Echarts: A declarative framework for rapid construction of web-based visualization. *Visual Informatics*, 2(2):136–146, 2018.

[3] I. Malavolta. Beyond native apps: web technologies to the rescue!(keynote). In *Proceedings of the 1st International Workshop on Mobile Development*, pages 1–2, 2016.

[4] J. Nejati and A. Balasubramanian. An in-depth study of mobile browser performance. In *Proceedings of the 25th International Conference on World Wide Web*, pages 1305–1315. International World Wide Web Conferences Steering Committee, 2016.

[5] R. Netravali and J. Mickens. Prophecy: accelerating mobile page loads using final-state write logs. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pages 249–266, 2018.

[6] M. Rajiullah, A. Lutu, A. S. Khatouni, M.-R. Fida, M. Mellia, A. Brunstrom, O. Alay, S. Alfredsson, and V. Mancuso. Web experience in mobile networks: Lessons from two million page visits. In *The World Wide Web Conference*, pages 1532–1543. ACM, 2019.

[7] W. Stępniak and Z. Nowak. 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*, pages 235–247. Springer, 2017.

[8] D. Thissen, L. Steinberg, and D. Kuang. Quick and easy implementation of the benjamini-hochberg procedure for controlling the false positive rate in multiple comparisons. *Journal of educational and behavioral statistics*, 27(1):77–83, 2002.

[9] J. Wagner. *Web Performance in Action: Building Faster Web Pages*. Manning Publications Co., 2017.

[10] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with wprof. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 473–485, 2013.

[11] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding up web page loads with shandian. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, pages 109–122, 2016.

[12] M. Wijnants, R. Marx, P. Quax, and W. Lamotte. Http/2 prioritization and its impact on web performance. In *Proceedings of the 2018 World Wide Web Conference*, pages 1755–1764. International World Wide Web Conferences Steering Committee, 2018.

[13] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Computer Science. Springer, 2012.

[14] K. Wolsing, J. Rüth, K. Wehrle, and O. Hohlfeld. A performance perspective on web optimized protocol stacks: Tcp+ tls+ http/2 vs. quic. In *Proceedings of the Applied Networking Research Workshop*, pages 1–7, 2019.