Iruka: Accelerated Mobile Pages Through Classifying and Blocking Non-critical JavaScript

Patrick Inshuti Makuba
Computer Science, NYUAD
patrick.inshuti@nyu.edu

Advised by: Yasir Zaki, Moumena Chaqfeh

Reference Format:
Patrick Inshuti Makuba. 2021. Iruka: Accelerated Mobile Pages Through Classifying and Blocking Non-critical JavaScript . In *NYUAD Capstone Seminar Reports, Spring 2021, Abu Dhabi, UAE.* 9 pages.

ABSTRACT

The average page size has been increased from 1720 Kilobytes In 2016 to 3077 Kilobytes in 2021[5], where Javascript is increasingly contributing to the total page size (from 24% in 2016 to 27% in 2021)[2]. In comparison to equivalent-sized web resources, JavaScript is more expensive to process due to the requirement of parsing, interpretation and execution. This cost significantly contributes to the increasing complexity of modern mobile pages, which becomes a critical performance bottleneck for low-end smartphone devices and ultimately leads to lesser user engagement.

Hence, in this paper I propose a tool which uses a machine learning algorithm to classify JavaScripts into different categories from which are used as a basis point to non-critical Javascript, with the aim of reducing the amount of JavaScripts in a fully loaded web-page. The tool is a form of browser plugin that is privacy-preserving, since it offers an independent client-end solution that does not require any external interaction with any third-party service. The proposed plugin is empowered by a neural network model which detects categories of JavaScript elements used in web pages with an accuracy of 89.33%. Evaluation results show that when the plugin is installed on a client's browser, there is a substantial improvement across different evaluation metrics used i.e on low-end mobile phones, there was a 31% reduction in the Speed Index and a 32% reduction in the Page Load Time among others.

1 INTRODUCTION

The web is increasingly becoming much more complex and sophisticated and so is the users' expectation towards the quality of the browsing experience. This creates a dilemma where more and more Javascript is used to provide more complex features to entice users but at the same time leading to an increased Page Load Time which ultimately reduces users retention and engagement levels.

In 2018 a research carried out by Google found that when the page load time (PLT) increases from 1 to 10 seconds, the probability of a user leaving a website increases by 123%, and in the same research it is shown that by average it takes 15 seconds to fully load landing page on a mobile device [1]. Hence, this fully conveys the idea of how important each second is when it comes to page load time(s). Therefore any increased Page Load Time can potentially present a much larger problem than a just a "slow website" to institutions that rely on their online presence to carry out their businesses due to the fact that in the same study, since the retention of their clients is directly correlated to how fast their web-pages load.

While it would be an incomplete analysis to directly attribute a long page load time to one single event, as this could either range from a user's slow Internet connection to server overload. Another study carried out at the University of Washington[12], shows that computation time; which includes loading, interpreting and executing JavaScript makes up 35% of the a critical path for a page load time and in particular, due to the dependency policies given by the Web Standards[3], synchronous JavaScript highly contributes to the overall page load time as it blocks HTML parsing when it is being processed, hence an increase JavaScripts requests results into a higher page load time penalty.

Henceforth, the studies carried out on this topic have sparked an increased interest in the research community to create tools and methodologies to decrease the web complexity and increase user retention through optimizing how JavaScript is handled on the browser without compromising the web page's contents or functionality [7]. The plugin I propose in this paper aims to specifically achieve this. In addition the proposed plugin achieves this without compromising users' privacy as the machine learning model that powers the plugin is entirely deployed on the client's device. On low-end devices, the plugin achieve a reduction of 31% reduction in the Speed Index, 32% reduction in the Page Load Time, 23% reduction in First Paint and 41% reduction in DOM Interactive time. And on high-end devices, the plugin performs similarly but slightly better numbers.

2 MOTIVATION

While there is a growing number of research being conducted on the topic of reducing the Page Load Time and increasing the quality of user interaction on the web by removing unnecessary web components that bloat the Page Load Time, there are still untapped issues that are generally not addressed in such research studies, such as Privacy issues, lack of sufficient user control over the platforms and a lack of focus on low-end devices which are the ones that highly benefit from removing these unnecessary JavaScripts.

Henceforth this is the motivating factor for this paper, with which I explain in details how these issues can be addressed while simultaneously improving the existing mechanism to decrease the Page Load Time. As an example the proposed system, *Iruka* guarantees privacy to the user by employing a novel system that uses a machine learning model that runs 100% from the client side, and therefore the client will never need at any point to communicate with a third party service to complete, as a result this fully insulates the client from any possible network attacks or possible data leaks.

Additionally, the tool proposed in this paper focuses on giving the end user the power to extensively configure how the tool works, but on a more important note, with *Iruka* we take into account that most users get the most of this feature of removing unnecessary web components are those whose electronic devices do not have enough capability to swiftly process Javascript, hence the tool proposed in this paper is tested and optimized for low-end devices.

3 RELATED WORK

Recently, there has been an increased interest in the research community to achieve a sustainable solution for simplifying ever increasing complexity of web pages due to JavaScript. Although with a certainly different approach and final aim. For example, Silo[8] is a tools proposed by Microsoft Research that aims to Exploit JavaScript and Document Object Model (DOM) storage to reduce page load time. Silo addresses a similar issue to this paper's but in a different way.

Silo concatenates JavaScript files to form a smaller number of JavaScript files, similarly to style sheets, different CSS files are combined into a single CSS file. This is ultimately aimed to reduce the number of HTTP requests required to render a complete HTML file. Additionally in order to increase the efficiency, the in-lined JavaScript and CSS are injected in the HTML so that they become single file with HTML and hence only require one HTTP request to be fully loaded. Though, this seems efficient, after enough considerations we realized a few implementation flaws for which it creates a segway to propose our idea.

Silo chunks in-lined HTML files and stores them in the DOM storage of the client, this makes it possible to only send necessary chunks of the HTML file when a user loads another similar page but with just a few different elements. In order to achieve this, Silo extensively uses the DOM storage to cache the concatenated HTML file. Such an implementation becomes a storage burden to a normal end-user who surfs around 80 different websites per month. Therefore given the fact that our proposed tool is targeted to support users from developing regions where storage is not as cheap. Silo does not line up to the need.

In order for Silo to be used, the server delivering the pages needs to be a silo enabled server. Therefore, this increases the learning curve of developers, hence being discouraging for the developers. Additionally this does not give the power to end-users to choose for themselves which content to block and with our tool, we intend to enable users to have a choice in regards to what scripts to block.

Another related tool is JS Lite [4]. JS Lite is a tool developed to optimize JavaScript and render an optimized version of a page. The system is comprised of a web browser plugin and a server. The browser plugin intervenes before a request for a JavaScript script is made and then sends the script to the server to be analyzed and labelled. The server runs a machine learning algorithm proposed in JS Lite in order to label scripts and place them in different categories, then sends back to the plugin the labelled scripts.

4 SYSTEM OVERVIEW

Whenever a request is about to be made, the system intercepts this with an *onBeforeRequest* event listener, the system checks if the request about to be made is for a JS script. If the request is for a JS Script, this will mark the beginning of a sequence of decisions initiated by the system to determine whether the request for the JS script is to be blocked or not.

The initial phase of the sequence involves checking whether the client's local database *indexedDB* holds a label for that particular JS script that is about to be loaded via the request.

In case the JS script's label is found to be stored in the local database, the system then decides whether to block or not block the request according the "blocking configurations" provided (which can be customized by the user).

Otherwise if the JS script's label is not found in the client's local database *indexedDB*, the request will not be blocked. In order to classify or label the JS script whose label was not found in the client's local database, the system uses the *onCompleted* event listener which is fired when an entire request for a script is complete, and then uses web-workers to label and save the JS script's label in the client's local database. This phase involves three crucial phases namely;

(i) **Feature Extraction & Replication:** In this phase, the system extracts and counts specific features from an incoming JS Script (in format the ML model understands)
(ii) **Script labelling:** During this phase, the system uses a Machine Learning algorithm to classify the script into one of 8 distinct labels according to the features extracted.
(iii) **Storing:** During this phase, the computed label of a particular script is saved into the local database *indexedDB* so that on subsequent page visits, the JS script's label is instantly retrieved instead of going through phase (i) to (ii) all over again.

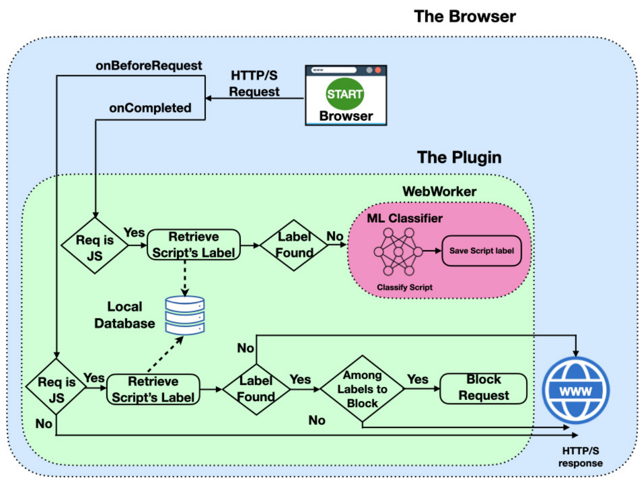

Figure 1: Architecture Design

5 DESIGN AND IMPLEMENTATION

This section discuses the implementation phases and design decisions of the plugin. There are two crucial implementation milestones with which the plugin is made with. (1) The Machine Learning Algorithm, (2) The Plugin

5.1 The ML Algorithm

The machine algorithm is one of the main core part of the plugin workflow, as it enables a server-less method of determining a script's category/label . In short, the machine learning's purpose is to efficiently analyze a JS script and classify it into the following categories seven categories; (1) ads+marketing, (2) tag-manager+content, (3) hosting+cdn, (4) video, (5) utility, (6) analytics, (7) social, (8) customer-success.

5.1.1 ***Training and Testing dataset.*** The dataset used to train and test the ML algorithm uses the Almanac dataset which contains more than 100,000 scripts from different websites. In the Almanac dataset, each record represents an individual script. Each record has 5000 columns which represent features that a particular script has, among the 5000 columns one would be a category column which would contain one of the 8 categories specified above. Beyond the category column, an example of other columns would be *addEventListener* or *onkeyup* and under this column it would have 1 or 0 which represents whether that record has this "feature" or not respectively. For the purpose of faster data accessibility and processing, the proposed tool *Iruka* uses a reduced version of the original Almanac dataset which has been implemented in the JSLite paper[4]. In order to generate the reduced dataset, multiple columns are combined into one according to function-wise similarity and odds of common occurrence. Using this dataset allows us to cut down the number of required columns by almost 90% - from 5000 to 508 columns of features.

5.1.2 ***ML Design & Implementation Phase.*** The machine Learning learning model is a three-layered Tensorflow Sequential model. Each layer is designed to reflect the dataset we have and need to process.

Layer 1 is designed to have an *input_shape* of 508 which reflects the number of columns i.e features from the reduced version of the original Almanac dataset that was cited above. The *inputs* configuration of the layer, which determines the output shape of the layer is set at 350 units, this allows for a decent but yet not steep-enough feature processing which could lead to loss of data granularity. Furthermore, to ensure fast computation we use the default activation function used when developing many multi-layered models - the rectified linear activation function or in short ReLU.

Additionally, using ReLU enables us to avoid the *The vanishing gradient problem* which occurs during the training phase of a model where the neural network's weights become stagnant and hence not change their values throughout multiple training sessions. This is due to the fact that the gradient increasingly becomes smaller due to the nature how the neural network's weights receive updates that are proportional to the partial derivative of the activation function in use. Henceforth this problem can be avoided by using a suitable activation function and in our case we use the ReLU function.

Layer 2 uses the ReLU activation function as well, similar to the first layer, but then has a configuration of 50 units which is a reflection of the *output_shape* to be expected as input from the third layer. Even though the goal of the model is to have an output of 8 units which is the number of script categories to predict, It is essential that we do not directly move from 305 units to 8 units, as it degrades the training quality and results in poor performance, hence why we have an intermediate phase which outputs 50 units instead 8 in order to not have a steep loss of granularity.

Layer 3 which is the final layer of the model, is designed to have 8 units which reflects the output shape of the whole model, and this reflects the expected 8 labels/categories mentioned above. Since the model is trying to solve a multi-class (8 classes) classification problem and in addition we are interested in getting a probabilistic representation how each class is classified, the most suitable activation is Softmax, hence for this final layer I use the Softmax activation function.

The model is then compiled using *Adam* as the optimization algorithm and *sparse_categorical_crossentropy* function as the loss function. Finally the model is then trained over 30 epochs with a validation split of 25 percent.

5.2 The Plugin

As mentioned in the over-view section, the plugin's purpose is to block incoming Javascript requests according to the blocking configurations that provides which categories/labels to block, and with the help of the Machine Learning model we are able to determine with a certain accuracy which category each script belongs to. In order to achieve this, there are phases that are implemented and are detailed below.

5.2.1 ***Preprocessing Phase:*** The ML model described in the subsection(s) above is by design not executable within a mobile browser environment, therefore in order to integrate the model within our plugin, we opted into converting the model to Tensorflow-JS model, which creates a version of the original model that can be run and executed inside a browser's environment. To achieve this we use a Tensorflow-JS in-built function *converters.save_keras_model* which carries out the conversion of the model from a Tensorflow to Tensorflow-JS format

5.2.2 ***Feature Extraction & Blocking Phase:*** In this subsection is where the script blocking logic implementation is explained. In order for the plugin to determine whether to block or allow an incoming Javascript request, the plugin needs to label the incoming JS script, and in order to do so, we have to first extract the content of the script and count how many features (among the 508 features) that are inside the JS script.

This allows us to create a data structure of features with which the ML model can use to predict the JS script's category. In regards to blocking an incoming JS scripts, the plugin sets a *onBeforeRequest* event listener which allows the plugin to analyze all incoming request. Whenever an incoming request is for a JS script, the plugin checks for 2 conditions to determine if the script gets blocked or not;

```
condition1 = script_category stored in local_db
condition2 = blocking_config has script_category

if (condition1):
    if (condition2):
         initiate(block script)

else:
    initiate(warm-up phase)
    initiate(do not block script)
```

Noticeably, from the pseudo-code above it is understandable that upon the initial contact with a JavaScript the client's local database will not possess that JavaScript's labelled category, hence making it impossible for the plugin to block scripts on the initial page visits. There is an option to run the ML model up on the initial page visit and henceforth avoiding the warm-up phase altogether, but unfortunately this option gives us a large overhead in terms of the performance during the initial visits.

On the other hand, with the warm-up phase implemented, the plugin is able to avoid such an overhead by not carrying out the blocking on first-page but on all other subsequent page visits. This is a trade-off between the user's overall experience and the ability to block JS scripts regardless of which page visit.

Warm-up & Labelling Phase: During the warm-up phase it is where the ML model is deployed, then used to analyze and classify the JS scripts. As discussed above, the warm-up phase is a one-time process per each JS Script which is initiated on the first contact with a JS Script.

To avoid performance lags, when processing each JS script we instantiate a new web worker which will then be executed from a separate thread from the main execution thread of the web-page[9]. This allows us not to block or slow down the main thread which could have otherwise lead to an increased page Load Time.

Inside the web-workers is where the machine learning model is loaded, and then the JS script is extracted for features, and labelled into a category. After the machine learning model predicts the category for the JS script, if the accuracy of the prediction is below 80% then the script is labelled as an "unknown" category, hence this guarantees us that whenever we are blocking a JS script, we are 80% assured that we are blocking a script that belongs to the category we want to block. Afterwards, the JS script is saved against its category in the client's local database *indexedDB*, after which the web-worker is terminated.

6 EVALUATION

6.1 ML Model Evaluation

As described above the implemented machine learning model is trained for 30 epochs over 100,000 scripts with which 75% are used as the training set and the rest 25% are used as the test set. After training the model over 30 epochs, the final results are; an accuracy of *0.8933* and a loss of *0.3188*.

The ML model's performance and efficiency can be broken down by each category, as it can be observed from the figure below that for the category of *ads+marketing* the ML model does considerably better at 97% than it does when it comes to categorizing *customer-success* scripts at 65%.

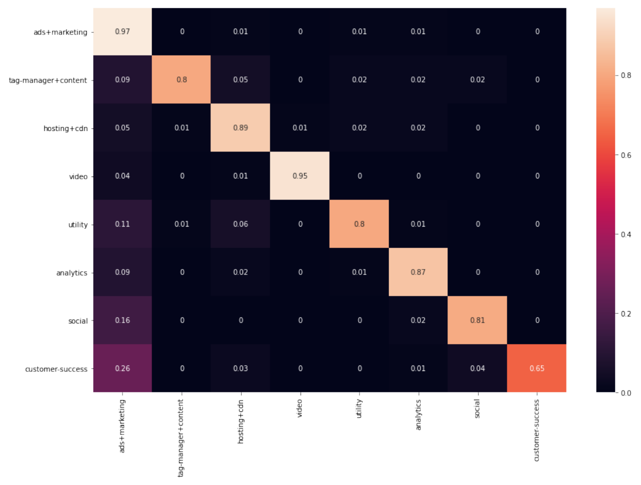

Figure 2: Breakdown of ML model performance by category

This performance discrepancy between the *ads+marketing* scripts and *customer-success* scripts is largely due to the imbalance between the two script categories in the training dataset used. Henceforth since the algorithm has more examples of *ads+marketing* scripts to learn from, this manifests into the results as well.

6.2 The Plugin Evaluation

The Evaluation of the plugin was carried out with an aim of understanding the overall impact of the system on various performance determinants being (1) First Paint, (2) Page Load Time, (3) DomInteractiveTime, (4) Speed Index.

The evaluation is done over two different mobile devices. A low-end and high-end device. The low-end device is a Xiaomi Redmi Go with a RAM of 1GB, storage of 16GB, has a Qualcomm® Snapdragon™ 425 Processor and the GPU is Adreno™ 308, up to 500MHz, and has a battery of 3000mAh and Wifi 802.11 b/g/n. Whereas, the high-end device is a Samsung Galaxy A8 with a 4GB RAM, with a storage of 32GB, and has an Exynos 7885 chipset with an Octa-Core, 2 processors: 2.2Ghz Dual-Core ARM Cortex-A73 as the processor, and ARM Mali-G71 MP2 as the GPU. The high-end mobile device has a fast charging 18W Li-Ion 3000 mAh battery and WiFi 802.11 a/b/g/n/ac.

For testing purposes, on each device the plugin is configured to block only three categories of scripts; Ads+marketing, Analytic and Social scripts. Each mobile device is tested on 500 websites. The first 250 websites come from the Hispar list of websites. Hispar is a list of a top web pages which takes a step further than famous Alexa Top 1 Million as it give URLs of internal pages instead of just the home page [11]. The next 250 websites come from the Tranco list. The Tranco list eliminates the issue that the majority of top web-pages lists i.e Alexa face which is the easy manipulation of such lists by malicious actors [10].

In the diagrams below, the blue dotted line represents the base line testing when the plugin is not installed on the mobile devices, the red line represents the first visit to a web-page when the plugin is installed, and the green dotted line represents the second web-page visit when the plugin is installed. It is important to note that in this experiment all caches were disabled so that the results are a clear indication of how they plugin performs without any interference.

6.2.1 *First Paint*.

First Paint refers to the time between navigation to a certain URL and when the browser renders the first pixels to the screen. The high-end device recorded an average of *2.707 seconds* when the plugin was not installed. When the plugin was installed the high-end device recorded an average of *3.526 seconds* on first visits and on the second visits it recorded *1.951 seconds.*

The results explained above means that on all subsequent visits to these web-pages, *Iruka* a will enable an user equipped with a mid to high-end mobile device to save 28% of their time spent waiting for the browser to render the first pixel. Here is a diagram illustrating the explanation above.

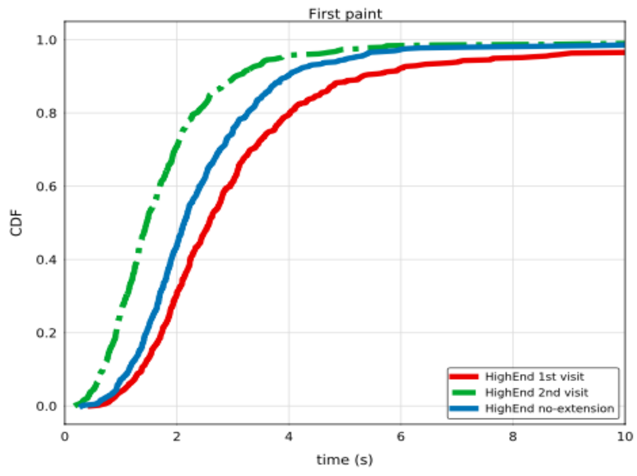

Figure 3: High-end Mobile Device, First Paint

While, the low-end device recorded an average of *3.058 seconds* when the plugin was not installed. And when the plugin was installed was installed the low-end device recorded an average of *4.267 seconds* on first visits and on the second visits it recorded *2.348 seconds*. Below is an illustration of the explanation above.

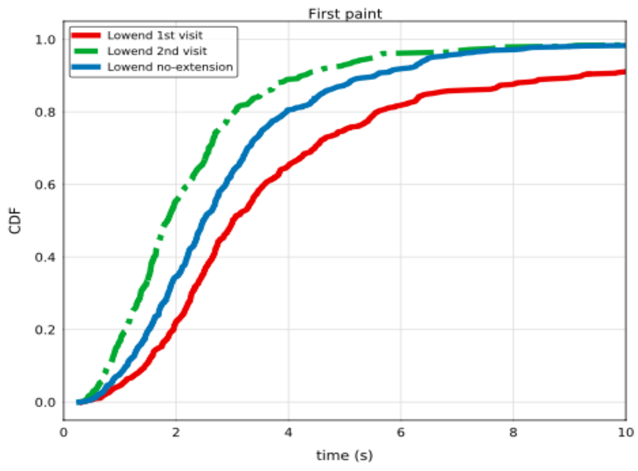

Figure 4: Hispar Low-end Mobile Device, First Paint

The results explained above means that on all subsequent visits to these web-pages, *Iruka* will enable an user equipped with a low-end mobile device to save around 23% of their time spent waiting for the browser to render the first pixel.

Average Performance Comparison (Seconds)			
	No-Extension	Initial visit	Second visit
First Paint (High-end)	2.707	3.526	1.951
First Paint (Low-end)	3.058	4.267	2.348

6.2.2 ***Page Load Time (PLT).*** The high-end device recorded an average of *10.699 seconds* when the plugin was not installed. When the plugin was installed the high-end device recorded an average of *22.635 seconds* on first visits, this is of course a one time penalty as on the second and subsequent visits it recorded *6.666 seconds* which is 38% decrease in the page load time for all subsequent page visits on a high-end mobile device.

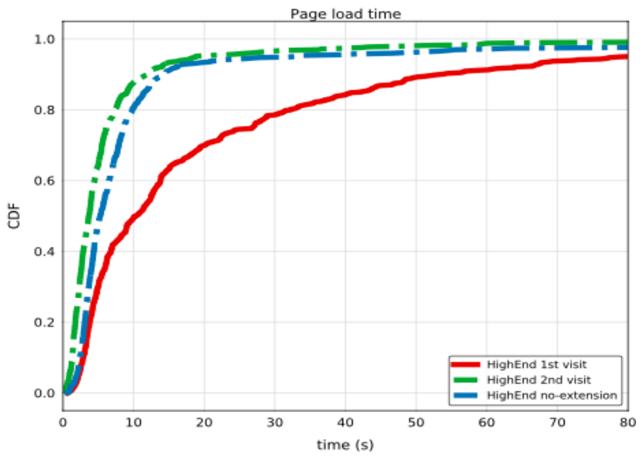

Figure 5: High-end Mobile Device, Page Load Time

On the other hand, the low-end device recorded an average of *14.252 seconds* when the plugin was not installed. And when the plugin was installed, the low-end device recorded an average of *26.513 seconds* on first visits and on the second visits it recorded *9.752 seconds*.

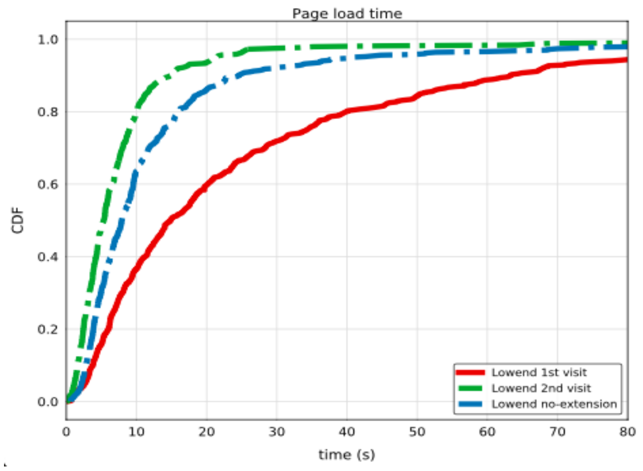

Figure 6: Low-end Mobile Device, Page Load Time

The results explained above means that on all subsequent visits to these web-pages, *Iruka* will enable an user equipped with a low-end mobile device to receive an improvement of 32% of the Page Load Time.

Average Performance Comparison (Seconds)			
	No-Extension	Initial visit	Second visit
PLT (High-end)	10.699	22.635	6.666
PLT (Low-end)	14.252	26.513	9.752

6.2.3 ***Speed Index***

Speed Index (SI) is an essential performance metric that enables us to understand how fast a web-page is populated. The high-end device recorded an average of *1.181 seconds* when the plugin was not installed. When the plugin was installed the high-end device recorded an average of *1.487 seconds* on first visits, and on the second visits it recorded *0.784 seconds* which is 34% improvement for all subsequent page visits.

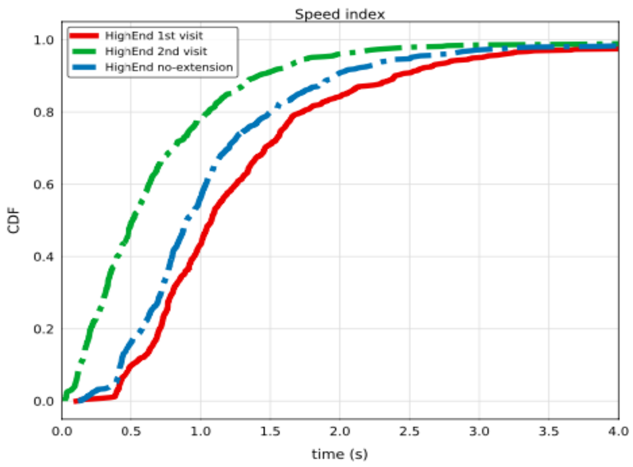

Figure 7: High-end Mobile Device, Speed Index

On the other hand, the low-end device recorded an average of *1.101 seconds* when the plugin was not installed. And when the plugin was installed was installed the low-end device recorded an average of *1.429 seconds* on first visits and on the second visits it recorded *0.759 seconds*

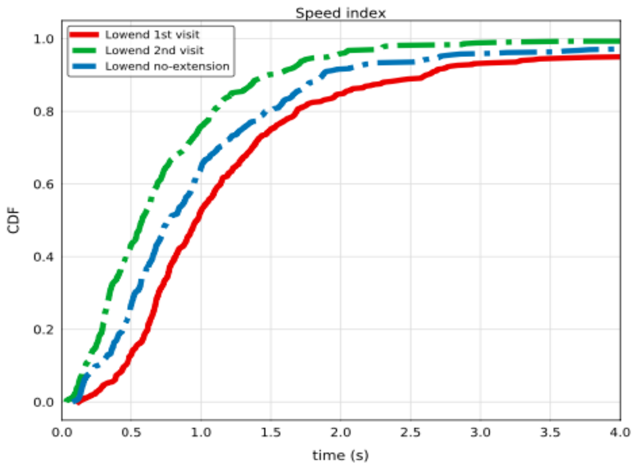

Figure 8: Low-end Mobile Device, Speed Index

The results explained above means that on all subsequent visits to these web-pages, *Iruka* will enable an user equipped with a low-end mobile device to receive an improvement of 31% of the speed Index.

Average Performance Comparison (Seconds)			
	No-Extension	Initial visit	Second visit
SI (High-end)	1.181	1.487	0.784
SI (Low-end)	1.101	1.429	0.759

6.2.4 ***DOM Interactive Time***

Similar to First Paint, DOM Interactive Time (DIT) is a performance metric that understand the website speed from user's perspective. DOM Interactive Time measures the time from when the user navigates to a URL until the page is ready for the user to interact[6]. It is important to note that in order for this event to be triggered the web-page does not have to be fully loaded, but should be in a state where a user can interact with it.

The high-end device recorded an average of *3.656 seconds* when the plugin was not installed. When the plugin was installed the high-end device recorded an average of *4.250 seconds* on first visits, and on the second visits it recorded *2.091 seconds* which is 43% improvement for all subsequent page visits.

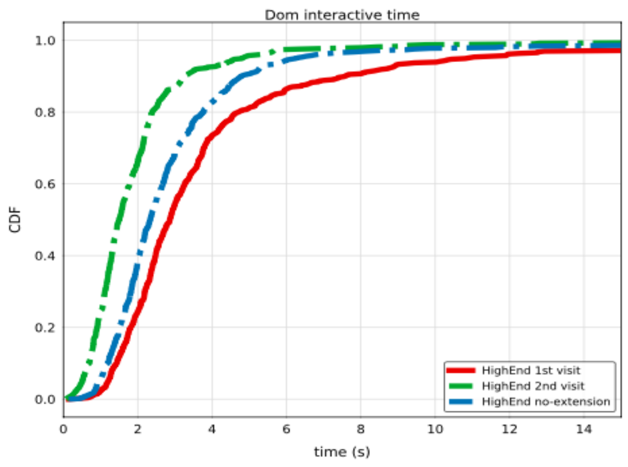

Figure 9: High-end Mobile Device, DOM Interactive Time

On the other hand, the low-end device recorded an average of *5.008 seconds* when the plugin was not installed. And when the plugin was installed was installed the low-end device recorded an average of *6.874 seconds* on first visits and on the second visits it recorded *2.938 seconds*

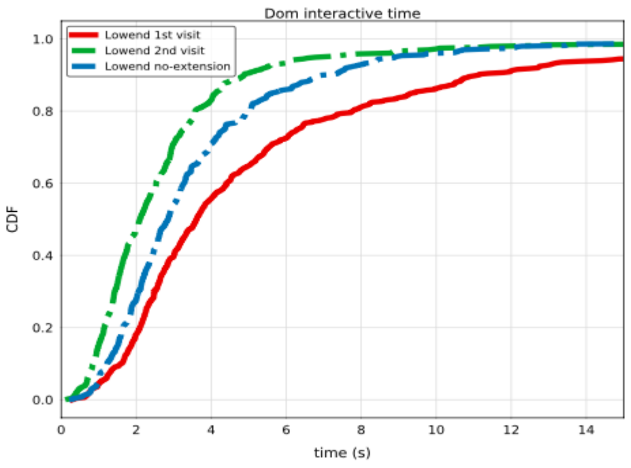

Figure 10: Low-end Mobile Device, DOM Interactive Time

The results explained above means that on all subsequent visits to these web-pages, *Iruka* will enable an user equipped with a low-end mobile device to receive an improvement of 41% of the DOM Interactive Time.

Average Performance Comparison (Seconds)			
	No-Extension	Initial visit	Second visit
DIT (High-end)	3.656	4.250	2.091
DIT (Low-end)	5.008	6.874	2.938

7 CONCLUSION

Taking into consideration the results of the experiment above to determine the impact of Iruka on accessibility of a web-page, It is in fact easily observable that on each performance metric Iruka provides a substantial increase in the overall performance regardless of the user's mobile device.

Let us take an example of the Page Load Time, a performance increase of 32% on a low-end mobile phone means that with the help of Iruka, we are cutting down the Page Load Time by 4.5 seconds from 14.252 seconds to 9.752 seconds. If a user visits 100 web-pages (not websites) per day then this will translate to about 225 minutes saved per month or about 45 hours every year.

Furthermore, Iruka has an immense impact especially in regions with expensive or slow internet connections, where even 1 Kilobyte eliminated on each web-page that a user visits might potentially save a user a lot of money or a substantial amount of time, while simultaneously respecting their privacy.

REFERENCES

[1] Daniel An. 2018. Find Out How You Stack Up to New Industry Benchmarks for Mobile Page Speed. https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/

[2] Http Archive. 2016. Average mobile webpage is now 2.2MB, 68% is images: let's trim the fat. https://mobile.httparchive.org/interesting.php?a=All&l=Apr%201%202016.

[3] World Wide Web Consortium. 2021. STANDARDS. https://www.w3.org/standards/.

[4] Dr. Moumena Chaqfeh Dr. Yasir Zaki. 2020. JSLite: An ML-driven Browser Plugin For Lightweight Mobile Pages. (2020). unpublished.

[5] Andy Favell. 2017. Average mobile webpage is now 2.2MB, 68% is images: let's trim the fat. https://www.clickz.com/average-mobile-webpage-is-now-2-2mb-67-is-images-lets-trim-the-fat/109268/. Accessed: 2021-04-2.

[6] UTKARSH GOEL. 2017. Beware of PerformanceTiming.domInteractive. https://developer.akamai.com/blog/2017/12/04/beware-performancetimingdominteractive.

[7] Utkarsh Goel and Moritz Steiner. 2020. System to Identify and Elide Superfluous JavaScript Code for Faster Webpage Loads. *arXiv preprint arXiv:2003.07396* (2020).

[8] James Mickens. 2010. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads. In *Proceedings of the 2010 USENIX Conference on Web Application Development* (Boston, MA) *(WebApps'10)*. USENIX Association, USA, 9.

[9] Mozilla. 2021. Web Workers API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API.

[10] Tranco. 2021. Tranco Top Pages. https://tranco-list.eu/.

[11] Duke University. 2021. Hispar Top Pages. https://hispar.cs.duke.edu/.

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