Framework to Improve the Web Application Launch Time

Suresh Kumar Gudla, Jitendra Kumar Sahoo, Abhishek Singh, Joy Bose, Nazeer Ahamed
Samsung R&D Institute
Bangalore, India
{suresh.gudla, jitendra.ks, abhi.rathore, joy.bose, nazeer.ahmed}@samsung.com

***Abstract*— Having too many applications running simultaneously on a smartphone consumes memory and slows down the performance of the device. Hence we need web applications which are lightweight and consume less memory. Web Applications use the browser engine and take a lot of time to launch compared to a native application, especially upon device boot up or if the browser is not already running in the background. In this paper, we propose an intelligent framework to launch web applications as fast as native applications. The framework considers the user's usage of web applications and pre-launches the preferred web applications, thus enhancing the launch time performance. We provide the architecture and implementation details of the framework. We then perform experiments on various web applications to measure the effectiveness of the framework for fast launch of the applications after the device boots.**

Keywords*— *Web apps, native apps, launch time, progressive web apps, responsive web apps, Web Push

I. INTRODUCTION

All major players in the mobile app industry have developed their own versions of both native apps and web apps to gain the confidence of end users. Native apps are supposed to be faster and responsive, but they require significant development and maintenance efforts because the developer would have to package all the resources for offline work. If a content provider intends to support multiple smart phone platforms like Android, Tizen and iOS, then they would have to maintain a separate application for each of these platforms. Also, porting among the platforms is often costly and not straightforward. Web apps give developers the flexibility to develop simple light weight web based apps with standard HTML, Javascript and CSS and thus facilitate cross platform development and deployment. On the other hand, web apps also suffer from limitations such as not being able to access local resources, although this will be less of a problem as more device APIs are developed. Users can open web apps using any of the standard browser engines like Chrome, Safari, Firefox, or Internet Explorer. The evolution of HTML5 and W3C specifications has helped to provide many essential native platform features like Push, Wallet API's and other device API's within the web apps.

Installing too many native apps on one device consumes power and slows down the device performance. Besides, users do not need to use too many apps simultaneously [1]. In Android, the low memory killer feature has to trigger frequently to free the memory consumed by background services and applications. Using web browser based apps instead of individual standalone native apps can help to reduce the number of installed apps. The web app internally uses a browser with minimal consumption of the RAM (Random Access Memory).

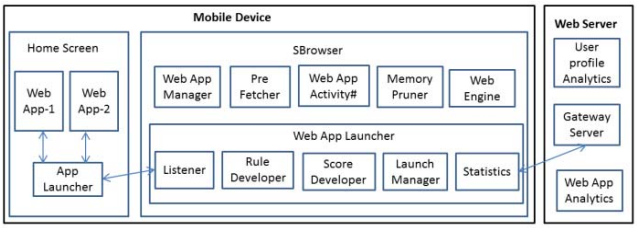

Fig. 1. System architecture showing the modules comprising the optimized webapp framework.

The full screen web app functionality in Chromium engine based browsers gives users the option to create a web app shortcut to the home screen. Selecting this shortcut will launch the web app in the full screen mode, giving the same look and feel as a native app. For the rest of the paper, we analyze this kind of app shortcut and methodology of launching using the browser engine as a web app. Functionality wise, full screen web apps behave similar to mobile pages rather than pure native apps which supports more offline functionality and are not dependent on Browser engine.

However, there are some challenges to be addressed for the web apps, which internally use Web Runtime (WRT) or the Web Browser directly, before they can perform as fast as native apps. One problem is optimizing the launch time performance of a web app, considering the user interest and previous web app usage by the same user. The Android Platform takes care of initializing and optimizing native apps and framework resources during the boot time, but it does not take care of web apps. This is because the full set of full screen web apps appears as shortcuts in home screens, but when launched will be taken care by the Browser Application in Android. This behavior is similar in case of all non-web operating systems (unlike WebOS and such web based OS'es).

2329-6453/16 $31.00 © 2016 IEEE
DOI 10.1109/MobServ.2016.21

In this paper, we discuss techniques to improve the launch time of the web apps in non-web operating systems by intelligently picking up the right set of web apps to pre-load, based on the user's usage of the web apps and the user context. Here, our objective is mainly to evaluate the performance of the engine changes in improving the launch time of the full screen web apps without affecting the core functionality of the full screen web apps. We also propose a framework to view the user profile statistics and web app performance statistics for content providers, which they can use to improve their app performance and promote their services and features to users at a later point of time.

The rest of the paper is organized as follows: In section 2 we survey related work in the area of web app optimization. Section 3 presents the proposed system framework. Section 4 describes the details of the proposed framework for optimized launch time performance. Section 5 describes the experimental setup and results, while Section 6 concludes the paper.

II. RELATED WORK

With the evolution of smart phone platform and web standards, current approaches to enable a full screen web app to perform at par with native apps include the following:

- Fast Launching of the web app
- Real Time Communication to user
- Offline mode of the web app
- Dynamic customization of the web app
- Exclusive Web App mode

The Chrome Browser App has solved the goal of offline mode of the web app by proposing the Application Shell Architecture [2] by leveraging on the features of W3C Service Workers and W3C Web App Manifest specifications. The Chrome Browser has also introduced the concept of progressive web apps using the functionalities of W3C Web App Manifest specification. However, the Chrome team has not focused on improving web application launch time considering the user interest and the likelihood of the user opening particular apps.

Liu et al [3] evaluated the performance of native apps and web apps while using web services, using various parameters like network connections and traffic volume and performing a trace based analysis. On the basis of the evaluation, they developed some guidelines on how the performance can be improved. However, they do not implement any optimizations in web apps to make their performance at par with native apps. In this paper we propose a method to do the same.

Yan [4], among others, proposes a method for predicting the next native applications the user is expected to launch, using contextual factors such as user location, and on that basis speeding up the launch time. Our approach is similar to this except it is geared for web apps and includes preloading of the apps that the user is expected to launch.

In the following section we describe the system framework and components used in our model.

III. SYSTEM FRAMEWORK

Figure 1 shows the framework of our enhanced full-screen web app architecture. Our architecture consists of the web server and the user's mobile device. The web server consists of a component called the Gateway Server, which acts as a middleman between the mobile device and the web server and performs operations such as running analytics on the user's profile and the web app usage and performance. The system on the mobile device consists of a highly tuned web app launcher, along with other modules like pre-fetcher, memory pruner, and statistics module to intelligently pre-fetch the web app and its content and thus reduce their launch time.

The component modules of our system are described in the following subsections.

A. Web App Launcher

The web app launcher module consists of the following sub modules: Listener, Rule-Developer, Score-Developer, and Launch Manager.

- The Listener module is responsible for listening to Smart Phone Platform level events like Web App Launch indications, Boot-Up Broadcasts, Low Memory indications, Browser App Launch indications and other system level broadcasts. These indications are later used in various phases and modules.
- The Rule-Developer module develops simple rules based on factors such as the usage of the web apps by the same user, current system level broadcasts, preferences of the user derived from the Browser App, etc.
- The Score Developer module gives proportional weightage to various rules and generates the overall weighted score index of a web app.
- The Launch Manager takes the weighted score index of each Web App and decides which web app has to be given to the pre-fetcher module.

B. Web App manager

The Web App Manager module listens to the Web App Launch Broadcasts and verifies that there is enough memory and resources available to launch a Web App Activity. Browsers like the Samsung Browser and Google Chrome support a limited number of Web App Activities. Hence if there is no Web App Activity available to launch a Web App, the Web App which is the least recently used will be killed (as per the Least Recently Used (LRU) mechanism) and the corresponding Web App Activity will be used for the new Web App launch request.

C. Pre-fetcher

The Pre-fetcher module is used to pre-render the corresponding Web App resources, scripts and html files and ensure that the actual page is ready for loading in a Tab of the Browser. The Pre-fetcher uses the internal browser engine modules such as network manager, resource loader, web app manager etc., that are necessary to ensure early loading of the

web app. The Pre-fetcher helps in reducing the actual launch time of the Web App as requested by the user. But selecting all the Web Apps to Pre-fetch is not the right choice and consumes a lot of memory and power if not used wisely. Hence we designed our Pre-fetcher to be tightly coupled with the Launch Manager which decides which app to Pre-fetch.

D. Memory Pruner

In the Android framework, managing native application resources and memory is handled through application life cycle callbacks. However, these suggestions are applicable to native apps but not to full screen web apps, which are highly dependent on the web browser engine. Hence we designed the memory pruner module, which is a hybrid in nature, to handle the memory requirements of the browser application along with individual web apps.

The Memory Pruner module plays a crucial role when the system goes down on Memory. It implements the onTrimMemory() API of Android and handles the important trim memory level indications like MEMORY_LOW, MEMORY_MODERATE and MEMORY_CRITICAL. If the memory level is critical and the web apps are not in foreground it informs the Launch Manager to clean all the pre-fetcher data. If the memory level is moderate or Low, it informs the pre-fetcher to clean pre-fetched data of the web apps that are in background and Least used. This mechanism helps in the web application level clean-up.

E. Statistics Module

The Statistics module is mainly designed for two purposes. Firstly, it stores relevant data corresponding to the web app usage and the user's preferences, and uses this data to generate a weighted score index. This weighed score index is used in successive device boot-up. The data can include statistical data necessary for each Web App, user details, device conditions, launch times of each Web App etc. Secondly, it also uploads the necessary details to the Gateway Server to generate quick charts for each web app and for further analysis. With the help of the statistics module, we can analyze the real time performance of our framework, since we know approximately how much time any web app should take for launching.

The Gateway Server takes care of providing the Web App analytics across various users and also the user profile analytics and usage statics of various web apps.

The Gateway server takes care of providing the web app analytics across various users and similarly the user profile analytics and their usage statics of various web apps. Content providers can request the Gateway server for these analytics and recommend push notification messages to the user to promote their apps and services.

In the following section, we analyze the algorithm to optimize the launch time of web apps as part of our framework.

IV. Efficient Web App Framework with Optimized Launch Time

As mentioned, in this paper we propose a system to pre load web apps that the user is more likely to launch, and thus to improve the average launch time of such apps. The types of Web Apps that a user is interested to launch is purely dependent on their current context. In our system, we design a context aware Web App Launcher which helps in launching web apps with reduced launch time. The Web App Launcher comprises of four modules, which are described below.

A. Listener

The listener module takes care of hearing the updates from the system, web apps, browser app and other user profile related information. This module recreates all this information into simple key-value pairs in the form "attribute = value". Every attribute plays a key role and have weightages assigned by the Rule-Developer. These attributes are evaluated against the rules developed in Rule-Developer and the Rule Vector table is updated accordingly

B. Rule-Developer

Rule-Developer is the central key module which has a set of predefined rules along with dynamic rules generated based on the device statistics and user context available. This is a lightweight engine and is designed based on predicate calculus. A rule consists of several predicates and logical operators. A predicate is a 3-tuple in the form of "attribute (relational operator) value".

For example, "Available Memory > 10%" is a simple predicate which will be evaluated at a system level and the evaluation result is critically high. Formally, a rule can be represented as a logical expression consisting of predicates, with brackets at any place as below

$$R = P_1\ op_1\ P_2\ op_2\ P_3\ op_3\ P_4\ \ldots..\ op_{n-1}\ P_n \qquad (1)$$

Here, P_i is a predicate and op_i is the logical connective or logical operator used to connect two or more predicates or sentences. Logical connectives AND, OR and NOT are usually "∩", "U" and "¬ " operators.

The rule developer has many predefined rules at different levels. These can include system level, Web App Level, and User Level rules. System level rules are mandatory for all the web apps. Web App rules are specific to each web app and User Level rules are applied to the user behavior and interests. Based on the attribute values, the rule developer updates the rule vector table. Figure 2 shows the mapping scheme between the rule vector table and predicate vector table.

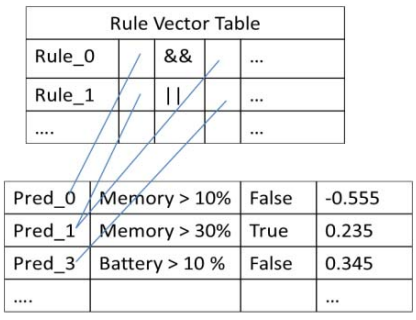

Fig. 2. Simplified Data Structure view used in Rule Developer Module.

For evaluation of rules to identify whether the algorithm should run or not, it is implied that all system level rules are to be passed. Web app level rules and user level rules are evaluated further to calculate the Web Score Index (WSI).

C. Score-Developer

Score-Developer calculates the score based on the rules and predicates. Score-Developer is tightly coupled with Launch manager, which has the actual algorithmic rules defined. In a nutshell, this determines what set of rules should be passed and what should be the minimum score index needed for a web app to be eligible for prefetching.

Score-Developer is mainly designed for two purposes. In the first step, it ensures all the mandatory rules are met including the system level, user level and web app level rules. In the second step, it generates the WSI index value for each web app. Using this index value, a further decision will be taken by the Launch Manager on whether or not to pre-fetch the web app being considered (in order to reduce the loading time).

In the first step, the Score-Developer ensures all critical rules like "Available Memory > 30%" and "Available Battery > 15%", "Browser App in foreground" are satisfied to ensure that the algorithm starts smoothly. For this we use a combination of rules to ensure the safe criteria is met. Usually we do not like much processing to happen when the battery is critically low. This kind of critical rule looks like the following:

$$R_i = P_1 \cap P_2 \cap P_3 \ldots \cap P_n \quad (2)$$

Where R_i is a critical rule identified containing one or more critical predicates. P_i can be a critical predicate or sub-rule having critical predicates.

In the second step, to get the weighted score index of each web app, we choose the rules which are true and extract the predicates from them. Since some of the predicates can repeat across the rules, the duplicate predicates are removed. Each of these predicates are given weights. We use the weighted sum model (WSM) to calculate the weighted score index of each web app, as presented in equation 3.

It is very important to state here that this is applicable only when all the data are expressed in exactly the same unit.

$$APP_i^{WSI} = \sum_{(j=1)}^{n} W_j P_{ij} \text{ for } i = 1,2 \ldots m \quad (3)$$

Where m represents the number of web apps on the device, P_{ij} represents the predicate used in evaluating for that web app, W_j the weight associated with that web app, APP_i is the i^{th} web app we are evaluating and APP_i^{WSI} is the WSI value of the i^{th} web app.

After computing the WSI values for all installed web apps, the top few Web apps with the highest WSI index will be chosen for pre-fetching. As we are considering user preferences and Web App usage history as per the user context, we assume that the WSI value is directly proportional to user preferences and likes of the web app at that moment or in that week.

D. Launch Manager

The Launch Manager module is designed to handle two functionalities and interacts with Score-Developer and Statistics modules. First, it handles the core algorithm execution along with the Score-Developer module and decides which web-app to pre-fetch. Second, it calculates the pre-fetch time taken for the selected web-apps, the launch time of the web app when the user selects from the home screen and updates the calculated information in the statistics database.

In the first step, the Launch Manager module checks the WSI index of each web app and decides whether to pre-fetch or not. If the WSI index is more than a cut-off threshold, the web app is requested for pre-fetching. The cut-off threshold is based on the past history of web-apps pre-fetched by the Launch Manager and launched by the user after that. This cut-off threshold is dynamically calculated, since the user's usage of web apps and preferences and context of browser launch may vary from day to day.

In the second step, the Launch Manager module calculates the pre-fetch time taken for each of the selected web apps, Launch time of the launched web app by user, duration of the web app used and updates the statistics data base. These values are further used to calculate the launch time performance evaluation results. The Statistics module, in turn, updates these profiling values to the Gateway server, which can be used further to understand the user behavior and preferences against each of the web-apps. The Gateway server module gives statistics of user, web app for understanding the trends, requirements and expectations of users of various ages.

In the following section, we describe a few experiments performed to analyze the effectiveness of our solution in optimizing web app launch times.

V. Experimental Setup and Results

As discussed earlier, our enhanced full screen web apps architecture plays a key role in making the Web App Ecosystem more robust and works as per user choices. To test the system, we selected apps from a list of top popular Android apps in India [5] in each of the top categories. One restriction we had was that the apps should also support the full screen web app functionality. Full screen web app functionality means the sites can be added as short cuts to home screen and when launched will open with browser engine in an exclusive mode as defined by W3C Web App manifest specification [6].

We picked NDTV from the Entertainment category, Twitter from the Social Network category, Times of India (TOI) from News category, VIA from the Travel category and Paytm from the Ecommerce category. All the chosen apps support the W3C Web App manifest specification. The corresponding native apps are installed from the Google play store. In our evaluation, our focus lies mainly in comparing the performance of a given web app with and without our framework. For the web app without our framework, we have accessed the corresponding website (such as PayTm.com or NDTV.com) using the default web browser and added a shortcut to the home screen, thus creating the full screen web apps as described earlier. For the web app using our framework, we

have made the described changes to the web browser engine and repeated the experiment. The corresponding native apps are added mainly for reference, with the understanding that the functionality provided is similar to the web apps.

In mobile platforms, the quality of an app or framework is mainly measured in terms of Power, Memory, Launch Time, Network Latency, CPU utilization and other screen transition criteria. As we are focusing on these evaluation parameters, the modules of Rule Developer and Score Generator are excluded during evaluation.

In this section we describe the experimental setup and results of the performance evaluation for our enhanced architecture. In our experimental setup, we used two Samsung Galaxy Note5 devices having Wi-Fi connected with 32.78 Mbps download speed and 42.98 Mbps upload speed.

A. *Launch Time test*

In our experimental setup, we used the top listed apps selected from various top categories as mentioned earlier and measured their launch times with and without our framework. The efficiency of the framework is measured by generating the Weighted Mean Absolute Error (WMAE) of the framework which is defined as below.

$$\text{WMAE} = 1 n w_i \cdot |y_i - y_i\hat{}| \quad (4)$$

Where w_i represents the weights given to each of the apps based on their priority preference to the user. In the evaluation system, we hardcoded these weights since we handpicked these apps. However, in a real time practical system, these weights shall be generated such that their summation is 1, the apps picked shall be the ones that are highly visited and used apps of the user. y_i denotes the launch time of the i^{th} app without our framework and $y_i\hat{}$ denotes the launch of the i^{th} app with our framework. We took the measurements 20 times on two devices. The graph in Fig. 3 shows the average launch time of the apps with and without our framework, along with their corresponding native apps launch time for reference.

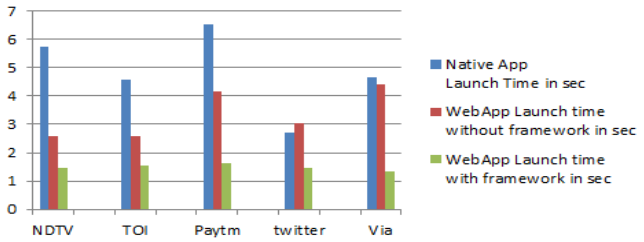

Fig. 3. Launch time for the selected apps, in seconds. Here X-axis denotes the application names and Y-axis denotes the launch time of the applications in seconds.

The weights taken as per the popularity of the apps are as follows: 0.3 for NDTV, 0.25 for TOI, 0.2 for Paytm, 0.15 for Twitter and 0.1 for VIA, such that their summation is 1. The WMAE of the launch time of these web apps as measured in our experiments is 0.216192. The lesser the WMAE score, the better is the performance of the framework.

As we can see, the percentage gain in launch time for NDTV full screen web app is 43.02 %, for TOI full screen web app it is 40.5%, for Paytm full screen web app it is 61.3%, for Twitter full screen web app it is 51.6% and for VIA full screen web app it is 69.5%. Thus, the prefetching of the resources and early composition of the pages without actual rendering seem to have a significant improvement in the launch time of the full screen web apps.

B. *Network Connections Test*

In our proposed system, we have not changed the execution flow of the web app composition and rendering. There will be only internal pre-rendering and hence the number of network requests made by a web app should not get affected with our system. We measured the network requests made by native apps, web apps with and without our framework for the selected apps mentioned earlier.

We took the TCP (Transmission Control Protocol) dump from the adb (Android Debug Bridge) shell of the mobile device until the app is properly launched and initialized and counted the number of network requests made by the app manually using the Wireshark Network Protocol Analyzer. We chose to measure in this way, since we are concerned about the total number of http and https requests made by the web apps rather than the amount of data downloaded.

TCP dump is taken using the command “tcpdump -i any -s 0 -w /data/<file_name.pcap>” in the adb shell command prompt window. Fig. 4 clearly shows there is not much difference in the number of network requests made by the web apps with and without using our framework. Thus our framework of prefetching have not changed the number of network requests made by the full screen web apps and hence there is no loss of functionality to user.

Usually, native apps are packed with many static resources and hence the number of network requests made by native apps is usually much less compared to the number of network requests made by web apps. This design differentiation is one of the basic differences between web apps and natives apps architecture.

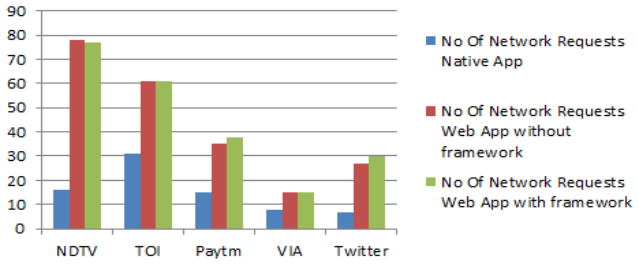

Fig. 4. Network connections of apps during their launch. Here the X-axis denotes the application names and the Y-axis denotes the number of network requests made by the applications.

C. *Memory test*

Measuring memory in case of full screen web app is different from measuring for a native app. While launching a full screen web app, an app process, a sandbox process and a

privilege process will be created. Launching another full screen web app will re-uses the app process and privilege process but creates a separate sandbox process for addressing security and privacy concerns of the second web app.

For example, launching the NDTV app creates an app process with the name com.*sec.android.app.sbrowser*, a privilege process with the name *com.sec.android.app.sbrowser:privileged_process0*, a sandbox process *com.sec.android.app.sbrowser:sandboxed_process0*. Now, launching the Twitter web app will add a sandbox process like *com.sec.android.app.sbrowser:sandboxed_process1* only, but adds additional memory consumed by the app to the browser app process. Hence the memory consumed when using a browser engine is cumulative and it is better to take a cumulative memory reading after launching all the web apps. We have taken the memory reading using the command "*adb shell dumpsys meminfo<process name>"*.

In our experiments, we considered the privilege process and sandbox process together as an engine process and have taken a cumulative reading of app process and engine process with and without our approach after launching all the web apps. The memory is measured in MB and is rounded to the nearest integer value for mapping.

Fig. 5 shows the average cumulative memory plot of a full screen web app with and without our approach. As we can see, the memory consumed with our approach is a little higher than without it. This is because the new modules like pre-fetcher and Listener caused a slight increase in memory which is negligible when compared to the significant first launch time improvement of the web apps that we saw previously.

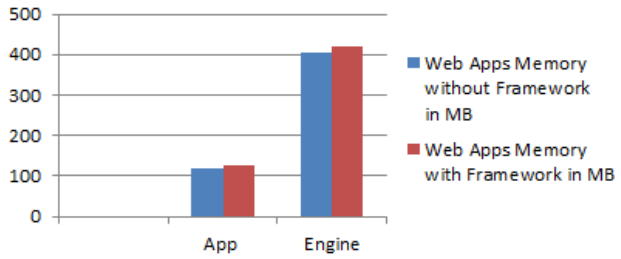

Fig. 5. Cumulative memory consumed by the selected full screen web apps measured in MB. Here, the X-axis denotes the component names in the browser and the Y-axis denotes the amount of RAM memory consumed in MB

D. *Power test*

In our experimental setup, we used the Monsoon power monitor [7] for measuring current in our test device. Current readings are taken in mA and measured for 10 times in our test device for a 10 sec window. During this window we have launched the web app and kept it in foreground till the 10 sec window is completed. The 10 sec window size is derived from our Launch Time test as we found that the average launch time of each of the apps was less than 10 seconds. Hence the power measurement is taken only for this window and power consumption shown beyond 10 seconds from the time of the launch is ignored.

The experiments are repeated for 10 times with and without our framework, and the results of average current are plotted as shown in Fig. 7. It is observed that for lighter web apps (such as VIA, having less amount of resources downloaded in order to load the page) the savings are less, as compared to heavier pages (such as Paytm). The percentage gain in current during first launch for NDTV is 21.3 %, for TOI is 14.3%, for Paytm is 38.8%, for VIA is 9.73% and for Twitter is 17.9%. Thus, less utilization of the CPU core helps in saving the power consumption of the device.

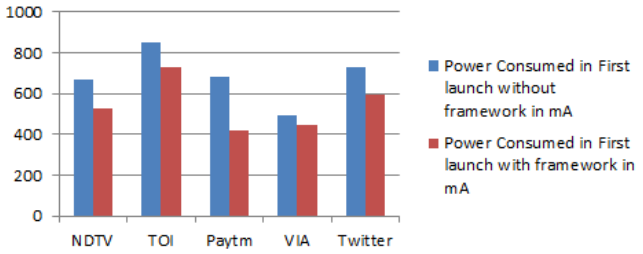

Fig. 6. Power consumed by web apps during their first launch in mA. Here, the X-axis denotes the application names and Y-axis denotes the power consumed in mA.

VI. Conclusion and Future Work

In this paper we have presented the design of a system to reduce the web app loading time based on user preferences, and presented the results of various tests to show its performance.

In future we plan to generate the rules online based on the user behavior, further improve the model by incorporating machine learning and also run the system for a larger dataset.

References

[1] Michael H. Yahoo says the average Android user has 95 installed apps, but only uses 35. [Online]. available: phonearena.com/news/Yahoo-says-the-average-Android-user-has-95-installed-apps-but-only-uses-35_id59898

[2] Addy Osmani, Matt Gaunt. Instant Loading Web Apps with An Application Shell Architecture. Google Developers Documentation. [Online]. Available: developers.google.com/web/updates/2015/11/app-shell

[3] Yi Liu, Xuanzhe Liu, Yun Ma, Yunxin Liu, Zibin Zheng, Gang Huang, and M. Brian Blake. Characterizing RESTFul Web Services Usage on Smartphones: A Tale of Native Apps and Web Apps. In Proc. ICWS 2015.

[4] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. 2012. Fast app launching for mobile devices using predictive user context. In Proc. 10th international conference on Mobile systems, applications, and services (MobiSys '12). ACM, New York, NY, USA, 113-126

[5] App Annie. Google Play Top App Charts. [Online]. Available: appannie.com/apps/google-play/top/india/

[6] W3C Working Draft. Web App Manifest. [Online]. Available: https://www.w3.org/TR/appmanifest/

[7] Monsoon Solutions: 2016. Power Monitor. [Online]. Available: https://www.msoon.com/LabEquipment/PowerMonitor