Measuring Cross-Platform Application Performance on Android

Does React-Native solve the cross-platform performance problem?

Daniel Watson, Lakshmanan Arumugam, Meiyappan Nagappan, Michael Godfrey
daniel.watson@uwaterloo.ca,larumuga@uwaterloo.ca,mei.nagappan@uwaterloo.ca,migod@uwaterloo.ca
University of Waterloo
Waterloo, Ontario

ABSTRACT

Facebook, React-Native's developer, claims that with this tool, developers can create a "real native app". Although their statement is vague, we sought to determine if, from a performance standpoint, this is true. Therefore, in this document, we present the results of an effort to determine certain performance characteristics of cross-platform applications developed using the React-Native framework. We found that overall, in the use-cases we tested, React-Native applications do not seem to perform UI updates and process input as efficiently as native Android apps. In many cases, we also found that React-Native performed less efficiently than apps built using Ionic. We did, however, find that React-Native does possess some important performance advantages over Ionic based applications that may offset some of the performance costs. Still, the idea that React-Native offers a major performance improvement over Ionic apps does not seem to hold true.

KEYWORDS

Performance Testing, Android, React-Native, Ionic

ACM Reference Format:
Daniel Watson, Lakshmanan Arumugam, Meiyappan Nagappan, Michael Godfrey. 2019. Measuring Cross-Platform Application Performance on Android: Does React-Native solve the cross-platform performance problem?. In *Proceedings of ACM/SPEC ICPE conference (ICPE'19).* ACM, New York, NY, USA, Article 4, 20 pages.

1 INTRODUCTION

In the world of mobile app development, there is an increasing demand for app development frameworks [23, 30, 43] that not only save time and effort, but also ensure consistent user experience across Android and IOS devices. To this end, cross platform mobile development tools that allow developers to write code once and run it on different platforms without significant reproduction of effort have gained popularity in recent years [30, 43]. The progress of "write once, run anywhere" (WORA) [7] software has been marked, but comes with limitations, especially on mobile devices. Popular cross platform development tools including Ionic[27], Appcelerator[24], Apache Cordova[4], enable portability using "native WebViews", but this paradigm comes with inherent performance impacts [28, 37]. These frameworks operate by drawing the layout of the application using web code such as HTML and Javascript, and by displaying the resulting view through what is essentially a stripped-down native web browser. This, however, typically leads to performance problems such as input lag, jank[39], and increased memory and CPU requirements [41]. As a result, these applications tend to consume more battery power and give the user a jerky and potentially less pleasant experience. These problems with older cross platform development frameworks cannot be effectively mitigated with clever coding; they are an inherent limitation to this paradigm.

In 2015, Facebook announced React-Native [29]. This framework allows developers to build mobile apps for Android and IOS using only JavaScript. With respect to cross-platform development frameworks, React-Native is the first of its kind in that its functional paradigm differs entirely from previous such frameworks. Rather than drawing UI elements in a webview, React-Native works by generating native IOS or Android UI elements using a specially designed JavaScript interpreter running in the background. This inevitably leads to some overhead, but unlike webview-style applications, nearly all overhead is shifted off of the main thread, improving UI responsiveness and resource management. Further, React-Native works by interpreting JavaScript to native code, native code can be introduced directly into React-Native code to implement features that are difficult or impossible to do in JavaScript alone.

Facebook states [14] that "With React-Native, you don't build a 'mobile web app', an 'HTML5 app', or a 'hybrid app'. You build a real mobile app that's indistinguishable from an app built using Objective-C or Java". The keyword here, "indistinguishable", makes this an interesting claim. Does the writer mean indistinguishable from a user's standpoint or a system standpoint? If React-Native apps are truly indistinguishable from native applications, then they will exhibit performance and resource consumption nearly-identical to functionally equivalent native applications. This should be a curious prospect to anyone with a software engineering background. React-Native requires a Javascript interpreter to run in the background during program execution, so there will necessarily be overhead involved. One important question to answer when arguing performance equivalence, however, is how this overhead is managed and how much of an impact it has on user experience and system resource consumption.

MOTIVATION

We were motivated to study React-Native as its rapid adoption[16] has created a demand for detailed analysis of the system. We were unable to find very much previous work that has been done on this

ICPE'19, April 2019, Mumbai, India

system. Further, it seems that React-Native's performance on Android is disputed. Facebook claims that React-Native applications are "indistinguishable" from native applications, but the development community at large debates this claim[12, 33]. Many believe that React-Native is inherently inefficient. Most concede that React-Native apps are smoother and better looking than those assembled with a webview-based tool such as Ionic [27]. We chose to study Android rather than IOS as the wealth of performance analysis tools in the Android ecosystem far outweighs that of IOS, which is generally a more opaque system overall. While researching the topic, we observed an interesting phenomenon occurring across the software development community. It is broadly assumed [32] by app developers that React-Native comes with run-time overhead and therefore will lead to more resource hungry applications. This assumed truth is present in comment threads and blog posts across the community.

There has, however, been some previous work [10] done on directly measuring resource consumption on the IOS platform. In this experiment, Calderaio builds two apps, one with Swift, the other with React-Native. He constructed the apps to be functionally equivalent to one another and built in some common functionality used by many millions of apps such as Facebook sign-in, editable lists, and an interactive map view. The author measured app resource consumption using Apple Instruments [5], a set of tools used by IOS developers to analyze the performance of their applications. The results indicate a much closer contest between the two technologies insofar as resource consumption and user experience is concerned, contrary to the assumptions of most developers. In these experiments the React-Native version of the application performed similarly across most experiments, and even performed better on certain tests.

Although intriguing, this experiment was not published as an academic study, nor written up with much detail, and there are a few glaring flaws in the experimental design. Firstly, the Calderaio did not perform a properly controlled experiment. Although, the author does not explicitly state this, the data presented seems to have been collected via a single recorded trial, making the small differences that arise during the comparison statistically questionable. Also, it seems that he conducted the trials by loading up the apps, then manually interacting with them to perform their test cases. This adds an unacceptable human element to the trials. The experimenter could not possibly have performed precisely the same set of actions for both test cases. The experiment is nonetheless interesting and suggests more work can be done on empirically determining how React-Native compares to native applications. Calderaio's study motivated and inspired us to embark on our project, as the results he obtained call into question the assumption that React-Native apps are inherently inefficient compared to native apps.

One of the primary promises of React-Native, beyond efficient operation, is reduced development turnaround due to process simplification because of the relative simplicity of working with JavaScript. JavaScript is by far the most well known language amongst the Github [18] community, allowing a huge population of developers previously relegated largely to web development to develop high-quality mobile applications. This has led to rapid uptake of the technology into the mobile ecosystem, as well as adoption by large technology companies and small-time developers. This growth has not been without caveats, however. Some users [3] report that React-Native has poor documentation, and its reliance on NPM, a package management tool with a history of notable breakages and security vulnerabilities riles some developers[35]. Still, adoption of React-Native for Android and IOS development has been rapid, likely owing to its cross-platform capabilities and open-source license. We want to contribute to this conversation by performing analytics on the run-time behavior of React-Native. To this end, we formulate two research questions:

(1) Are React-Native apps truly "indistinguishable" from native Android apps as Facebook claims?
(2) Are React-Native apps more efficient than apps built on HTML5 frameworks like Ionic?

2 METHODOLOGY

Designing Clones. To perform a true comparison of React-Native and native Android applications, we need equivalent implementations of various features common to Android applications. We also wanted to observe the performance differences associated with HTML5 based cross-platform mobile apps, so we chose to include Ionic in our experiments as well. We chose Ionic because it has a strong developer community (over 5 million developers)[25]. However, Android, React-Native, and Ionic require code in different languages, and therefore rely on different libraries for common functionalities. This is problematic for our experiment as preconceived notions about inherent efficiency could lead to bias that might affect the results of the experiment. Further, if we split the development effort between more than one individual, this may lead to inconsistencies across platforms, invalidating our results. For this reason, only one developer will write the applications, and he does so using an identical design pattern so that he can achieve consistency as much as is possible. The developer who built these applications, one of the authors, had prior experience in the industry as a software developer.

To reduce development time and ensure we take an approach typical of other developers, we studied and implemented demo apps available on Github, as well as Android samples [19], and implemented them step by step to achieve the desired functionality. We did not copy and paste the code as such, but developed simple applications following developer tutorials/documentation. Once the Android/React-Native application is complete, we develop an equivalent app on other platforms, mimicking the same behaviour and functionality. We looked for equivalent ways of replicating common functionalities and implemented them using standard practices.

We implemented three apps on each of the three platforms, each testing a different aspect of the system. We do not believe that our set of apps represents a full sampling of everything available to the development community, nor was this our intent. Our goal was to implement features that were commonly used by other developers or that demand significant computational resources.

For our first app, Property Finder [2], we developed a simple app for browsing properties available for sale in a given region. We developed the React-Native version first, using a tutorial [2]. We then implemented a native Android version of this application mimicking, as closely as possible, the functionality and look of

the React-Native version. We chose this application as it involves making an HTTP request to retrieve data based on a location query, then generating list elements dynamically and displaying them to the user in the form of a scrollable list. We chose this feature as scrollable lists are commonly used in many applications and dynamically generated lists tend to be resource intensive to generate and display.

Next, we developed and implemented a simple Maps application. We first followed Google developer documentation [20] for a maps demo app and to create a native Android version. The app contains a bare-bones map view, showing the user a map of the Earth and allowing them to scroll, zoom, and rotate the map. Just like the Google Maps app, the map updates in real-time by loading in more detailed map data as the user scrolls and zooms. Then, we located documentation and tutorials on implementing Google Maps functionality in React-Native and Ionic for which we found equivalent library that allows this functionality. We then implemented the functionality mimicking the same look and feel of the Android Application. We chose Maps because we knew that this is a resource intensive component of any application, and map views are commonly used for many tasks.

For our third application, Matrix Product, we eschewed UI elements entirely in order to get a sense of how efficiently React-Native can carry out CPU intensive operations. Specifically, we wrote an app that carries out a lengthy matrix multiplication task. The matrices were 250X250 in size and consisted of randomly generated positive integers between 0 and 10,000. The goal here was to observe how the JavaScript engine handles this intensive operation in comparison to the Android application which carries out the operation directly in the JVM, no interpretation needed.

Measuring Performance. "Efficiency" and "performance" are aspects of an application that must be evaluated across many axes, as there is by no means one complete definition of these terms. So, for the purposes of our analysis, we have devised what we hope is a relatively complete list of such axes, only some of which we will test in our experiments:

- CPU demands
- GPU demands
- RAM demands
- Frame-rate
- UI Latency
- Boot time
- APK size

Thankfully, tools exist for measuring most of these things. The tool that suits our purposes for most of the axes is Trepn profiler [34]. Trepn, developed by Qualcomm, allows us to measure CPU, GPU, RAM, application state, and screen state, among other things (irrelevant for our experiment). Trepn collects utilization data from the system in 100ms intervals, and has a relatively low overhead cost overall. It is also easy to trigger through shell commands, allowing us to automate the profiling process along with our automated tests. We chose to collect CPU and RAM measurements with Trepn as these metrics varied the most between trials in our early experiments.

Recording Performance Data. In designing our experiments, it quickly became evident that careful controls were necessary to achieve statistically valid results without bias. Most importantly, the series of interactions (screen touches) must be consistent between both of our samples. To achieve this, we design, record, and carry out automated UI tests. Several tools exist for this purpose, but the one that fit our purposes the best was ReplayKit[36]. This tool allows us to record and playback a series of interactions through the Android Debugging Bridge (ADB). So, to design the trials, we simply boot up our demonstration app, carry out a (qualitatively) predetermined series of interactions that demonstrates the functionality of the application in ways that are typical and intensive. A typical use case is what one might consider expected actions such as slowly scrolling a list, slowly panning a map, or otherwise performing actions one would expect from a typical use case. In the intensive phase, we carry out actions designed to abuse the interface as much as possible, to see how it reacts. This might be rapidly scrolling a list up and down, rapidly and unpredictably panning a map, or actions that may otherwise put unusual demands on the system. We do this to take measurements of a "standard case" and a "worst case" set of interactions.

Measuring the response to precisely the same set of interactions between the Android and React-Native versions of our apps already corrects for much of the uncertainty present in the inspirational experiment, but this alone is not sufficient. We found that we also need to carry out the experiment many times and aggregate our results to correct for variance among our samples. To account for this, we automate the entire process with a Windows shell script that sends Android shell commands via the ADB. In writing the script however, we did need to be careful to ensure that trials did not differ from each other in any other way. To do this, we start the profiler at the beginning of the experiment and begin each trial with the screen off and the application not running (even in the background). We then turn on the screen and unlock the phone (with adb commands) and cold-boot the app. After allowing the app to boot completely (by telling the script to wait for three seconds), we trigger replaykit to run the interactions. After the interactions are completed, we wait another 2 seconds to ensure we record any after-effects of the last interaction. Finally, we kill the app completely using the adb "kill" command which not only halts the app but also halts any background activities. Then, the screen shuts off and the process repeats. Due to, we presume, operating system level caching activities that we cannot control, the first several runs of the application in each batch tend to be noticeably slower than later runs. We performed a few preliminary experiments with the script and determined that application behavior stabilizes after 5 runs, so we drop the first 5 runs from each sample to maintain as much consistency as possible. This does introduce some validity concerns as we are testing the app after running it several times, which may not represent a true use case. We explore this issue in Threats to Validity.

We carried out experiments in two phases. The first phase was to allow us to design and implement a testing framework and to gauge the difficulty of recreating equivalent apps across platforms. In this phase, we built nine apps, three in Android, three in React-Native, and three in Ionic. We designed quick sequences of interactions and designed the trials to run quickly. Our preliminary results suggested

that React-Native apps do, in fact, possess a run-time overhead larger than that of native Android applications, but our initial results were not strong enough to be considered conclusive. The first phase did give us several insights into what makes a good test and what sort of behaviors we should look for. After we completed our initial experimentation, we found our results difficult to interpret for a few reasons. When we plotted our original data, features of our graphs were difficult to map to particular events. This was because we were running short trials wherein our apps were launched, run, and closed rather quickly. Also, we only ran our first experiment on a single phone, a Motorola G Play, so we have no way of knowing how well our results generalize to other phones. Lastly, although our original goal was to compare results between React-Native and Native Android applications, we felt that this comparison came without context, as React-Native is designed to be a replacement for other cross-platform development tools. Therefore, without results from some such tool, our results lacked a valid point of comparison. So, we introduced another popular development tool which also allows developers to author Android/IOS cross platform apps using JavaScript, Ionic. We chose Ionic because we felt that it represented the industry standard in WebView [26] [9] based cross platform tools.

For our new experiment, we wanted to isolate app startup and shut down costs from other features like screen interactions and UI updates. So, we added several seconds of delay to our scripts before and after launch and shutdown. We also designed our test interactions to be slower, more deliberate and more varied. This redesign made our graphs far more meaningful as it is now simple to discern which features correspond to a given event.

We also wanted to expand the range of devices that we use for our experiments, so we ran our second set of experiments on two new phones in addition to the original Motorola Gplay. We present important specifications of the phones we used in Table 1.

Table 1: Specifications of phones used in our experiments

Manufacturer	Motorola	LG	Samsung
Model	Moto G Play	Nexus 5	Galaxy Note 2
Screen Resolution	1280x720	1920x512	1280x720
SDK	25	23	19
OS Version	7.1.1	6.0.1	4.4.2
RAM	2 GB	2 GB	2 GB
CPU (4-core)	1.2 Ghz	2.3 GHz	1.6GHz

Analyzing Performance Data. Trepn outputs the performance data in CSV format, so it is easy to parse, manipulate, and visualize. We use rows during which the screen was recorded as being off as delimiters to separate individual trials. We have automated this process in an R script that allows us to split the results of the the experiment into individual trials which can then be aggregated to produce an overall profile. In our experiments, we observed that the length of each trial is highly consistent, as run durations are consistent across trials to within 2%.

Our early experiments showed that the CPU utilization data provided by Trepn is very noisy. So noisy in fact, it is very difficult to distinguish individual features of the resulting plots. We found that applying a moving average to this data helped to make large features more distinguishable, but came with a serious drawback. A moving average broad enough to smooth the data to a readable state obscured many smaller details. Furthermore, the apps must run in an Android OS environment, meaning that occasionally, Android would perform a background task, causing a spike in our data not attributable to the app we are testing. To avoid both of these obstacles, we take an average of 10 runs of the app and make observations on the resulting plots. This averaging approach does average out some fine detail in the data, but features are clearly visible in detail that were obscured by a moving average on one sample.

Because we repeated our experiments multiple times, we can make statistically strong claims regarding differences in CPU and memory consumption. Our data is in time-series form, so, to test for significant differences, we take the means of the observations of each app at each point in time to produce an aggregated representation of our performance data. It is this aggregated result that we present in results. To perform statistical comparisons between any two apps, we produce a time-series representing the difference between them and model the resulting time-series with an ARMA model. ARMA (Auto-Regressive Moving Average) is a time-series model that approximates a time series in terms of two polynomials, an autoregressive (AR) component and a moving average (MA) component[40]. Each component has a set number of coefficients, p and q for the AR and MA components respectively. For each test, we choose p and q to minimize AIC as recommended by Brockwell and Davis [8] The coefficients are estimated using least squares regression and the resulting model consists of $p + q + 1$ terms. The extra term is an added constant, called the intercept (I), and it represents the mean of the underlying stochastic process[17]. We would like to know whether a statistically significant difference exists between the any two measurements over an interval, therefore the intercept term, and its significance, is our coefficient of interest. Each coefficient, including I, comes with a p-value, representing the probability that we would observe a greater I, given that the underlying process is zero-mean. In context, this means that the p-value associated with our I coefficient represents the likelihood that we would observe a larger difference in our measurement than we did, given that the two apps actually have equivalent resource demands. For each interval wherein we observe a difference graphically, we run this test and report I and our p-value for this coefficient. ARMA models are only effective on stationary time-series, so we perform an Augmented Dickey Fuller test [17] to check the time series for trend before fitting our ARMA model. We only report results for intervals that pass this test with a confidence level of 95% or better.

We have made an effort to provide all of the artifacts from our efforts to encourage replication and future work. The app source code, Powershell scripts used for experimentation, and R scripts used for analysis are available on our Github repository, Laksh47/uwaterloo_icpe[1]. Also available there is an expanded version of this document with appendices. We encourage interested parties to reach out to us for assistance in replicating our experiments.

RESULTS

We first report all important observations we made about each experiment. Please ensure you are reading the graphs in color, as they are difficult to read properly in black and white. For much of the time during our experiments, the CPU data contains little interesting information as the app is either idle or, in the case of Property Finder, receiving keyboard input requiring minimal CPU time. Also, some of our trials are more than two minutes in length, so presenting all of this data on paper would result in cumbersome, unreadable plots. So, in this section, we present only subsections of our graphs that are illustrative of our observations. Our full data, plotted in a longer format, can be found in the appendix. Note that this means that the scale of the horizontal axis is inconsistent in the graphs in this section

Property Finder. This experiment demonstrates some performance limitations of React-Native. At $t = 74s$, (figure 1), after some downtime waiting for the http request to be filled, we see a large, brief spike (1) in activity followed by a moment of downtime, and then another, larger spike peaking about three seconds later in all three versions of the app. The first spike occurs when the http response arrives, the data is read and a data structure is constructed with it. Then, there is a brief pause (2) as the UI transitions, and then the list is drawn from the data acquired by the request (3). The two spikes, separated by a moment of downtime, were observable on all phones and on all platforms. The React-Native implementation of the app, however, utilized the CPU far more during this process than the Android (I = 7.75, p = .023) implementations. Recalling React-Native's multi-threaded nature, it follows that it would have a tendency to occupy more than one core during execution.

During $t \in [80, 117]$ seconds into each trial (figure 2), the user scrolls through the list slowly (4) at first, then scrolls quickly and erratically (5). During scrolling, the React-Native app demands more CPU time than the Android app over this interval (I = 6.65, p < 2e-16), as well as the Ionic app (I = 5.14, p=0.00172). The Ionic app does not consume significantly more CPU than the Android app during scrolling (I = -1.02, p=.61). This trend persists in the last interaction of the trial with the back button being pressed (6), (end of figure 2) returning the user to the search screen. The React Native app once again exhibits the tallest spike, followed by Ionic, with Android spiking the least. No significant differences in the memory behavior between the apps for this test. The memory necessary to carry out the task is trivial, so this is not surprising. This app, consisting of one simple list view, demonstrates how hard a React-Native app will push the hardware to carry out basic UI operations.

Maps. The most important result of this test was that map interactions utilized the CPU more in the React-Native app than in the Android app in 100% of interactions across all phones. Comparing React-Native with Android over the full scrolling interval, we found the a significant difference in the means (I = 7.41, p = 1.350e-14). The Ionic app also demanded more CPU than the Android app (I=6.11, p= 0.001227) over this interval. These trends do not vary much between phones. Ionic's demands are also more varied, however, sometimes performing similarly to (7) the Android app (Figure 3, $t = 42s$) and sometimes spiking (8) much higher than the other apps (Figure 3, $t = 54s$). In this test, the Android app

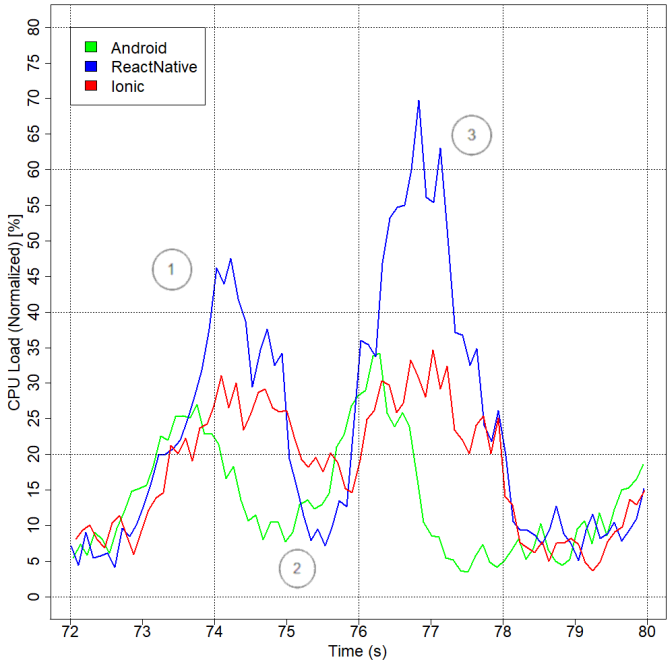

Figure 1: An HTTP request is received and the UI is updated.

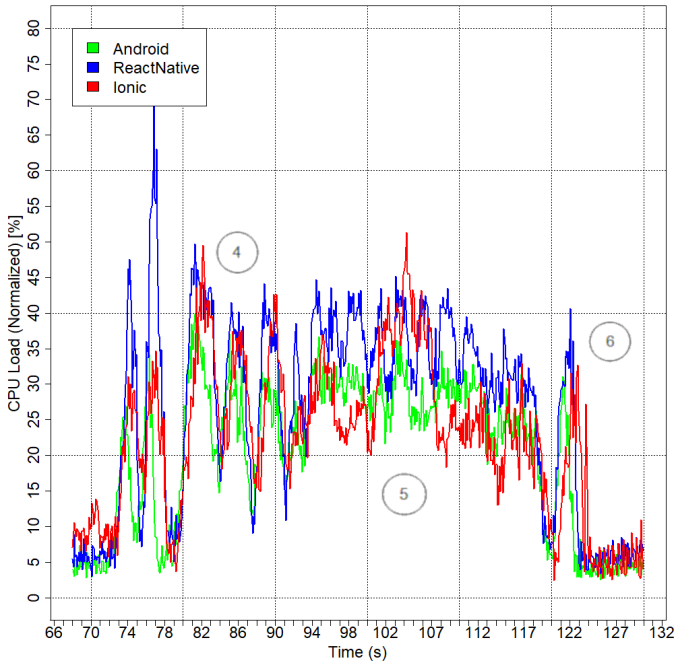

Figure 2: The user scrolls the list repeatedly after the UI updates

Matric Calculation		Mean	Maximum	Minimum	Median
Nexus	Android	4.06	4.94	3.18	4.19
	React-Native	4.29	5.01	3.88	4.06
	Ionic	1.15	1.31	1.09	1.13
Note	Android	0.75	0.78	0.71	0.75
	React-Native	4.38	4.46	4.28	4.39
	Ionic	0.93	1.19	0.58	0.96
G play	Android	1.11	1.74	0.97	1.05
	React-Native	7.08	7.29	6.93	7.03
	Ionic	2.38	2.60	2.00	2.43

Table 2: Time taken(in secs) to calculate the 250*250 matrix multiplication

performed the best and React-Native and Ionic performed similarly, each with their own benefits and detriments. Map views have significant memory requirements, so our memory data showed some interesting characteristics. Figure 4 shows the memory allocated by the system throughout the trial since Trepn only allows us to track total system RAM usage. Our memory data can be deceptive to interpret. For example, in the Maps memory data (Figure 4), the React-Native version of the app shows lower system memory usage throughout the trial, but our measurements also show that elevated system usage is present before the app launches (t = 8s) and after it closes (t = 77s). This, we presume, is an artifact of the way we carried out our experiments, but is still telling. We believe this to be due to Android's caching behavior. That is, we believe that the Map-View component we used in the app causes the system to cache map data that it pulls from the web. This would explain why the readings taken from the React-Native and Ionic apps show memory consumption increasing as more of the map is displayed while the Android version simply reads from its cache, making memory usage higher overall, but more stable. Further, this may relieve some of the work of the CPU and reduce data usage, but this remains to be seen. That being said, it is telling that the three apps seem to start out with different amounts of allocated memory, but the three converge to nearly the same amount by the end of the trial. Recall that the apps are run 5 times before we begin recording. Interestingly, Android seems to do this more aggressively with the Android implementation of the Maps app which uses the native MapView class, a subclass of Android's View class. The shape of the React-Native and Android curves are nearly identical in shape, although the React-Native app required about 75MB less memory throughout the trial, with Ionic requiring about 15MB less than Android at its peak (9) around $t = 70s$ (Figure 4). Considering CPU and memory utilization, Android's handling of the Map-View functionality seems preferable to the others in terms of resource efficiency.

Matrix Product. Table 2 and 3 shows the time taken by the apps from calculating the matrix to rendering it in the UI. We collected the data by dumping out the timestamps to console logs after each operation (matrix multiplication, the next UI interaction) and manually calculated the differences. These console logs were added to observe the time differences after the performance experiments were completed.

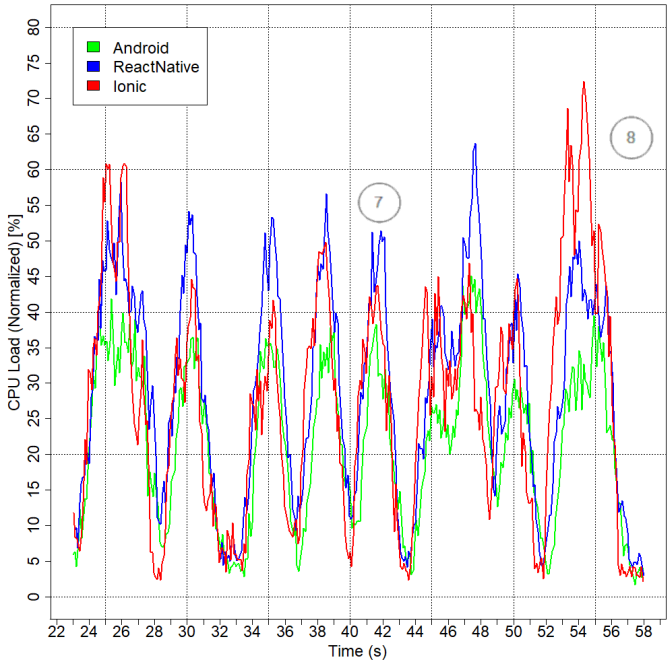

Figure 3: The user zooms and pans the map repeatedly.

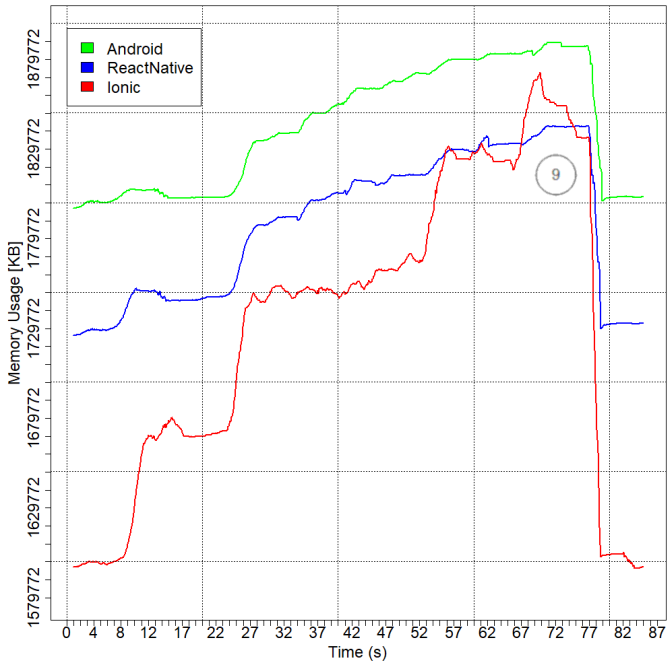

Figure 4: A comparison of memory requirements for our Maps applications.

UI rendering		Mean	Maximum	Minimum	Median
Nexus	Android	32.26	35.57	26.96	34.55
	React-Native	19.06	21.64	17.86	17.96
	Ionic	9.69	11.05	9.27	9.45
Note	Android	23.09	23.25	22.97	23.06
	React-Native	40.14	40.56	39.75	40.24
	Ionic	5.05	5.69	4.18	4.98
G play	Android	28.56	28.65	28.43	28.54
	React-Native	20.12	20.32	19.95	20.12
	Ionic	15.61	21.33	13.69	14.09

Table 3: Time taken (seconds) to render the entire matrix result as a string

Our results for this trial were surprising. We found that the UI update, basically writing a very long string to ScrollView, took more time than computing the result itself. We present tables showing how long it took each app to compute the result (Table 2) and update the UI (Table 3). We found that all three platforms were able to carry out the computation very quickly, but it took far more time for the app to update the view with the result, which came in the form of a very long string. With regards to computation time, we found that the React-Native app and native app took a similar amount of time to complete the computation on the Nexus phone while React-Native took 5.84 times longer than the Android app on the Note phone. Also interestingly, Android app took, on average, only .75 seconds to compute the result on the Note, an older phone, but took 4.06 seconds on the Nexus, a newer phone with a faster processor. It is unclear why this is the case, but it may have to do the way the phones manage background processes.

It would appear from the table that Ionic updated the UI much more quickly than the other apps, but there were some important differences in the way the three apps behaved during this step. We found that the React-Native app and the Android app both became interactive (scrollable) immediately after the result was visible to the user. The Ionic version, on the other had, took more time to become responsive after this step, taking, on average, 25, 8, and 15 seconds on the Nexus, the Note, and the G Play. Still, even this considered, the Ionic app became fully rendered and responsive faster than the other apps on the Samsung phone, which is an interesting result. After the UI updated and became responsive, we had the "user" scroll the output and measured CPU consumption during this action. The result is presented in Figure 5. The React-Native app demanded much more CPU time during scrolling than Android (I = 11.09, p = 2.156e-05) as well as Ionic (I=8.91, p = 8.142e-05), while no significant difference was found between the Android and Ionic apps.

Memory efficiency also showed some disparities on this test. When the apps start ($t = 10s$ Figure 6), memory is allocated by all three apps, but Ionic allocates much more than the other two on launch. Interestingly, at computation start ($t = 28s$ Figure 6), the React-Native app allocates just as much memory until the computation completes ($t = 38s$ Figure 6). Android appears to handle memory more efficiently than the other frameworks in this context as its requirements are consistently below the others during the computation and UI update phases.

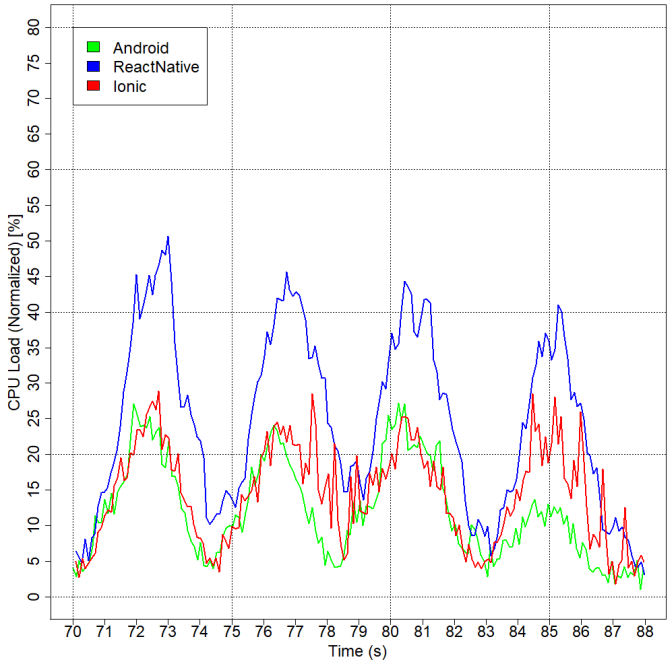

Figure 5: The user scrolls the result of the matrix multiplication.

f

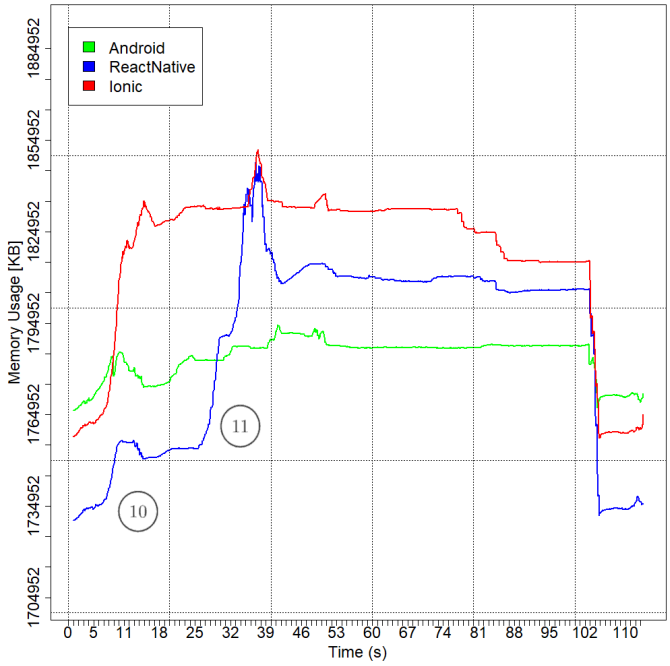

Figure 6: Memory requirements for our Matrix multiplication experiment.

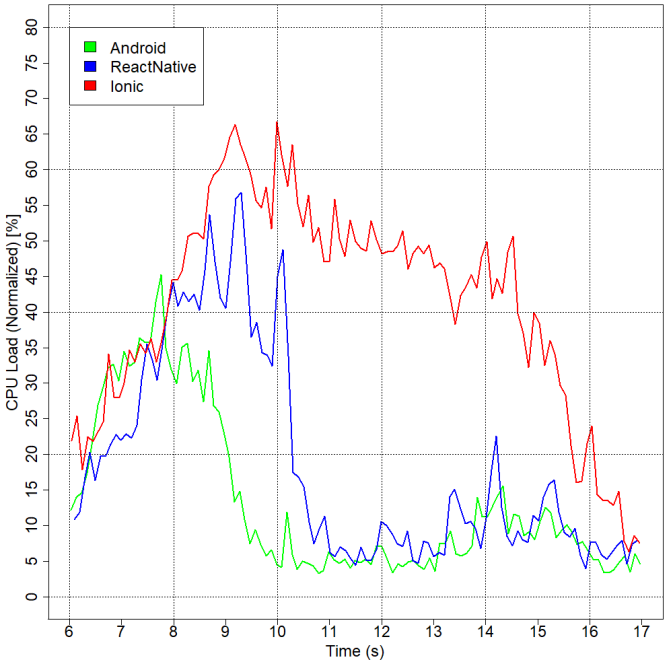

Figure 7: Comparison of app startup demands.

Discussion

The only feature we identified that was absolutely consistent was the trend in startup costs. Figure 7 shows the start-up demands of all three Property Finder apps. We observed similar demands during startup across all apps and across all phones. Ionic consumed the most CPU time during startup by far in terms of both peak usage and the duration of elevated consumption. React-Native however, also consumed extra CPU time compared with Android, but the difference was far less substantial. Therefore, if it is a priority for an app to startup quickly without slowing other processes, a native Android application is still the best option. However, React-Native offers a substantial improvement in this area over Ionic.

Another observation that we made consistently was that any user interaction that changed the UI state was more costly, in terms of CPU usage, in the React-Native version of the apps than in either Android or Ionic. This is particularly observable in the Property Finder and Matrix Product trials, but is less consistent in the Maps application. In all cases, we found React-Native to be no less demanding in this regard than Ionic. This is an interesting finding as UI state changes are the most critical moments in terms of app performance and user experience. If keeping the CPU free during these moments is a priority, once again, native Android is the best option and React-Native does no better than Ionic in this performance point. The results we have gathered here largely align with the Android community's belief that cross-platform development will necessarily have performance disadvantages. Considering how React-Native works to translate React code to Android native code, this should not be very surprising. At runtime, React-Native interprets the JavaScript code which defines the app, necessarily introducing some CPU bound overhead to every user interaction.

Armed with this intuition, as well the results we have present here, we can answer our research questions:

(1) Are React-Native apps truly "indistinguishable" from native Android apps?
No, at least not from a performance standpoint. Our native Android apps performed better in terms of CPU and Memory requirements in nearly all tests we carried out. The difference was particularly observable when user interactions were occurring.

(2) Are React-Native apps more efficient than HTML5 apps built on frameworks like Ionic?
No, not really. The only time we observed React-Native performing preferably was during startup, where Ionic demanded far more CPU for a longer duration. Other than this aspect, Ionic performed more efficiently than React-Native, especially during user interactions.

RELATED WORK

Many previous studies have taken various routes to measure the performance of applications under various circumstances. In Performance Monitoring of Java Applications [22], Harkema et al. present a performance measuring tool called JPMT (Java Performance Monitoring Toolkit) that focuses on Quality Of Service (QOS). They devise a system based on event driven monitoring that extracts performance data from logs generated at runtime. This tool allows them to monitor the creation and deletion of multiple threads, accurately report on system response times, and generate an overall picture of system performance with (reportedly) minimal overhead. However, this approach (event driven monitoring) is infeasible for our problem space. Event driven monitoring is applicable for large systems with high (>1s) latency processes where the overhead introduced by event logging does not produce a significant observer effect. On mobile platforms, however, we are working with a system with inherent performance limitations. Mobile platforms have very limited computational resources and the operating systems they employ aggressively manage background resource consumption to give the user a smooth experience and prevent battery drain. This requires a lighter hand to be used for our experiments.

Filip [6] has conducted a similar study that aims to find out if Ionic 2 is a suitable framework (in terms of performance) for designing cross-platform applications. In this thesis, Filip has developed one application "Val-home" in 2 platforms (one in React native and one in Ionic). The objective of this thesis was to justify ionic as a cross-platform development framework with respect to performance. Both applications share a common backend system, thus, Ionic was primarily used to develop the user interface. The applications were tested on a single mobile device using Trepn [38] profiler. They find that React-Native and Android had more modest system requirements and used less battery. They also performed a user study to determine whether users were able to identify any differences between the apps. The study concludes that ionic can be used to develop cross platform applications. One would think the thousands of apps on the app store would be proof enough of this. Our work is similar in that we are comparing the same three platforms. However, we explore the differences in performance of these platforms in a more repeatable and statistically sound manner.

We also perform all of our tests with multiple devices to test our apps on a broad spectrum of hardware.

Hansson and Vidhall have written a thesis [21] on the performance of react native on Android and iOS. They develop a single application on both Android and iOS, then a functionally equivalent one on react native. They found that 75% of the code can be re-used in both Android and iOS versions of the app with the rest making up platform-specific code. The application they developed was a "home automation application". Their performance study on this application indicates that react native uses more CPU compared to native Android and iOS. They report no substantial performance bottlenecks throughout their experiment, regardless of platform. They also conducted a user study to understand whether the apps produced a similar user experience. In this work, we eschew questions of user experience and focus solely on rigorous performance trials. We did not include native code in our React-Native applications because we are more interested in how well our apps can perform with JavaScript elements alone.

Xanthopoulos and Xinogalos [42] have investigated the various approaches of developing cross platform mobile applications such as web apps, hybrid apps, interpreted apps and generated apps. This work was published (2013) long before the initial release [15] of react native (2015) framework. They performed a comparative analysis of these approaches based on distribution, technologies involved, hardware and data access, user interface, and user perceived performance. The study shows that "generated apps", in which apps are compiled like native apps, is the most promising approach. Although, with respect to native look and feel, they conclude that "interpreting apps" is the best approach. React-Native is an interpreted system that did not exist when this study was published. React-Native has theoretical advantages over web apps and hybrid apps, so this gives us an opportunity to compare the cutting edge in hybrid platforms like Ionic and interpreted platforms such as React-Native.

In 2013, Dalmasso et al., conducted an study [11] of cross platform tools available for developing mobile applications. In this work, they discuss the general architecture of cross platform development and identify the desirable requirements in cross platform frameworks. They also present the survey results and comparisons of frameworks such as PhoneGap, Titanium and Sencha Touch. They also built test applications to measure resource consumption in Android. The block of code that measures CPU is injected into the app, memory is measured using DDMS in Android development tools, and power consumption is measured using the 'Power tutor' Android app. However, it is unclear from the paper whether the test cases used to measure the performance were automated or whether the entire experiment was carried out in a controlled manner. Finally, they report that the resource consumption in PhoneGap is lower than other frameworks as PhoneGap does not provide exclusive user interface components. In this work, we intend to improve on this study's experimental design using multiple pre-scripted experiments for repeatability and statistical power.

THREATS TO VALIDITY

Observer Effect

We are not testing our applications under conditions that perfectly reflect use cases. Our testing tools may impart some observation effects on our data. Trepn profiler queries the system for resource information 10 times per second. This necessarily induces some additional demands, skewing our data. Additionally, the automated testing script sends user interactions over the Android Debugging Bridge using Replaykit. However, Replaykit's recordings do not have the same temporal resolution as the phone's touchscreen. This just means that the interactions we send to the phone are not as smooth as genuine interactions that go through the touchscreen. Emulating interactions over the ADB also introduces some CPU strain, like Trepn. We feel that these effects did not have sufficient impact on our data to impact our conclusions. All apps were tested with the same automated testing script and and each set of apps was tested with the same set of user interactions. Trepn's observer effect should be identical throughout all trials as each test was run with the same configuration. The observational impact of our testing environment was uniform across all trials and therefore should not benefit one platform over another. This does imply, however, that the performance data we collected may not perfectly reflect the way the app runs in-vivo.

Pre-Measurement Runs

We found in our early experiments that the first few times an app is run on a phone after a fresh boot, any gathered performance data will be too inconsistent for detailed analysis. We believe this is due to operating system level behaviors such as caching and other optimizing behaviors. While this behavior is an important consideration for mobile app performance engineering, it makes it impossible to do analysis at the resolution we desire. After some probative experiments, we found that the performance data gathered from the apps stabilizes after 5 runs. So, to improve the consistency of our experiments, we begin recording performance data after the script has already run the app and interaction set 5 times. Therefore, our data is not reflective of the apps' performance on a completely "cold" run, an important distinction. It is possible that one of the frameworks is capable of more aggressive run-time optimization and therefore, our comparison may not be totally fair. More work is needed to determine whether this effect is uniform across app frameworks and whether our results hold for "cold" runs.

Limited Ecosystem Coverage

Clearly, we tested only a very small subset of the application development ecosystem. We chose to test the list view and the map view elements because they are commonly used in Android applications and they have non-trivial resource requirements. Testing two commonly used elements, clearly, does not give us enough evidence to claim that our results hold for the entire software ecosystem around React-Native. Our experiments merely help to support what most of the development community already believe: there are significant performance implications of switching to React-Native from native Android of which developers working on performance-sensitive apps should be aware. In particular, We tested a ListView in our

Property Finder experiment. The React-Native documentation [13] offers some alternatives to ListView such as FlatList or SectionList. We did not test these alternative which could have improved the performance of our app.

React-Native can Utilize Native Code

For the apps that we wrote, we used only JavaScript (non-native) code for our React-Native application. However, one of React-Native's most important features is that developers can insert native code directly into their applications wherever desired. This is to allow developers to recycle UI elements they've designed for use in native apps for their React-Native application. This could also mitigate some of the performance costs of React-Native apps as intensive operations or those involving real-time feedback could be implemented natively and then wrapped in JavaScript for use by a React-Native app. However, it is not clear whether this would actually improve performance or not. Introducing native code to the process could introduce other overhead, canceling out performance gains. More work is required to determine whether this interesting feature of React-Native could be used to improve performance wherever necessary.

CPU Utilization as a Proxy for App Performance

CPU utilization is an important performance metric to consider, but it is far from the only one. The fact that one app has greater CPU requirements than another does not imply that the user experience is any different. Indeed, one of React-Native's advertised design properties is that it processes in two threads, a JavaScript interpretation thread and a "native" UI thread, shifting much of the work off of the UI thread keeping the user experience smooth. Increased CPU utilization is directly indicative of increased battery usage, however. Our experiments do not measure interface latency or frame rate, both important performance characteristics that directly impact user experiences. CPU utilization is directly related to UI latency in that when UI updates require many background steps, CPU utilization will be elevated, but the relationship is not linear. Further, background operations carried out by the app that do not affect the user experience whatsoever can cause elevated CPU utilization. Therefore, CPU utilization, although nonetheless a useful performance metric, is a poor proxy for user experience.

Averaging CPU Data

Despite our efforts to achieve consistency in our trials, factors beyond our control such as changes in battery voltage, ambient temperature, and operating system behaviors such as background services do not allow us to collect fully controlled samples. These factors can cause the app, or our testing script, to take slightly different amounts of time for each run. It is for this reason that we average together data from several runs. However, this averaging process has caveats. Averaging several trials before looking at the results may hide important details in our data regarding events that do not occur during specific time intervals. Averaged data can also mislead a reader about the shape of the underlying curves. For example, if a sharp peak appears in every sample, but at sufficiently disparate points in time, the peaks will average together to form a shallow and wide rise, obscuring the underlying phenomenon. Such features, below a resolution of about 1 second are difficult to distinguish visually in plots of individual runs as the CPU data is very noisy. In fact, there are many individual data points where CPU utilization is at 100% but when averaged, our points never exceed 80%. Therefore, even though we collect CPU data 10 times per second, we would be mistaken to identify features less than one second in duration as true features.

FUTURE WORK

2.1 IOS Performance

Our investigation considers only apps that run on Android, but our experiments ignore a large chunk of the mobile application marketplace, IOS. As we mention in *Motivation*, others have performed similar experiments on IOS, but without the rigor and scope of our investigation. Performance on both platforms is an important consideration for developers considering switching to React-Native, so data on this side of the market would be helpful.

2.2 Considering other Performance Metrics

We only looked at CPU and Memory measurements from devices, but other metrics exist that may give insight. Metrics such as GPU, frame rates and battery consumption could be explored. The device's GPU takes off some load from the CPU and is crucial in modern mobile applications that are graphics intensive especially mobile games. Frame rate, related to GPU usage, is the number of frames the phone is able to draw per second. Sometimes, this is caused by GPU overload, but more often low frame rates in mobile apps are a sign of inefficient view updates, and big drops in frame rate could indicate other problems with memory management or threading issues. Measuring battery consumption would tell us how much power an app consumes during various user interactions and how efficiently an app can run in the background, an important factor for service apps that need to efficiently idle.

2.3 Xamarin

We compared two JavaScript based cross-platform frameworks, but others exist. We are also interested in measuring app performance on other platforms. One very popular framework, Xamarin, allows development of apps for Android and IOS using C# and XML and is very popular for cross-platform development, as it allows apps to be developed for IOS, Android, and Windows devices, and the ecosystem is rather mature. React-Native is becoming more versatile as time goes on, with Microsoft introducing a library[31] that allows development of Universal Windows Platform (UWP) apps with React-Native. It would be interesting to study and compare Xamarin against React-Native and Ionic on a wider variety of platforms.

CONCLUSIONS

Whether developers should be willing to sacrifice some CPU efficiency for the convenience of writing a cross-platform application in JavaScript depends entirely on the demands of the application's target market and the developer's means. Cross-platform tools can certainly reduce the effort necessary to produce a high-quality app and our data show conclusively that React-Native offers certain advantages over webview based cross-platform tools, in terms of CPU

efficiency. However, in many regards React-Native seems to offer no performance improvement over Ionic apps. This runs contrary to our expectations and Facebook's stated aim for React-Native to replace other cross-platform tools as a more efficient framework. Certainly, for a heavily CPU bound application where performance and battery efficiency are priorities, developing with React-Native does have performance trade-offs that should be considered against the convenience of platform unification.

REFERENCES

[1] [n. d.]. Laksh47/uwaterloo_icpe: Scripts to replicate our work submitted to ICPE conference, mumbai 2019. https://github.com/Laksh47/uwaterloo_icpe. (Accessed on 10/13/2018).

[2] Christine Abernathy. [n. d.]. First react native Application. https://goo.gl/4M8uXb

[3] AlexSoft. [n. d.]. Good and bad of React-Native. https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-reactjs-and-react-native/

[4] Apache. [n. d.]. Apache Cordova. https://cordova.apache.org/. (Accessed on 09/13/2018).

[5] Apple. [n. d.]. Apple developer tools. https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/InstrumentsUserGuide/index.html

[6] Filip Asp. 2018. A comparison of Ionic 2 versus React Native and Android in terms of performance, by comparing the performance of applications.

[7] Sören Blom, Matthias Book, Volker Gruhn, Ruslan Hrushchak, and André Köhler. 2008. Write once, run anywhere a survey of mobile runtime environments. In *Grid and Pervasive Computing Workshops, 2008. GPC Workshops' 08. The 3rd International Conference on*. IEEE, 132–137.

[8] Peter J. Brockwell and Richard A. Davis. 1987. Stationary Time Series. 1–41. https://doi.org/10.1007/978-1-4899-0004-3_1

[9] Cory Buckley. [n. d.]. Cordova webview. https://www.pop-art.com/hybrid-mobile-applications-is-webview-or-native-right-for-me/

[10] Calderaio. 2017. Performance of react native apps on IOS. https://goo.gl/G2aeZj

[11] Isabelle Dalmasso, Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. 2013. Survey, comparison and evaluation of cross platform mobile application development tools. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*. IEEE, 323–328.

[12] Ariel Elkin. [n. d.]. Why I'm not a React Native Developer. https://goo.gl/nL6JwT

[13] Facebook. [n. d.]. Performance Â· React Native. https://facebook.github.io/react-native/docs/performance. (Accessed on 09/18/2018).

[14] Facebook. [n. d.]. A React Native app is a real mobile app. http://facebook.github.io/react-native/

[15] Facebook. [n. d.]. React-native Release. https://code.fb.com/ai-research/f8-big-technology-bets-and-open-source-announcements/

[16] Facebook. [n. d.]. Who's using React Native? https://facebook.github.io/react-native/showcase.html

[17] Wayne A. Fuller and Wiley InterScience (Online service). 1996. *Introduction to statistical time series*. Wiley.

[18] GitHub. [n. d.]. The fifteen most popular languages on GitHub. https://octoverse.github.com/

[19] Google. [n. d.]. Android code samples. https://developer.android.com/samples/index.html

[20] Google. [n. d.]. Android maps application. https://developers.google.com/maps/documentation/android-api/map-with-marker

[21] Niclas Hansson and Tomas Vidhall. 2016. Effects on performance and usability for cross-platform application development using React Native.

[22] Marcel Harkema, Dick Quartel, BMM Gijsen, and Robert D van der Mei. 2002. Performance monitoring of Java applications. In *Proceedings of the 3rd international workshop on Software and performance*. ACM, 114–127.

[23] Henning Heitkötter, Sebastian Hanschke, and Tim A Majchrzak. 2012. Evaluating cross-platform development approaches for mobile applications. In *International Conference on Web Information Systems and Technologies*. Springer, 120–138.

[24] Appcelerator Inc. [n. d.]. Home - Appcelerator | The Mobile First Platform. https://www.appcelerator.com/. (Accessed on 09/13/2018).

[25] Ionic. [n. d.]. Ionic developer community. https://ionicframework.com/community

[26] Ionic. [n. d.]. Ionic webview. https://ionicframework.com/docs/wkwebview/

[27] Ionic. [n. d.]. What is Ionic. https://ionicframework.com/what-is-ionic

[28] Marcin Kornek. [n. d.]. Why You Should Migrate Your App From Ionic, Cordova, or PhoneGap to React Native. https://goo.gl/akBTdM

[29] Frederic Lardinois. [n. d.]. Facebook Open-Sources React Native. https://goo.gl/Mf1ri4

[30] Eddie Lockhart. [n. d.]. Build apps once with cross-platform mobile development tools. https://goo.gl/akBTdM

[31] Microsoft. [n. d.]. Microsoft/react-native-windows: A framework for building native UWP and WPF apps with React. https://github.com/Microsoft/react-native-windows. (Accessed on 10/01/2018).

[32] Vytenis Narusis. [n. d.]. Good and bad of React-Native. https://www.devbridge.com/articles/pros-cons-of-react-native-crash-course/

[33] Eric Nograles. [n. d.]. React Native: The Good, The Bad, and The Ugly. https://goo.gl/JS6vDs

[34] Qualcomm. [n. d.]. Trepn Profiler. https://developer.qualcomm.com/software/trepn-power-profiler

[35] Reddit:FoodieAdvice. [n. d.]. Is react-native dependency hell real. https://goo.gl/p9Sg6X

[36] ReplayKit. [n. d.]. Replay kit tool to record and play user interactions. https://seattle.github.com/appetizerio/replaykit

[37] Ruben Smeets and Kris Aerts. 2016. Trends in Web Based Cross Platform Technologies. *International Journal of Computer Science and Mobile Computing* 5, 6 (2016), 190–199.

[38] Qualcomm Technologies. 2014. Trepn Profiler Starter Edition: User Guide. (2014), 1–49. https://developer.qualcomm.com/qfile/28788/trepn_6.0s_starter_edition_user_guide.pdf

[39] Technopedia. [n. d.]. What is Jank? - Definition from Techopedia. https://www.techopedia.com/definition/29607/jank-web-development. (Accessed on 09/13/2018).

[40] Peter Whittle. 1951. *Hypothesis testing in time series analysis*. Almqvist & Wiksells boktr., Uppsala. http://www.worldcat.org/title/hypothesis-testing-in-time-series-analysis/oclc/22153644

[41] Michiel Willocx, Jan Vossaert, and Vincent Naessens. 2015. A quantitative assessment of performance in mobile app development tools. In *2015 IEEE International Conference on Mobile Services (MS)*. IEEE, 454–461.

[42] Spyros Xanthopoulos and Stelios Xinogalos. 2013. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*. ACM, 213–220.

[43] Carly Yuk. [n. d.]. Increased demand for mobile app development skills. https://goo.gl/4hwgoC

APPENDIX

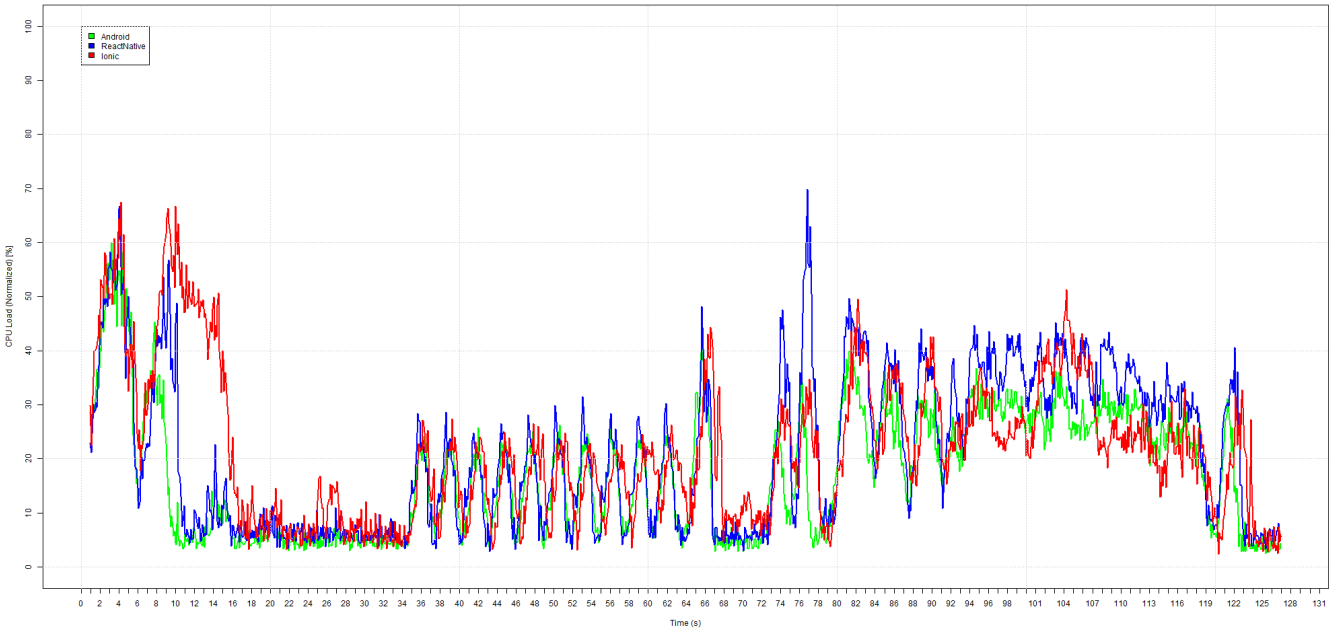

Figure 8: Observed Mean in CPU utilization for the Property Finder experiment on the Motorola G Play

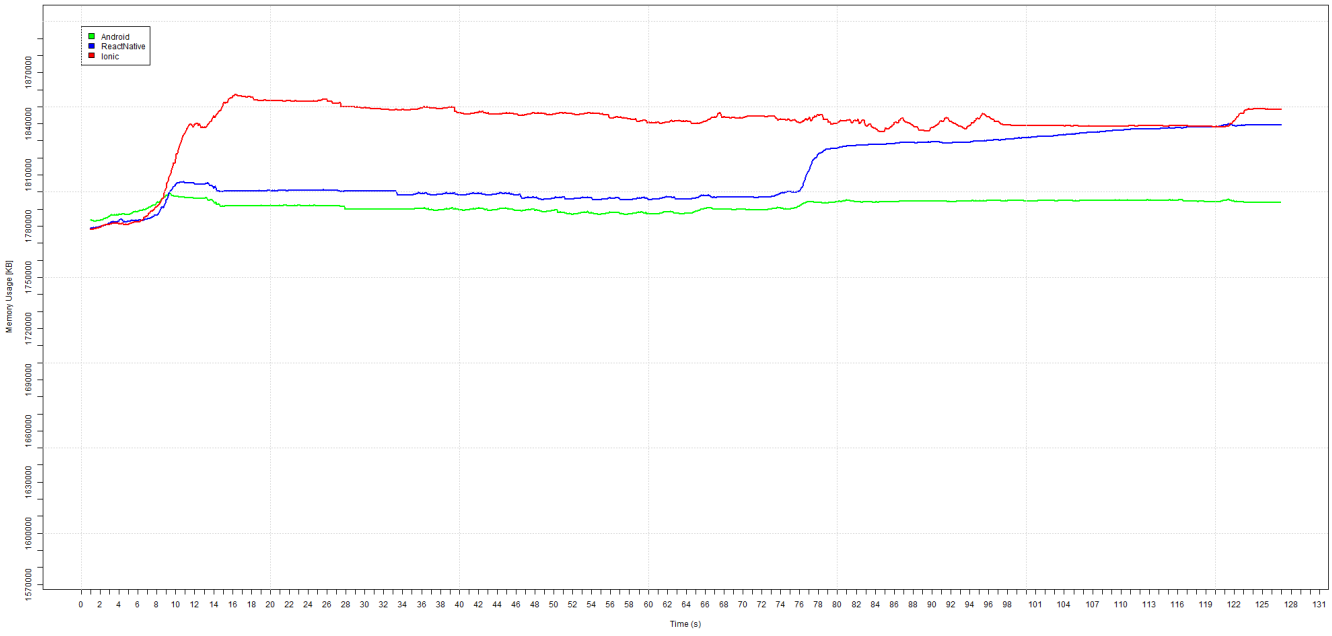

Figure 9: Observed Mean in Memory utilization for the Property Finder experiment on the Motorola G Play

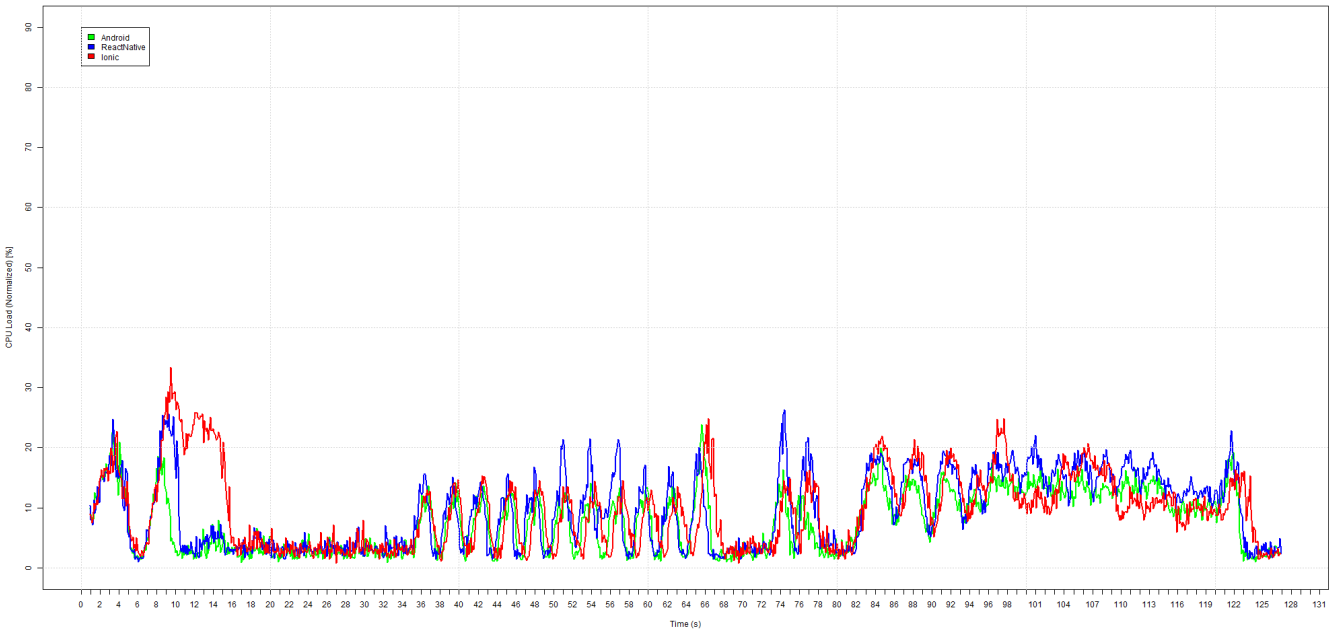

Figure 10: Observed Mean in CPU utilization for the Property Finder experiment on the LG Nexus 5

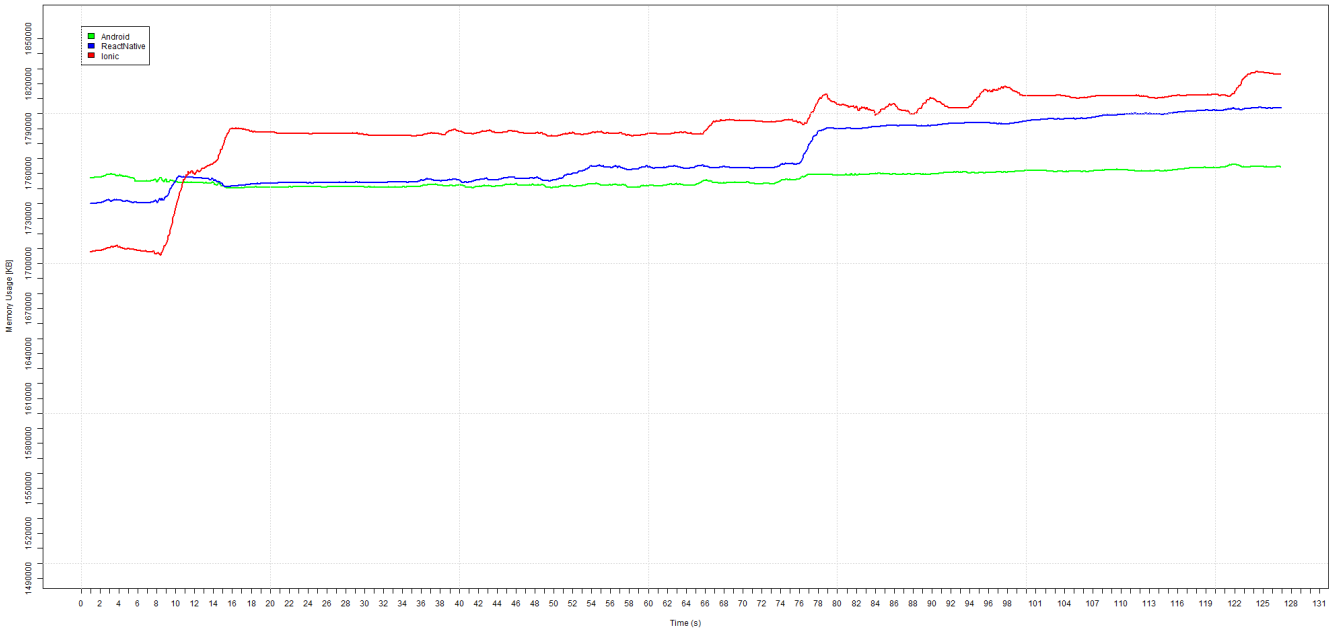

Figure 11: Observed Mean in Memory utilization for the Property Finder experiment on the LG Nexus 5

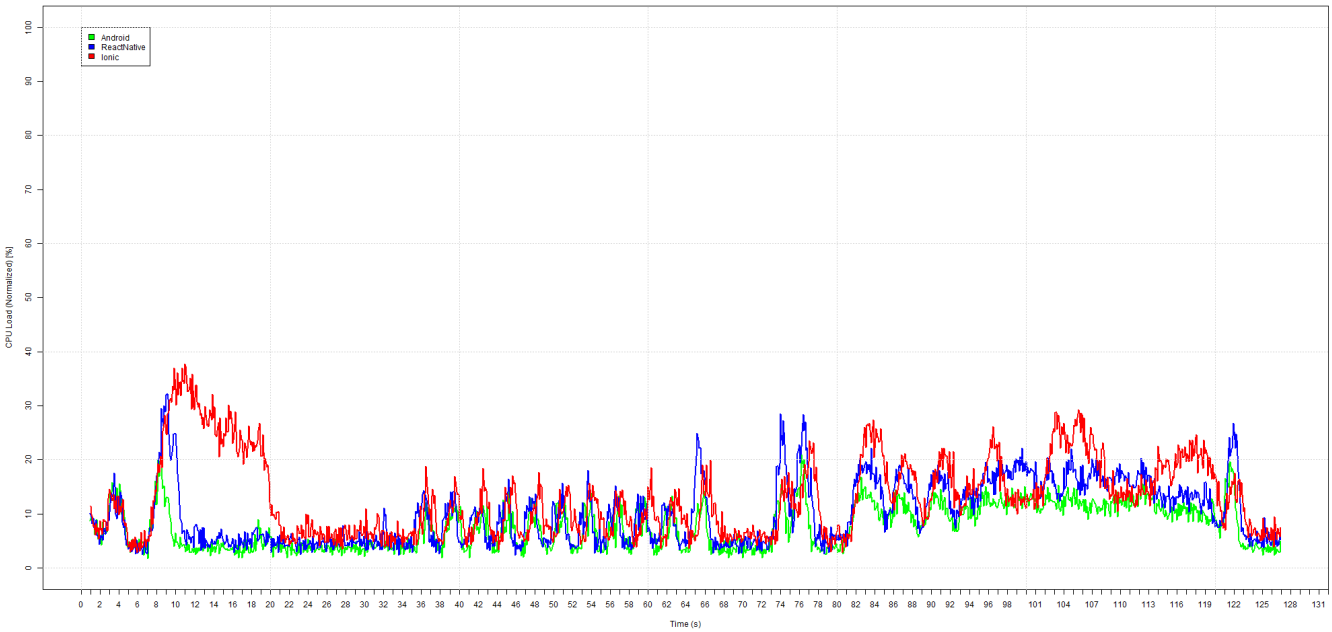

Figure 12: Observed Mean in CPU utilization for the Property Finder experiment on the Samsung Galaxy Note 2

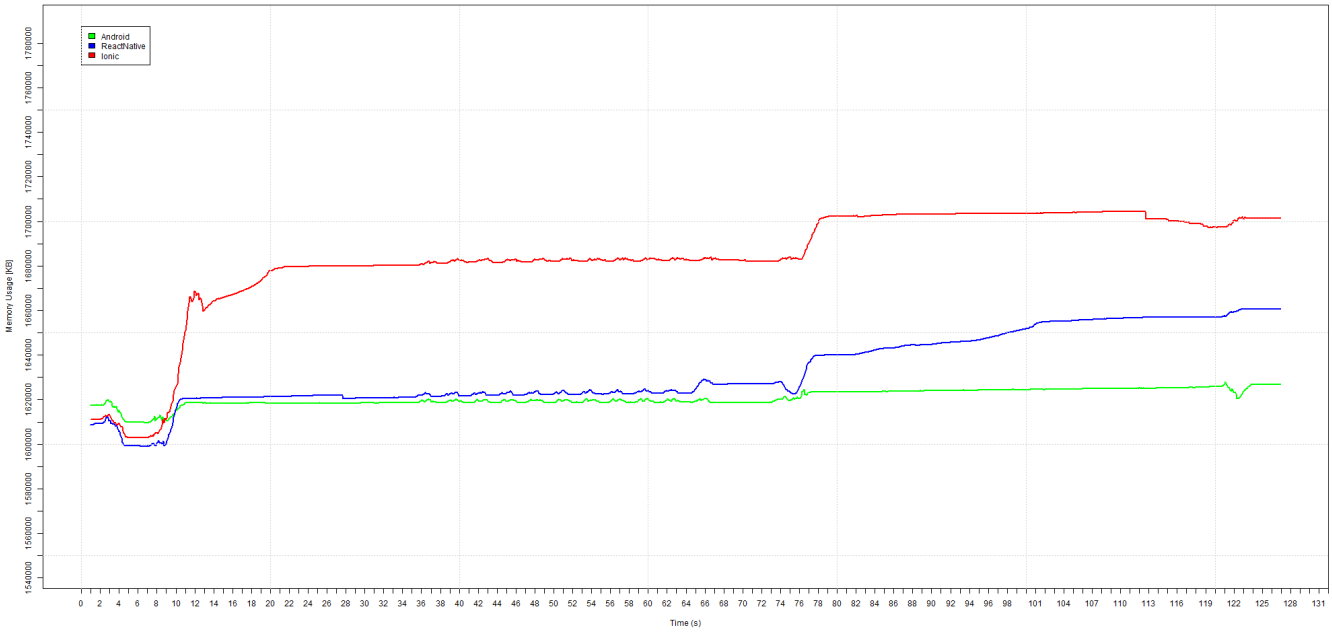

Figure 13: Observed Mean in Memory utilization for the Property Finder experiment on the Samsung Galaxy Note 2

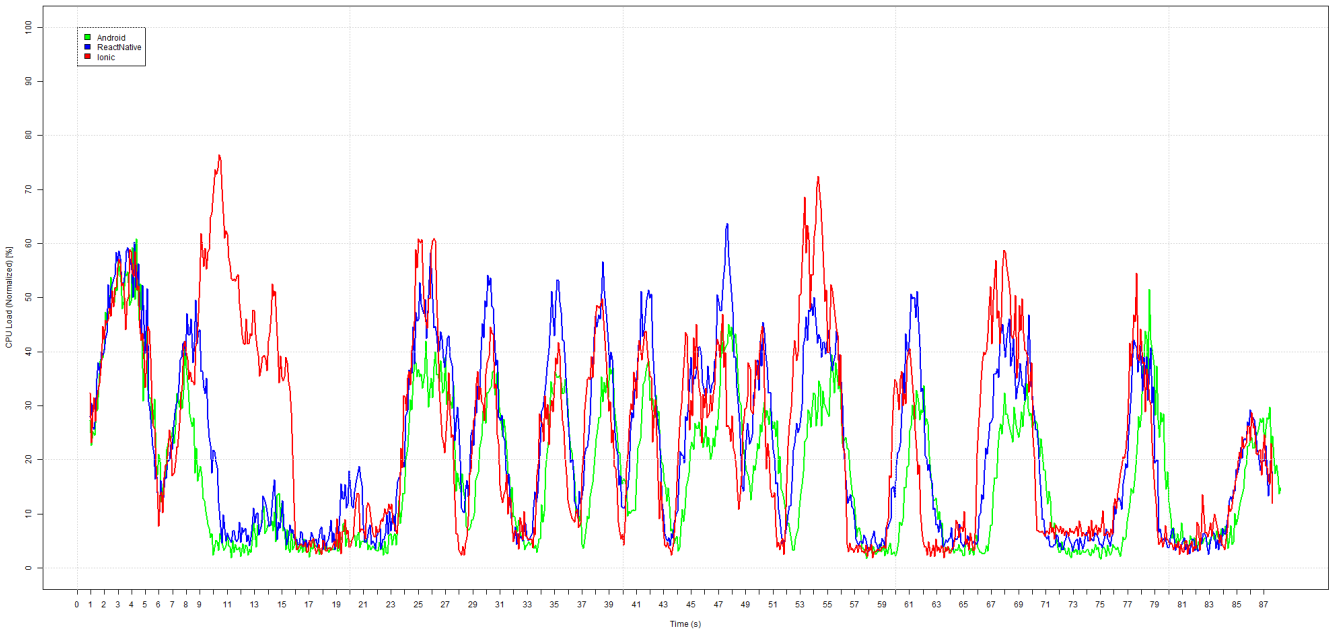

Figure 14: Observed Mean in CPU utilization for the Maps experiment on the Motorola G Play

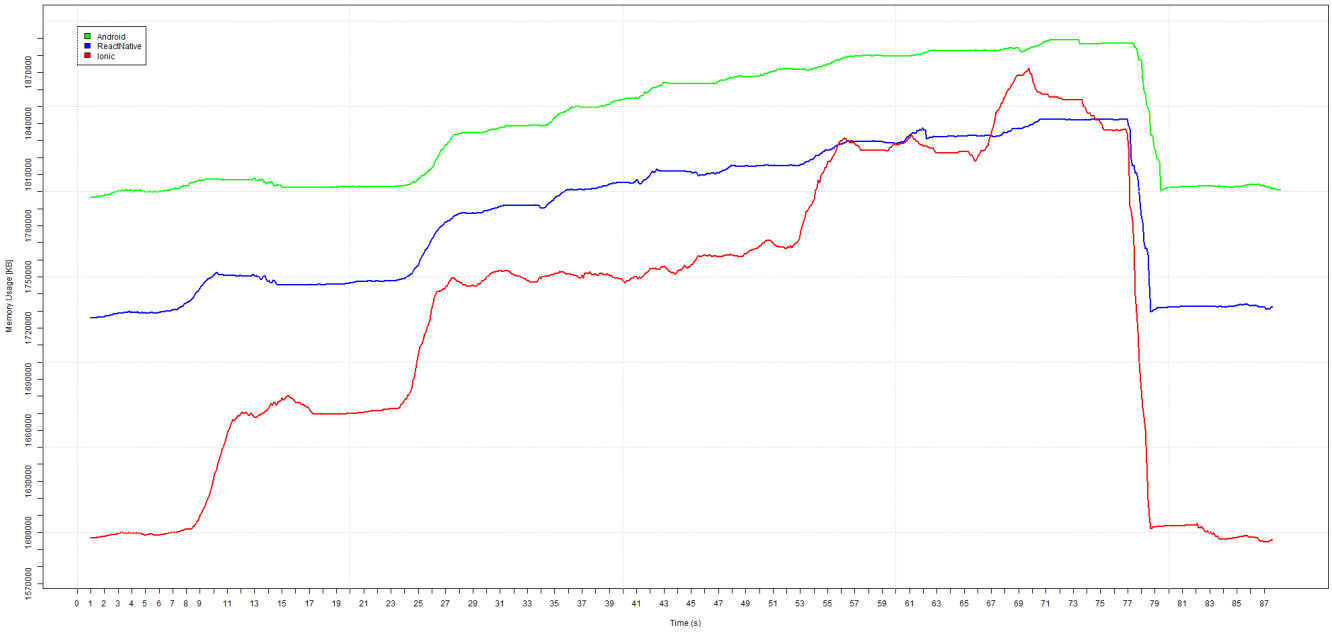

Figure 15: Observed Mean in Memory utilization for the Maps experiment on the Motorola G Play

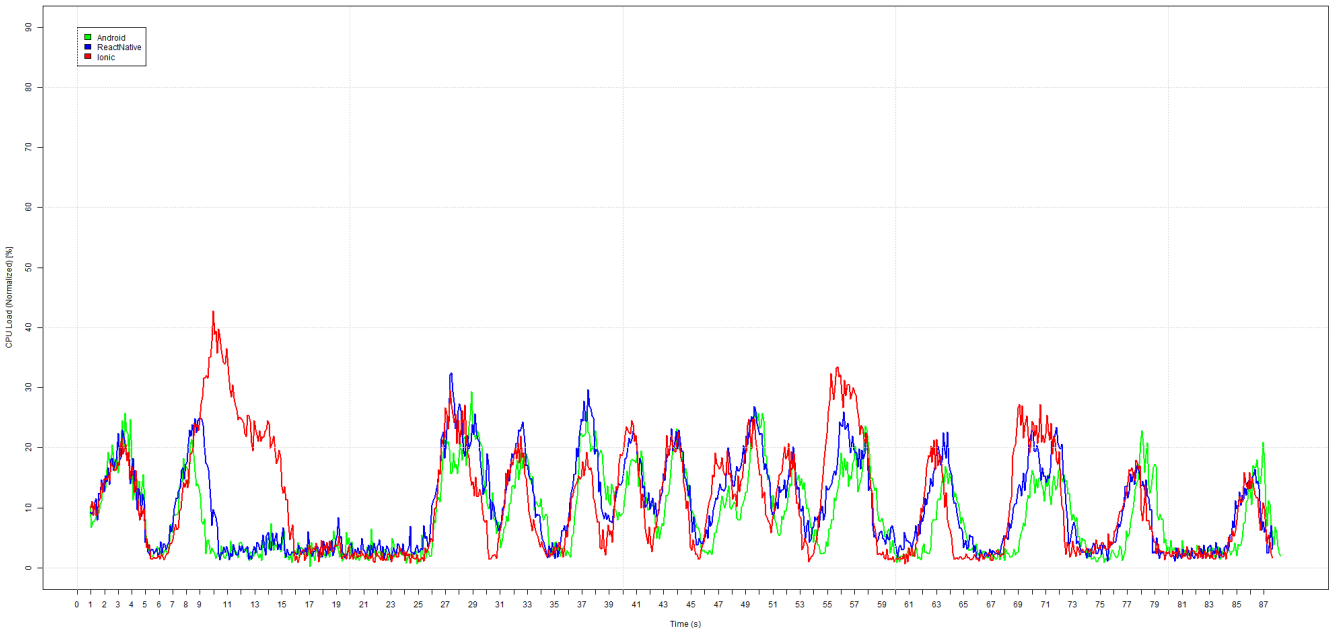

Figure 16: Observed Mean in CPU utilization for the Maps experiment on the LG Nexus 5

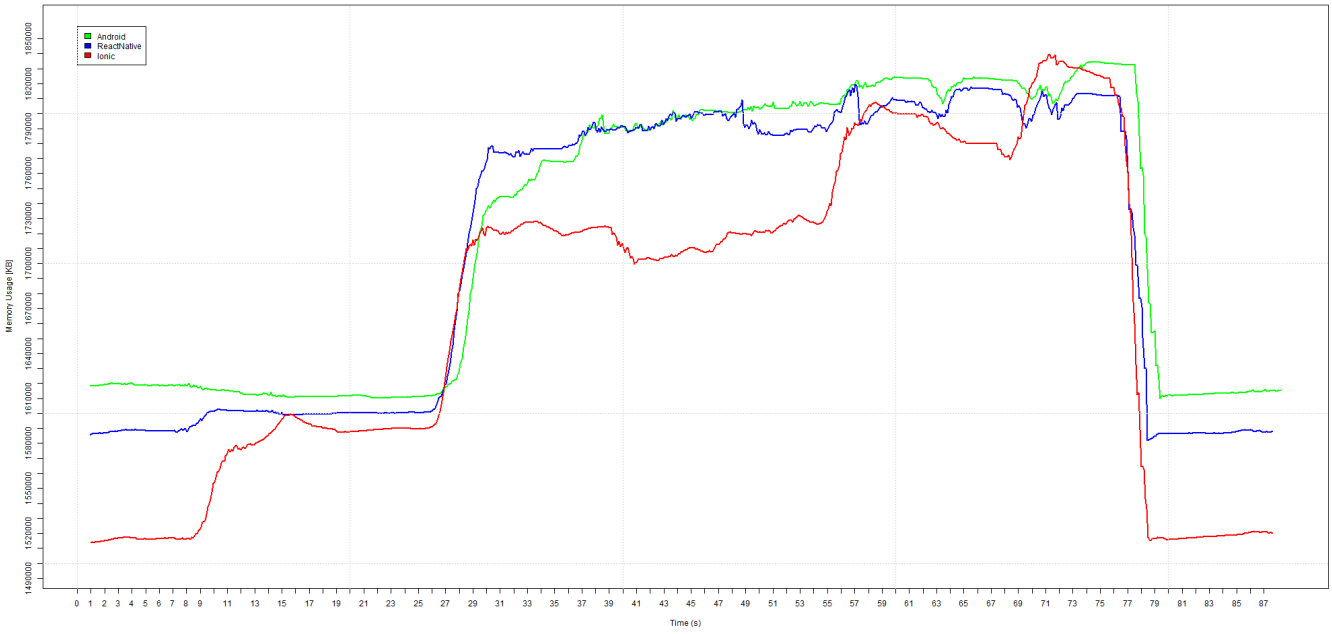

Figure 17: Observed Mean in Memory utilization for the Maps experiment on the LG Nexus 5

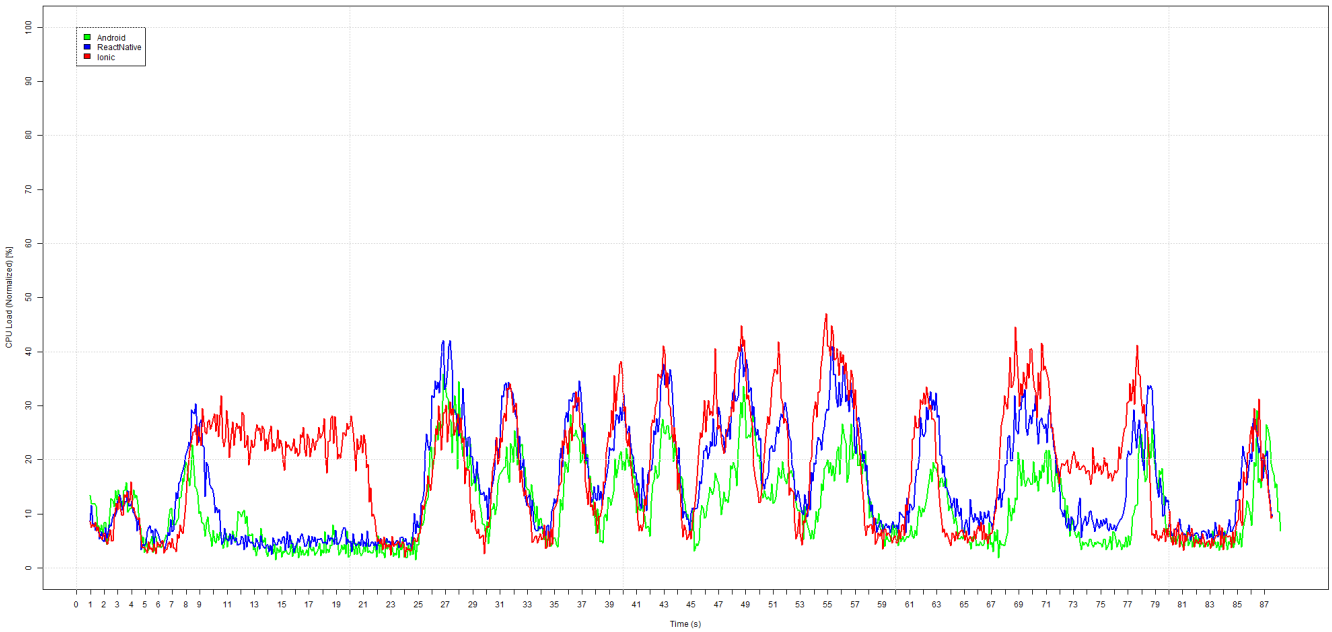

Figure 18: Observed Mean in CPU utilization for the Maps experiment on the Samsung Galaxy Note 2

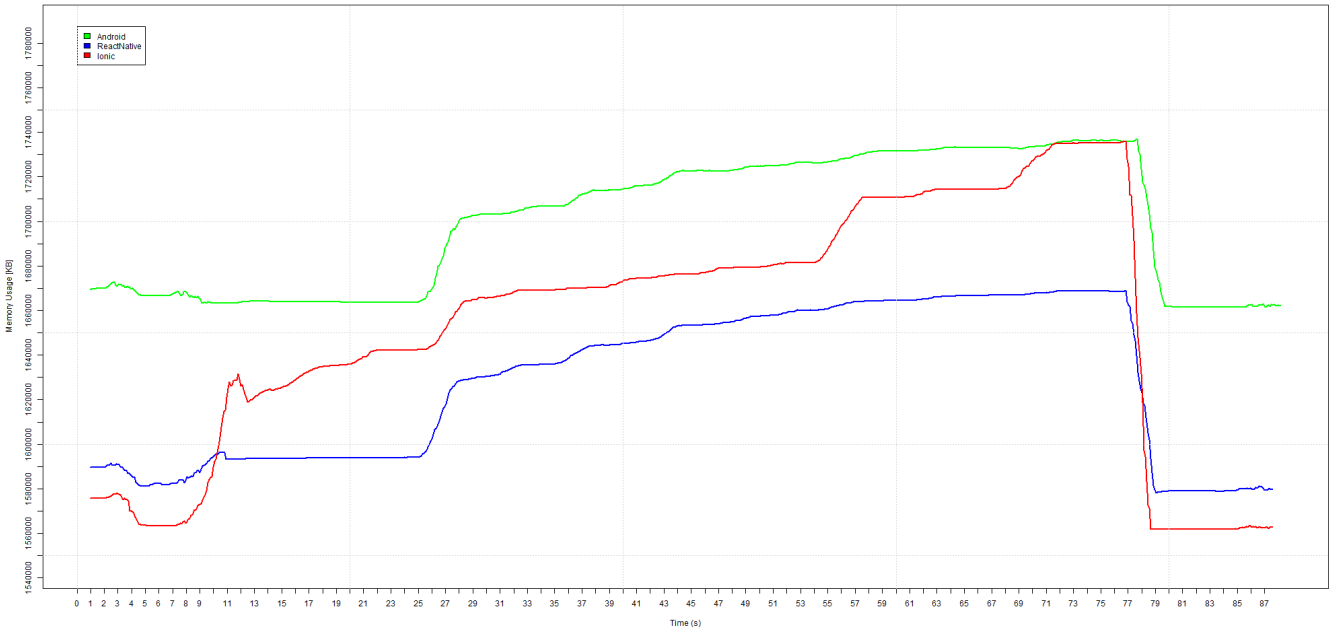

Figure 19: Observed Mean in Memory utilization for the Maps experiment on the Samsung Galaxy Note 2

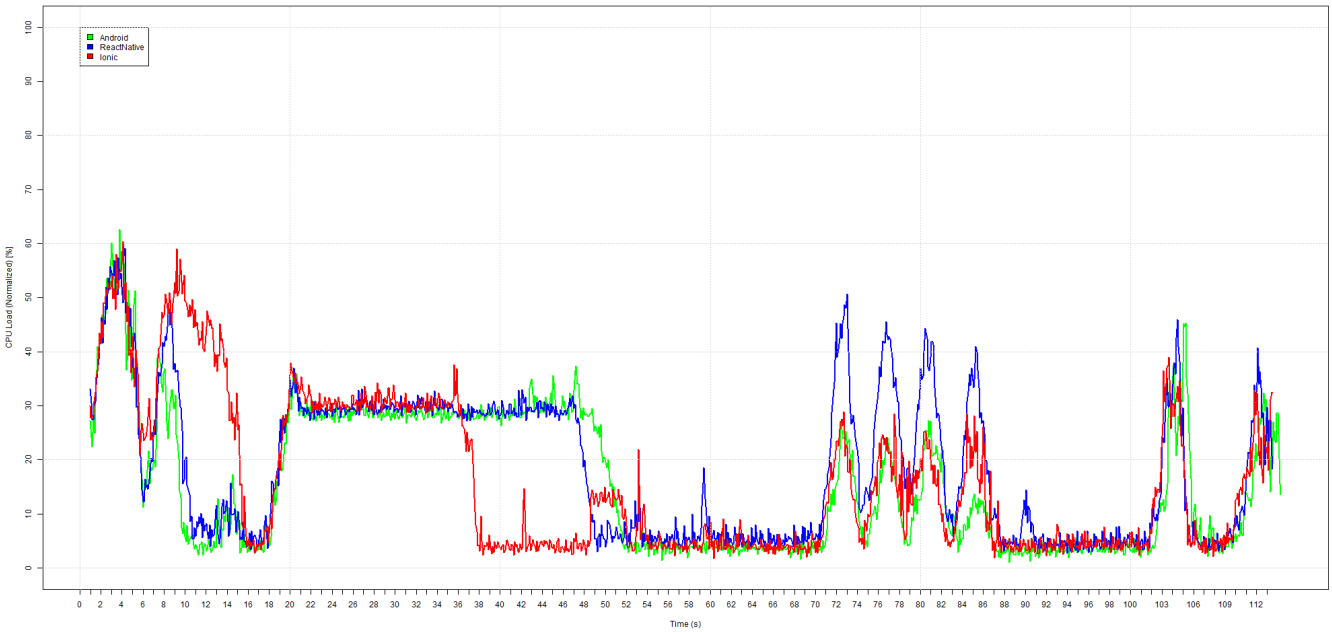

Figure 20: Observed Mean in CPU utilization for the Matrix experiment on the Motorola G Play

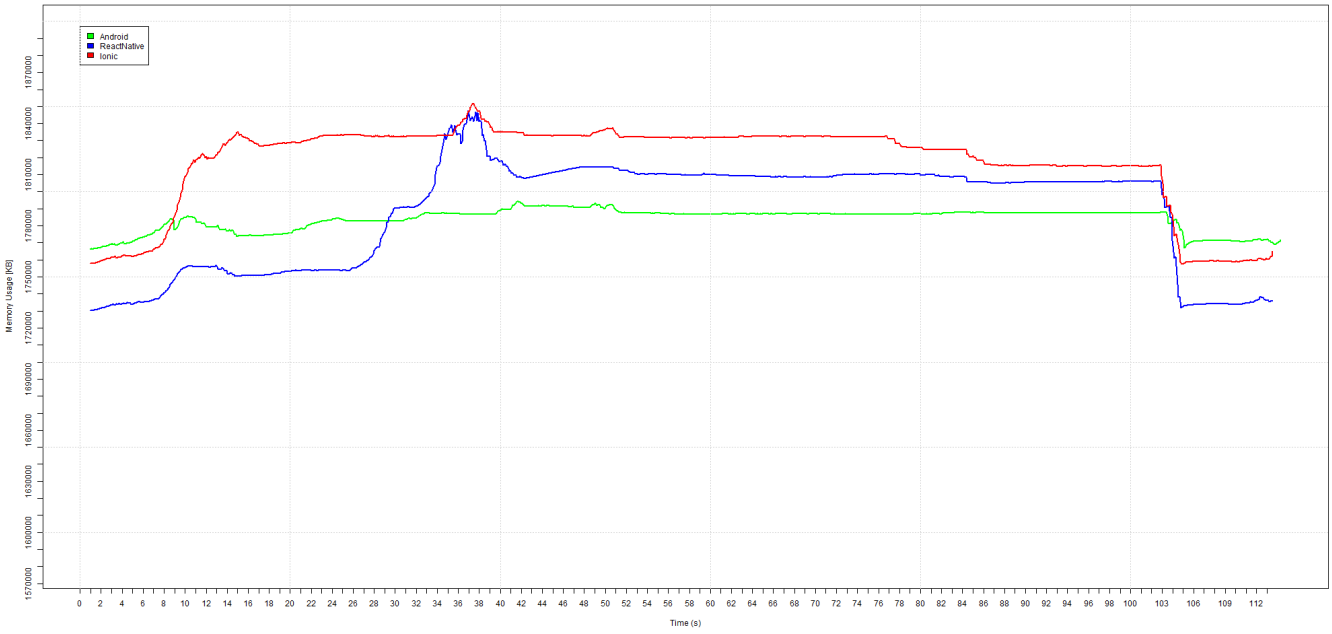

Figure 21: Observed Mean in Memory utilization for the Matrix experiment on the Motorola G Play

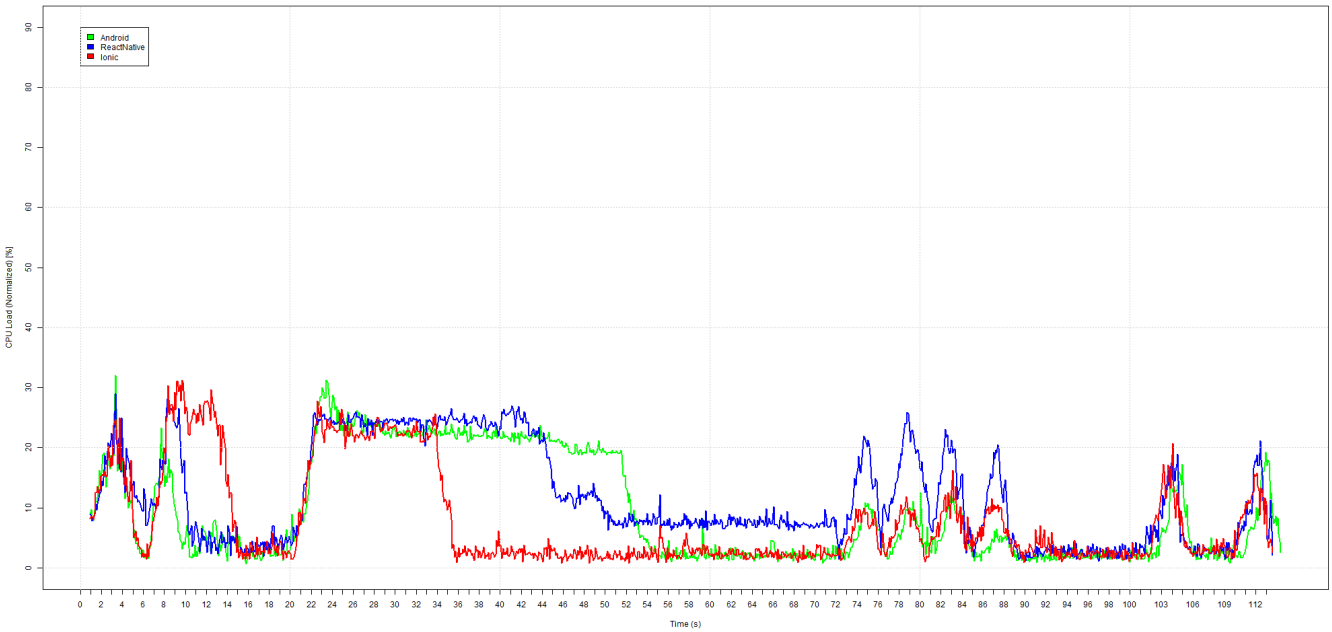

Figure 22: Observed Mean in CPU utilization for the Matrix experiment on the LG Nexus 5

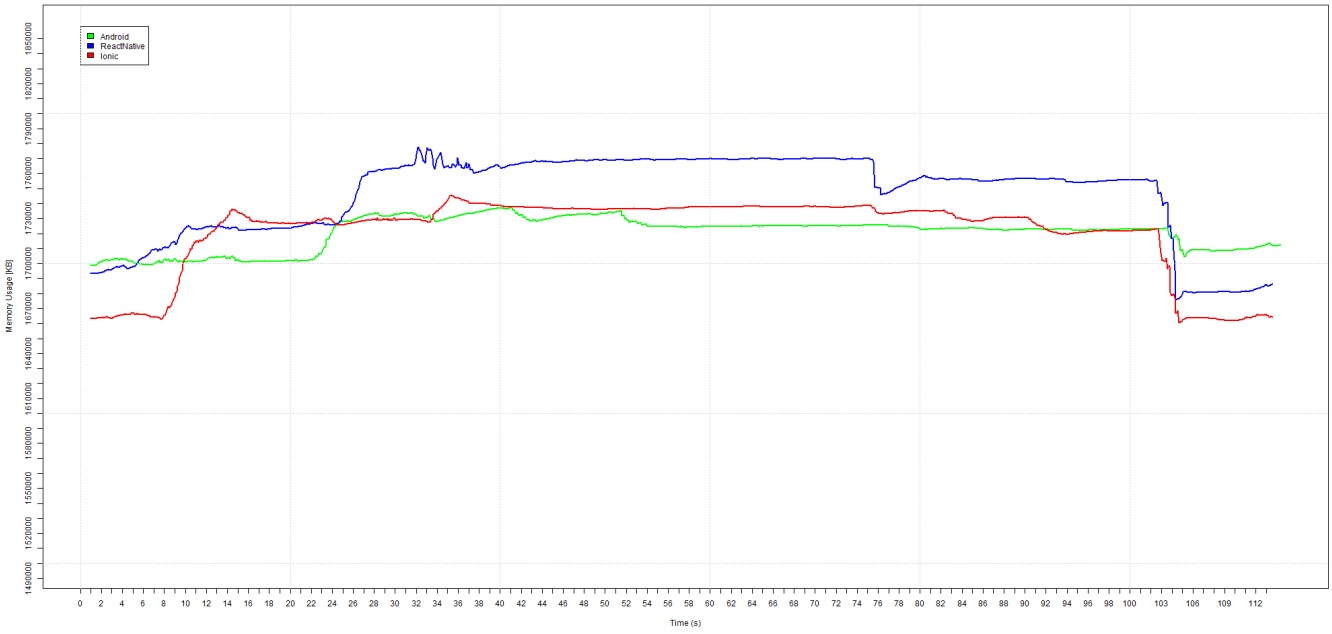

Figure 23: Observed Mean in Memory utilization for the Matrix experiment on the LG Nexus 5

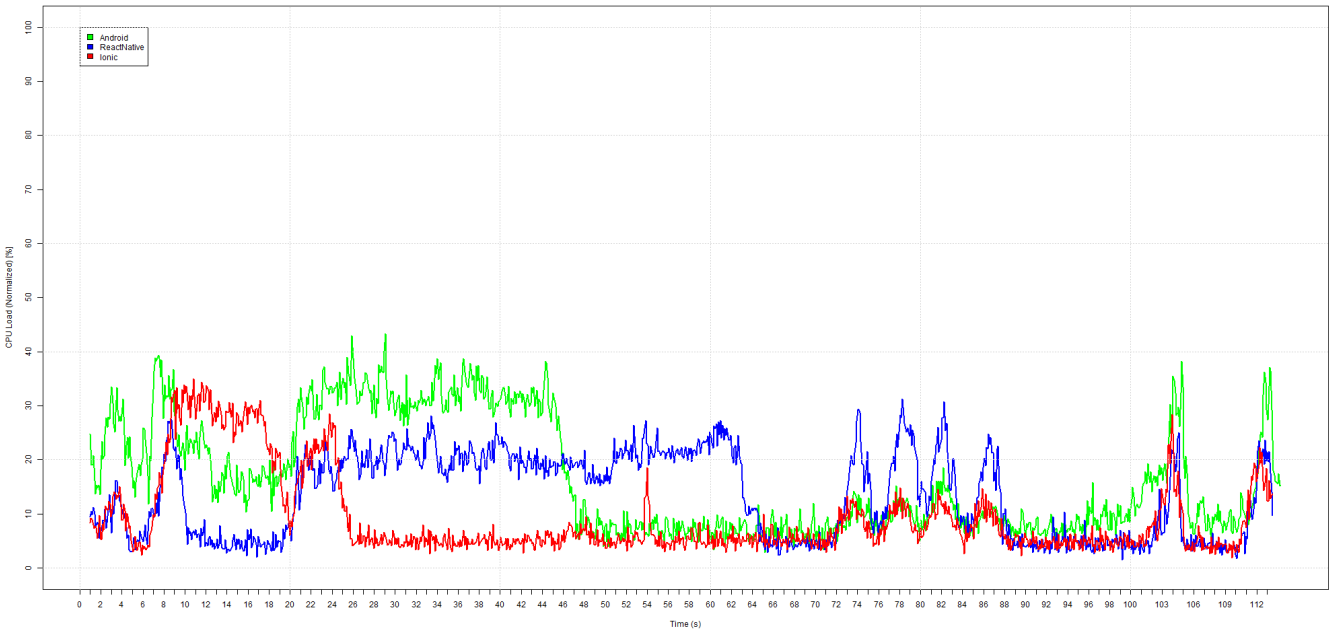

Figure 24: Observed Mean in CPU utilization for the Matrix experiment on the Samsung Galaxy Note 2

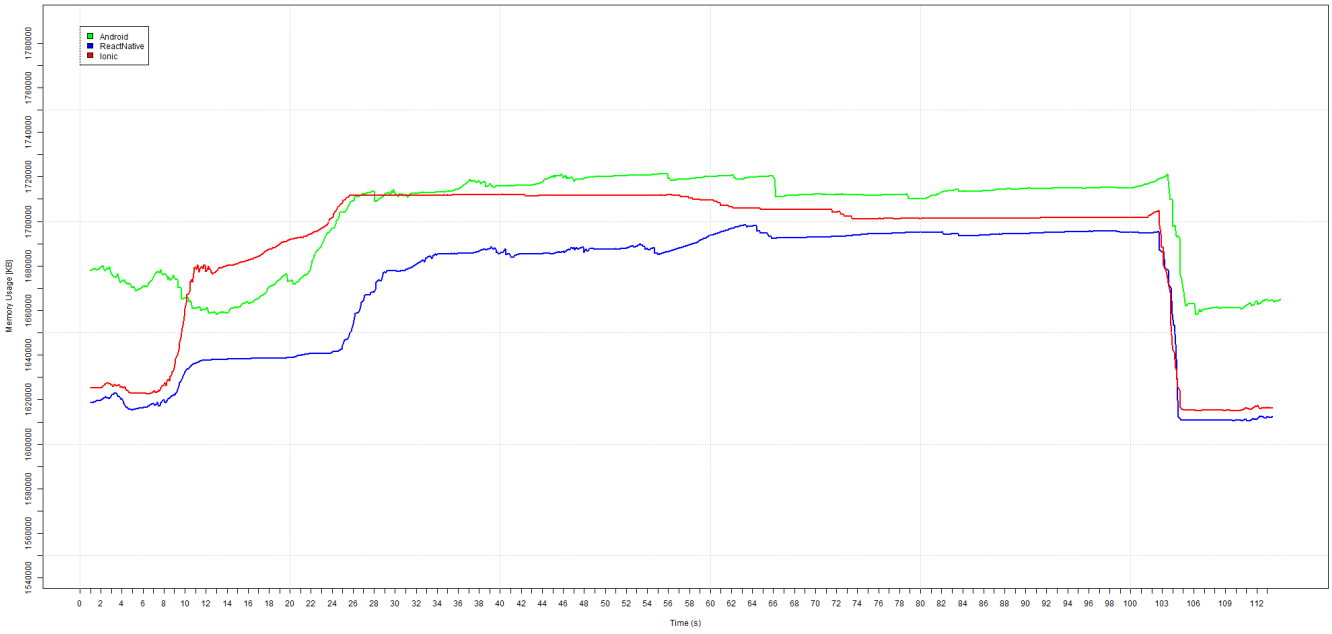

Figure 25: Observed Mean in Memory utilization for the Matrix experiment on the Samsung Galaxy Note 2