A Study of Non-Functional Requirements in Apps for Mobile Devices

Leonardo Corbalán[1,2][0000-0001-9026-8059], Pablo Thomas[1][0000-0001-9861-987X], Lisandro Delía[1][0000-0003-0515-0609], Germán Cáseres[1][0000-0003-3032-7023], Juan Fernández Sosa[1][0000-0002-0482-3392], Fernando Tesone[1][0000-0001-9499-3127] and Patricia Pesado[1][0000-0003-0000-3482]

[1] Computer Science Research Institute LIDI (III-LIDI)*
School of Computer Science, National University of La Plata,
La Plata, Buenos Aires, Argentina
*Partner Center of the Scientific Research Agency of the Province of Buenos Aires (CICPBA)
{corbalan, pthomas, ldelia, gcaseres, jfernandez, ftesone, ppesado}@lidi.info.unlp.edu.ar

Abstract. Nowadays, no one questions the crucial role of Requirements Engineering in software systems development. Specifically, if apps are generated for execution on mobile devices, certain non-functional requirements become highly relevant. In this article, an experimental study on three non-functional requirements that are essential for the development of native and multi-platform mobile apps is detailed. These requirements are performance, energy consumption and storage space utilization.

Keywords: Native Mobile Apps, Multi-Platform Mobile Apps, Non-Functional Requirements.

1 Introduction

Not many years ago, Requirements Engineering was underestimated. Computer Science professionals considered design or coding as more challenging stages in software development. Currently, this situation has changed. Having the right requirements in early development stages considerably reduces the risk of problems appearing later on. In this context, and with the boom of mobile devices, software development is particularly conditioned by complying with certain requirements, some non-functional requirements in particular, that are critical in mobile apps.

In this article, the results obtained in [1] [2] and [3] are expanded.

This study is aimed at quantifying the impact of the development approach used on three of the most popular non-functional requirements in the area of apps for mobile devices: performance, energy consumption and use of storage space.

To choose cases for study, the multi-platform development classification proposed by Raj and Tolety in [4] and reviewed by Xanthopoulos et al. in [5] was considered.

[2] Corresponding author

The final authenticated version is available online at http://doi.org/10.1007/978-3-030-27713-0

This classification considers 4 categories: 1) mobile web approach, 2) hybrid approach, 3) interpreted approach and 4) cross-compilation approach.

The development approaches studied in this article were the native approach and the hybrid, interpreted, and cross-compilation multi-platform variations. The mobile web approach was excluded because a thorough analysis of that approach will be done in the future. The tests detailed in this article were run on the Android platform, which is the operating system that currently has the largest market share for mobile devices [6].

Even though the general design of the experiments revolves around the development approaches used, the tests had to be implemented using specific frameworks. There are several of these development frameworks for each of the approaches being considered. The ones chosen for this study are well known and very popular in the field. Based on these, six different analysis scenarios were defined: 1) Android SDK (native approach), 2) Cordova (hybrid approach), 3) Titanium (interpreted approach), 4) NativeScript (interpreted approach), 5) Xamarin (cross-compilation approach) and 6) Corona (cross-compilation approach). The same scenarios were used to test all 3 non-functional requirements being considered: performance, energy consumption and use of storage space.

It should be noted that the results presented in this paper are linked to the state of the art of the development framework used at the moment of carrying out the experiments and, therefore, could change in the future as these frameworks evolve.

The rest of this article is organized as follows: Chapter 2 discusses the issue of performance in mobile apps, Chapter 3 considers energy consumption, and Chapter 4 is devoted to the use of storage space. Finally, the conclusions and future lines of work are presented.

2 Performance

According to several quality standards, such as ISO/IEC 9126 and ISO/IEC 25010, an efficient performance (least possible processing time) is one of the attributes that any software application must meet [7-3]. This requirement gains significance in the context of mobile devices due to its direct relation to energy consumption [8], which is a critical aspect affecting battery autonomy.

User ratings in online app stores usually penalize low performance, generating negative publicity as a result [6]. Andre Charland and Brian Leroux identified processing time as one of the main issues to solve when developing multi-platform applications, and they stated that end users care about software quality and user experience [9].

Some authors have studied application performance based on the development approach used. The work done by Corral et al. [10] around native and hybrid apps (developed using Phonegap) for a version of the Android operating system can be mentioned. However, there are not many articles analyzing app performance based on the multi-platform development approaches mentioned in [4] and [5] that were used as reference to establish the cases for analysis for this study.

2.1 Experiment

Tests were carried out for the 6 scenarios mentioned above, which include the most relevant development frameworks at the time of writing this article. For the tests, 3 different mobile devices were used – two smartphones and a tablet, all three with Android as OS; they are identified as Device A, Device B and Device C.

- **Device A**: *smartphone*, brand: Motorola, model: Moto-G2, processor: Quad-core 1.2 GHz Cortex-A7, RAM 1GB Snapdragon 400, OS: Android 4.4.
- **Device B**: *smartphone*, brand: Samsung, model: S6, processor: Octa-core (4x2.1 GHz Cortex-A57 & 4x1.5 GHz Cortex-A53), RAM 3GB Exynos 7420 Octa, OS: Android 5.0.2.
- **Device C**: *tablet*, brand: Samsung, model: Tab 2, processor: Dual-core 1.0 GHz, RAM 1GB TI OMAP 4430, OS: Android 4.2.2.

A total of 18 test cases were defined – one per device per scenario.

To assess processing speed, the calculation of the following series was proposed:

$$serie = \sum_{j=1}^{5} \sum_{k=1}^{100000} (\log_2(k) + \frac{3k}{2j} + \sqrt{k} + k^{j-1}) \quad (1)$$

This expression includes several iterations, mathematical functions and f arithmetics. This type of calculation is frequent in applications that make intensive use of CPU computation power, such as games, augmented reality apps, image treatment apps, and so forth.

For each of the 18 test cases defined, 30 separate runs of the experiment were carried out, obtaining in each case a sample T, where T = T1, T2, … T30, and Ti = time required for calculating the series on the nth run of the experiment. Time Ti is expressed in milliseconds. In most of the practical cases of interest, 30 samples are enough to propose $\bar{T}$ as a good approximation to the real mean of the distribution. A large number of experimental works published in this field used this number of measurements for their data collection phases.

To characterize each of the samples obtained, statistic variables $\bar{T} = (1/n) \sum_{i=1}^{n} T_i$ and $S = \sqrt{(\frac{1}{n-1} \sum_{i=1}^{n} (T_i - \bar{T})^2)}$, corresponding to the sample average and sample standard deviation, respectively, were calculated.

2.2 Results

Table 1 and Figure 1 summarize the results obtained during the experiments. The values of $\bar{T}$ and S for the samples collected allow comparing the performance of the apps generated using the different development approaches in each of the devices used.

Clearly, for the three devices used for the tests, the apps generated with NativeScript, Titanium and Cordova were the most efficient ones, all of them completing the required calculation. On the opposite end, Xamarin, Android SDK and Corona always produced the slowest apps. Corona in particular stands out for its low performance.

Table 1. Processing time (ms) – Intensive calculation

Framework	Device A		Device B		Device C	
	$\bar{T}$	S	$\bar{T}$	S	$\bar{T}$	S
Android SDK	532.93	16.14	211.80	19.97	763.80	28.98
Cordova	230.33	14.22	85.77	8.83	190.60	9.36
Titanium	211.67	24.95	95.63	7.64	192.70	16.80
NativeScript	187.30	9.39	89.67	9.16	183.50	3.04
Xamarin	395.17	8.95	211.00	6.69	379.33	8.31
Corona	1401.73	12.60	600.53	5.95	1344.30	23.39

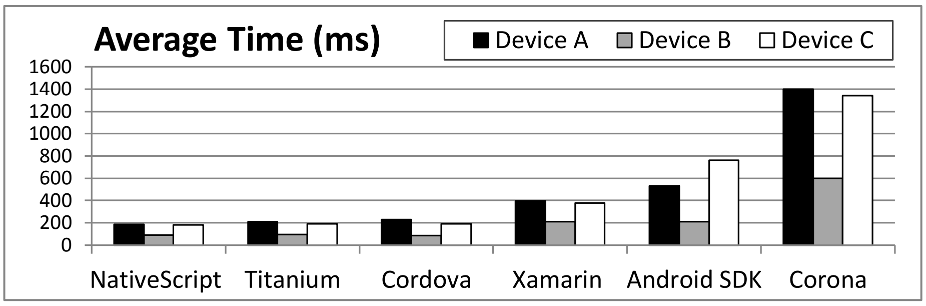

Fig. 1. Average processing time, in milliseconds.

It should be noted that the hybrid (Cordova) and interpreted (Titanium and NativeScrip) approaches, even if they operate differently, have one characteristic in common: they both run JavaScript code. In all these frameworks, the JavaScript V8 engine is responsible for optimizing the code that is then interpreted by a WebView. This engine has a crucial role and is largely responsible for the good results obtained. The tests for these approaches had a better behavior than that of the native approach (Android SDK) and the cross-compilation approach (Xamarin and Corona), which yielded the lowest performance.

The same relative differences were observed with the different development frameworks for all three devices used to carry out these tests.

3 Energy Consumption

Mobile device technology experienced a fast-paced development, significantly increasing its capabilities and performance, but also its energy requirements. Since battery technology has not evolved at the same speed, energy consumption has to be optimized to achieve a balance between performance and device autonomy.

Energy efficiency has become a relevant issue both for hardware manufacture as well as for software development. There is also a related requirement to protect the environment and general health of the planet. A higher energy consumption is against the current trend of *green computing*, which is attempting to achieve *eco-friendly* computer systems.

The introduction of ARM's big.LITTLE technology [11] to improve energy efficiency in mobile devices is an example of the commitment of hardware manufacturers with this issue [12]. However, the solution depends largely on software developers. It has been shown that, through changes in application source code, significant improvements can be obtained.

Many researchers have proposed solutions involving good programming practices. In [13], it was shown that the fastest algorithms are not always the ones that consume less energy. In [14] and [15], recommendations for the development of applications with a reduced energy demand were presented. In [16], it was concluded that most (5 out of 8) of the best programming practices published by Google to optimize Android app performance also had a positive impact on energy consumption. Other researchers explored the advantages of the technology called *Mobile Cloud Computing*, which integrates the concept of *cloud computing* to the mobile device environment. In [17] and [18], it was shown that this technology is effectively useful to save energy.

Energy efficiency and development *frameworks* for mobile devices have been widely studied separately; however, the articles considering both aspects simultaneously are scarce. This section is intended as a contribution in that direction, analyzing the effects of the development approach on app energy consumption. The 6 development frameworks mentioned in previous sections were considered, and 3 different types of common apps: 1) Intensive processing, 2) Video playback and 3) Audio playback.

3.1 Experiment

The platform chosen for testing was a medium-range smartphone – brand: Motorola, model: Moto-G2, processor: Quad-core 1.2 GHz Qualcomm Snapdragon 400, GPU: Adreno 305, RAM: 1GB, OS: Android 6.0. This device was selected as an average representation of all devices considered during a preliminary testing phase.

Intensive processing, video playback and audio playback apps were developed, each of them in six different versions – one for each development *framework* being considered. This resulted in a total of 18 test cases. The intensive processing app consisted in calculating the series used for performance analysis, discussed above, represented by Equation (1). The audio and video playback apps consisted in the playback of a one-minute long multimedia resource. In the case of the video, the file was 89.2 Mb in size, with a resolution of 1280x720 pixels, H.264 as codec at 5585 Kbps, and AAC audio tracks at 128 Kbps. For audio playback, the file used was 1.32 Mb in size and MP3 AC3 as codec, at 128 Kbps.

For energy consumption measurements, Qualcomm's *Trepn Profiler* tool was used; this is the same company that developed the smartphone processor used in all tests. To minimize external interference during the tests, a number of conditions were applied: 1) the device was on plane mode, 2) screen brightness was at the minimum level, 3) audio volume was set at 20%, 4) battery charge between 80% and 100%, 5) the device was not connected to the battery charger, 6) the app was implemented on dark mode, and 7) the app was running on the foreground, i.e., on the screen, during the test.

For each of the tests that were defined, 30 separate runs were executed, obtaining X = X_1, X_2, … X_{30} samples. In all cases, energy consumption, execution time and CPU percentage use were measured. Sample average $\bar{X} = (1/n)\sum_{i=1}^{n} X_i$ and sample standard deviation $S_X = \sqrt{(\frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2)}$ were calculated for all samples obtained.

3.2 Results

Table 2. Intensive processing app

Framework	Power (mWh)		CPU charge (%)		Duration (s)	
	$\bar{E}$	S_E	$\bar{C}$	S_C	$\bar{T}$	S_T
Cordova	1.597	0.136	35.924	2.571	8.467	0.679
Titanium	1.692	0.096	37.480	2.395	8.355	0.643
NativeScript	1.792	0.176	33.357	2.217	9.109	1.789
Xamarin	3.036	0.185	32.072	1.768	17.891	0.973
Android SDK	3.463	0.149	32.468	1.332	18.568	2.938
Corona	7.304	0.189	44.347	54.793	38.877	1.492

Table 3. Video playback app

Framework	Power (mWh)		CPU charge (%)		Duration (s)	
	$\bar{E}$	S_E	$\bar{C}$	S_C	$\bar{T}$	S_T
Android SDK	4.776	0.287	14.540	0.862	61.600	0.814
Corona	4.992	0.235	14.704	0.711	62.733	0.907
Xamarin	5.119	0.473	15.465	1.608	62.333	0.959
Titanium	5.262	0.502	15.204	1.643	63.633	1.033
NativeScript	11.112	1.590	17.839	2.210	63.333	1.295
Cordova	13.866	0.536	22.358	0.903	62.833	0.834

Table 4. Audio playback app

Framework	Power (mWh)		CPU charge (%)		Duration (s)	
	$\bar{E}$	S_E	$\bar{C}$	S_C	$\bar{T}$	S_T
Android SDK	3.920	0.291	10.497	0.882	64.033	0.999
Xamarin	4.010	0.201	10.592	0.613	64.967	1.098
Titanium	4.189	0.277	11.865	0.835	64.767	1.104
NativeScript	4.224	0.229	11.233	0.644	65.867	1.042
Cordova	4.288	0.191	11.473	0.487	65.733	1.388
Corona	5.194	0.387	14.680	1.080	64.800	1.031

Tables 2, 3 and 4 summarize test results for intensive processing, video playback and audio playback, respectively. The histograms in Figure 2 show sample distribution for energy consumption.

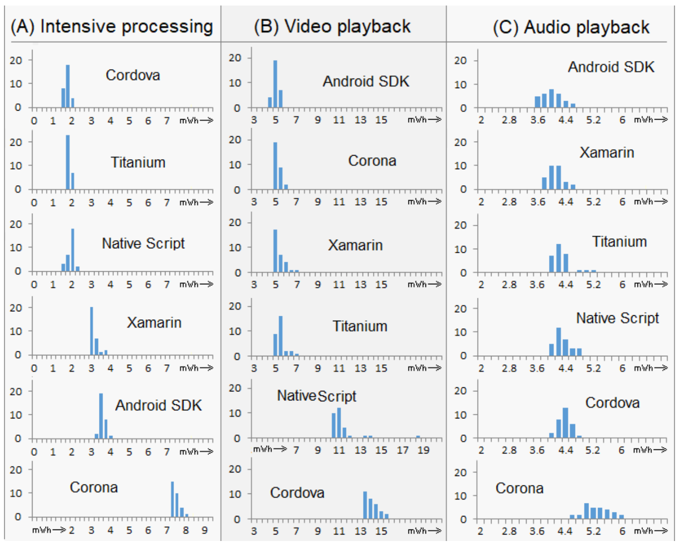

Fig. 2. Histograms representing test samples obtained.

As regards intensive processing, there are three clear groups of *frameworks* (see Table 2 and Figure 2 in Section A). The first group, with the highest energy efficiency, is formed by Cordova, Titanium and NativeScript. The second group, with medium efficiency, includes Xamarin and native Android SDK. Finally, with a much lower performance, Corona is in the group that has the greatest impact on battery autonomy. It is worth noting that the native development framework, Android SDK, is not among the most efficient approaches. This would be explained by the low performance of Java for mathematical functions in relation to execution time and, therefore, energy consumption.

As regards video playback apps, there are two clearly defined groups (see Table 3 and Figure 2 in Section B). The first group, with greater energy efficiency, includes Android SDK, Corona, Xamarin and Titanium. The second group is formed by NativeScript and Cordova. These two frameworks showed a really low efficiency level, requiring more than double the power than the other frameworks that were tested. In particular, Cordova consumes almost triple the power than Android SDK (the best option). This significant difference is probably due to the fact that Cordova uses the HTML video player, which requires more CPU power than what the operating system provides.

In relation to audio playback apps, there are no major differences in energy consumption among the various development approaches with the exception of Corona, which stands out as the least efficient of the bunch (see Table 4 and Figure 2 in Section C).

Even though all audio and video tests were done using a resource that was exactly 60 seconds long, tables 3 and 4 show differences in playback time. This is because the time required to start up the app is different for the different frameworks used. The native approach has advantages in this regard, but these become significant only if the app is used for a short time. As use time increases, the relevance of this advantage becomes less significant.

Figure 3 represents the values for $\overline{\boldsymbol{E}}$ (sample average) obtained for the 18 test cases run. A quick visual inspection tells us that only one or tow of the development frameworks analyzed stand out for being inefficient (high energy consumption) for all three types of apps considered. This is the case of Corona for intensive processing, Cordova and NativeScript for video playback, and Corona for audio playback. On the other side of the coin, the most efficient framework does not stand out clearly from the other frameworks that are also efficient. This indicates that, even if there is no clear winner as the most efficient option, there is enough information as regards which frameworks should be avoided in each case.

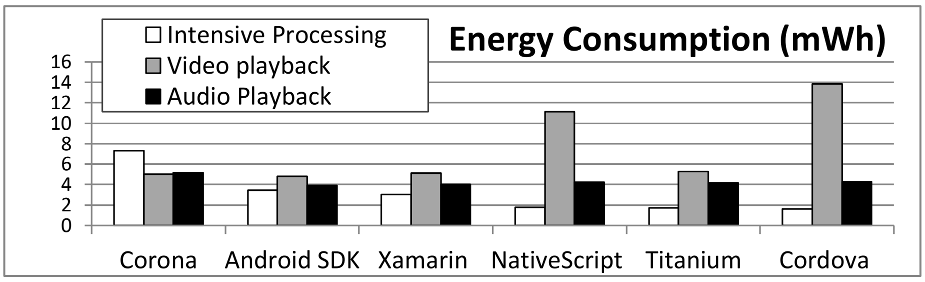

Fig. 3. Energy consumption by development framework used for the three types of apps considered.

Tables 2, 3 and 4 show that the impact of the development *framework* on energy consumption is greater in intensive processing apps (the consumption of the app developed with Corona is 4.57 times that of the app developed with Cordova). This is also significant in video playback apps (the consumption of the app developed with Cordova is 2.9 times that of the app developed with Android SDK). Finally, the smallest impact of the choice of development *framework* is found in audio playback apps, where the consumption of the development using Corona (the least efficient *framework*) is only 1.33 times that of the consumption of the development using Android SDK (the most efficient *framework*).

Additionally, for all three types of apps considered, the development framework with Titanium stands out for always being in the group of the most efficient frameworks, even if it never got the first place.

4 Storage Space

There is a lot of variation in storage space size among the different models of smartphones, this resource being critically scarce in less expensive devices. In the latter, the operating system and factory pre-installed apps take up a large portion of the available space, which hinders the installation of new apps [19]. This problem is worsened by a trend in the market towards the development of increasingly bigger apps.

According to a study carried out on Google Play Store, the size of the apps has quintupled between 2012 and 2017 [20]. This increase is due to the evolution of the market, requiring new features and better resources in apps. However, users are reluctant to resign storage space in their devices. The same study showed that the number of effective installations decreases by 1% for each 6 megabytes of increase in app size. Additionally, downloads are interrupted 30% more often in 100-megabyte apps than in 10-megabytes ones.

It is apparent that developers must optimize storage space usage to reach a larger number of potential users. Faced with this need, the scientific community has not been indifferent. In [21] and [22], elastic mobile app design models are proposed. These models use cloud computing technology to increase computation resources and storage space, splitting the apps into modules and migrating to the cloud those that require more resources. In addition to the obvious disadvantages, the excessive use of space can also negatively impact energy consumption [23]. In [24], methods to reduce the energy consumption associated with reading and writing access to storage space are proposed.

To minimize the size of the apps built, the impact of the development approach chosen on power consumption should also be considered. Below, we present the results of the experimental tests quantifying how large this impact is based on the development framework used.

4.1 Experiment

The tests whose results are presented in the following section were carried out with the 6 scenarios mentioned before. To assess the impact on the space used by the apps built, the size of the APK file generated by the different development frameworks being considered was measured. This information is independent from the device where the app is later installed.

The specific tools and libraries used by the development frameworks to support certain functionalities may impact differently the size of the resulting apps. To detect such potential situations, three different types of apps were implemented for each of the 6 scenarios defined, covering the usual functionalities: 1) text display, 2) video playback and 3) audio playback. Thus, there are 18 test cases: The source code for all developments produced for the experiments is publicly accessible on [25]

For all tests, the applications were generated following the standard procedure recommended by the documentation for each framework. In all cases, it was specifically corroborated that no additional files, such as images or videos, were included. These files are usually added by framework tools when a new app is built. Below, the results obtained are displayed.

4.2 Results

Table 5 and Figure 4 show test results for the three types of apps considered. The development frameworks used are ordered based on the size of the APK file obtained. It can be seen that the sorting order is the same for all three types of apps considered. In all cases, the native development with Android SDK was the most efficient approach, producing the smallest app, followed closely by Cordova, hybrid approach. The apps generated with cross-compilation (Xamarin and Corona) are in intermediate positions. Finally, the frameworks using the interpreted approach (Titanium and NativeScript) generated the largest APK packages.

Table 4. Size (in Mb) of the app package produced

Framework	**Text-based app**	**Video play-back app**	**Audio play-back app**
Android SDK	1.48	1.04	1.38
Cordova	1.74	2.77	1.82
Xamarin	4.08	5.01	4.08
Corona	6.51	6.58	6.73
Titanium	8.54	9.23	9.16
NativeScript	12.49	12.47	21.11

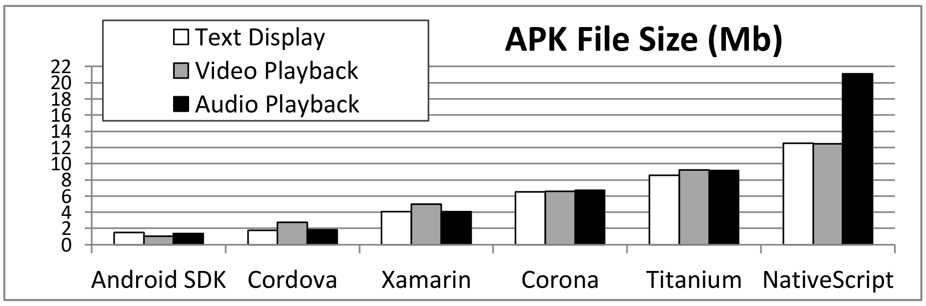

Fig. 4. Package size by development framework used for the three types of apps considered.

5 Conclusions and Future Work

In this article, the results obtained in our previous work, presented in [1] [2] and [3], are expanded.

Thorough tests of three of the most important non-functional requirements were carried out. These requirements affect the development of apps for mobile devices: performance, energy consumption and use of storage space.

To analyze each of these requirements, 6 of the most popular development frameworks in the market were used: 1) Android SDK (native approach), 2) Cordova (hybrid approach), 3) Titanium (interpreted approach), 4) NativeScript (interpreted approach), 5) Xamarin (cross-compilation approach) and 6) Corona (cross-compilation approach).

The results obtained are a contribution for Software Engineers, allowing them to prioritize the use of an approach over others, based on the expected levels of performance, energy consumption and use of storage space.

Mobile app users heavily weight these non-functional requirements when it comes to deciding whether to install an app on their mobile devices. This work presents an advance in that regard, with concrete results.

Finally, an expansion is planned in the future to include iOS' mobile platform, in addition to carrying out tests with other frameworks for the native and multi-platform development approaches discussed here.

References

1. Delia, L.; Galdamez, N.; Corbalan, L.; Pesado, P.; Thomas, P.; Approaches to Mobile Application Development: Comparative Performance Analysis. *SAI Computing Conference* (SAI), 2017. IEEE, 2017. p. 652 – 659.
2. Corbalan L.; Fernandez Sosa J.; Cuitiño A.; Delia L.; Caseres G.; Thomas P.; Pesado P., Development Frameworks for Mobile Devices: A Comparative Study about Energy Consumption (ICSE), MobileSoft 2018 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems on, Gothenburg Sweden, 2018.
3. Fernandez Sosa J.; Thomas P., Delía L.; Cáseres G., Corbalán L., Tesone F., Cuitiño A., Pesado P., Mobile Application Development Approaches: A Comparative Analysis on the Use of Storage Space, CACIC 2018, Tandil, Argentina. ISBN: 978-950-658-472-6.
4. RAJ, CP Rahul; TOLETY, Seshu Babu. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In: India Conference (INDICON), 2012 Annual IEEE. IEEE, 2012. p. 625-629.
5. XANTHOPOULOS, Spyros; XINOGALOS, Stelios. A comparative analysis of cross-platform development approaches for mobile applications. In: Proceedings of the 6th Balkan Conference in Informatics. ACM, 2013. p. 213-220.
6. Florian Rösler, André Nitze, Andreas Schmietendorf. Towards a Mobile Application Performance Benchmark. ICIW 2014: The Ninth International Conference on Internet and Web Applications and Services, At Paris, France.
7. Jung, H.W, Kim, S.G., Chung, C.S. Measuring Software Quality: A Survey of ISO/IEC 9126. IEEE Software, Septem-ber/October 2004. pp. 88 – 92. 2004.
8. Luis Corral, Anton B. Georgiev, Alberto Sillitti, Giancarlo Succi, Can execution time describe accurately the energy consumption of mobile apps? An experiment in Android. GREENS 2014 Proceedings of the 3rd International Workshop on Green and Sustainable Software. Pages 31-37
9. Andre Charland, Brian Leroux, Mobile application development: web vs. native. Magazine Communications of the ACM CACM Homepage archive Volume 54 Issue 5, May 2011 Pages 49-53 ACM New York, NY, USA
10. Luis Corral, Alberto Sillitti, Giancarlo Succi, Mobile multiplatform development: An experiment for performance analysis, The 9th International Conference on Mobile Web Information Systems (MobiWIS), Ontario, Canada, 2012.

11. big.LITTLE technology https://www.arm.com/why-arm/technologies/big-little [Last access: March 2019].
12. BANERJEE, Abhijeet; ROYCHOUDHURY, Abhik. Future of mobile software for smartphones and drones: Energy and performance. En Proceedings of the 4th International Conference on Mobile Software Engineering and Systems. IEEE Press, 2017. p. 1-12.
13. BAYER, Hannah; NEBEL, Markus. Evaluating Algorithms according to their Energy Consumption. Mathematical Theory and Computational Practice, 2009, p. 48.
14. LARSSON, Petter. Energy-efficient software guidelines. Intel Software Solutions Group, Tech. Rep, 2011.
15. SIEBRA, Clauirton, et al. The software perspective for energy-efficient mobile applications development. In: Proceedings of the 10th International Conference on Advances in Mobile Computing & Multimedia. ACM, 2012. p. 143-150.
16. CRUZ, Luis; ABREU, Rui. Performance-based guidelines for energy efficient mobile applications. In: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems. IEEE Press, 2017. p. 46-57.
17. KUMAR, Karthik; LU, Yung-Hsiang. Cloud computing for mobile users: Can offloading computation save energy? Computer, 2010, vol. 43, no 4, p. 51-56.
18. GILL, Queen Kaur; KAUR, Kiranbir. A Review on Energy Efficient Computation Offloading Frameworks for Mobile Cloud Computing. 2016.
19. K. Vandenbroucke, D. Ferreira, J. Goncalves, V. Kostakos and K. D. Moor, “Mobile cloud storage: a contextual experience.” Proceedings of the 16th international conference on Human-computer interaction with mobile devices & services (MobileHCI '14), pp. 101-110, 2014.
20. S. Tolomei, “Shrinking APKs, growing installs,” 20 November 2017. [Online]. Available at: https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2 [Last access: March 2019].
21. X. Zhang, A. Kunjithapatham, S. Jeong and S. Gibbs, “Towards an Elastic Application Model for Augmenting the Computing Capabilities of Mobile Devices with Cloud Computing,” Mobile Networks and Applications, vol. 16, nº 3, p. 270–284, 2011.
22. J. H. Christensen, “Using RESTful web-services and cloud computing to create next generation mobile applications.” In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, New York, 2009.
23. Y. Lyu, J. Gui, M. Wan and W. G. J. Halfond, “An Empirical Study of Local Database Usage in Android Applications,” IEEE International Conference on Software Maintenance and Evolution , Shanghai, China, 2017.
24. G. Z. David T. Nguyen, X. Qi, G. Peng, J. Zhao, T. Nguyen and D. Le, “Storage-aware smartphone energy savings,” Proceedings of the 2013 ACM international joint conference on Pervasive and ubiquitous computing, New York, 2013.
25. https://gitlab.com/iii-lidi/papers/apps-size.git [Last access: March 2019].