Improving mobile hybrid applications: Do current tools provide a benefit?

Final Submission

William Jernigan
Oregon State University
Corvallis, OR 97331
wdcjernigan@gmail.com

William Leslie
Oregon State University
Corvallis, OR 97331
lesliew@onid.oregonstate.edu

Francis Vo
Oregon State University
Corvallis, OR 97331
vof@onid.oregonstate.edu

ABSTRACT

Mobile hybrid applications are applications designed for mobile devices and written in web languages. By using the well supported web languages and running on top of something like PhoneGap, these applications run natively and download less data from the web than a website. Many of the guidelines and recommendations about applications written in web languages discuss improvements such as download speeds that do not affect hybrid applications. In this paper, we present analysis of hybrid applications performance and how they improve by using Google Closure Compiler and the recommendations of metrics tools.

1. INTRODUCTION

Mobile devices load entire JavaScript libraries when users browse the web. This can make the mobile device slow or unresponsive if the library is large and/or if the mobile device's resources are limited, which may lead users to look for another website to use or download an app instead of using the mobile browser. Traditional guidelines were made to improve load times and usability of these websites. JavaScript libraries are often minified to create smaller files allowing for faster download times for the files. 80% of web page loading time is attributed to loading page resources and only 20% is from back-end computations [7], so loading is a large concern for websites. Another approach to counter download times is hybrid applications; these store most of the JavaScript files on the device to avoid delays due to server communication. This means that the traditional guidelines for websites do not apply to hybrid applications.

Guidelines for hybrid applications are few and far between. Research literature also has not explored the effects of following web application guidelines when developing hybrid applications. The web application guidelines include using the following: Google Closure Compiler, a code transformation tool designed to make JavaScript smaller; MobileOK, which rates websites on their friendliness on mobile devices; and Google PageSpeed Insights, which rates a website on loading speed and user experience. This paper begins by exploring those old guidelines and then provides new guidelines for hybrid apps.

First, we create a baseline by recording the app load times, MobileOK scores, and Google PageSpeed Insights scores for 9 PhoneGap applications (original). Then, we run Google Closure Compiler on the apps and record the same scores (optimized). Finally, we measure the same benchmarks for apps we improved with our recommendations (improved).

1.1 Research Questions

To provide new guidelines, we propose the following research questions: RQ1: Do guidelines for the loading of websites on mobile devices apply to hybrid applications?

RQ2: If not, what new guidelines can be provided to improve the performance of hybrid applications?

2. MOTIVATING EXAMPLE

A web application designer, John, made an app, but decided that he wanted it to be mobile compatible. He decided the best way to do this was to make a hybrid app, which would make the application run more natively than on a browser. This makes the application faster on mobile devices, since not as much data has to be downloaded over a slow mobile network. He gets complaints that the application is slow to load, so he researches possible solutions. One tool John found was the Google Closure Compiler. This software minifies John's JavaScript files and removes unused lines of code from his program. He reasons that a smaller file would be quicker to parse than a large file. He uses this tool on all of his JavaScript files and pushes out the update. He continues to get complaints that the application is slow. Upon closer inspection, John finds that Google Closure Compiler did not improve the loading times of his application.

3. TECHNICAL CHALLENGES

Several factors make this research technically challenging. First, using traditional techniques on hybrid apps and determining if speed ups have occurred requires minifying and profiling on a large corpus of applications. This means the applications must be retrieved, processed, and judged for improvements with a process we design. Second, we must determine new guidelines to use for hybrid applications that could potentially improve performance. Current literature does not provide many recommendations for this type of application, although PhoneGap developers do make some recommendations based on experience. Third, we must use

similar techniques as in the first step, where we test the guidelines we propose by processing the applications and measuring their performance. We will then report which guidelines make a difference.

3.1 Answering RQ1

To determine the (lack of) gains when using the Google Closure Compiler on a hybrid application, a nontrivial problem makes the process challenging. First, if we run the Closure Compiler on all of an application's JavaScript files at once, it combines them into one JavaScript file (which fits a recommendation at https://codedrop.com.au/blog/phonegap-performance-tips). Then we must change the references in all the HTML files to include just this newly compiled JavaScript file instead of including all the old files. If we run the Closure Compiler on each of the application's JavaScript files separately, the HTML modification is not required, but then the Closure Compiler cannot make gains in combining sections of code from across files, reducing its effectiveness. So, it may be necessary to either try both approaches or find the best way to modify the HTML so the Closure Compiler is given an honest shot at making a difference.

4. EMPIRICAL METHODOLOGY

In order to show that the traditional web page guidelines do not apply to hybrid applications, we will first use the Google Closure Compiler to create a version of the hybrid application which is minified. Then we will test its speed with Android Device Monitor with the expectation that there will be miniscule improvements. Below is a discussion of the current challenges with each required technology.

4.1 The Corpus

First, we will build a corpus of applications to test our process. We target PhoneGap applications, but need to determine which apps are open source. Doing this manually requires searching for the applications in repositories hosted at sites like Github or Google Code. Determining that the correct application has been found potentially requires human intelligence (or is hard to automate) because one or more factors may confirm that the correct application has been found, including author name, images of the application, release dates, etc. Automating this process appears to be potentially possible, but using a simple automated process and just using the applications we can get is likely the best approach.

4.2 Google Closure Compiler

After obtaining the applications, we will measure performance (loading speed, etc.) before and after modifying the application with the Google Closure Compiler. Using this tool creates the challenges described in section Answering RQ1, including requiring modification of all HTML files to refer to the compiled JavaScript file(s), or compiling one JavaScript file at a time, reducing the Closure Compiler's effectiveness.

4.3 KJSCompiler

KJSCompiler is a wrapper for Google Closure Compiler that allows the user to provide annotations with information about dependencies to be used. When KJSCompiler is run, it compiles the JavaScript files in the order that the dependency annotations dictate. This allows for the correct ordering of function and value declarations when compiling over multiple files. The downside to this software is that the user needs to have dependence information in each file. This is worsened if the user does not understand the dependence structure of the target system very well. Using the KJS-Compiler in our case would require automating the determination of dependencies between files of an arbitrary file structure, which may be too complex considering the limited gains to be had.

4.4 Ineffective traditional guidelines

We will use Google Closure Compiler to turn a hybrid app into a minified hybrid app. The closure compiler uses traditional guidelines for optimizing mobile web apps. Then we will test both the normal and minified versions of the hybrid apps. Our hypothesis is that the performance of normal and minified versions will be very similar. The traditional guidelines are used to create an optimized javascript files that are transferred over the internet to the user device. Unlike mobile web apps, hybrid apps have the JavaScript files on the device on execution. This means that file size and other minification guidelines might not affect performance as much as they would for web apps.

4.5 New Guidelines

After proving that traditional guidelines are ineffective, we will test and validate that new guidelines are more effective for hybrid apps. Aharon has suggested some new guidelines that will be more effective on hybrid apps:

1. Creating smaller files using minification
2. Load only functions needed for that page

We also did some research and found some performance techniques used by Phonegap developers:

1. Avoid Network access
2. Don't wait for data to display
3. Use CSS Transitions
4. Use Native functions
5. Avoid Click event delay
6. Use CSS Sprite Sheets
7. Limit some UI functions
8. Limit access to DOM
9. limit libraries and frameworks

5. RESULTS

5.1 RQ1: Traditional Guidelines

We were expecting that changing the original according the the tradition guidelines would yield no significant speed improvements. The next 2 graphs show the comparison between the original and the compiled version of the apps.

After making the changes to the original apps, the PageSpeed Insights showed a significant change. This was very unexpected, but since both Google Closure Compiler and Google PageSpeed Insights were made by the same company, farther research needs to be done to understand this change.

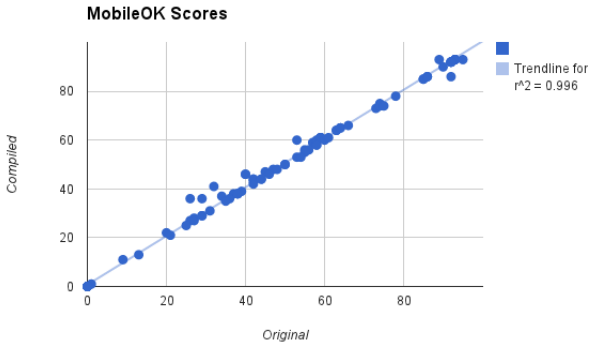


Figure 1: The changes yielded no significant changes at a p-value = 0.8776

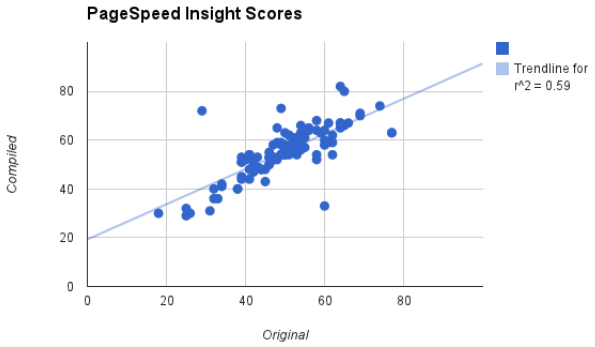


Figure 2: The changes yielded significant changes at a p-value = 9.905e-05

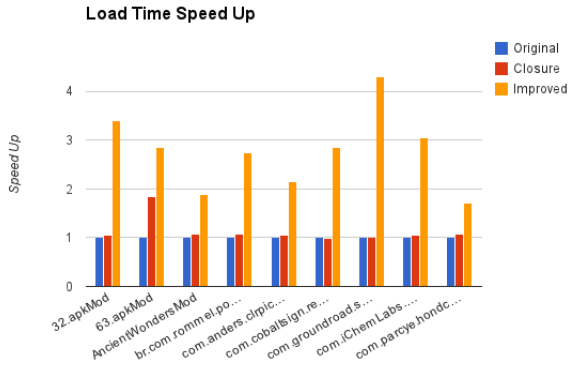
5.2 RQ1: Improved Guidelines

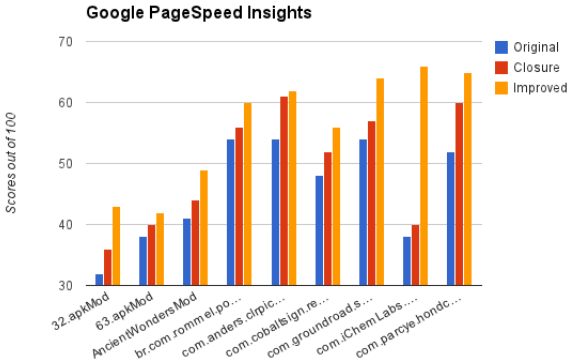
From the nine applications that we manually refactored with the improved guidelines, we have consistently improved speed-ups over the respective original applicatons. One outlier from our results was an application which did not load after the improvements were implemented. While the load speed increased on this application, it stopped working correctly. This is due to an issue with aggregating the JavaScript into a single, minified file. There are references within this while to other JS files that have been removed by the afore mentioned minification process. Since these files no longer existed the page would throw an error and terminate before the HTML loads on the page, resulting a blank page. This shows that while our refactoring methods may improve performance, some functionality may be lost in certain applications.

6. RELATED WORK

6.1 Digua: Minifier and Obfuscator for Web Resources [1]

Ciminian and Dobre discuss their minifier tool Digua in [1]. Digua is a minifier which works for CSS, HTML, and JavaScript. It will find correlations between these different languages to further reduce the size of these files. The paper states that 80% of web page loading time is attributed to loading page resources and only 20% is from back-end





computations and the reduced file size will result in much increased web page responsiveness. This does not apply to hybrid mobile apps. This is because the web code for the application is already stored in the device. The method presented in the paper would only apply to applications which need to be downloaded every time the website is hit.

6.2 Semi-Automatic Rename Refactoring for JavaScript [2]

Feldthaus and Moller discuss a method for renaming functions using a combination of static analysis and programmer input in [2]. While this does not apply directly to our project, it does bring up issues involved with renaming in JavaScript. For instance, similarly named features of the code may be renamed when they should not be. This will lead to a lot of errors in the code. This is a limitation to minification methods as well, since, in order to maintain functionality, certain features such as functions cannot be renamed.

6.3 Modern JavaScript Project Optimizers [10]

Zolotareva et. al. in [10] discuss the state of the art of JavaScript optimizers. Tools discussed included JSMin, YUI Compressor, UglifierJS, Google Closure Compiler, and KJS-Compiler and their respective benefits and drawbacks. For instance, KJSCompiler is an extension to Google Closure Compiler where the users can annotate dependencies between files. This is a useful tool to research due to the fact that it may further reduce the size of JavaScript files and combine them into a single library. These tools could all fit in our process to determine if old guidelines do not create

improvements for hybrid applications. With regard to scale of work for the term, we are currently considering just the Google Closure compiler.

6.4 Characterizing and Detecting Performance Bugs for Smartphone Applications [3]

This paper begins by presenting common patterns among performance bugs among popular, large scale Android applications. Then, the paper presents PerfChecker, a static analysis tool that looks for those performance bugs (20 of which were fixed by developers). This paper focuses on bugs relating to a native application's local code and potentially very specific causes for those bugs, such as activating multiple features in a specific sequence. So, the guidelines presented here do not directly address hybrid mobile applications, although the approach or results could in some cases be adapted for hybrids. For instance, the general recommendation of recycling resources on a page instead of creating memory bloat by adding and deleting constantly can be derived from the memory bloat bug type in this paper.

6.5 AppMobiCloud [8]

Wang et. al. in [8] look to improve the performance of mobile web applications by partitioning computation-intensive portions of code and offloading it to servers for remote execution. AppMobiCloud is the tool that does this, ensuring that an app's functionality remains the same but it performs faster and saves energy on the mobile device. The tool uses a code analyzer to find the computations to offload. AppMobiCloud technique of offloading computation to servers is different our technique of finding code changes that run on the device that will improve performance. Also, although they target mobile web applications and this paper targets mobile hybrid applications, both types could likely benefit from computational offloading. This is a potential future work because, to the best of our knowledge, this avenue has not been explored in the current literature.

6.6 A Comparative Analysis of Cross-platform Development Approaches for Mobile Applications [9]

[9] compares the use of four types of mobile applications: web, hybrid, interpreted, and generated. This survey of current technologies highlights that native apps provide a great experience but have a high development cost due to requiring device-specific implementations. Hybrid and interpreted apps avoid this deployment issue because the code for these apps is written once and interpreted in the required way by the devices. [9] also reports that generated apps give a great alternative to native apps but there is little support for this paradigm without commercial purchases. These results discuss the pros and cons of a high level design decision, where our reported code changes focus on lower level code changes and their pros and cons.

6.7 Discovering Refactoring Opportunities in Cascading Style Sheets [5]

Mazinanian et. al. explore ways to improve CSS, primarily for maintainability in [5]. That paper provides description of CSS duplications, refactoring improvements, ranking of those refactorings for prioritization, and preconditions for preservation of styling after refactoring. Our paper could have suggested following what was presented in [5], but there is not a strong indication our metrics would have improved. Their technique reduced file size by 8

6.8 MobiTran: Tool Support for Refactoring PC Websites to Smart Phones [4]

In [4], Ma et. al. present MobiTran which helps developers create a smart phone ready website from a desktop website. Their approach highlights some differences between websites on PC and smart phones but does not address differences for hybrid applications. Their paper does explain how transformations of certain portions of websites may be generalized to streamline their process, but these transformations do not necessarily improve loading times or help with the other metrics highlighted in our paper. It may be that the touch-centric transformations improve scores on MobileOK because of its recommendation of using links on entire images instead of click boxes that do not cover the entire image.

6.9 How Do Code Refactorings Affect Energy Usage? [6]

In [6], Sahin et. al. explore the energy consumption of programs after refactorings are performed. It focuses on refactorings in Java programs on PCs, not hybrid applications on mobile devices. By focusing on energy consumption, [6] considers one of the many elements that impact user experience. We focus on metrics that affect the user's perceptions in the moment, such as loading times or render blocking as reported by PageSpeed Insights.

6.10 High Performance Web Sites[7]

The paper discussed the importance of front-end performance. It also lists best practices and shows why they are important.

1. Make fewer HTTP requests
2. Use a content delivery network
3. Add an Expires header
4. Gzip components
5. Put stylesheets at the top
6. Put scripts at the bottom
7. Avoid CSS expressions
8. Make JavaScript and CSS external
9. Reduce DNS lookups
10. Minify JavaScript
11. Avoid redirects
12. Remove duplicate scripts
13. Configure ETags
14. Make Ajax cacheable

7. PROJECT EXPERIENCE

7.1 Corpus

We had a rough time obtaining a corpus. We first started by scanning the internet and Github for open source phonegap applications, and found small applications that would not have been great for our research. Many of the repositories were just examples, and tutorials. The phonegap website had a list of 720 phonegap applications with links to their

apks. Apks were not useful to use because they were packaged up and didn't have the source code. So we decided to pull the list of phonegap applications and associated developers. We used this list of search for open source projects on github, but many repositories were links to download the apks, small example libraries, or applications with incomplete source.

8. CONCLUSION

By combining the described changes into an automated refactoring tool, these improvements could be made in many applications. Further research must be done into how to make these refactorings compatible with the largest number of apps. While most were improved, some outliers lost functionality, threw errors, or just simply developed strange behaviors. Other avenues to explore would be testing applications on mobile devices. During the testing phase we lacked the know-how to automate load time analysis from a mobile device and the time to do so manually.

9. ACKNOWLEDGMENTS

This paper was possible because of Dr. Danny Dig; Aharon Abadi of IBM; the makers of Google Closure Compiler, MobileOK, PageSpeed Insights; and the writers of all the apps we used for the experiment.

10. REFERENCES

[1] Alex Ciminian and Ciprian Dobre. Digua: Minifier and obfuscator for web resources. 2012.

[2] Asger Feldthaus and Anders Møller. Semi-automatic rename refactoring for javascript. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 323–338. ACM, 2013.

[3] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *ICSE*, pages 1013–1024, 2014.

[4] Yun Ma, Yimeng Fang, Xiaomin Zhu, Xuanzhe Liu, and Gang Huang. Mobitran: Tool support for refactoring pc websites to smart phones. In *Proceedings Demo & Poster Track of ACM/IFIP/USENIX International Middleware Conference*, MiddlewareDPT '13, pages 6:1–6:2, New York, NY, USA, 2013. ACM.

[5] Davood Mazinanian, Nikolaos Tsantalis, and Ali Mesbah. Discovering refactoring opportunities in cascading style sheets. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 496–506, New York, NY, USA, 2014. ACM.

[6] Cagri Sahin, Lori Pollock, and James Clause. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 36:1–36:10, New York, NY, USA, 2014. ACM.

[7] Steve Souders. High-performance web sites. *Communications of the ACM*, 51(12):36–41, 2008.

[8] Xudong Wang, Xuanzhe Liu, Gang Huang, and Yunxin Liu. Appmobicloud: Improving mobile web applications by mobile-cloud convergence. In *Proceedings of the 5th Asia-Pacific Symposium on Internetware*, Internetware '13, pages 14:1–14:10, New York, NY, USA, 2013. ACM.

[9] Spyros Xanthopoulos and Stelios Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*, BCI '13, pages 213–220, New York, NY, USA, 2013. ACM.

[10] IO Zolotareva, OO Knyga, and . Modern javascript project optimizers. 2014.