Mobile JavaScript Library Minimization

Milestone Two

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

1. RESEARCH OVERVIEW

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 [4], 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. 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 does not remove entire functions, even though some of them go unused.

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. PROPOSED SOLUTION

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 Retrieving applications

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 KJSCompiler 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. RELATED WORKS

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

Authors Alex Ciminian and Ciprian Dobre discuss their minifier tool Digua. 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.

5.2 Semi-Automatic Rename Refactoring for JavaScript[2]

Authors Asger Feldthaus and Anders Moller discuss a method for renaming functions using a combination of static analysis and programmer input. 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.

5.3 Modern JavaScript Project Optimizers[5]

The authors of this paper discuss the state of the art of JavaScript optimizers. Tools discussed included JSMin, YUI Compressor, UglifierJS, Google Closure Compiler, and KJSCompiler 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.

5.4 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 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. There is little mention of web language bugs.

5.5 High Performance Web Sites[4]

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

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

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