UNIVERSITY WEST

2015-05-28

A Performance Study of Hybrid Mobile Applications Compared to Native Applications

Dan Brinkheden **Robin Andersson**

DEGREE PROJECT
Computer Engineering
Bachelor level G2E, 15 hec

Department of Engineering Science, University West, Sweden

DEGREE PROJECT

A Performance Study of Hybrid Mobile Applications Compared to native Applications

Summary

This study evaluates the performance difference between hybrid and native mobile applications when accessing the low level API. The purpose of this study is to find out the difference in performance between the different methods for developing applications due to an increasing market for platform independent applications.

Several benchmarks were created to measure the performance on the following criteria, execution time, memory allocation and storage space. The benchmarks were developed with a similar behaviour to match the functionality.

The Titanium benchmarks were around 8.5 times larger in storage space and used 26-28% larger heap when it came to memory than the equivalent Android benchmarks. Android generally has a lower execution time than Titanium, however there are cases such as the math library where Titanium has a lower execution time.

Date:	May 28, 2015
Author:	Dan Brinkheden, Robin Andersson
Examiner:	Thomas Lundqvist
Advisor:	Dena Ala-Hussain
Programme:	Computer Engineering and System Development, 180 HE credits
Main field of study:	Computer Engineering
Education level:	First cycle
Course code:	EXD500, 15 HE credits
Keywords:	*Android, Titanium, Hybrid applications, Native applications, Benchmark*
Publisher:	University West, Department of Engineering Science SE-461 86 Trollhättan, Sweden Phone: + 46 520 22 30 00, Fax: + 46 520 22 32 99, www.hv.se

Preface

This bachelor thesis is our final assignment of the Computer engineering and system development program at University West. We would like to take the opportunity to thank our advisor Dena Ala-Hussain for the support and guidance. We would also like to thank Priska Schimming and Lea Häberle for proof reading. Lastly we would like to thank each other for the effort put into the thesis.

The work has been evenly divided amongst each other in the writing and research, for development Dan Brinkheden has been more responsible for the Titanium development and Robin Andersson has been more responsible for the Android Development.

Table of contents

Appendices

1 Introduction

Development for the mobile application market has drastically increased in size and magnitude therefore the requirements for developing applications has changed along with the market [1]. Hybrid applications is one of the three main development paradigms along with native development and HTML5. Hybrid applications features a single code-base, bridging the different platforms as opposed to the native applications. HTML5 shares the platform independence with hybrid applications, however it lacks the ability to communicate with the low level Application Programming Interface (API). The three development paradigms differ in performance, development resources and user interface. Due to performance limitations of mobile devices the performance is a major factor for the selection of development method. The technology for developing mobile applications is evolving rapidly and that makes the performance to evolve along with it.

This study will evaluate the performance difference between native and hybrid applications when accessing the device native hardware through the low level API. The operating system (OS) and framework that this study will evaluate are Android and Titanium. Since there are many different platforms out on the mobile market and with the demand for context awareness amongst cross-platform applications has increased therefore the demand for hybrid applications has also increased [2]. The objective for this study is to evaluate the performance difference between native and hybrid solutions and evaluate if hybrid can compete with the native performance when communicating with low level API.

To evaluate the performance of specific functions, small applications called benchmarks will be developed. The benchmarks for this study are developed with identical use cases to evaluate the performance. The purpose of the benchmarks are to isolate the specific code used to access the low level API functions, in order to test how efficiently they communicate with the API and the resource allocation. The data from the benchmarks are evaluated on different criteria such as access time, memory allocation and disk storage space.

2 Background

Native applications are binary executable files which are written in the platform specific language. The native application freely accesses the platform API through directly interfacing without containers [3]. Advantages with using native developed applications are full access of the device with great performance but at the cost of portability.

Hybrid applications consist of native and web technology to increase portability while maintaining the access to the native API. The applications are written with a single code-base, typically JavaScript that operates as a bridge between the native container and the native API [3]. Advantages with Hybrid developed applications are high availability, a single code-base while it may lack in full native access, security and performance.

2.1 Android

As of 2007 Android is a platform owned by Google and released as the Android Open Source Project. The OS is based on a single modified Linux kernel, specifically the 2.6 kernel series. The Linux kernel has been modified to work on devices with generally lower memory and with stricter requirements on energy consumption.

In general Android applications are written in Java and each application runs in an instance of the virtual machine. The name of the virtual machine is Dalvik and it acts as an interpreter between the Java bytecode and the Dalvik bytecode [4]. Dalvik is designed to be very small virtual machine in order to limit the overhead since every application runs on a separate instance of Dalvik in a sandbox environment.

The system architecture for Android is divided into four layers, the bottom layer is the Linux kernel for power management and drivers. On top of the kernel lies the libraries and Android run times, the library layer is written in C/C++ and includes libraries for media, screen access and amongst many other crucial functionalities. The third layer is the application framework which provides abstractions to the native functions and the Dalvik functionality. The top layer is the application layer where the device applications are executed and where the high level API functions a located. The two last layers are both written in Java and run on the Dalvik virtual machine.

2.2 Software Development Kit (SDK)

Software development kit (SDK) are ways for developers to target specific platforms, the SDK contains a set of tools that makes it possible to compile the source code for the targeted platform. Each mobile OS has its own SDK, they all differ in language, tools and package formats. The tools contained in the SDK are commonly utilities such as libraries files, development tools, sample applications and API.

2.3 Application Programming Interface (API)

Mobile OS APIs acts as a bridge between applications and the OS through proprietary calls exposed by the OS. The API is divided into low and high-level, the low-level API calls interacts with hardware elements while high-level API interacts with services. Example of low-level API elements are touch screen, microphone, Global Positioning system (GPS), camera and storage. The high-level services accessed through the API are processes like calendar, contacts, photo album and phone calls.

2.4 Titanium

Titanium is an open-source framework made and licensed by Appcelerator under the Apache 2 license. Titanium allows the developer to build cross-platform applications with a single code base. In contrast to the true hybrid solutions, Titanium is not limited by running in a web container. It creates cross platform applications using the native user-interface through generated native code [5].

Titanium applications are written in JavaScript, the application source code is interpreted on the mobile device using JavaScriptCore on IOS and V8 JavaScript engine on Android. The Titanium software development kit (SDK) is a set of Node.js-based utilities and supporting tools for the native SDK. The Titanium SDK combines JavaScript source code, JavaScript interpreter and static assets into application binary. The application binary file is then installed to a mobile device.

“V8 is Google’s open source, high performance JavaScript engine” [6]. The engine is designed for fast execution of large JavaScript applications. The three major components that makes up the performance of V8 are Fast Property Access, Dynamic Machine Code Generation and Efficient Garbage Collection. Fast Property Access uses hidden classes behind the scenes instead of using dynamic lookup to access properties. Dynamic Machine Code Generation interprets JavaScript code directly into machine code without the bytecode intermediate step. The garbage collection frees up memory that is no longer in use by the application to enhance the performance.

3 Related Works

Dalmasso et al. in [5], made a survey, comparison and evaluation of cross platform mobile application development tools. The article provides several criteria other than just portability and performance, for example user experience, development cost and ease of updating. The cross-platform tools studied in the article are PhoneGap, PhoneGap & JQuery mobile, PhoneGap & Sencha Touch 2.0 and Titanium. The section about performance evaluation in the article features three subcategories being memory usage, CPU usage and power consumption. Conclusions are that cross-platform development tools have lower costs and quicker time to market at the cost of user experience. Out of the evaluated platforms PhoneGap uses the least amount resources but has a very simple user experience.

Thelander in [7] made a study on alternatives to native mobile development, specifically Android and Titanium. The comparison were on the follow criteria, internet connectivity, location awareness, data storage and user interface. For each method two application were created to be identical and then evaluated against criteria. Conclusions from the article are that applications developed in Titanium does not meet- the standards of Android development for single platforms while it may serve as a suitable alternative when developing for multiple platforms.

4 Methodology

This study evaluates the performance and ability to access the device low-level API in Android and Titanium, in context of this study performance will be defined by execution time, disk storage space and memory usage [8]. Seven test applications were developed to evaluate the performance of low-level API functions and general performance. Camera, GPS and storage were the low-level API functions that were evaluated on the performance. The general performance were tested to obtain a point of reference to gauge the results of other tests.

Android was chosen due to the high market share of Android devices at the time of the study [9] and the availability of developer tools. Titanium had at the time of the study a complete support of the low-level API required for the study, along with a big enough market share.

The performance evaluations were performed on a Samsung Galaxy Tab 2 (7.0) with an Android version 4.1.2. The Integrated Development Environment (IDE) used to develop the test applications were Eclipse with Android developer tools (ADT) version 5.1.1(API 22) and Titanium build 3.4.1.

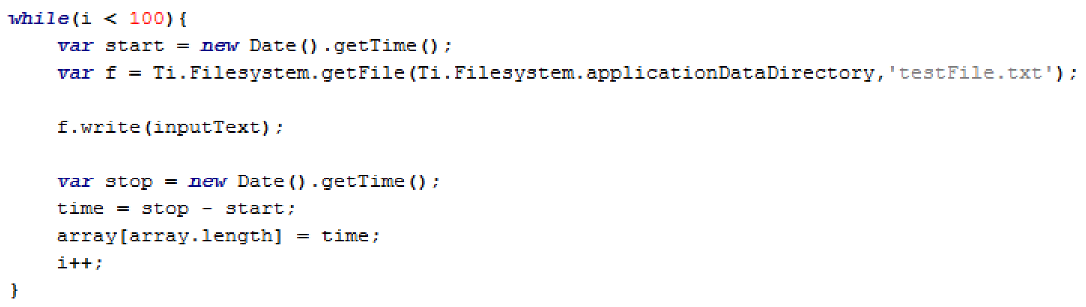
Each test consisted of two applications built to have similar behavior and perform the same action. The test applications are designed only to contain the necessary code while still maintaining the ability to perform its intended action. Memory and disk storage space can then be accurately reflected, hence the applications minimalistic proportion. The execution time is measured by timing the isolated code relevant to the intended action of the test.

Dalvik Debug Monitor Server (DDMS) was used to obtain the memory usage, by monitoring the benchmark applications process. The obtained memory data from DDMS included the heap memory size and the currently allocated memory of the heap and reserved free memory. Through the Application manager in Android OS the disk storage space was obtained.

To obtain the execution time of the benchmark functions two timing methods were used, these timing methods measure the time that the code takes to execute by comparing a start timestamp with an end timestamp. With the timestamps the difference was calculated to get the execution time of the function, the method used for Android was the Chronometer and for Titanium the `Date().getTime()` function, see figure 1 & 2 for examples. To reduce the relative errors in the execution time measurements, each individual benchmark used a loop to iterate the benchmark code 100 times per execution.

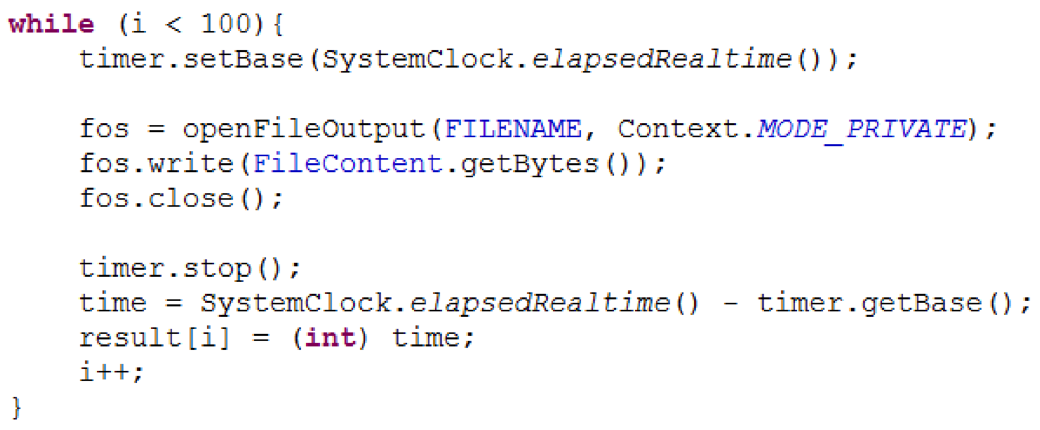
The execution time was not obtained in the camera and GPS benchmarks, hence the normal use case for the camera does not include repeating access calls to the low level API. The execution time is not measured in the GPS benchmark because the execution time of the GPS function relies more on external factors such as the access time to the GPS source.

To test the low-level API functions seven different tests per programming language were created to analyze the performance of a certain function. All tests have a specific structure that they share amongst each other, the idea behind the structure is to isolate execution time code along with the functions critical code and keep the code as clean as possible for accurate measurement of RAM and storage usage. Examples of the structure can be seen in figure 1 & 2 for both Titanium and Android.



```
while(i < 100){
    var start = new Date().getTime();
    var f = Ti.Filesystem.getFile(Ti.Filesystem.applicationDataDirectory,'testFile.txt');

    f.write(inputText);

    var stop = new Date().getTime();
    time = stop - start;
    array[array.length] = time;
    i++;
}
```

Fig 1: Example code for the execution time measurement in Titanium.



```
while (i < 100){
    timer.setBase(SystemClock.elapsedRealtime());

    fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
    fos.write(FileContent.getBytes());
    fos.close();

    timer.stop();
    time = SystemClock.elapsedRealtime() - timer.getBase();
    result[i] = (int) time;
    i++;
}
```

Fig: 2 Example code for the execution time measurement in Android.

5 Result & Analysis

The result from the benchmarks are individually presented in the following sections, each benchmark result contains a brief overview of the core functions along with the memory usage. Code execution measurements are illustrated in a line graph under the storage, database and general benchmarks. Results from the application storage space is presented for all benchmarks in the last section.

Table 1: Result order and criteria for the benchmarks.

	Benchmarks criteria
General	Code execution time, memory allocation, application storage space
Camera	Memory allocation, application storage space
GPS	Memory allocation, application storage space
Storage	Code execution time, memory allocation, application storage space
Database	Code execution time, memory allocation, application storage space

5.1 General Benchmark

The general benchmark was created and tested by using a Prime number program to evaluate the general performance difference between Android and Titanium. A prime number program was created to find the all prime number within 100 000 numbers. The benchmark was created identically between Android and Titanium to get as accurate data as possible. To find out what caused the disparity of the result, three additional benchmarks are created to measure the math library and the loops.

The math library is benchmarked by taking the square root of an incrementing variable and round it up to the nearest integer using `Math.ceil(Math.sqrt(i))`and performing it 100 000 times. The while and for loops are evaluated by running an empty loop 10 000 000 times.

From the results listed below it is clear that Titanium has an advantage over Android in terms of execution time in the prime number benchmark, the reason for this advantage is that the functions used from the math library in the prime number benchmark are more efficient in Titanium than Android, this was concluded by breaking the prime number benchmark down into smaller pieces e.g. loop and math functions benchmarks. The results from the loop and math functions benchmarks shows that the difference in the execution time can be found in the math functions as the loop benchmarks has little to none difference in the execution time.

Table 2: Results from the individual runs of execution time for prime number benchmark

	Min (ms)	Max (ms)	Average (ms)
Titanium run 1	405	467	408.8
Titanium run 2	405	459	407.92
Titanium run 3	405	460	408.22
Android run 1	613	652	615.84
Android run 2	614	658	616.64
Android run 3	614	666	616.28

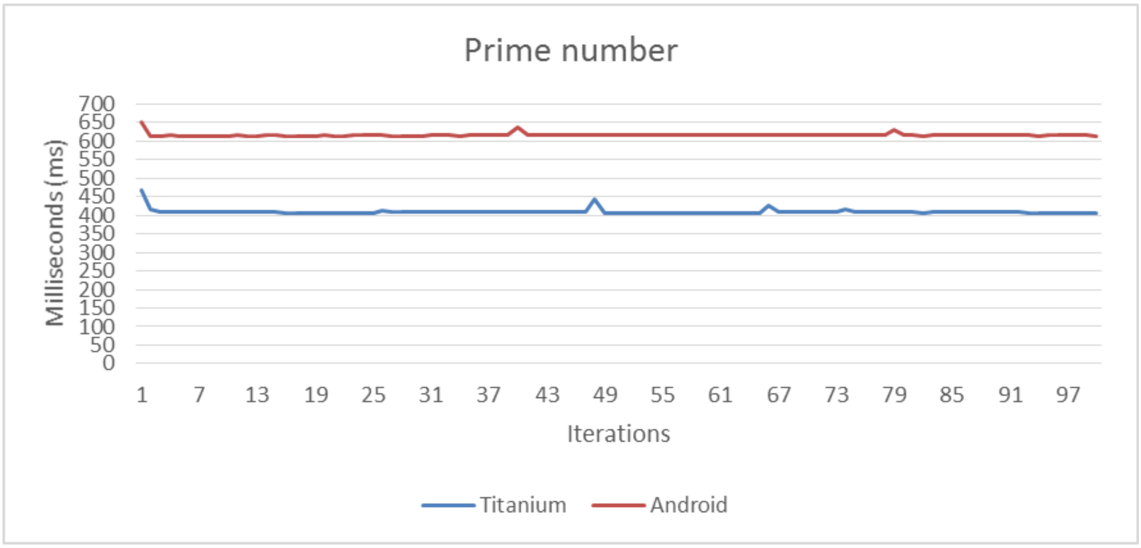


Figure 3: A graph illustrating the first run of the execution time for the prime number benchmark.

Table 3: Results from the individual runs of execution time for the Math library benchmark.

	Min (ms)	Max (ms)	Average (ms)
Titanium run 1	22	128	24.40
Titanium run 2	22	52	23.51
Titanium run 3	22	28	23.27
Android run 1	36	38	36.41
Android run 2	36	38	36.50
Android run 3	36	38	36.71

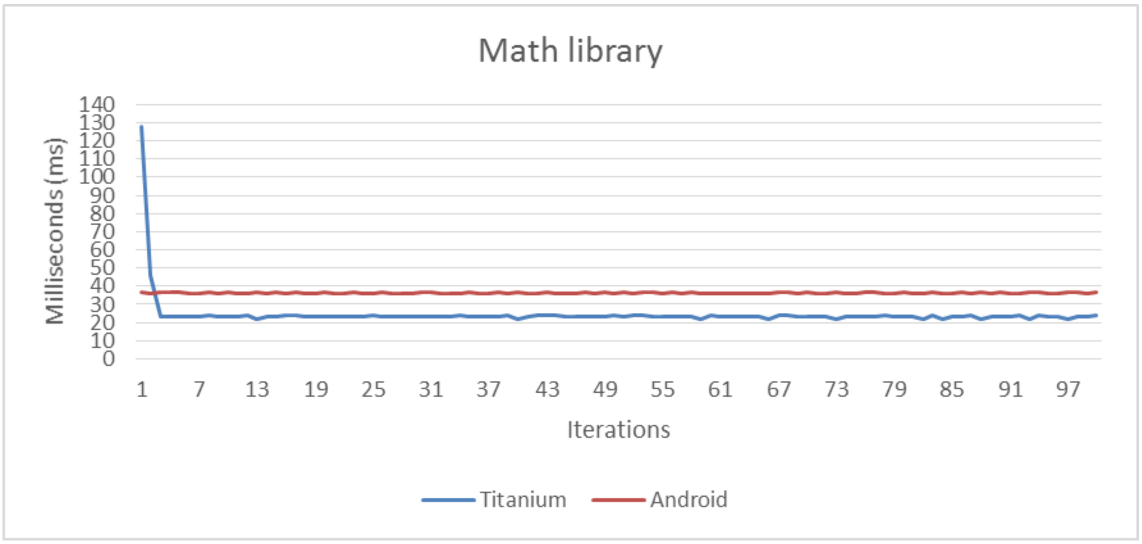


Figure 4: A graph illustrating the first run of the execution time for the Math library benchmark.

Table 4: Results from the individual runs of execution time for the For loop benchmark.

	Min (ms)	**Max (ms)**	**Average (ms)**
Titanium run 1	49	104	51.75
Titanium run 2	49	115	52.74
Titanium run 3	50	91	52.30
Android run 1	49	85	50.64
Android run 2	49	85	50.35
Android run 3	49	85	50.75

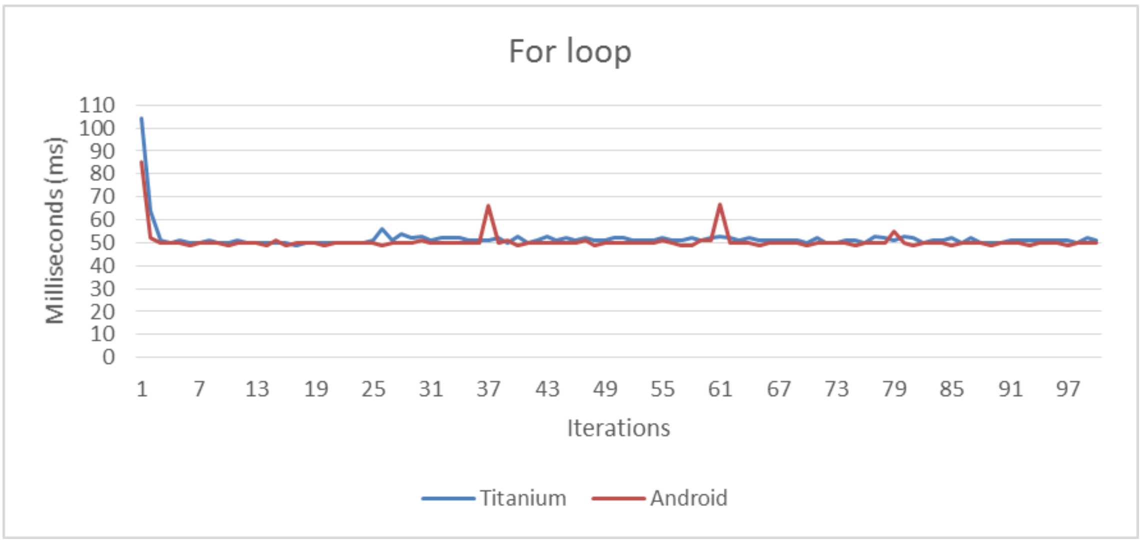


Figure 5: A graph illustrating the first run of the execution time for the For loop benchmark.

Table 5: Results from the individual runs of execution time for the While loop benchmark.

	Min (ms)	Max (ms)	Average (ms)
Titanium run 1	49	67	51.24
Titanium run 2	55	49	51.33
Titanium run 3	49	63	51.46
Android run 1	49	81	50.90
Android run 2	49	85	50.54
Android run 3	49	84	50.21

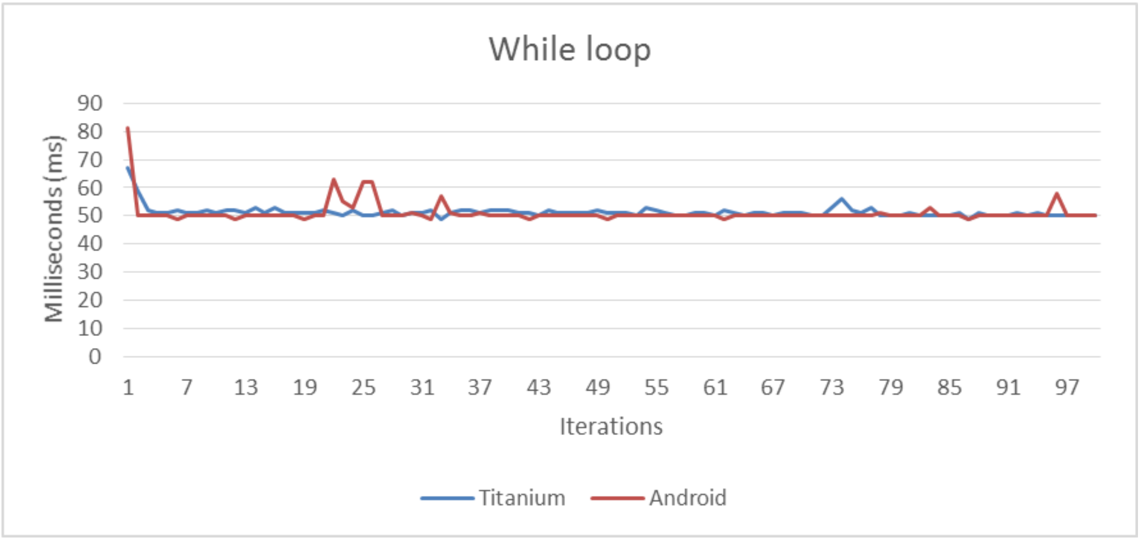


Figure 6: A graph illustrating the first run of the execution time for the For loop benchmark.

5.2 Camera & GPS Benchmark

The camera benchmark is designed to measure the storage space and memory allocation for an application that invokes the third-party camera application. Titanium uses a `show-Camera()`function which handles all the API calls. Android doesn't have an equivalent function to `showCamera()`instead it uses multiple functions that relies on a `MediaS-tore.Action_Image_Capture` Intent.

The GPS Benchmark is designed to measure the storage space and memory allocation for an application that retrieves a single update from the GPS. Titanium register a geolocation object to an event listener, when an event is triggered the listener retrieves coordinates and then unregister the listener to create a single update. Android uses `requestSingleUpdate` to create a listener and on a location change event it retrieves the coordinates.

5.3 Storage Benchmark

The storage benchmark is divided into two different sets, a writing and reading benchmark. The writing benchmark writes 40 kb of data to a file. Titanium creates a file object with the `getFile` function and writes the data using the `write()` function. Android creates an output stream with `openFileOutput()`and writes the data with `write()`to a file.

The read benchmark uses the same 40 kb file from previous benchmark. Titanium gets the file with `getFile` and then reads it with `read(),` Android opens a file input stream with `openFileInput()`. A buffered reader is then created by using the input stream and the data is read by using `readLine()`function.

The results from the storage benchmark it indicates that Titanium has a higher execution time when it comes to the write function, however the read function has on average a lower execution time in Titanium. The results shows that Android has a much more stable execution time, while Titanium tends to have a greater spread amongst the execution times. However the average execution times does not differ significantly between Android and Titanium.

Table 6: Results from the individual runs of execution time for the Storage write benchmark.

	Min (ms)	**Max (ms)**	**Average (ms)**
Titanium run 1	3	66	15.90
Titanium run 2	3	43	14.98
Titanium run 3	3	41	15.39
Android run 1	1	37	5.61
Android run 2	1	26	4
Android run 3	1	37	5.72

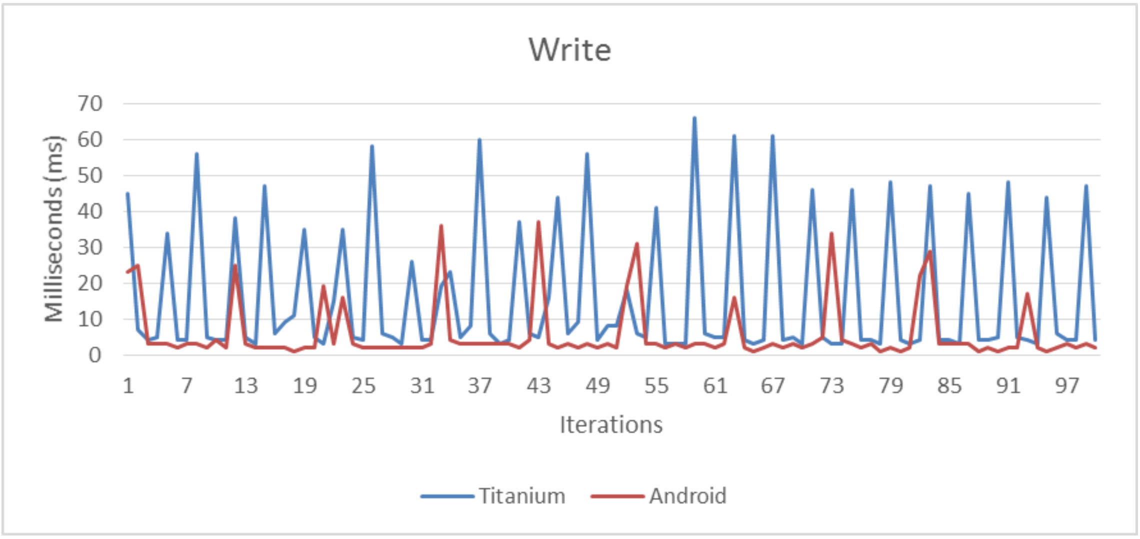


Figure 7 A graph illustrating the first run of the execution time for the Storage write benchmark.

Table 7: Results from the individual runs of execution time for the Storage read benchmark.

	Min (ms)	**Max (ms)**	**Average (ms)**
Titanium run 1	4	64	17.26
Titanium run 2	4	66	16.34
Titanium run 3	4	65	15.72
Android run 1	18	47	20.07
Android run 2	18	59	20.19
Android run 3	16	53	20.71

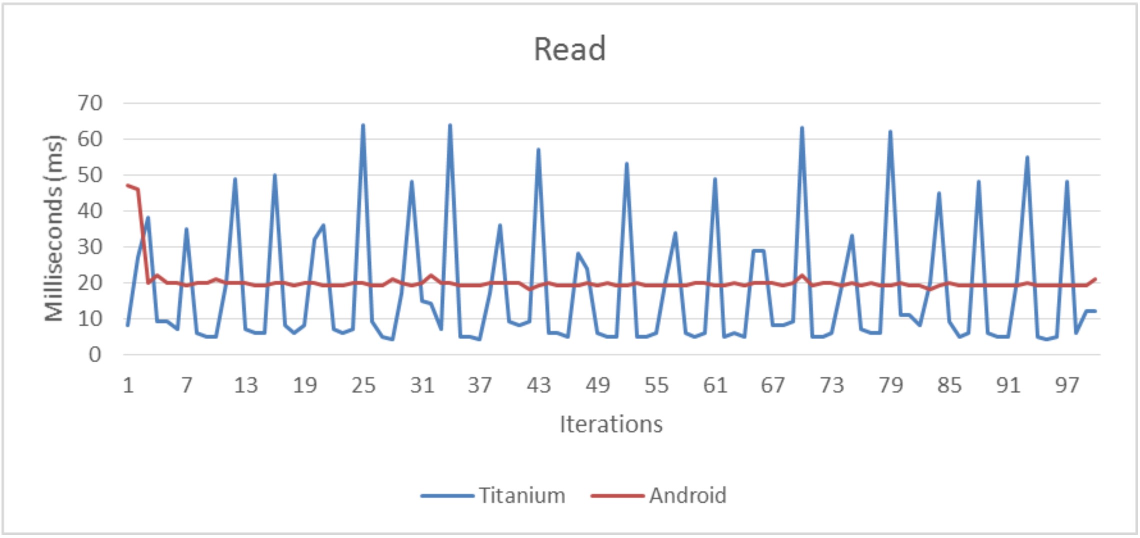


Figure 8: A graph illustrating the first run of the execution time for the Storage read benchmark.

5.4 Database Benchmark

The database benchmark consists of two parts, a reading and a writing part. The benchmark creates a new database if it doesn´t exist and it also creates the table if it doesn´t exists. The data that was inserted into the database was composed of two separate strings "This is a title" and "Hello im content".

Titanium creates a database handle with the `Ti.Database.open()` function and uses `execute()`function to perform the database queries. Android creates the database by using the `getReadableDatabase()` function and uses the `query()`function to perform the database queries. The database queries used in the reading part is a select statement.

Titanium uses the same function for the writing benchmark as it did for the reading benchmark. The difference between them is the structured query language (SQL) query, the "create table" and the "insert" query is used during the writing benchmark. Android uses `getWritableDatabase()`to create the database handle and uses the `insert()` function to insert data to the database.

The results from the read benchmark shows that Android has on average a slightly lower execution time however in the write benchmark, Android has on average less than half the execution time compared to Titanium.

Table 8: Results from the individual runs of execution time for the Database read benchmark.

	Min (ms)	Max (ms)	Average (ms)
Titanium run 1	20	82	25.09
Titanium run 2	19	69	25.38
Titanium run 3	19	68	23.48
Android run 1	10	49	19.37
Android run 2	10	52	18.93
Android run 3	11	57	22

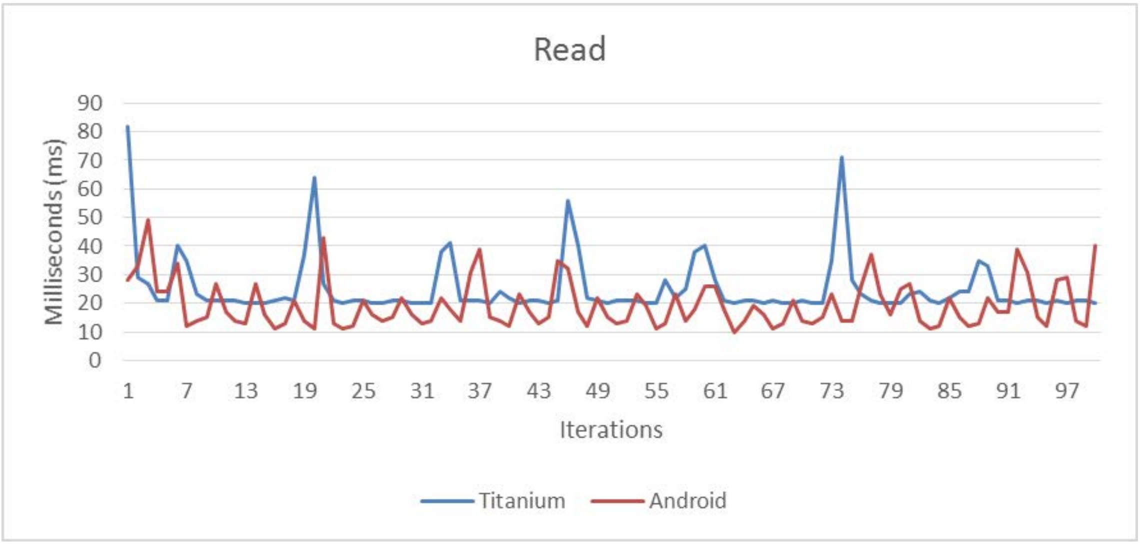


Figure 9 A graph illustrating the first run of the execution time for the Database read benchmark.

Table 9 Results from the individual runs of execution time for the Database write benchmark.

	Min (ms)	Max (ms)	Average (ms)
Titanium run 1	39	97	50.94
Titanium run 2	38	103	49.97
Titanium run 3	37	101	50.88
Android run 1	9	74	17.86
Android run 2	10	59	20.93
Android run 3	11	52	20.97

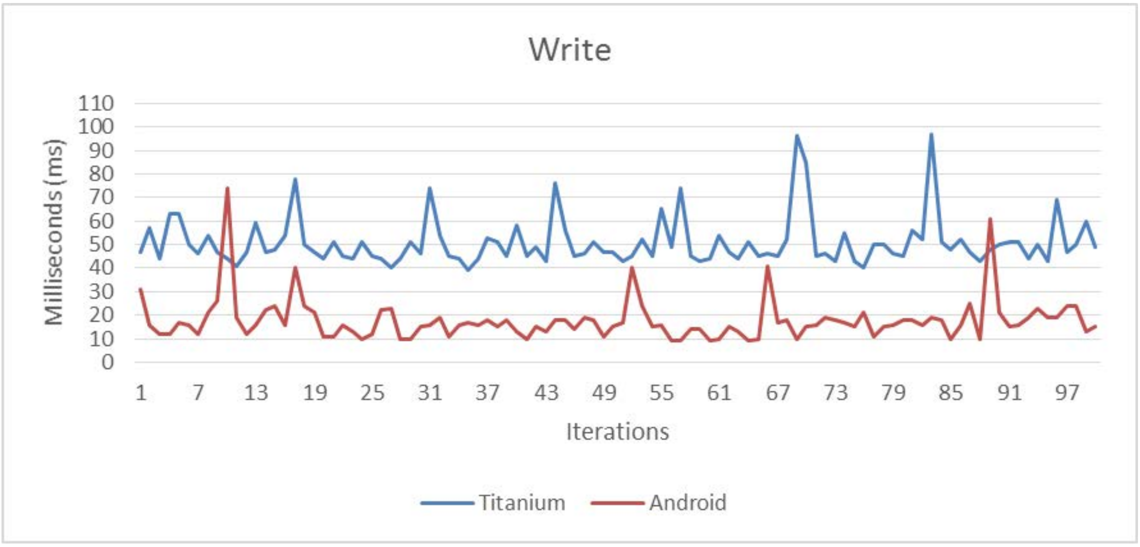


Figure 10: A graph illustrating the first run of the execution time for the Database write benchmark.

5.5 Storage space

This section shows the storage space used by the benchmarks applications with the addition of two applications created for a point of reference. The “Hello world” application in the table only contains a label with “Hello World” to get a reference of the size of a small application. The application called “Big” in the table consists of Camera, GPS, Storage and Database applications joined as one to get a reference for a larger application.

The results from the storage benchmark on the Android and the Titanium applications used in this study, shows that Android applications has a significantly smaller size compared to Titanium applications. Titanium has a larger size than Android because of the Titanium SDK that includes e.g. V8 libraries. This can be seen in the Hello world applications that contains nothing more than the bare minimum code to function. The applications size does not have a linear growth in size because the SDK size is only added once. The source code in the applications does not affect the size considerably, this can be seen in the size difference between the Hello world and the Big application. One big factor in the size of the applications is the assets that the application uses, e.g. the text file in the storage benchmark adds 40 kb to the application size when the file is created

Table: 10 Results from the Storage space measurement for all benchmarks.

	Titanium (MB)	**Android (MB)**
Camera	18.32	2.11
GPS	18.33	2.11
Storage	18.30	2.16
Database	18.28	2.12
General	18.31	2.11
Hello world	18.21	2.11
Big	18.41	2.17

5.6 Memory Allocation

The memory allocation results listed below, clarifies that in all the benchmarks the value of the allocation is close to same, for each benchmark only slight variances can be found. Titanium allocates roughly two megabyte more heap memory than Android in all of the benchmarks. Out of the two allocated megabytes Titanium uses both more allocated memory and free memory within the heap for the process.

Table 11: Results from the Memory Allocation measurements for the General benchmark.

	Heap Size (MB)	Allocated (MB)	Free (MB)	% Used
Titanium	10.320	9.179	1.141	88.94
Android	8.070	7.359	0.728	91.19

Table 12: Results from the Memory Allocation measurements for the Camera benchmark.

	Heap Size (MB)	Allocated (MB)	Free (MB)	% Used
Titanium	10.320	9.171	1.149	88.87
Android	8.133	7.369	0.781	90.61

Table 13 Results from the Memory Allocation measurements for the GPS benchmark.

	Heap Size (MB)	Allocated (MB)	Free (MB)	% Used
Titanium	10.383	9.186	1.196	88.48
Android	8.070	7.356	0.731	91.15

Table 14 Results from the Memory Allocation measurements for the Database benchmark.

	Heap Size (MB)	Allocated (MB)	Free (MB)	% Used
Titanium	10.320	9.178	1.142	88.93
Android	8.133	7.439	0.710	91.46

Table 15 (Results from the Memory Allocation measurements for the Storage benchmark.

	Heap Size (MB)	Allocated (MB)	Free (MB)	% Used
Titanium	10.320	9.199	1.121	89.14
Android	8.070	7.363	0.724	91.23

6 Discussion

All applications code-base are designed to have similar behavior with the official documentation [10, 11] as foundation. The reason for using the documentation as foundation is to have a baseline for the code standard, however the Titanium and Android documentations does not offer identical solutions to every application functionality. Because of this some of the implementations have been modified to match their counterpart, e.g. in the storage benchmark the Android documentation recommends reading data with a read function that reads one character at a time. While Titanium uses a read function that acts as buffer reading many characters at a time. If the data processing would not be similar throughout the benchmarks, then the results would show code optimization amongst the frameworks rather than performance.

The execution time benchmarks were designed to reduce the external influences from operating system or other tasks, by using an iterative approach on each benchmark. The number of iterations were determined to eliminate possible influence of the cache and still be long enough to show repeating trends. When the iterations exceeded 100 loops the trends still remained the same and therefore 100 iterations were chosen as a threshold. To validate the authenticity of the results the benchmarks were executed three times to reduce the possibility of errors in the result values.

The results from the storage and database benchmarks shows varied results where there are no frameworks with a superior execution time in all cases but there is a common behavior where Android has a slightly lower execution time than Titanium. Execution time measurements including the result from the prime number benchmark are in most cases the execution time is relatively close to each other however there are functions such as square, ceiling in the math library and database read function where the difference is greater. The major factor causing difference in execution time is how efficient the function is implemented in the frameworks, therefore it´s not possible to determine an overall better performing framework when it comes to execution time.

7 Conclusions

As a conclusion from analyzing the results are that in terms of memory allocation and storage disk space Titanium used more resources in all of the benchmarks. Titanium used 2.187-2.313 megabytes more memory than Android and this translated to an increase of 26-28% difference in memory allocation. Titanium used 16.10–16.24 megabyte more storage space than Android and this translated to an estimated of 8.5 times increase in storage space, this increase is caused primarily due to the fact that the Titanium SDK uses more storage space.

Android generally uses less of the device resources to perform the same tasks compared to Titanium and executing the required code faster in most cases, however there are some cases where Titanium performed better when it comes to code execution time.

References

[1] Intel. (2015, May 17) Implementing a Cross-Platform
Enterprise Mobile Application Framework [Online] Available
http://www.intel.com/content/dam/www/public/us/en/documents/best-practices/implementing-a-cross-platform-enterprise-mobile-application-framework-paper.pdf

[2] K. Dulaney et al. (2015, May 10) Predicts 2013: Mobility Becomes a Broad-Based Ingredient for Change [Online]. Available: https://www.gartner.com/doc/2242915

[3] IBM. (2015, May 17). Native, web or hybrid mobile-app development [Online] Available FTP: ftp://public.dhe.ibm.com/software/pdf/mobile-enterprise/WSW14182USEN.pdf

[4] S. Brähler, "Analysis of Android Architecture" Student Research Project, Dept. Inst. Info., Karlsruhe Institute of Technology, Germany, 2013.

[5] I. Dalmasso et al. "Survey, comparison and evaluation of cross platform mobile application development tools" in Wireless Communications and Mobile Computing Conference. 2013 © IEEE. doi: 10.1109/IWMCMC.2013.6583580

[6] Google Inc. (2015, May 10) Google v8 [Online]. Available: https://developers.google.com/v8/

[7] T. Thelander, "Alternatives to Native Mobile Development" Bachelor Thesis, Dept. Inst. Eng., University West, Sweden, 2013.

[8] C. Ryan and P. Rossi "Software, performance and resource utilisation metrics for context-aware mobile applications" in Software Metrics, 2005. 11th IEEE International Symposium. 2005 © IEEE. doi: 10.1109/METRICS.2005.44

[9] IDC Corporate. (2015, May 10) Smartphone OS Market Share, Q4 2014 [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[10] Google Inc. (2015, May 10) Google Developers [Online]. Available: http://developer.android.com/index.html

[11] Appcelerator Inc. (2015, May 10) Appcelerator Titanium [Online]. Available: https://docs.appcelerator.com/

A. Android source code

General benchmark

```
while (n < 100){
    i = 0;
    timer.setBase(SystemClock.elapsedRealtime());
    while(i < 100000){
        isPrime = checkPrime(i);
        if(isPrime == true){
            totalPrime++;
        }
        i++;
    }
    timer.stop();
    time = SystemClock.elapsedRealtime() - timer.getBase();
    result[n] = (int) time;
    n++;
}

public boolean checkPrime(int i){
    int j = 3;

    if(i == 1)
        return false;
    if(i == 2)
        return true;
    if(i % 2 == 0)
        return false;

    while(j <= Math.ceil(Math.sqrt(i))){
        if(i % j == 0)
            return false;
        j = j + 2;

    }
    return true;
}
```

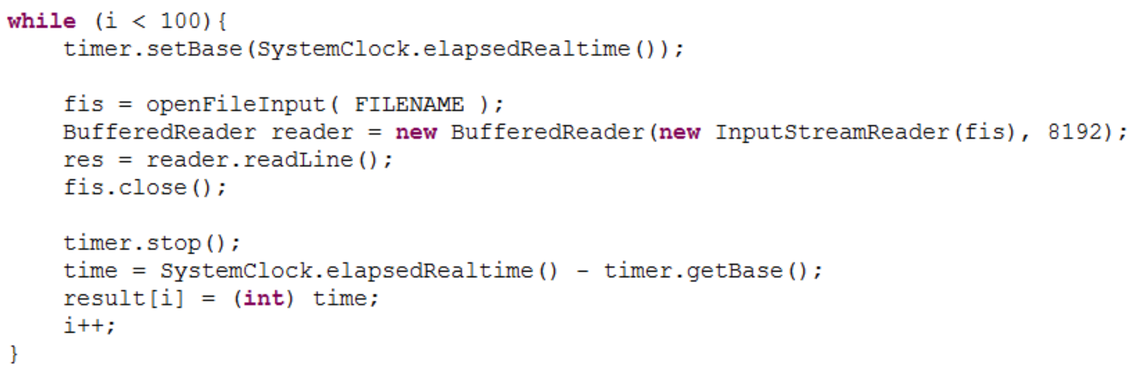
Camera benchmark

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE) {
        if (resultCode == RESULT_OK) {
            // Image captured and saved to fileUri specified in the Intent
            Toast.makeText(this, "Image saved to:\n" +
                     data.getData(), Toast.LENGTH_LONG).show();
        } else if (resultCode == RESULT_CANCELED) {
            // User cancelled the image capture
        } else {
            // Image capture failed, advise user
        }
    }

    if (requestCode == CAPTURE_VIDEO_ACTIVITY_REQUEST_CODE) {
        if (resultCode == RESULT_OK) {
            // Video captured and saved to fileUri specified in the Intent
            Toast.makeText(this, "Video saved to:\n" +
                     data.getData(), Toast.LENGTH_LONG).show();
        } else if (resultCode == RESULT_CANCELED) {
            // User cancelled the video capture
        } else {
            // Video capture failed, advise user
        }
    }
}
public void start_test (View view){
    start_camera();
}
public void start_camera(){
     // create Intent to take a picture and return control to the calling application
    Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
    fileUri = getOutputMediaFileUri(MEDIA_TYPE_IMAGE); // create a file to save the image
    intent.putExtra(MediaStore.EXTRA_OUTPUT, fileUri); // set the image file name

    // start the image capture Intent
    startActivityForResult(intent, CAPTURE_IMAGE_ACTIVITY_REQUEST_CODE);
}

/** Create a file Uri for saving an image or video */
private static Uri getOutputMediaFileUri(int type){
    return Uri.fromFile(getOutputMediaFile(type));
}
```

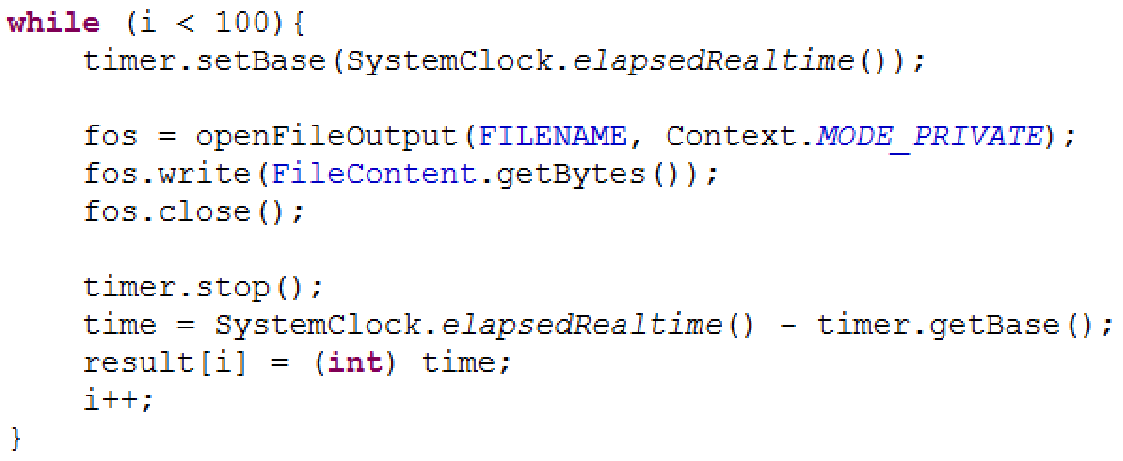
GPS benchmark

```
@Override
public void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    retrieveLocationButton = (Button) findViewById(R.id.startBtn);
    locationManager = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    retrieveLocationButton.setOnClickListener(new OnClickListener() {
        @Override
        public void onClick(View v) {
                showCurrentLocation();
        }
    });
}
protected void showCurrentLocation() {
    Criteria criteria = new Criteria();
    criteria.setAccuracy(Criteria.ACCURACY_FINE);
    criteria.setPowerRequirement(Criteria.POWER_HIGH);

    locationManager.requestSingleUpdate(criteria, new LocationListener(){
            @Override
            public void onLocationChanged(Location location) {
                double lat = location.getLatitude();
                double lon = location.getLongitude();
                double alt = location.getAltitude();
                double tim = location.getTime();

                if (location != null) {
                    String message = String.format(
                            "Current Location \n Longitude: %1$s \n Latitude: %2$s",
                            location.getLongitude(), location.getLatitude()
                    );
                    Toast.makeText(MainActivity.this, message,
                            Toast.LENGTH_LONG).show();
                }

            }
```

Storage benchmark



```
while (i < 100){
    timer.setBase(SystemClock.elapsedRealtime());

    fis = openFileInput( FILENAME );
    BufferedReader reader = new BufferedReader(new InputStreamReader(fis), 8192);
    res = reader.readLine();
    fis.close();

    timer.stop();
    time = SystemClock.elapsedRealtime() - timer.getBase();
    result[i] = (int) time;
    i++;
}
```



```
while (i < 100){
    timer.setBase(SystemClock.elapsedRealtime());

    fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
    fos.write(FileContent.getBytes());
    fos.close();

    timer.stop();
    time = SystemClock.elapsedRealtime() - timer.getBase();
    result[i] = (int) time;
    i++;
}
```

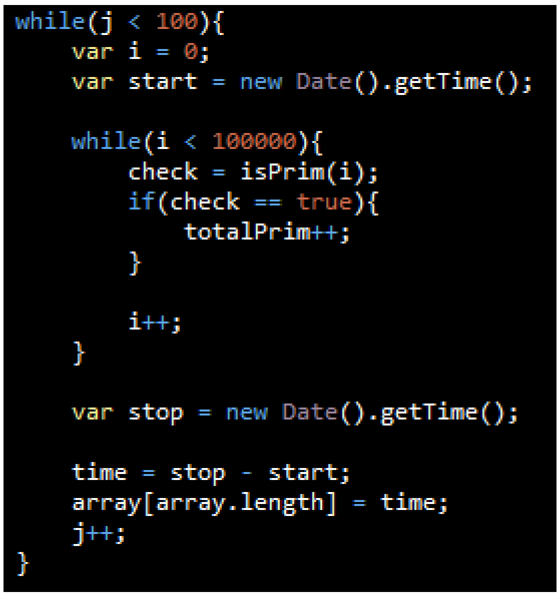
Database benchmark

```
while (i < 100){
    timer.setBase(SystemClock.elapsedRealtime());

    FeedReaderDbHelper mDbHelper = new FeedReaderDbHelper(this);
    SQLiteDatabase db = mDbHelper.getWritableDatabase();
    // Create a new map of values, where column names are the keys
    ContentValues values = new ContentValues();
    values.put(FeedEntry.COLUMN_NAME_ENTRY_ID, ID );
    values.put(FeedEntry.COLUMN_NAME_TITLE, TITLE);
    values.put(FeedEntry.COLUMN_NAME_CONTENT, CONTENT);

    // Insert the new row, returning the primary key value of the new row
    long newRowId;
    newRowId = db.insert(
             FeedEntry.TABLE_NAME,
             FeedEntry.COLUMN_NAME_CONTENT,
             values);


    timer.stop();
    time = SystemClock.elapsedRealtime() - timer.getBase();
    result[i] = (int) time;
    i++;
}
```

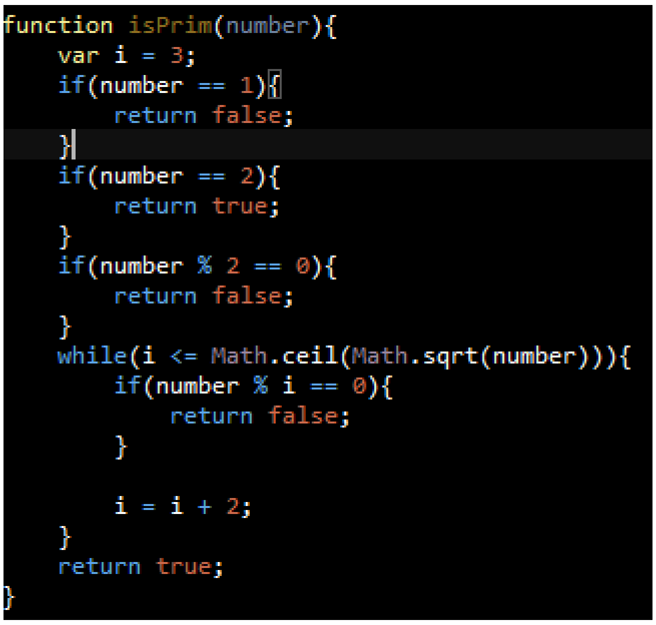
```
while (i < 100){
    timer.setBase(SystemClock.elapsedRealtime());

    FeedReaderDbHelper mDbHelper = new FeedReaderDbHelper(this);
    SQLiteDatabase db = mDbHelper.getReadableDatabase();

    String[] projection = {
        FeedEntry._ID,
        FeedEntry.COLUMN_NAME_TITLE,
        FeedEntry.COLUMN_NAME_CONTENT,
        };

    String sortOrder =
        FeedEntry.COLUMN_NAME_CONTENT + " DESC";

    Cursor c = db.query(
        FeedEntry.TABLE_NAME,  // The table to query
        projection,           // The columns to return
        null,                // The columns for the WHERE clause
        null,               // The values for the WHERE clause
        null,              // don't group the rows
        null,             // don't filter by row groups
        sortOrder        // The sort order
        );

    StringBuffer buffer=new StringBuffer();
    while(c.moveToNext())
    {
       buffer.append("Row ID: "+c.getString(0)+"\n");
       buffer.append("Title: "+c.getString(1)+"\n");
       buffer.append("Marks: "+c.getString(2)+"\n\n");
    }

    timer.stop();
    time = SystemClock.elapsedRealtime() - timer.getBase();
    result[i] = (int) time;
    i++;
```

B. Titanium source code

General benchmark

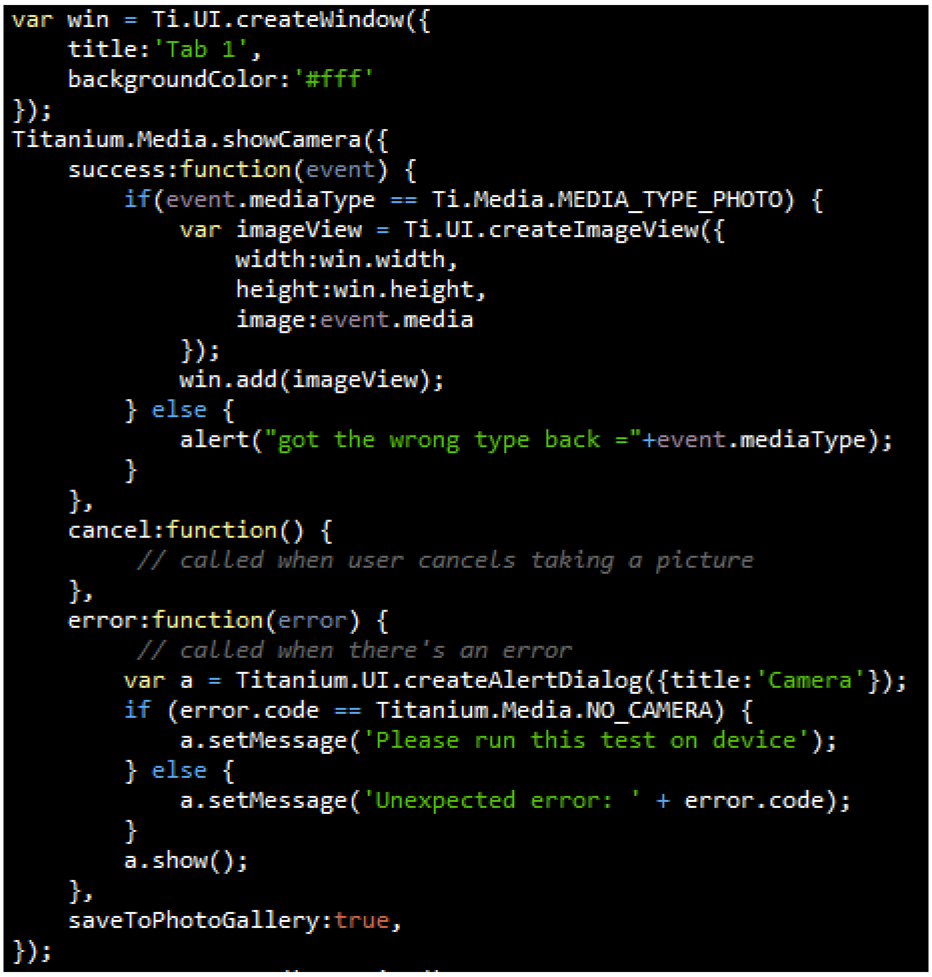


```
while(j < 100){
    var i = 0;
    var start = new Date().getTime();

    while(i < 100000){
        check = isPrim(i);
        if(check == true){
            totalPrim++;
        }

        i++;
    }

    var stop = new Date().getTime();

    time = stop - start;
    array[array.length] = time;
    j++;
}
```



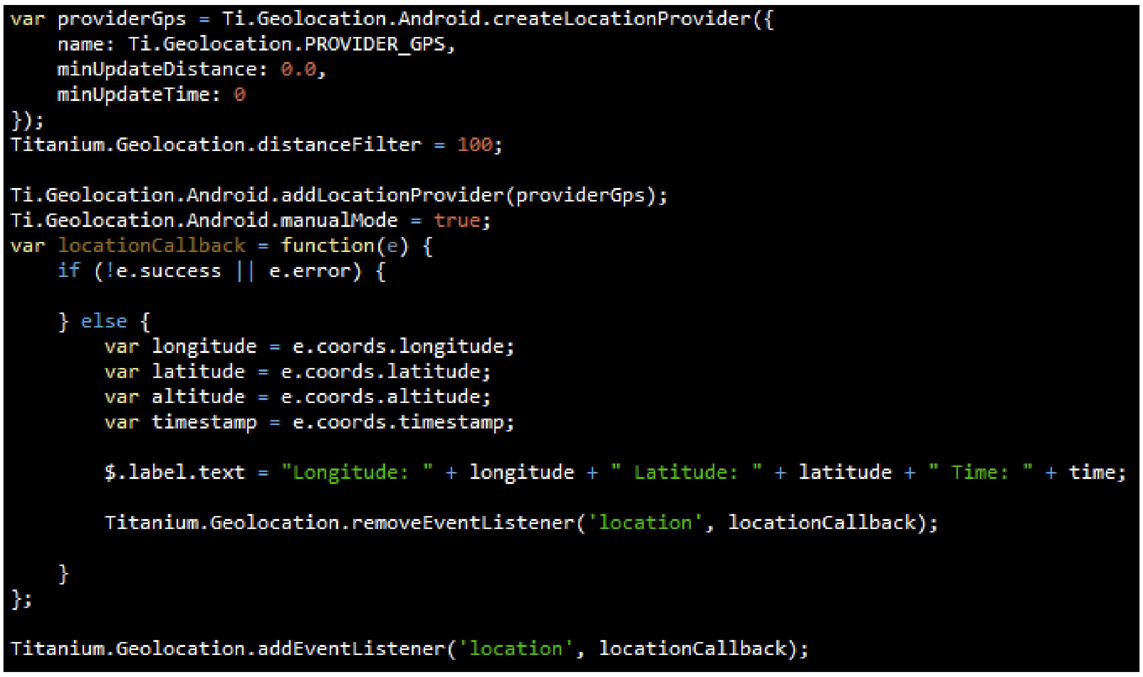
```
function isPrim(number){
    var i = 3;
    if(number == 1){
        return false;
    }
    if(number == 2){
        return true;
    }
    if(number % 2 == 0){
        return false;
    }
    while(i <= Math.ceil(Math.sqrt(number))){
        if(number % i == 0){
            return false;
        }

        i = i + 2;
    }
    return true;
}
```

Camera benchmark



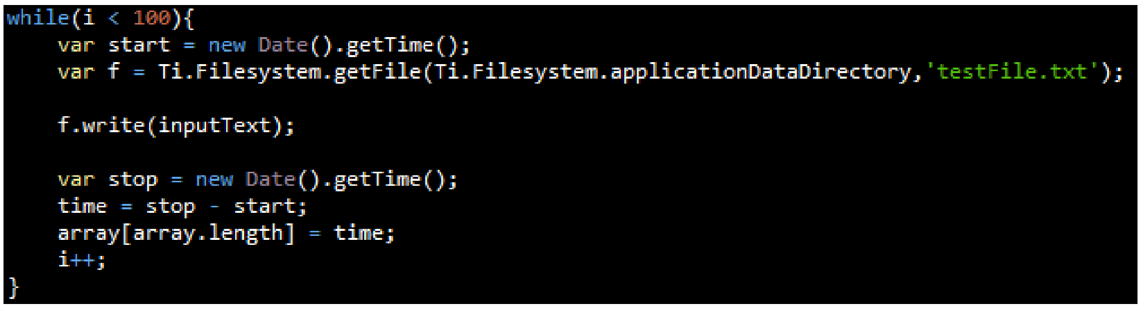
```
var win = Ti.UI.createWindow({
    title:'Tab 1',
    backgroundColor:'#fff'
});
Titanium.Media.showCamera({
    success:function(event) {
        if(event.mediaType == Ti.Media.MEDIA_TYPE_PHOTO) {
            var imageView = Ti.UI.createImageView({
                width:win.width,
                height:win.height,
                image:event.media
            });
            win.add(imageView);
        } else {
            alert("got the wrong type back ="+event.mediaType);
        }
    },
    cancel:function() {
         // called when user cancels taking a picture
    },
    error:function(error) {
         // called when there's an error
        var a = Titanium.UI.createAlertDialog({title:'Camera'});
        if (error.code == Titanium.Media.NO_CAMERA) {
            a.setMessage('Please run this test on device');
        } else {
            a.setMessage('Unexpected error: ' + error.code);
        }
        a.show();
    },
    saveToPhotoGallery:true,
});
```

GPS benchmark

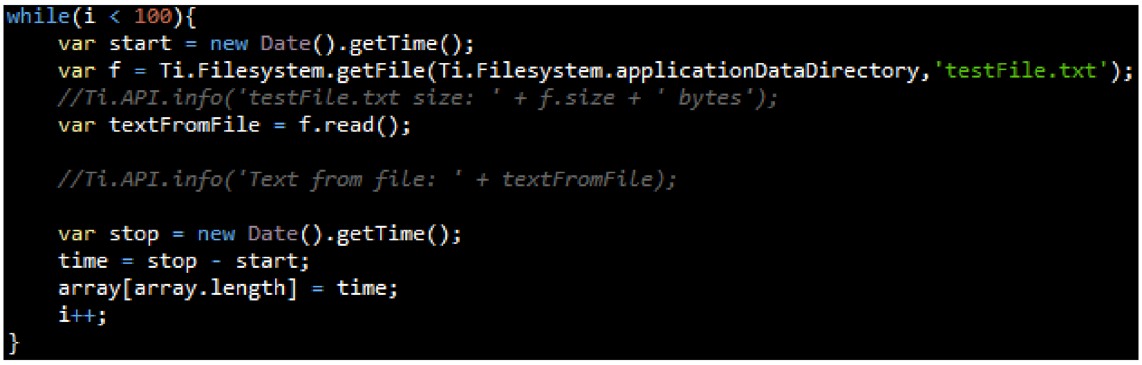


```
var providerGps = Ti.Geolocation.Android.createLocationProvider({
    name: Ti.Geolocation.PROVIDER_GPS,
    minUpdateDistance: 0.0,
    minUpdateTime: 0
});
Titanium.Geolocation.distanceFilter = 100;

Ti.Geolocation.Android.addLocationProvider(providerGps);
Ti.Geolocation.Android.manualMode = true;
var locationCallback = function(e) {
    if (!e.success || e.error) {

    } else {
        var longitude = e.coords.longitude;
        var latitude = e.coords.latitude;
        var altitude = e.coords.altitude;
        var timestamp = e.coords.timestamp;

        $.label.text = "Longitude: " + longitude + " Latitude: " + latitude + " Time: " + time;

        Titanium.Geolocation.removeEventListener('location', locationCallback);

    }
};

Titanium.Geolocation.addEventListener('location', locationCallback);
```

Storage benchmark

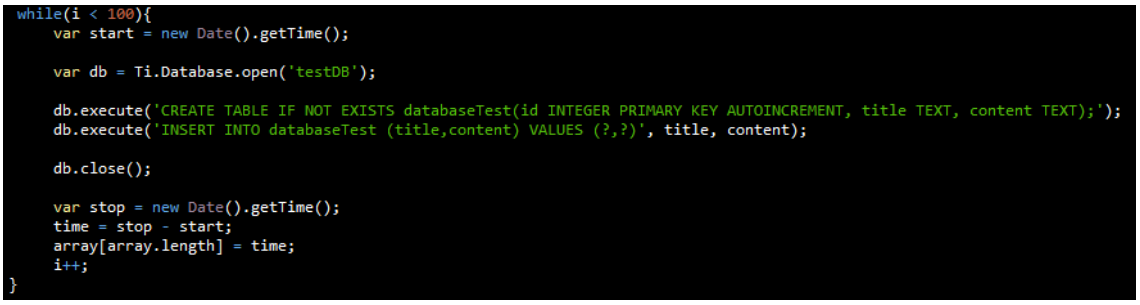


```
while(i < 100){
    var start = new Date().getTime();
    var f = Ti.Filesystem.getFile(Ti.Filesystem.applicationDataDirectory,'testFile.txt');

    f.write(inputText);

    var stop = new Date().getTime();
    time = stop - start;
    array[array.length] = time;
    i++;
}
```

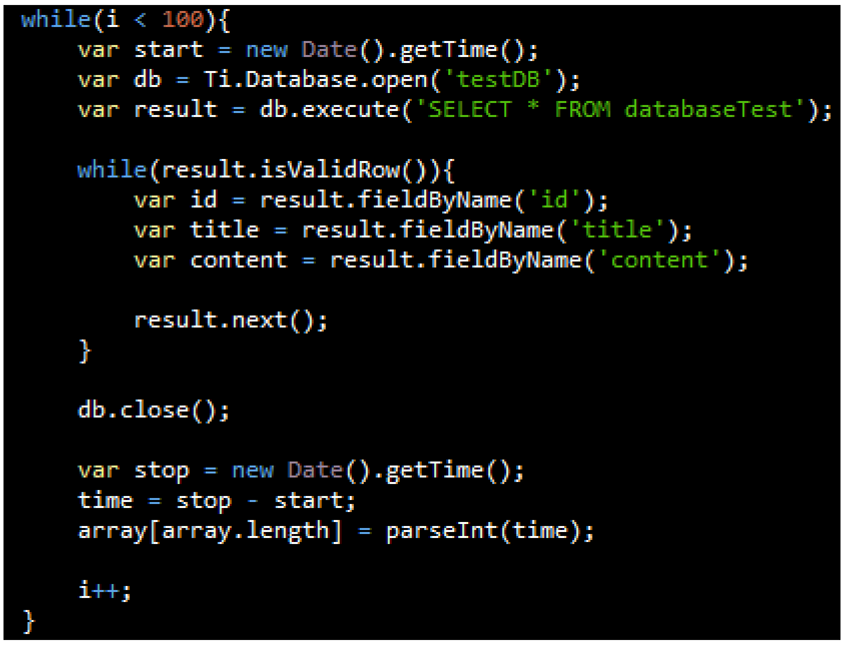


```
while(i < 100){
    var start = new Date().getTime();
    var f = Ti.Filesystem.getFile(Ti.Filesystem.applicationDataDirectory,'testFile.txt');
    //Ti.API.info('testFile.txt size: ' + f.size + ' bytes');
    var textFromFile = f.read();

    //Ti.API.info('Text from file: ' + textFromFile);

    var stop = new Date().getTime();
    time = stop - start;
    array[array.length] = time;
    i++;
}
```

Database benchmark



```
while(i < 100){
    var start = new Date().getTime();

    var db = Ti.Database.open('testDB');

    db.execute('CREATE TABLE IF NOT EXISTS databaseTest(id INTEGER PRIMARY KEY AUTOINCREMENT, title TEXT, content TEXT);');
    db.execute('INSERT INTO databaseTest (title,content) VALUES (?,?)', title, content);

    db.close();

    var stop = new Date().getTime();
    time = stop - start;
    array[array.length] = time;
    i++;
}
```



```
while(i < 100){
    var start = new Date().getTime();
    var db = Ti.Database.open('testDB');
    var result = db.execute('SELECT * FROM databaseTest');

    while(result.isValidRow()){
        var id = result.fieldByName('id');
        var title = result.fieldByName('title');
        var content = result.fieldByName('content');

        result.next();
    }

    db.close();

    var stop = new Date().getTime();
    time = stop - start;
    array[array.length] = parseInt(time);

    i++;
}
```