EvaluatingAndroid best practices for performance

Aline Rodrigues Tonini, Marco Beckmann, Júlio C. B. de Mattos, Lisane Brisolara de Brisolara

Centro de Desenvolvimento Tecnológico, Universidade Federal de Pelotas
Pelotas, RS, Brazil

{artonini, mbeckmann, julius, lisane}@inf.ufpel.edu.br

Abstract—**To improve Android code performance, Google proposed a number of coding best practices that aim to optimize the code through analysis and refactoring. This work studies and evaluatesa subset of these best practices, including an analysis of their impact on the performance of experimental codes and real Android applications. Experimental results demonstrate a positive impact of these evaluated practices on the performance. Our experiments reinforce that developers can avoid overheads and improve performance by the use of coding best practices.**

***Keywords*— Android; Best Practices; Performance; Optimization; Efficiency**

INTRODUCTION

With the technological advances of recent years, the development of applications for mobile devices hasgrown exponentially[1]. Most of these devices run Android Operating System. The Android platform was developed by Open Handset Alliance [2], a group formed by several companies led by Google. The purpose of the alliance is to provide a standardized environment for the development todifferent communication devices.

Android represents an open solution with development tools, a large support to many devices, as well an operating system.This platform supports the development using Java, one of the most used programming language. In addition, Android has connexion with Google services.The combination of these characteristics becomesthe development for Android easy and advantageous [3].

However, mobile applications are significantly different from traditional applications, mainly due to limited resources available on mobile devices (e.g. battery, memory, etc.).Thus, the software should be developed considering these restrictions and optimization should be applied in order to obtain an efficient code. Moreover, Android application optimization is a hard task, mainly because an application can be executed in different devices with different processors, as well as several versions of the Virtual Machine [3]. The main problem is how to ensure that the application works well across a wide variety of devices, and how to ensure that the code is efficient.

Addressing this problem, Google proposes best practicefor Android development [4], focusing on performance improvement.The proposed best practices are simple tips that improve the code efficiency.This work studies these practices and evaluates through experimentstheir impact on the code performance when executed on Android based devices.

The remaining of the paper is organized as follows. Section II presents the related work. The Google best practicesare presented in Section III. Experimental results are presented and discussed in Section IV. Section VI concludes and points out directions for future work.

RELATED WORK

Several authors have addressed the problem of code efficiency for mobile applications [5][6][7][8]. In [5] the impact of the software abstraction usage in embedded systems efficiency is analysed through two benchmarks for Android performance evaluation.

In [6] a tool was proposed, which automatically refactors Android applications transferring some computation-intensive tasks from a smartphone to a server in order to improve the application efficiency. The refactoring is performed at byte code levelgenerating an implementation that supports on-demand offloading. Afterthat, according to [6], the offloaded apps execute about 46-97% faster, as well as its energy consumption is reduced about 27-83%.

Another study evaluatesthe efficiency of the native code and compares it to Java Dalvik code on real Android devices[7]. According to this study, the use of native code is 34,2% faster than Java code.

Likewise, in [8]experiments evaluate performance, potency and energy consumption fordifferent Java implementationsof algorithms of the same complexityrunning on Android devices. The goal is to determinethe best algorithms for specific applications running on this platform.

These efforts show thatthe worries with code efficiency mainly related to performance, and energy consumptionare realand constant for the Android platform.These works propose or evaluate optimizations in different levels, as native code, byte code and algorithms. However,neither effort considers optimizationsatimplementation level, or evaluates the impact of the best practices proposed by Googleon theperformance of real applications.

BEST PRACTICES

This section summarizes the best practices for performance proposed by Google to be incorporated in the Android application process development. According to the studyconductedby Google [4], the use of best practices provides better overall performance in the application.

One of the best practices suggested that designer must avoid the creation of unnecessary objects, because it is always costly.The allocation of excessive objects impliesin a periodic

garbage collection, causing a negative impact on the application performance.

Another practice indicates the use of static methodsinstead of virtual ones. Thus, methods that do not access attributes of object should be declared as *static*. According to Google, these invocations will be about 15% - 20% faster.

Another practices concerns to declaration and usage of constants and recommends the use of *static final* forprimitive constants and Strings. When using the *final*reserved word in the constant definitions, the access will be faster. This occurs because the class does not require the <*clinit*> method, generated during the class initialization, since those constants are stored in the *.dex* file. However, this practice is valid only to primitive types and constant Strings.

Inobject-oriented languages like C++ and Java is common the use of getters/setters methods to access class attributes.However, in Android this isnot a good practice, because method invocations are expensive. Thus, the use of getters/setters methodsshould be avoided. According to Google, the time to directly access an attribute is three times faster than trough getter/setter methods on devices without JIT (Just-in-time) and about seven times faster with JIT recourses.

Concerning to manipulation of arrays, Google best practices also present suggestion about the use of the appropriated *for* syntax. Using the Java syntaxes, it is possible to iterate an Array using three different forms, as illustrated in the code from Fig. 1. The *for-each* syntax (used in *two()*) can be used to define collections that implement an iterative interface to Arrays. According to Google, the use of *for-each*in collections is three times faster (with or without JIT) compared to the use of the traditional *for* (used in *zero()*).

```
static class Foo {
    int mSplat;
}

Foo[] mArray = ...

public void zero(){
    int sum = 0;
    for
(int i = 0; i < mArray.length; ++i){
        sum + = mArray[i].mSplat;
    }
}

public void one(){
    int sum = 0;
    Foo[] localArray = mArray;
    int len = localArray.length;
    for (int i = 0; i < len ; ++i){
        sum + = localArray[i].mSplat;
    }
}

public void two(){
    int sum = 0;
    for( Foo : a ){
        sum + = a.mSplat;
    }
}
```

Fig. 1: Code fragment of the appropriate *for* practice

The iteration used in the *zero*method is slow, because JIT does not optimize the ways to obtain the Array length at each loop iteration. The iteration in *one*is faster, because uses local variables and the array size is obtained before the loop and not at each iteration. The last implementation uses the *for-each*syntax,introduced in Java 1.5, which is fasterthan *one*on devices without JIT and indifferent in devices with JIT.

The best practices also indicate the use of package access instead private access in private inner classes. This practice is applied when an inner class another need to access attributes of external class.The virtual machine considers the direct accessof inner class to attributes of an external class as illegal, because they are different classes. To avoid this problem, attributes and methods from an inner class, should use package visibility, which is provided by the publicand protected modifiersor when no modifier is used[9]. Applying this practice, one can avoid overhead in applicationsthat use inner class at critical points of performance.

Another best practice indicatesthat the use of float point for Android is not recommended. According Google, the use of float point is two times slower than integer[4].

METHODOLOGY

For all experiments, the emulator provided into the Android SDKis used. This emulator is configured to run on Android 4.1.2 using an API 15, and simulating the ARM EABI V7a processor. Our experiments do not consider JIT.

The *android.os.Debug* library [10]is used to generate the trace files required for performance estimation. The *startMethodTracing()* and *stopMethodTracing()* methods from this library are used to indicate start and end point of trace. The execution time is obtained using Traceview tool[11], which provides values for the Exclude and Include CPU Time.Most experiments consider Exclude CPU Time and only one experiment uses Include CPU Time. Thirty executions are conducted for each experiment and medium values are comparedusing a t-student statistic test to verify statistical significance of the observed differences.

EXPERIMENTAL RESULTS

Two Google best practices are evaluated in this work. Firstly, these are analysedusing experimental codes, and finally these are applied on a real Android application.The results presented in this section were obtained using the Android 4.1.2. However, we also evaluated these practices for Android 1.5, 2.1 and 3.0 and the results were representative in these different versions of the platform.

Analysing a set of Google best practices

Firstly, the practice that suggests to avoidgetter/settersmethods is evaluated using the code fragmentsillustrated in Fig.2, Fig.3 and Fig.4. Fig. 2 illustrates the class definition, which has an attribute that can be accessed directly or by the *getGetter()* method. These different solutions are represented in Fig. 3 by *withGetters()* and *withoutGetters()* methods, where *withoutGetters()*represents the solution that uses the good practice, differently of *withGetters()*.To evaluate both solutions, the code fragment presented in Fig. 4 isused for tracing these methods and estimate its performance and the obtained results are presented in Table I (medium execution

time and standard deviation).In these experiments, the withoutGetter method is 2,93 times faster than withGetter.

```
public class Getter {
    int getter;

    int getGetter() {
        return getter;
    }
}
```

Fig. 2: Experimental code fragment– Avoiding getters/setters.

```
void withoutGetter(Getter get) {
    int i;
    i = get.getter;
}

void withGetter(Getter get) {
    int i;
    i = get.getGetter();
}
```

Fig. 3:Methods used for tracing.

```
Debug.startMethodTracing("Getter", 40000000);
for (int i = 0; i < 10000; i++)
    withoutGetter(get);
    //withGetter(get);
Debug.stopMethodTracing();
```

Fig. 4: Code fragment for performance evaluation of avoiding getters.

Table I: Results of the avoiding getters/setters methods

Method	Med. Exec.Time(ms)	σ
withGetter	547,435	3,882
withoutGetter	187,156	5,507

The impact of using the appropriate *for* syntax is observed comparing the code fragments from Fig. 5, which represent three different implementations (*zero*, *one*, and *two*) of a loop. These implementations are evaluated using the tracing code illustrated in Fig. 6 and results are presented in Table II. In the experiments, *Two* is 1,25 faster than *zero* and 1,05 faster than *one*.

```
public void povoa(FooLoop mArray[]) {
    for (int i = 0; i < mArray.length; ++i) {
        mArray[i] = new FooLoop();
        mArray[i].mSplat = 1;
    }
}

public void zero(FooLoop mArray[]) {
    int sum = 0;
    for (int i = 0; i < mArray.length; ++i) {
        sum += mArray[i].mSplat;
    }
}

public void one(FooLoop mArray[]) {
    int sum = 0;
    FooLoop[] localArray = mArray;
    int len = mArray.length;
    for (int i = 0; i < len; ++i) {
        sum += localArray[i].mSplat;
    }
}

public void two(FooLoop mArray[]) {
    int sum = 0;
    for (FooLoop a : mArray) {
        sum += a.mSplat;
    }
}
```

Fig.5: Experimental code fragment - Appropriate for practice

```
FooLoop[] mArray = new FooLoop[10000];
povoa(mArray);

Debug.startMethodTracing("For0", 40000000);
zero(mArray);
//one(mArray);
//two(mArray);
Debug.stopMethodTracing();
```

Fig. 6: Code fragment with tracing.

Table II: Results of the appropriate for practice

Method	Med. Exec.Time(ms)	σ
Zero	4,171	1,362
One	3,4974	1,062
Two	3,322	0,810

Evaluating impact on the performance of a real application

In this section,one application is used to demonstrate the impact of the two studied best practices on real applications.The chosen application is the OpenSudoku [12], which was used toevaluate the impact of the for syntax selection as well as of the avoiding getters/setters methods. These impacts are firstly evaluatedseparately and after that simultaneously.

To evaluate the two best practices, the code fragment illustrated in Fig. 7 is used. This code illustrates the *validate()* method after the best practicesbe applied.

```
protected boolean validate() {
    boolean valid = true;
    contvalidate+=1;
    Map<Integer, Cell> cellsByValue = new HashMap<Integer, Cell>();
    //int len = mCells.length;
    Cell cell;
    int value;
    //for (int i = 0; i < len; i++){
    for (Cell a :mCells) {
        //cell = mCells[i];
        cell = a;
        //value = cell.getValue;
        value = cell.mValue;
        if (cellsByValue.get(value) != null) {
            //mCell[i].setValid(false);
            a.mValid=false;
            cellsByValue.get(value).setValid(false);
            valid = false;
        } else {
            cellsByValue.put(value, cell);
        }
    }
    return valid;
}
```

Fig. 7: *validade()* method after best practices be applied.

Experimental results obtained for the different *for* syntaxes are depicted in Fig. 8. This comparison is based on medium values for CPU Exclude Time of original and optimized code. The*for-each* syntaxhas reduced execution time(1775,0134 ms) compared to the traditional syntax (1792,9221ms), with standard deviation of 23,5669 ms for optimized code and 17,6413 ms for original code (commented in Fig. 7).By criteria conventional, this difference is considered statisticallysignificant.

Fig. 9 presents the CPU Include Time obtained for the *validate()* method, considering the versions with and without Getters/Setters. Experiments show the positive impact of replacing getter and setter invocation by direct accesses. The

execution time for original code (using getter/setter methods) is 7046,0340ms(σ =101,2620ms). Applying the best practice, the execution time is reduced to 4195,0565ms (σ =87,7723ms). In this experiment, we used Include time in order to consider invocations inside of the evaluated method. This difference is considered statistically significant.

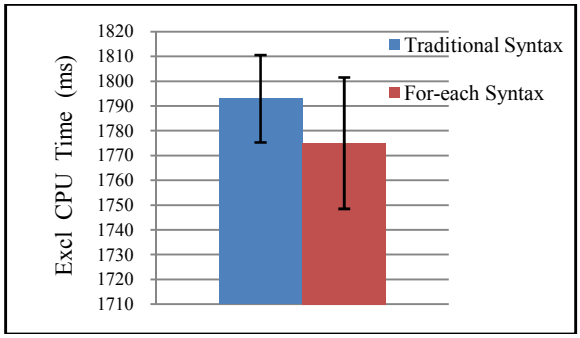

Fig. 8: Comparison between Traditional For syntax and For Each syntax

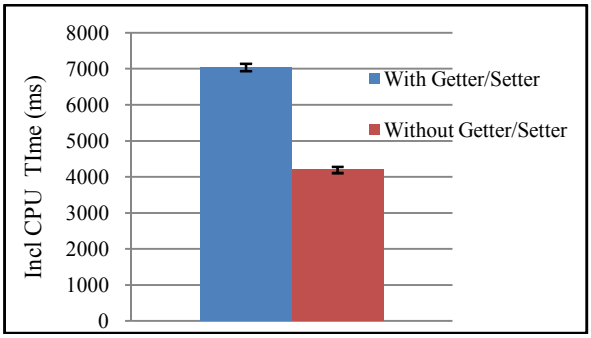

Fig. 9: Comparison between solutionswith and without Getter/Setter methods

After the separate evaluation, we applied the two practices simultaneously (Fig. 7) and obtained results are illustrated in Fig. 10. In these experiments, the best practices reduced the execution timeof the application in133,6%, which can be considered a significantdifference.

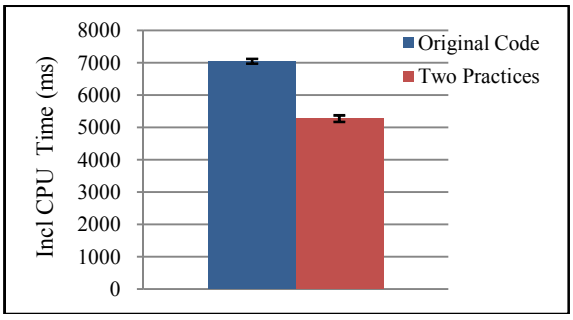

Fig. 10: Comparison between original code and two practices applied.

CONCLUSIONS

This paper presents a preliminarystudy of the Android best practices for performance in which a subset of these practices are revised and evaluatedthrough experiments. This work presents only the analysis of the most significant best practices, “for syntax” and “avoiding getters/setters”, due limitation on the number of pages. Our experiments firstly use experimental codes and finally analyze the impact of these two best practices on a real Android application.Experimental results demonstrate a significant and positive impact when getter/setters methods are avoided and when the *for-each* syntax is used. As future work, we plan extend these experiments to evaluate the impact of the best practices on energy consumption, an important issue for mobile devices.

ACKNOWLEDGMENT

The authors acknowledge financial support received from Fapergs, and NESS project (PRONEX-10/0043-0).

REFERENCES

[1] A. Wasserman, "Software Engineering Issues for Mobile Application Development," in *Proc. of the FSE/SDP Workshop on Future of Software Engineering Research*, New York, 2010, pp. 397-400.

[2] OHA. (2013) Open Handset Alliance. [Online]. http://www.openhandsetalliance.com/

[3] Y.L. Jie, Z. X. Yi, C. Da, and Z. Siting, "Development and Implementation of Eclipse-based File Transfer for Android Smartphone," in *The 7th International Conference on Computer Science & Education*, Melbore, Australia, 2012.

[4] Google. (2013) Best Practices. [Online]. http://developer.android.com/training/articles/perf-tips.html

[5] V.B. Camara, U. Corrêa, and L. Carro, "Impacto de uso da abstração de software no desempenho de sistemas embarcados complexos," in *Brazilian Symposium on Computing System Engineering*, Florianópolis, 2011.

[6] Y. Zhang et al., "Refactoring Android Java Code for On-Demand," in *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, New York, 2012, pp. 233-248.

[7] C. M. Lin, J. H. Lin, C. R. Dow, and C. M. Wen, "Benchmark Dalvik and Native Code for Android System," in *Second International Conference on Innovations in Bio-inspired Computing and Applications*, 2011, pp. 320-323.

[8] A. Vieira, D. Debastiani, L. Agostini, F. Marques, and J. C. B. Mattos, "Performance and Energy Consumption Analysis of Embedded Applications based on Android Platform," in *Brazilian Symposium on Computing System Engineering*, Natal, 2012.

[9] Oracle. (2013) Controlling Access to Members of a Class. [Online]. http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

[10] Google. (2013) Using DDMS. [Online]. http://developer.android.com/tools/debugging/ddms.html

[11] Google. (2013) Profiling with Traceview. [Online]. http://developer.android.com/tools/debugging/debugging-tracing.html

[12] R. Masek. (2013) Open Sudoku. [Online]. http://code.google.com/p/opensudoku-android/