https://doi.org/10.5392/IJoC.2017.13.4.039

Towards Performance-Enhancing Programming for Android Application Development

Dong Kwan Kim
Department of Computer Engineering
Mokpo National Maritime University, Mokpo, Jeonnam 58628, Rep. of Korea

ABSTRACT

Due to resource constraints, most of Android application developers need to address potential performance problems during application development and maintenance. The coding styles and patterns of Android programming could often affect the execution time and energy efficiency which are utilized by the Android applications. Thus, it is necessary for application developers to apply performance-enhancing programming practices for mobile application development. This paper introduces performance-enhancing best practices for Android programming, and further, it evaluates the impact of these practices on the CPU time of the application. The original version with the performance-worsening code has been refactored to become an efficient version without changing its functionality. To demonstrate the efficiency of the proposed approach, each coding pattern was evaluated by measuring the CPU time under the controlled runtime environment. Furthermore, the Android applications were evaluated and compared via the CPU time of the original version, with that of the refactored version. These experimental results indicate that, by -using the proposed programming practices, the Android developer can develop performance-efficient mobile applications.

Key words*: Android Programming, Performance Efficiency, Programming Practices, Code Refactoring.*

1. INTRODUCTION

Android is one of the most popular mobile operating systems which are primarily designed for smartphones and tablets. The Android mobile platform is based on the Linux kernel and supports various Java and C / C ++ libraries to extend its basic functions. Due to its openness, Android is growing and spreading over the world. Many software developers can share fundamental Software Development Kits (SDKs), additional development tools, and extra APIs via the Android developer site [1]. Such an open policy is promoting the spread of Android programming among software developers. In addition, Google Play [2], an Android app store, is an online communication space for app developers where they can distribute and manage their Android applications. According to [3], as of February 2017, Google Play features over 2.7 million Android applications including games, movies, music, and books. Such a steep increase in developing Android apps indicates that Android application development could be relatively simplified by using free Android SDKs, tools, and APIs.

Since Android systems run on mobile devices with the limited battery lifetime, software developers cannot ignore the characteristics of mobile systems when developing mobile applications. There are a variety of ways to efficiently use limited resources of mobile devices. It can be divided into hardware-based and software-based approach. This paper attempts to improve the performance of mobile applications via a software-based approach. In the perspective of the Android programming, the software-based performance improvement can be applied into source code level and bytecode level according to the object of the performance enhancement. This paper aims at enhancing the performance of the Android application at the source code level.

The source code of Android applications may include coding styles or patterns that could result in performance degradation. Such coding styles or patterns can be referred as code smells that are any symptom introduced in application design or implementation phases in the source code of a program. The code smell is a sort of potential indications to maliciously impact on program execution time or runtime resource consumption. Such a code smell can be often removed by transforming its code structure. We call it code refactoring. Code refactoring does not change the semantics of an original program. This paper provides performance-enhancing programming practices to improve the performance of the Android application. Such Android programming practices can effectively remove bad code smells that are related with Android performance issues. Since the Android platform supports Native Development Kit (NDK) [4], this paper focuses on refactoring not only Java code but also C/C++ code.

** Corresponding author, Email: dongkwan@gmail.com*
Manuscript received Aug. 18, 2017; revised Sep. 26, 2017; accepted Sep. 27, 2017

The rest of this paper is organized as follows. Section 2 analyzes existing research outcomes by comparing with the proposed approach. Section 3 describes the proposed approach to apply Android best practices for performance enhancement. Section 4 presents the results of the case study to demonstrate the effectiveness of the proposed methodology. Section 5 describes the strength and weakness of the proposed approach. Finally, Section 6 remarks the conclusions and future work directions.

2. BACKGROUND AND RELATED WORK

Even if software looks like operating correctly at present, it could contain potential problems in terms of non-functional aspects such as performance, security, and scalability. Problem-free software systems can also be problematic through software evolution. Software engineers believe that such potential problems could be introduced due to poor design and implementation choices during software development lifecycle. They call it a code smell that refers to any symptom in a program. Code smell possibly indicates a deeper problem over software evolution. Unnecessary code, dead code, and code duplication are the typical examples of the bad code smell that can make software systems messier over time.

In particular, some code smells are closely related with the performance of an application. Such a performance code smell should be taken care since it can result in performance degradation. A quality smell catalogue including the performance code smell for the Android platform was introduced in [5], [6]. The catalogue includes 30 quality smells and refactorings to improve the quality of an Android application. Along with the refactoring techniques, Android programming practices were presented in [7], [8] for saving energy consumption. Energy-saving programming skills were proposed and evaluated their impacts on Android applications by measuring energy consumption and execution time.

Many software tools were developed to leverage such programming practices. They can be used to detect and eliminate bad code smells using software metrics and code refactoring methods [9]-[12]. Since Android programming is based on the Java programming language, object-oriented metrics for Java applications could be applied for Android applications. The well-known object-oriented metrics contain Line of Code (LOC), Depth of Inheritance Tree (DIT), Coupling Between Objects (CBO), Response for a Class (RFC), Weighted Methods per Class (WMC), Number of Children (NOC), Cyclomatic Complexity (CC), and Lack of cohesion in methods (LCOM). These metrics can be used to find bad code smells of Android applications by applying the thresholds of the pre-defined metric values.

Refactoring techniques can be used to remove the bad code smells which are identified by the object-oriented metrics [13]. Martin Fowler and his colleagues introduced code refactoring and presented a catalogue of common refactorings and code transformations in order to address bad code smells [8]. Representative refactoring techniques include Extract Class, Move Method, Move Field, Inline Class, and Introduce Parameter Objects.

In particular, more specific refactoring techniques were presented for Android applications by considering the mobile characteristics of the Android platform such as limited memory, storage capacity, battery lifetime, and processor power.

Code refactoring techniques for eliminating energy bad smells were proposed in [14]-[16]. They found energy bad code smells and investigated their impacts on inappropriate energy consumption in the Android application. They also revealed energy-efficient refactorings to improve the energy consumption of the Android application. Since the energy efficiency is related with the performance of the Android application, Hecht explored the performance impacts of three Android performance code smells including Internal Getter/Setter, Member Ignoring Method, and HashMap Usage [17].

3. ANDROID PROGRAMMING PRACTICES FOR PERFORMANCE

This section focuses on presenting programming practices which enable software developers to build performance-enhanced Android applications. Typical performance code smells are listed and code transformation methods are presented.

Since Android applications can be written in C/C++ as well as Java by using Android SDK and NDK, this paper provides refactoring methods for not only Java code but also C/C++ code. The NDK uses Java Native Interface (JNI) of the Java platform to connect Java layers with C/C++ layers. JNI is a programming framework for a low-level software development environment.

Fig, 1 shows an example of using JNI to call native code in an Android Activity. *stringFromJNI()* is declared as a native method. The keyword *native* indicates that a Java method is implemented in native code. When another method calls the native method, the native implementation will be executed. The lower part in Fig. 1 shows a native implementation of the native method, *stringFromJNI().*

//Declaration of native method in Android Activity

```
public class MainActivity extends Activity{
   //A native library
   static {  System.loadLibrary("native-lib"); }
   //Declaration of a native method
   public native String stringFromJNI();
}
```

// Implementation of native method in native code

```
extern "C"
JNIEXPORT jstring JNICALL
Java_example _MainActivity_stringFromJNI(
      JNIEnv *env, jobject) {
   std::string hello = "Hello from Native Code";
   return env->NewStringUTF(hello.c_str());
}
```

Fig. 1. Using Java Native Method to Call Native Code.

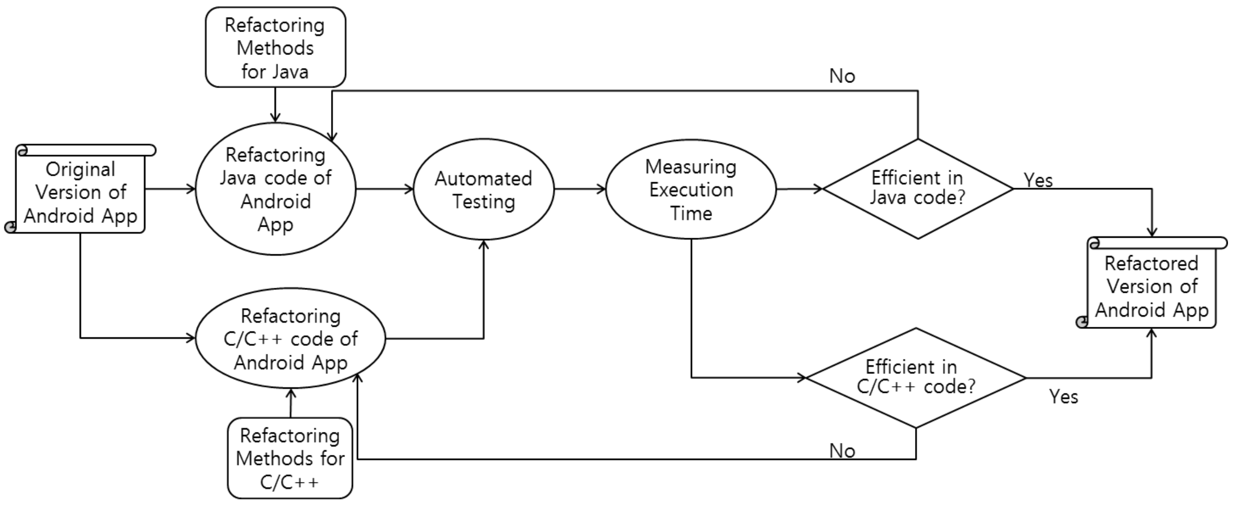
Refactoring Methods for Java
Original Version of Android App
Refactoring Java code of Android App
Automated Testing
Measuring Execution Time
No
Efficient in Java code?
Yes
Refactored Version of Android App
Refactoring C/C++ code of Android App
Refactoring Methods for C/C++
Efficient in C/C++ code?
Yes
No

Fig. 2. The Overall Procedure for Building Performance-Enhancing Android Applications.

Fig. 2 illustrates the overall procedure of the proposed approach to create performance-enhancing Android applications. The original version of an Android application will be transformed into the performance-aware version through the stepwise and systematic process. The Android application may be written in both Java and C/C++. Android supports all Java language features and APIs. Developers can also integrate C/C++ code with Java code for their applications by using NDK tools and libraries. Such NDK facilities enable Android applications to be harmonized with third-party C/C++ libraries. Therefore, there are the refactoring activities for Java code and C/C++ code. The refactoring activity for Java code proceeds according to the refactoring methods for the Java programming language. Similarly, C/C++ code is refactored by following the refactoring rules and guidelines for C/C++. The refactored versions are tested automatically by a GUI testing tool in order to evaluate if their original functions are preserved after code refactoring. Once the refactoring process is completed, the execution time of the original and refactored versions is measured to see if the refactored version is more efficient than the original version. If the refactored version consumes less CPU time, it is selected. Otherwise, it is considered to be refactored again. The measurement of the execution time is performed by separating Java code from C/C++ code.

Table 1 presents a list of the proposed Android programming practices which can be applied to improve the performance of an Android application.

❑ Enhanced *For* Loop :

For the better performance, the Android developers' web site suggests to use the enhanced *for* loop syntax (a.k.a. for-each statement) by default [18]. The enhanced *for* statement is introduced in Java 5 and can be used to iterate all elements of arrays and *Iterable* objects including collections such as ArrayList, LinkedList, and HashSet. Compared to a hand-written counted loop, the enhanced *for* expression can iterate all elements in a simple way and construct human-readable code. Fig. 3 shows the code snippet of replacing the typical *for* loop with the *for-each* loop. The typical *for* statement refers to the hand-written counted loop.

Table 1. Definition of Android Programming Practices

Code Styles	Descriptions
Enhanced *For* Loop	-*for* loop syntax for optimization -for-each statements are recommended for better performance
Internal Getter/Setter	-Directly access internal fields without getters/setters -Internal getters/setters are expensive
Local Variables	-Avoid unnecessary global variables within a loop expression -Replace global variables with local variables if needed
Avoid Creating Unnecessary String Objects	-Object management is expensive -Use StringBuffer instead of creating unnecessary String objects
Use Static Final For Constants	The "static" and "final" keywords should be used to declare constants
Inefficient Data Structure	Use SparseArray instead of HashMap<Integer, Object>
Avoid Using Recursive Methods	-Recursive methods are expensive -Use non-recursive methods instead of recursive methods
Avoid Transferring High Volume Data on Slow Network	-Transferring data on a slow network is not efficient for the battery life -Users can send bulk data on WiFi, 3G, etc.
Avoid Early Resource Binding	-Energy-consuming resources of an Android device should be bound as late as possible -More energy will be consumed because of more executed time

The Original Version: Typical *for* statement

```
for (int i = 0; i < myArray.length; i++) {
        sum += myArray[i].val;
}
```

The Refactored Version: *for-each* statement

```
for (Obj myObj : myArray) {
        sum += myObj.val;
}
```

Fig. 3. Replacing typical *for* loops with *for-each* loops.

❑ Internal Getter/Setter:

It is recommended to use getters and setters to allow for accessing private fields outside classes. However, the use of getters and setters within classes can lead to the performance degradation since Android virtual methods are expensive. Thus, one should access directly internal fields without using getters and setters. Getters and setters need to be called to provide public APIs.

❑ Local Variables:

Since its iteration number can vary considerably, the loop statement such as *for* and *while* expressions can impact on the performance of an Android application. Therefore, one should pay attention to the loop expressions when producing performance-critical applications. The loop statement iteratively performs the same operations within the inner block of the loop. Therefore, one has to consider replacing global variables within a loop statement with local variables in order to achieve the performance enhancement. Unlike the enhanced *for* statement, the use of this coding style can be applied into both Java and C/C++.

❑ Avoid Creating Unnecessary String Objects:

Object creation involves in memory allocation and garbage collection. Thus, one should avoid creating unnecessary objects. In particular, the creation of unnecessary String objects can result in unnecessary work and performance degradation. For example, the inappropriate use of the string constructor (e.g., new String("Android")) can produce unnecessary String objects. Since the String object is immutable, the operation of string concatenation needs to create intermediate String objects. It is recommended to use StringBuffer instead of String when one needs to modify the String object.

❑ Use Static Final For Constants:

For the optimization, one needs to explicitly declare constants with the "static" and "final" keywords. Thus, the compiler can generate optimized bytecode to enhance the access time on these constants. It is highly recommended to use appropriate keywords to declare constants for not only performance but also readability.

❑ Inefficient Data Structure:

The use of the collection framework can be beneficial to manage objects in an easy way. However, the misuse of the collection framework can cause the performance problems. In particular, one should carefully use the standard Java HashMap class when the performance matters. It is recommended to use the Android SparseArray class instead of HashMap<Integer, Object>. SparseArray is a good replacement of HashMap when mapping integers to Objects.

❑ Avoid Using Recursive Methods:

In Java, a method can call itself, which is known as recursion. Even if recursion is a beneficial function-call mechanism for computation in some aspects, the use of the recursion can cause the performance degradation. Since recursive methods are expensive, one should use non-recursive methods for embedded applications including mobile platforms.

The last two code styles shown in Table 1 are related to the power consumption of an Android device. For the better energy efficiency, it is beneficial to avoid transferring high volume bulk data over a slow network connection. In addition, it is recommended to avoid binding physical resources too early before they are requested.

4. EXPERIMENTAL RESULTS

To demonstrate the effectiveness of the proposed programming practices, case studies were conducted with the experimental example codes and third-party Android applications. These case studies evaluated C/C++ code as well as Java code because some parts of the programming practices can be applied into C/C++ code. The example code was used to assess the coding styles which are mentioned in the preceding section. And for the realistic assessment, two Android applications were considered. They include Java-only applications and NDK-based applications. During these case studies, the proposed Android programming practices were applied and then the execution time of the modified code was measured. By analyzing the elapsed CPU time, the proposed performance-enhancing programming practices are proved to be meaningful and effective.

All the measurements were performed on a workstation computer with an Intel Core i7 (3.5 GHz) processor and 8 GB RAM, running the 64-bit version of Windows 7. For these experiments, Android Studio 2.3.3 was used as a development tool and a Google Nexus 6 emulator was used as an execution environment with Android 6.0 (Marshmallow).

Table 2 presents the experimental results of measuring the CPU time of the example code. In Table 2, BR, AR, and PER stand for Before Refactoring, After Refactoring, and Performance Efficiency Ratio, respectively. For these experiments, the Android example code for each code style was run iteratively more than 10,000 times and the elapsed CUP time was measured. As the iteration number increases, the performance gains will do.

Table 2. Results of Measuring CPU Time

Code Styles	CPU Time(ms)		BR-AR (ms)	PER (%)
	BR	AR		
Enhanced For Loop	1,769	1,070	699	39.51
Internal Getter/Setter	608	85	523	86.02
Local Variables	2,818	913	1,905	67.60
Avoid Creating Unnecessary String Objects	2,705	2,279	426	15.75
Use Static Final For Constants	5,711	136	5,575	97.62
Inefficient Data Structure	4,133	1,614	2,519	60.95
Avoid Using Recursive Methods	2,827	936	1,891	66.89

The sample code for Internal Getter/Setter contains three internal fields and their corresponding getter methods and runs 9,000 times iteratively to measure the elapsed CUP time.

To evaluate the "Avoid Creating Unnecessary String Objects" case, five String variables are declared via the String constructor. And then, they are concatenated and returned. Meanwhile, the refactored version uses the StringBuffer instance instead of the String instance.

To show the impact of recursion on the performance, a sample code for calculating greatest common divisors was used. The original version contains a recursive method and the refactored version uses a loop construct instead.

Along with the experimental example code, Android applications were selected and evaluated for the more realistic case study. As the experimental example code includes Java and C/C++, Java-only applications and NDK-based applications are selected for this assessment. Table 3 summarizes a list of two Android applications, Snake Game and Bitmap Plasma for the experiments.

Snake Game is an Android version of the simple and typical 2D snake game. Game players need to control a snake-like line by selecting the directional arrow keys (e.g., north, south, east, and west). When the snake eats an apple during gaming, it grows longer. Since this Snake Game is written in Java, the performance-enhancing code styles for Java can be applied for the better performance.

Bitmap Plasma is one of the Android sample applications provided by the Android developer website and uses JNI to render a plasma effect in an Android Bitmap from C code [19]. Therefore, one can refactor the source code of Bitmap Plasma with NDK-specific programming practices.

Table 3. List of the Android Applications for the Experiments

Apps	Descriptions	Remarks
Snake Game	An Android version of the simple 2D snake game	Android Activity with GUI
Bitmap Plasma	-To render a plasma effect in an Android Bitmap from C code -To use JNI	An Application written in Java and C/C++

To measure more accurate CPU time, the original and the refactored versions need to be tested and measured with the exactly same GUI events. Manual event inputs from users can decrease the accuracy of the measured CPU time. In fact, it is almost impossible for users to create the exactly same input events to the two versions of the tested Android application. Therefore, it is required to use a testing tool which can support test automation with minimal manual efforts. In these experiments, the original and modified versions of the Android application were tested by an automated GUI testing tool, *Robotium* which is an Android test automation framework for native and hybrid applications [20]. Robotium is based on JUnit framework and can be integrated with Android Studio. One can edit GUI-based test inputs such as button clicks, text inputs, and mouse movements in Android Studio. And then, such test inputs can be run automatically in the order in which they were created in the original version of a tested application. For the refactored version, the same input events are applied automatically. Therefore, the CPU time of the two versions is measured under the same GUI input events.

In the Snake Game, a game player needs to maneuver a snake-like line by clicking the four direction arrow keys. The original and refactored versions of the Snake Game were tested with the twenty mouse clicks of the direction arrow keys.

The performance-enhancing programming practices were applied in the refactored version of the Snake Game. For example, enhanced *for* loops are used and internal getters/setters are removed in the refactored version. Fig. 4 shows an overridden code of the *onDraw()* method of a View class which is called to refresh the screen periodically by the Android system. While the original version of *onDraw()* uses internal getters, the refactored version of *onDraw()* accesses internal variables without using internal getters.

The Original Version: with Internal Getter

```
public void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    for (int x = 0; x < this.getmXTileCount(); x += 1) {
        for (int y = 0; y < this.getmYTileCount(); y += 1) {
            if (this.mTileGrid[x][y] > 0) {
                canvas.drawBitmap(
                        this.mTileArray[mTileGrid[x][y]],
                        this.getmXOffset() + x * this.getmTileSize(),
                        this.getmYOffset() + y * this.getmTileSize(),
                        mPaint);
            }
        }
    }
}
```

The Refactored Version: without Internal Getter

```
public void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    for (int x = 0; x < mXTileCount; x += 1) {
        for (int y = 0; y < mYTileCount; y += 1) {
            if (mTileGrid[x][y] > 0) {
                canvas.drawBitmap(
                        mTileArray[mTileGrid[x][y]],
                    mXOffset + x * mTileSize,
                    mYOffset + y * mTileSize,
                    mPaint);
            }
        }
}
```

Fig. 4. Original and Refactored Versions of Snake Game

Another example application is Bitmap Plasma which is based on JNI to call C functions from Java methods. Bitmap Plasma was selected to evaluate performance-enhancing programming practices for C/C++ as well as Java. Bitmap Plasma contains a native method, *renderPlasma()* which calls a function *fill_plasma()* to render plasma on a view. Fig. 5 shows the original and refactored versions of the Bitmap Plasma application. In the original version, the *for* loop uses the global variables xt1 and xt2. For the better performance, these global variables were refactored into the local variables in the refactored version of the Bitmap Plasma. Since the *fill_plasma()* function is called many times, the use of the local variable can enhance the overall performance of the application.

The Original Version: Using Global Variables

```
Fixed    xt1;  //Global Variable
Fixed    xt2;  //Global Variable
static void fill_plasma( … ){ …
  for (yy = 0; yy  <  info->height; yy++) {
    uint16_t*  line = (uint16_t*)pixels;
    Fixed     base = fixed_sin(yt1) + fixed_sin(yt2);
    xt1 = xt10;
    xt2 = xt20;
    …
  }
}
```

The Refactored Version: Using Local Variables

```
static void fill_plasma( … ){ …
  for (yy = 0; yy  <  info->height; yy++) {
    uint16_t*  line = (uint16_t*)pixels;
    Fixed     base = fixed_sin(yt1) + fixed_sin(yt2);
    Fixed xt1 = xt10;   //Local Variable
    Fixed xt2 = xt20;   //Local Variable
    …
  }
}
```

Fig. 5. Refactoring Global Variables to Local Variables in Bitmap Plasma.

Snake Game and Bitmap Plasma were tested and measured to assess the performance efficiency by the Android programming practices in more realistic apps. Fig. 6 depicts the exclusive CPU time of the original and refactored versions of the two Android applications. The Method Tracer in Android Studio was used to measure the invocation counts, inclusive times, and exclusive times of the methods of an application. For the experiments, the exclusive CPU times of the application methods were monitored without considering the processes of the Android System. Similar to the numbers obtained for the experimental example code, the results shown in Figure 6 indicate that the refactored versions perform efficiently compared to the original versions of the same application.

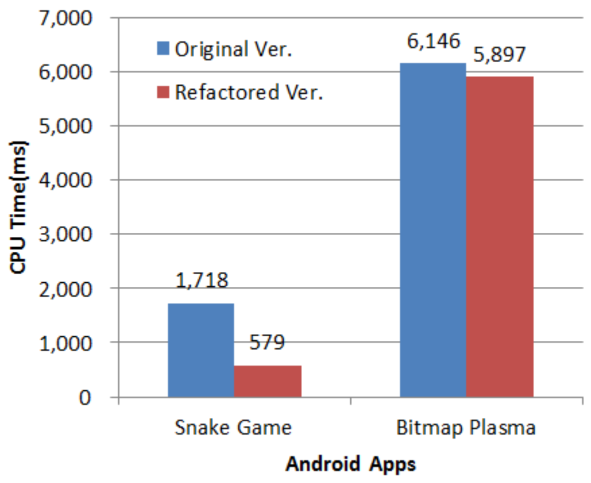

Fig. 6. CPU Time of Original and Refactored Android Applications-Snake Game and Bitmap Plasma

Table 4 shows more detailed CPU times according to the application methods. The four methods in the Snake Game application occupy CPU more than others. According to the results shown in Table 4, we can find that the inclusive CPU time of the refactored application method was reduced.

Table 4. CPU Times of Methods in Snake Game

Method Names	CPU Time in Org. Ver.(ms)	CPU Time in Ref. Ver.(ms)
SnakeView .update	1,151	565
TileView .clearTiles	1,020	423
TileView .onDraw	1,066	616
TileView.setTile	282	248

5. DISCUSSION

This paper is intended to provide Android best programming practices to enhance the performance of Android applications written in both Java and C/C++. In particular, the original version of the application is refactored into the performance-aware programming styles at the source code level. Even if the proposed practices can allow developers to create efficient programs, there are some limitations of the programming practices.

The proposed refactoring methods are not automatic but manual procedures. They are not supported by software development tools and code generation. In some aspects, automation is one of the most important factors to choose software practices. However, since developers can apply the proposed programming practices when editing source code, Android applications can be improved in an efficient way.

The proposed performance-enhancing programming practices focus on the source code of an Android application. They are intended to refactor the structure of source code by adding new methods, removing unnecessary code, or changing loop expressions. Hence, they are limited to be applied for the bytecode of an Android application. However, the fundamental ideas of the programming practices can be adopted for the bytecode of Android applications.

The execution time of an Android application is mainly concerned and measured to see whether the application runs efficiently or not. Since the CPU time consumption can directly affect the battery lifetime, it is important to reduce the CPU time consumption. Hence, even if the proposed programming practices do not explore the energy consumption in depth, the results of reducing the execution time could lead to the energy –saving benefits.

To demonstrate the effectiveness of the refactoring rules, several evaluation experiments were designed and conducted. The performance results were measured through multiple executions and were based on arithmetical means to minimize experimental errors. It is not easy to prove the performance enhancement of the proposed refactoring methods by using formal methods including formulas. In some senses, such an approach might not be relevant for the performance-enhancing programming practices.

6. CONCLUSIONS AND FUTURE WORK

Most of mobile applications should share limited resources of a mobile device with other ones during their operations. The performance efficiency of such an application is one of the most important software qualities to assess how well applications can response to users' inputs and use resources in an efficient way. Obviously, application developers need to build performance-enhancing programs that can reduce power consumption and speed up execution time under the normal conditions. This paper has focused on programming styles or coding patterns at a source code level that can impact on performance and execution time. The paper has presented a set of programming expressions that may lead to performance degradation due to its unnecessary and inefficient code. It has also proposed a refactoring approach to transform the original expressions into the improved ones. In addition, case studies have been conducted to demonstrate the effectiveness of the proposed approach. The experimental results suggest that the proposed programming practices could speed up the execution time of Android mobile applications.

As a future work, the programming styles on energy consumption will be explored in detail to extend Android programming best practices. In addition, the proposed approach can be applied into other popular mobile platforms such as the iOS mobile platform and Windows Mobile. Even though such mobile platforms are based on different programming languages, the basic guidelines and rules for performance enhancement at a source code level can be easily adopted for a specific mobile platform. Furthermore, it is worth developing a refactoring tool to support a systematic process of the proposed approach in the paper. Such a refactoring tool can be composed of three main parts--Android Code Analyzer, C/C++ Code Analyzer, Code Smell Detector, and Code Refactor. Code Analyzers are needed to parse Android and C/C++ code to create an intermediate representation such as Abstract Syntax Tree. Code Smell Detector should extract code blocks that will be refactored. Code Refactor enacts refactoring rules to the identified bad code smells.

Another promising research direction is to formally prove the effectiveness of the performance-enhancing programming practices. We can expect that such a formal proof method can supplement experimental results from a specific runtime environment.

REFERENCES

[1] Android Mobile Platforms, The official website, *https://developer.android.com*

[2] Google Play, https://play.google.com/

[3] AppBrain, Number of Android applications, Feb. 2017.

[4] Android NDK website, https://developer.android.com/ndk/index.html

[5] J. Reimann, M. Brylski, and U. Aßmann, *A Tool-Supported Quality Smell Catalogue for Android Developers*, Softwaretechnik-Trends, 2014.

[6] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in Android applications," International Conference on Mobile Software Engineering and Systems, 2016.

[7] S. Mundody and K. Sudarshan, "Evaluating the Impact of Android Best Practices on Energy Consumption," International Conference on Information and Communication Technologies, 2014.

[8] D. Li and W. G. J. Halfond, "An investigation into energy-saving programming practices for Android smartphone app development," International Workshop on Green and Sustainable Software, 2014.

[9] F. A. Fontana, V. Ferme, M. Zanoni, and A. Yamashita, "Automatic metric thresholds derivation for code smell detection," International Workshop on Emerging Trends in Software Metrics, 2015.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 2002.

[11] F. Palomba, D. D. Nucci, A. Panichella, A. Zaidman, and A. D. Lucia, "Lightweight detection of Android-specific code smells: The aDoctor project," International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017.

[12] G. Hecht, B. Omar, R. Rouvoy, N. Moha, and L. Duchien, "Tracking the Software Quality of Android Applications Along Their Evolution," International Conference on Automated Software Engineering, 2015.

[13] J. A. M. Santos, M. G. de Mendonça, and C. V. A. Silva, "An exploratory study to investigate the impact of conceptualization in god class detection," International

Conference on Evaluation and Assessment in Software Engineering, 2013.

[14] J. Lee, D. Kim, and J. Hong, "Code Refactoring Techniques Based on Energy Bad Smells for Reducing Energy Consumption," KIPS Tr. Software and Data Eng., vol. 5, no. 5, Apr. 2016, pp. 209-220.

[15] M. Gottschalk, M. Josefiok, J. Jelschen, and A. Winter, "Removing Energy Code Smells with Reengineering Services," Lecture Notes in Informatics, 2012.

[16] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of Android smells," International Conference on Software Analysis, Evolution and Reengineering (SANER), 2017.

[17] G. Hecht, N. Moha, and R. Rouvoy, "An Empirical Study of the Performance Impacts of Android Code Smells," International Conference on Mobile Software Engineering and Systems, 2016.

[18] Best Practices for Performance, https://developer.android.com/training/articles/perf-tips.html#PackageInner

[19] Bitmap Plasma, https://github.com/googlesamples/android-ndk/tree/master/bitmap-plasma

[20] Robotium, https://github.com/RobotiumTech/robotium

Dong Kwan Kim
He received the B.S., M.S in computer science from Soongsil University, Korea in 1993, 1998 respectively and also received Ph.D. in computer science from Virginia Tech, USA in 2009. He is a professor in the Department of Computer Engineering at Mokpo National Maritime University. His research interests include software modeling, mobile programming, and run-time systems.