Android Programming Techniques for Improving Performance

Jae Kyu Lee
Dept. of Electronic Engineering
Jeonbuk National University, JBNU
Jeonju, Republic of Korea
e-mail: jae4850@jbnu.ac.kr

Prof. Jong Yeol Lee
Dept. of Electronic Engineering
Jeonbuk National University, JBNU
Jeonju, Republic of Korea
e-mail: jong@jbnu.ac.kr

Abstract— **Android has been researched in various mobile device fields such as Smartphone and Tablet PC. In here, we should remember that mobile devices have limited storage and constrained battery life. Therefore, when developers develop applications, they should do efficient programming. In this paper, we have proposed programming guidelines for an effective way to improve performance in Android applications. We have programmed Android applications using Java and Native C, and compared the performance between the two languages. The applications are composed of five categories such as JNI delay, Integer, Floating-point, Memory access algorithm and String processing. By analyzing the results, we propose a more efficient way to program Android applications.**

Keywords- Android; Android NDK; JNI; Performance Evaluation; Native C

I. INTRODUCTION

Nowadays, Tablet PC and Smartphone are making a change in our life. Since these mobile devices have become more powerful and distributive, mobile computing has become more important. As the market share of mobile operating systems steadily grows and more IT applications are developed and deployed on mobile devices[1]. Moreover, a IDC(www.idc.org) survey suggests that 70% of organizations a recurrently deploying at least one mobile application. These trend is expected to continue[2].

Today, the most popular mobile operating systems are Apple's iOS, Microsoft's Windows Mobile and Google's Android. The Android provides the tools and APIs necessary to begin developing applications on the Android platform using the Java programming language. It gives developers an opportunity to build applications for the Android developer Challenge. So, many Android applications are being developed by developers around the world. Because mobile devices have limited storage and constrained battery life, the developers should pay more attention to the efficiency of the program when they develop applications.

In this paper, we have implemented and analyzed Android applications to provide guidelines for an effective way to improve the performance of Android applications.

The structure of the paper is as follows : Section II introduces the architecture of Android, JNI and NDK. Section III describes the implementation of the applications for performance measurement in detail. Section IV analyze the measurement results. Section V gives the conclusion.

II. BACKGROUND

A. *Android*

Android is a software stack for mobile devices that includes an operating system, middleware and key applications[3]. Google and other members of the Open Handset Alliance[4] collaborated on Android's development and release. The Android architecture and its main components are shown in "Fig. 1".

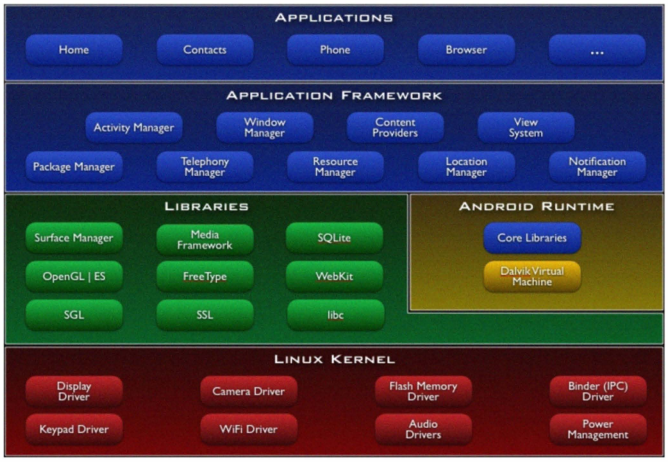

Figure 1. Architecture of Android.

As shown in "Fig. 1", Android consists of a Linux Kernel(2.6) and some key applications (e.g. phone, browser, maps) running on a Java-based, object-oriented application framework on top of core libraries running on a Dalvik virtual machine featuring JIT(version 2.2 or later) compilation. Dalvik virtual machine is one of the most prominent feature in Android. Especially, It is optimized for slow CPU, relatively little memory and to run on OS without any swap space. This is

also a register-based virtual machine. Therefore, the instruction tends to be longer. The latest version of Android is the Android 2.3(also known as Android gingerbread release). In this paper, all experiments were run on the Android 2.3.

B. Android NDK & JNI

The Android NDK is a toolset that lets you embed components that make use of native code in Android applications. Android applications run in the Dalvik virtual machine. The NDK allows you to implement parts of your applications using native-code languages such as C and C++[5]. In addition, the NDK provides stable headers for libc (the C library), libm (the Math library), OpenGL ES (3D graphics library), the JNI interface, and other libraries. This can provide benefits to certain classes of applications, in the form of reuse of existing code and in some cases increased speed.

The Java Native Interface (JNI) is a powerful feature of the Java platform. Applications that use the JNI can incorporate native code written in programming languages such as C and C++, as well as code written in Java programming language. So JNI enables one to write native methods to handle situations when an application cannot be written entirely in the Java programming language. Then the JNI allows programmers to take advantage of the power of the Java Platform[6][7]. The "Fig. 2" shows how the JNI ties the C side of an application to the Java side. We designed an application with NDK and JNI, and compared with the designed application using only Java.

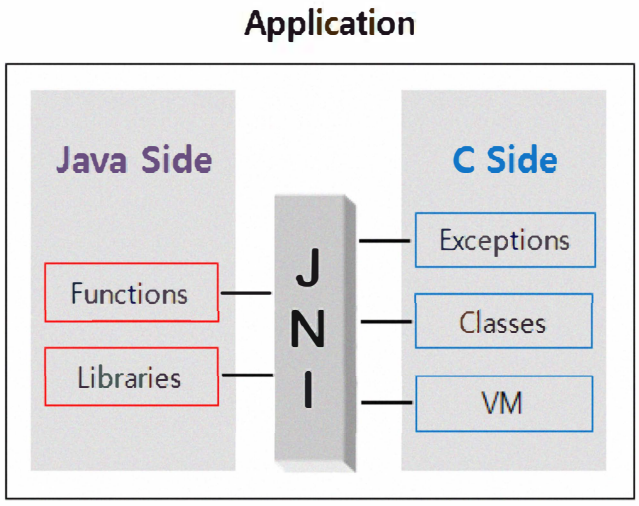

Figure 2. Architecture of JNI(Java ↔ C, C++).

III. EXPERIMENTS

In this paper, experimental equipment for performance evaluation is as follows: Intel Core i5 760 2.8GHz and 3.0GB memory. In addition, the experiment was performed on an Android virtual emulator. The emulator target version is Android 2.3(API level 9/Gingerbread), and native method was designed using the NDK R5b.

A. Implementation of Applications to evaluate the performance

This section describes the design and implementation of applications for performance evaluation. The applications were programmed by using JAVA and Native C. largely 2 kinds of ways. Java applications are developed based on commonly used Android development environment (SDK).

Applications in Native C are developed by following procedure: First, write your application source code using Java language (Activity, Service and so on). Then, create a header file containing Native C function prototype using 'javah' commands. Next, implement native method, which is an implementation of the actual behavior. Finally, build the native library, and insert this library in the application package. The behavior architecture of our Android applications using Native C is shown in "Fig. 3".

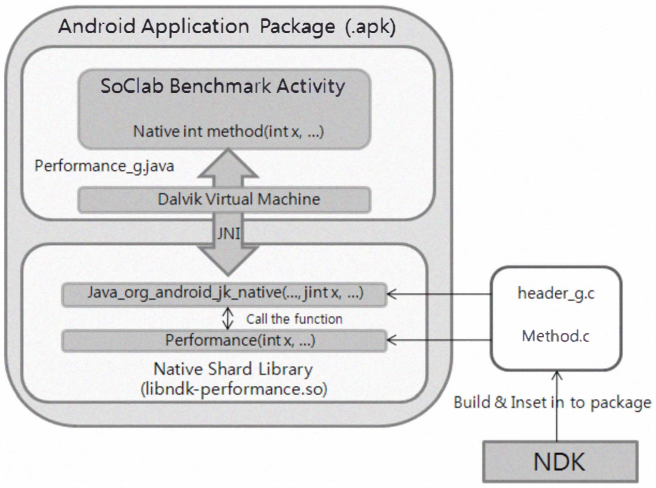

Figure 3. Architecture of applications in native C.

Applications include five different algorithms (JNI delay, Integer & Floating-point calculation, Memory access, String Processing). Each algorithm was implemented as follows. The first one is the JNI delay. JNI delay occurs when a native method is called. Therefore, JNI delay is measured before all other experiments[8]. Second, we measured the integer calculation time using Fibonacci sequence that consists of only integer arithmetic. Recalculated after initializing if an Integer gets out of range. Third, we measured the execution time of Floating-point processing using the circle area calculation algorithm that returns the area of a circle whose radius is r. In the fourth experiment, the memory access time is measured by using the worst-case scenario of bubble sort. Finally, we measured the time of String processing. It measures the processing speed of the String by the continuous String input.

We used 'nanoTime()' method in Java (Since JDK1.5) for more accurate measurements. It returns the current value of the most precise available system timer, in nanoseconds. This method also can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time. The value returned represents nanoseconds since some fixed

but arbitrary time. In other words, This method provides nanosecond precision, but not necessarily nanosecond accuracy[9]. In this application, the source code for execution time measurement is shown below.

```
long startTime = System.nanoTime();

/* Running the algorithm source code
   for performance evaluation… */

long endTime = System.nanoTime();

long estimatedTime = endTime - startTime;
```

"Fig. 4" shows designed applications for performance evaluation. This application consists of five tabs for each performance evaluation as described above. Furthermore, there are two buttons in each tab. One is button for measuring the execution time of native C algorithm, and the other one is button for measuring the execution time of java algorithm(method).

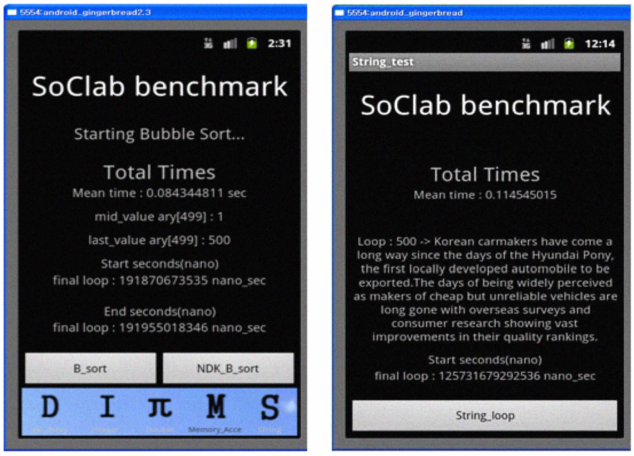

Figure 4. Performance evaluation.

IV. Results and Analysis

All of the above experimental result was calculated using the harmonic mean. The harmonic mean is one form of average, which is appropriate for situations when the average of rates is desired[10]. The harmonic mean H of the real numbers x1, x2, ..., xn > 0 is defined as follows.

$$H = \frac{n}{\frac{1}{x_1}+\frac{1}{x_2}+\cdots+\frac{1}{x_n}} = \frac{n}{\sum_{i=1}^{n}\frac{1}{x_i}} = \frac{n\cdot\prod_{i=1}^{n}x_i}{\sum_{j=1}^{n}\frac{\prod_{i=1}^{n}x_i}{x_j}}. \quad (1)$$

It is also more apparent that the harmonic mean is related to the arithmetic and geometric means. In this paper, the average processing time of each algorithm was calculated using the above formula (n=100) for more accurate measurements.

First, JNI delay is about 1.14 microseconds. It is a slight delay, and it does not greatly differ from no-operation delay. "Fig. 5" shows the results of the Integer & Floating-point execution time.

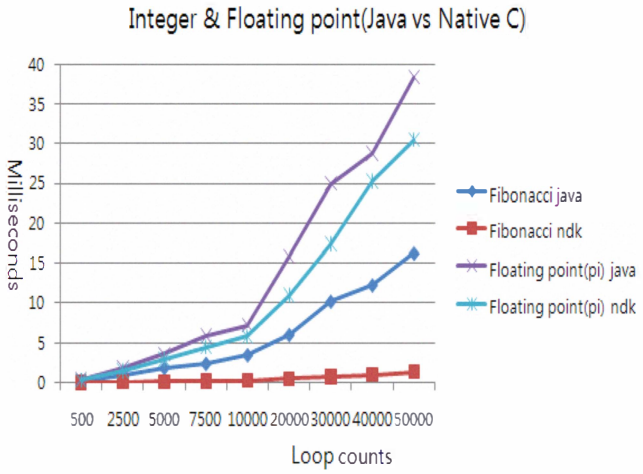

Figure 5. Integer & Floating-point procesing time.

The x-axis and the y-axis of the graph represent loop counts and execution time in milliseconds, respectively. We can see that Integer arithmetic is faster than Floating-point operations. Besides, the Native C code is faster than Java code. It means that we can achieve better performance if we use the Native C when we design application included many Floating-point operation such as game and graphic processing.

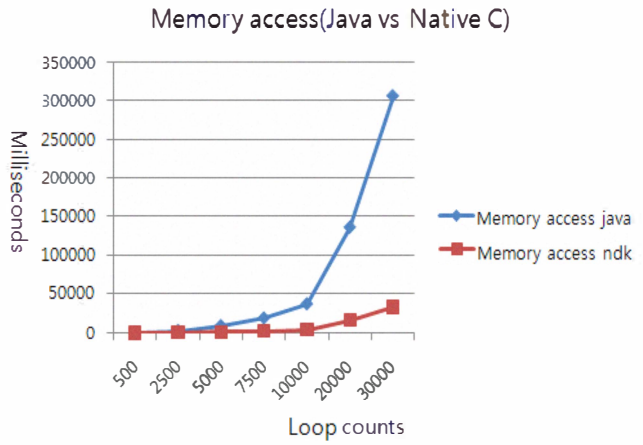

Figure 6. Memory access time.

The "Fig. 6" shows the measurement results of memory access time. "Fig. 6" also shows that Native C is faster than the Java. In addition, the deviation between Native C and Java becomes greater(X 9.2) as the number of memory accesses increase. It means that using Native C is more efficient in case of an application where memory access occurs frequently.

String processing measurement results are shown in "Fig. 7". In contrast to the above results, the performance of the Java string processing code is better than that of Native C code. This is reason that Java, C/C++ language and JNI String processing type are different each other. (Java : Unicode, C/C++ : KSC

5601, JNI : UTF-8) So, type conversion is needed. Therefore, Java language is more efficient when design an Android application which included many String processing.

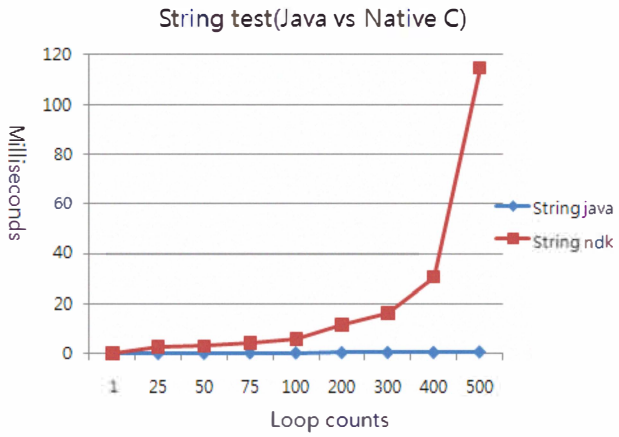

Figure 7. String processing result.

V. CONCLUSIONS

In this paper, we have programmed Android applications using Java and Native C, and compared the performance between the two languages. As a result, Native C was faster than the Java in Integer, Floating-point and Memory access operations. Especially, Native C shows the best performance in the memory access operation. On the other hand, Java language is more efficient when design an Android application that includes many String processing because Java language does not require type conversion of strings.

REFERENCES

[1] Chia-Chi Teng, Richard Helps, "Mobile Application Development : Essential New Directions for IT", School of Technology, Brigham Young University, April, 2010.

[2] Stephen D. Drake, "Embracing Next-Generation Mobile Platforms to Solve Business Problems", Sybase White Paper, Oct 2008.

[3] Android(Operating System) http://en.wikipedia.org/wiki/Android_(operating_system).

[4] OHA(Open Handset Alliance) http://www.openhandsetalliance.com/.

[5] Official Android developers website http://developer.android.com.

[6] Sheng Liang. The Java Native Interface : Programmer's Guide and Specification, Addison-Wesley, 1999.

[7] Java Native Interface(JNI) http://en.wikipedia.org/wiki/Java_Native_Interface.

[8] Dawid Kurzyniec and Vaidy Sunderam, Efficient Cooperation between Java and Native Codes - JNI Performance Benchmark, Emory University Dept. of Math and Computer science.

[9] Java nanoTime http://download.oracle.com/javase/1.5.0/docs/api/java/lang/System.html.

[10] Harmonic mean http://en.wikipedia.org/wiki/Harmonic_mean.