International Conference on Control, Automation and Systems 2010
Oct. 27-30, 2010 in KINTEX, Gyeonggi-do, Korea

Evaluating Performance of Android Platform Using Native C for Embedded Systems

Sangchul Lee[1] and Jae Wook Jeon[2]

[1] The Department of Mobile Systems Engineering, Sungkyunkwan University, Suwon, Korea
(Tel : +82-31-290-7237; E-mail: sangchul1011@gmail.com)
[2] The School of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea
(Tel : +81-31-290-7129; E-mail: jwjeon@yurim.skku.ac.kr)

Abstract: The Android platform used for mobile devices can be applied to embedded systems, such as robot control systems. Developers should create applications by using Java language provided by Android SDK for embedded systems operated via Android platforms. However, in many existing embedded systems, developers have written applications for controlling the system by using C language. Android NDK makes it possible for developers to easily reuse such legacy code written in C/C++ languages. In this paper, we show the difference in terms of performance between an Android application using native code library from C source and an Android application using the same algorithm written in Java language only. We conducted an experiment on five parts: JNI communication delay, integer calculation, floating-point calculation, memory access algorithm, and heap memory allocation algorithm. This paper presents a guideline for an effective way to use native code libraries in Android applications.

Keywords: Android, Smart phone, JNI, Native C, Performance evaluation

1. INTRODUCTION

Recently, the Android platform [1] supported by Google is widely used in mobile devices. Due to its openness, the use of Android platform has proliferated in mobile devices as well as in other embedded systems. The Android platform can be applied to embedded systems such as robot control systems.

In most embedded systems, C and C++ languages are commonly used as effective languages to control devices. In contrast, in embedded systems operated via the Android platform, developers should create applications by using Java language provided by Android SDK [2]. Generally, Android applications, based on Java language, are slower than applications written in native C/C++ languages when the program needs to execute complex operations because Android applications are executed on Dalvik Virtual Machine [3]. Such Android applications based on Java language make it difficult to manage all kinds of devices by using only Android frameworks.

Android NDK [4] was first released in June 2009. It enables Android application developers to manage devices in detail beyond the limit of the framework and to reuse legacy code written in C/C++ language easily. Android NDK can also improve the performance of the Android application if Android application developers use it appropriately.

In this paper, we show the difference in terms of performance between an Android application using native C library and an Android application using the same algorithm written in Java language only. This paper presents a guideline for an effective way to use native code libraries in android applications.

The remainder of this paper is organized as follows. In Section 2, we look into Android NDK and the JNI interface. In Section 3, we demonstrate experimental results. Section 4 examines related work. Lastly, we conclude this paper in Section 5.

2. ANDROID NDK AND JNI

2.1 Android NDK

Android NDK is a toolset that can embed components using native libraries from C/C++ code in Android applications. Android NDK enables Android application developers to use native code for performance-critical portions of their applications and to reuse legacy code written in C/C++ language.

Android application developers need to consider the advantages and disadvantages of using native code. Using native code does not always enhance application performance, but it always increases application complexity. Android NDK is effective in CPI-intensive operations, such as physics simulation, and signal processing. It is effective in reusing a large mass of legacy C/C++ code [4].

Android NDK Revision 4 was released in May 2010. This version of the NDK provides a simplified build system through the new ndk-build build command. It adds support for easy native debugging of generated machine code on production devices through the new ndk-gdb command. In addition, there are some added features that support Android 2.2[4].

2.2 JNI

Originally, the Java Native Interface (JNI) is a programming framework to allow Java code running in a Java Virtual Machine (JVM) to call and to be called by native applications and libraries written in C/C++ and assembly language [5].

It is also possible to use the JNI interface in an Android application. If Android application developers use Android NDK to embed native code libraries in their Android applications, then the native functions can be called from the application running on Dalvik virtual machine [3] through the JNI interface.

JNI [6] is used to reuse an important large mass of native code written in C/C++, accessing system devices,

performing platform-specific tasks, to enhance the application performance using time-critical code; we can create applications by using JNI. However, there can also be negative effects from using JNI, such as loosing the portability and safety of applications [7].

Fig. 1 shows an example of native function calls through JNI. Fig. 1 shows six native method declarations in the PEtest class for an Android application. They should be declared in a class for an Android application. It is necessary to write a "*native*" directive before writing the method's return type and name. Fig. 2 presents the example code, as described above. The bottom of Fig. 1 shows six native method implementations in the *libnativeC.so* shared library written in C native code. Native methods should have a one-to-one correspondence with the native method declaration in the class for an Android application.

In Android application development environments, we can use native code from shared libraries with the standard *System.loadLibrary()* call. The argument of *System.LoadLibrary()* is the library name. If we want to use the "*libnativeC.so*" shared library, then we would pass in "*nativeC*".

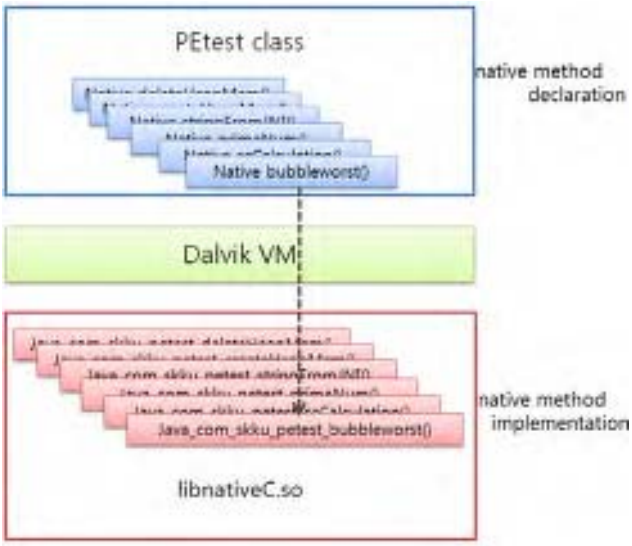


Fig. 1 Example native function call through JNI.

```
…
/*native methods*/
private native String stringFromJNI();
private native int primeNum(int num);
private native double scCalculation(int degree);
private native int bubbleworst(int size);
private native int createHeapMemory(int num);
private native void deleteHeapMemory(int num);

static {
   System.loadLibrary("nativeC");
}
…
```

Fig. 2 Using *System.loadLibrary* call to load native code from "*libnativeC*" shared library.

3. PERFORMANCE EVALUATION

3.1 The Experimental Environments

Our experimental system executing the Android application to evaluate performance is equipped with an AMD Athlon II X2 245 2.9GHz, 4.0GB memory. We executed our performance estimation application on an Android Virtual Device emulator that targets Android 2.1 (API level 7) with the Android NDK R3 version. Fig.3 shows our Android application consisting of five parts for evaluating performance.

Fig.3 Performance estimation application.

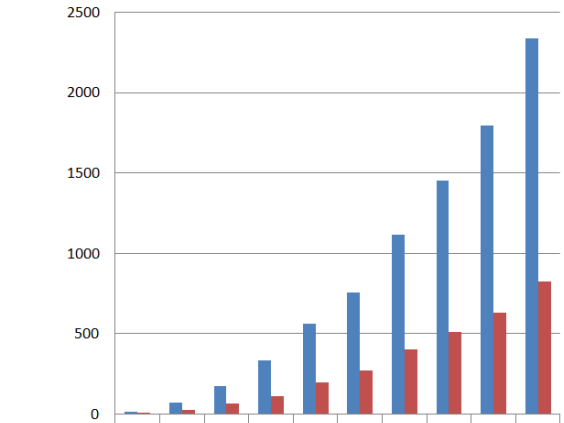
3.2 JNI Communication Delay

We measured JNI communication delays that occurred due to JNI. Our application exploited the native C library that does not perform arithmetic operation but passes a string on to the application. The experimental result shows that JNI communication delays are about 0.15 microseconds. This is a slight delay.

3.3 Integer Calculation

This experiment shows the difference in terms of integer calculation performance between an Android application using the native code library from C source and an Android application using the same algorithm written in Java language only. We measured the execution time of finding the prime number algorithm that returns a prime number of the N^{th} time of the input number. This algorithm uses only integer calculation rather than a fast floating-point algorithm using square root.

Fig.4 shows the integer calculation experimental results. The x-axis of the graph represents input number N. The y-axis of the graph represents execution time in milliseconds. As we can see from the results, using the native C library is faster than using the same algorithm running on Dalvik virtual machine only. The larger the size of the input number, the greater the performance gap between the case of using native C library and the case of not using it.



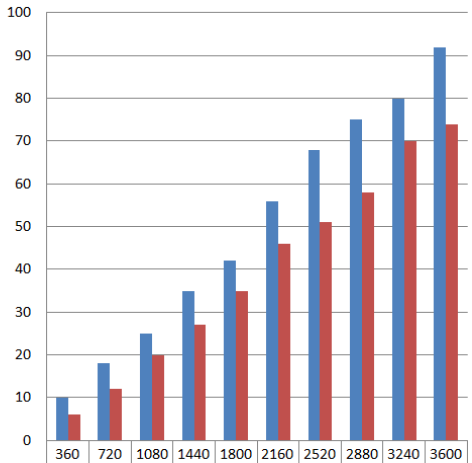
	100	200	300	400	500	600	700	800	900	1000
Dalvik VM	15	73	172	332	561	756	1116	1453	1794	2338
With Native C	5	25	63	112	197	271	403	510	630	827

Fig.4 Results of the integer calculation experiment.

3.4 Floating-point Calculation

This experiment shows the difference of floating-point calculation performance between an Android application using the native code library from C source and an Android application using the same algorithm written in Java language only. We included the 'math.h' header file to use sine/cosine functions that perform floating-point calculation. This algorithm returns the sum of the value of sine and cosine from one degree to the user input degree.

Fig.5 shows the results of the floating-point calculation experiment. The x-axis of the graph represents the input degree. The y-axis of the graph represents execution time in milliseconds. The result shows that using the native C library is a little faster than using the same algorithm running on Dalvik virtual machine only.



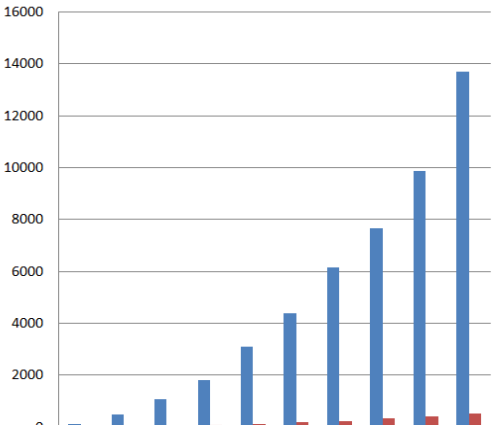
	360	720	1080	1440	1800	2160	2520	2880	3240	3600
Dalvik VM	10	18	25	35	42	56	68	75	80	92
With Native C	6	12	20	27	35	46	51	58	70	74

Fig.5 Results of the floating-point calculation experiment.

3.5 Memory Access Algorithm

We tested the memory access algorithm between an Android application using the native code library from C source and an Android application using the same algorithm written in Java language only. This algorithm performs the worst case scenario of bubble sort, time complexity of $O(n^2)$, in heap memory.

Fig.6 shows the results of the memory access algorithm experiment. The x-axis of the graph represents the input length of integer array. The y-axis of the graph represents execution time in milliseconds. It is worth noting that this test showed a considerable performance difference between using and not using the native code library. It is desirable for Android application developers to use the native C library in case of an application that frequently requires memory access.



	500	1000	1500	2000	2500	3000	3500	4000	4500	5000
Dalvik VM	125	473	1059	1831	3107	4390	6144	7657	9863	13702
With Native C	4	19	44	84	117	186	246	340	408	521

Fig.6 Results of the memory access algorithm experiment.

3.6 Heap Memory Allocation Algorithm

We measured the time of allocating heap memory whose size is calculated by multiplying 4Kbyte by the user input number.

Fig.7 shows the results of the heap memory allocation algorithm experiment. The x-axis of the graph represents the input number. The y-axis of the graph represents the execution time in milliseconds. As we can see from the results, using the native C library is faster than using the same algorithm running on Dalvik virtual machine only.

Generally, heap memory is used when we try to create an object in Java language and to allocate dynamic memory in C/C++ language. C/C++ developers use dynamic memory allocation to avoid memory wastage by static memory allocation and restrictions on a program behavior. However, dynamic memory allocation is slower than static memory allocation. Dynamic memory allocation also carries risks such as memory leaks, and memory exhaustion [8]. Though allocating heap memory in native C/C++ libraries is much faster than using the same behavior running on Dalvik virtual machine only, developers need to avoid dynamic memory allocation in native C/C++ frequently. They should carefully exploit dynamic memory allocation.

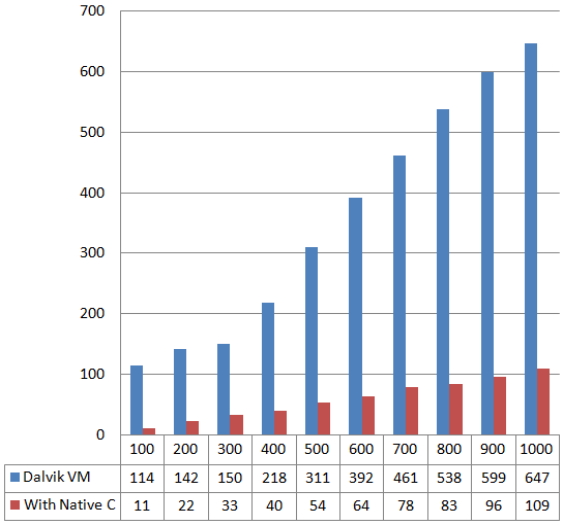


Fig.7 Results of the heap memory allocation algorithm experiment.

4. RELATED WORK

Leonid Batyuk et al. [9] performed similar tests of Android environments and Java environment using Sun JRE 1.6. They evaluated the performance of sorting algorithms using the JNI with native code or not. Under Android environments, using the JNI with the native code is always faster than using Dalvik virtual machine only. Conversely, under Sun JRE environments, using JNI with native code is not always faster than using JVM only. However, this study only tested integer operations. They did not consider the floating-point operations and memory access tests.

5. CONCLUSION

In this paper, we showed the difference in terms of performance between an Android application using the native code library from C source and an Android application using the same algorithm running on Dalvik virtual machine only. We conducted an experiment in five parts. First, we measured JNI communication delays that occurred due to using the JNI. And we found that these delays do not have influence on the experimental results because of its slightness. In every part of the experiment-integer calculation, floating-point calculation, memory access algorithm, heap memory allocation algorithm-, using native C library achieves faster results than using the same algorithm running on Dalvik virtual machine only.

It is worth noting that the memory access algorithm test showed a considerable performance difference depending on whether or not the native code library was used.

We recommend that Android application developers use the native C library when the application frequently requires memory access and that needs complex calculations under correct usage.

ACKNOWLEDGMENT

This research was supported by the MKE(The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center) support program supervised by the NIPA(National IT Industry Promotion Agency) (NIPA-2010-(C1090-1021-0008)).

REFERENCES

[1] "Android.com," Available: http://www.android.com

[2] "Android SDK | Android Developers," Available: http://developer.android.com/sdk/index.html

[3] DalvikVM.com, "Dalvik Virtual Machine insights," Available: http://www.dalvikvm.com/

[4] "Android NDK | Android Developers," Available: http://developer.android.com/sdk/ndk/index.html

[5] "Java Native Interface - Wikipedia," Available: http://en.wikipedia.org/wiki/Java_Native_Interface

[6] Rob Gordon, "Essential JNI: Java Native Interface," ISBN 978-0136798958, 1998.

[7] Mark Allen Weiss, "C++ for Java Programmers," ISBN 978-0139194245, October 2003.

[8] MISRA-C 2004: Guidelines for the use of the C language in critical systems. Motor Industry Research Association, October 2004.

[9] Leonid Batyuk, Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Ahmet Camtepe and Sahin Albayrak, "Developing and Benchmarking Native Linux Applications on Android," Proceedings of the 2nd International Conference on Mobile Wireless Middleware, Operating Systems, and Applications (Mobilware 2009), pp. 381-390, Beriln, Germany, April 28-29, 2009.