UDC 004.03

V.V. KAZYMYR[*], I.I. KARPACHEV[*]

IMPROVING TIME-CRITICAL CODE PERFORMANCE WITH JNI

[*]Chernihiv National University of Technology, Chernihiv, Ukraine

Анотація. *Наведено аналіз використання JNI для мобільних пристроїв з ОС Android. Дослідження JNI-процесу проведено з наданням прикладів коду, що реалізує Java-виклики C і C ++ частин програм. Показано підвищення продуктивності за рахунок заміни критичних частин коду JNI аналогами.*
Ключові слова: *JNI, мобільний пристрій, критичний по часу виконання код, продуктивність коду.*

Аннотация. *Приведен анализ использования JNI для мобильных устройств с ОС Android. Исследование JNI-процесса проведено с предоставлением примеров кода, реализующего Java-вызовы C и C ++ частей программ. Показано повышение производительности за счет замены критических частей кода JNI аналогами.*
Ключевые слова: *JNI, мобильное устройство, критический по времени код, продуктивность кода.*

Abstract. *Analysis of usage of JNI for mobile devices with OS android is provided. JNI-workflow is explored including usage of code examples which realizes Java calls C and C++ parts of programs. Improving of performance due to replacing time-critical parts of code with JNI analogs is shown.*
Keywords: *JNI, mobile device, time-critical code, code-performance.*

1. Introduction

One of Java's greatest advantages is that its design allows for cross-platform capability. This feature, however, is also a bug with regard to other aspects of programming. It is constrained in its interaction with the local machine, and thus the local machine instructions cannot be utilized to achieve the full performance potential of the machine. To ameliorate this weakness, there is the Java Native Interface, a Java platform that interacts with the machine on the local level. It can be employed to allow the use of legacy code and more interaction with the hardware for efficient performance. This article explores the JNI workflow, provides code examples of how Java calls in both C and C++, and introduces the Android Native Development Kit (NDK), which compiles the C/C++ code into applications that can run on an Android device.

2. JNI overview

The Java Native Interface (JNI) is the native programming interface for Java that is part of the JDK. By writing programs using the JNI, developer ensures that the code is completely portable across all platforms.

The JNI allows Java code that runs within a Java Virtual Machine (VM) to operate with applications and libraries written in other languages, such as C, C++, and assembly.

JNI is used to write native methods to handle those situations when an application cannot be written entirely in the Java programming language. For example, developer may need to use native methods and the JNI in the following situations [1]:

1. The standard Java class library may not support the platform-dependent features needed by the application.

2. An existing library or application written in another programming language may be already available and developer wishes to make it accessible to Java applications.

3. Developer may want to implement a small portion of time-critical code in a lower- level programming language, such as assembly, and then having Java application call these functions.

Programming through the JNI framework lets developer to use native methods to do many operations. Native methods may represent legacy applications or they may be written explicitly to solve a problem that is best handled outside of the Java programming environment.

The JNI framework lets native methods utilize Java objects in the same way that Java code uses these objects. A native method can create Java objects, including arrays and strings, and then inspect and use these objects to perform its tasks. A native method can also inspect and use objects created by Java application code. A native method can even update Java objects that it created or that were passed to it, and these updated objects are available to the Java application. Thus, both the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them.

Native methods can also easily call Java methods. Often, developer will already have an implemented library of Java methods. The native method does not need to repeat functionality already incorporated in existing Java methods. The native method, using the JNI framework, can call the existing Java method, pass it the required parameters, and get the results back when the method completes.

The JNI enables the advantages of the Java programming language from the native methods. In particular, developer can catch and throw exceptions from the native method and have these exceptions handled in the Java application. Native methods can also get information about Java classes. By calling special JNI functions, native methods can load Java classes and obtain class information. Finally, native methods can use the JNI to perform runtime type checking.

It is easy to see that the JNI serves as the glue between Java and native applications. The figure 1 shows how the JNI ties the C side of an application to the Java side.

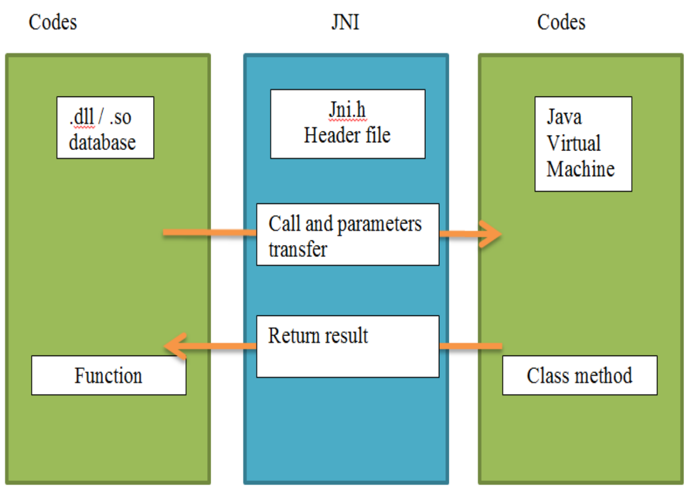


Fig. 1. JNI Architecture

3. Boosting performance with JNI

Mobile JNI or Java Native Interface is the interface between the Java code running in a JVM and the native code running outside the JVM. It works both ways that is developer can use JNI to call native code from Java programs and to call Java code from the native code. The native code normally resides within a library (.so file) and is typically written in C/C++ [2].

The main reason to use JNI in a Java program is to bypass performance bottlenecks – execute heavy number crunching in native code and get rid of the overhead that the instruction interpretation in the JVM introduces.

On Android, in order to prevent fragmentation, developers are only allowed to use the following libraries in the native code:

– libc (C library) headers;
– libm (math library) headers;
– JNI interface headers;
– libz (Zlib compression) headers;
– liblog (Android logging) header;
– OpenGL ES 1.1 (3D graphics library) headers (since 1.6);

– A Minimal set of headers for C++ support.

The following example demonstrates how to transform a time consuming Java method with a lot of number crunching into a native declared method where the real work is performed in native code.

Here is the time consuming Java method:

```
    public double compare(int[] sourceData,int[] targetData, double tar-
    getError) {
    double error = 0.0D;
    for (int index = 0; index < targetData.length; index++) {
      int c1 = sourceData[index];
      int c2 = targetData[index];
      int b = (c1 >> 16 & 255) - (c2 >> 16 & 255);
      int g = (c1 >> 8 & 255) - (c2 >> 8 & 255);
      int r = (c1 & 255) - (c2 & 255);
      error += r * r + g * g + b * b;
      if (error > targetError)
         return error;
    }
    return error;
  }
```

The sourceData and targetData arguments represent the pixels of two Bitmaps. In short the method calculates the sum of the square distance in color between two images, pixel by pixel. If compare two 200×200 pixels images the for-loop will run 40000 times. This is a typical candidate for when to use JNI.

This is what the function will look like when written in C:

```
static jdouble compareNative(JNIEnv *env, jobject thiz, jintArray
sourceArr , jobject targetArr, jdouble targetError){
         jdouble error = 0.0;
         int index, c1, c2, b, g, r  = 0;
         jint *sarr, *tarr;
         sarr = (*env)->GetIntArrayElements(env, sourceArr, NULL);
         tarr = (*env)->GetIntArrayElements(env, targetArr, NULL);
         if (sarr == NULL || tarr == NULL) return targetError;
         int size = (*env)->GetArrayLength(env, sourceArr);
for (index = 0; index < size; index++) {
  c1 = sarr[index];
  c2 = tarr[index];
  b = (c1 >> 16 & 255) - (c2 >> 16 & 255);
  g = (c1 >> 8 & 255) - (c2 >> 8 & 255);
  r = (c1 & 255) - (c2 & 255);
  error += r * r + g * g + b * b;
  if (error > targetError){
 (*env)->ReleaseIntArrayElements(env, sourceArr, sarr, 0);
 (*env)->ReleaseIntArrayElements(env, targetArr, tarr, 0);
             return error;
           }
        }
        (*env)->ReleaseIntArrayElements(env, sourceArr, sarr, 0);
        (*env)->ReleaseIntArrayElements(env, targetArr, tarr, 0);
        return error;
      }
```

All native functions must have the JNIEnv (a reference to the virtual machine itself) and the jobject (a reference to the "this pointer" of the Java object where the native method call comes from) as the first two arguments. Then it is possible to add custom arguments.

It is necessary to find a way to make the virtual machine direct the calls to the native declared Java method to the native C function. This is done using the registerNatives function of the JNIEnv. If developer uses the boilerplate C code from above two things are to be done:

1. Setting the classpath variable to the full class name of the Java class (including package name). Replacing the dots with slashes.

2. For each native declared method in Java, inserting a JNINativeMethod struct into the methods[] array.

For this example it will look like this

```
       static const char *classPathName =
 "com/jayway/MyComparator";

       static JNINativeMethod methods[] = {
          // nameOfNativeMethod, methodSignature, methodPointer
          {"compare", "([I[ID)D", (void*)compareNative },
       };
```

The first parameter is the name of the native declared Java method, the second is the signature of the native declared Java method and the last parameter is the function pointer to the C function to execute when the native declared Java function is executed.

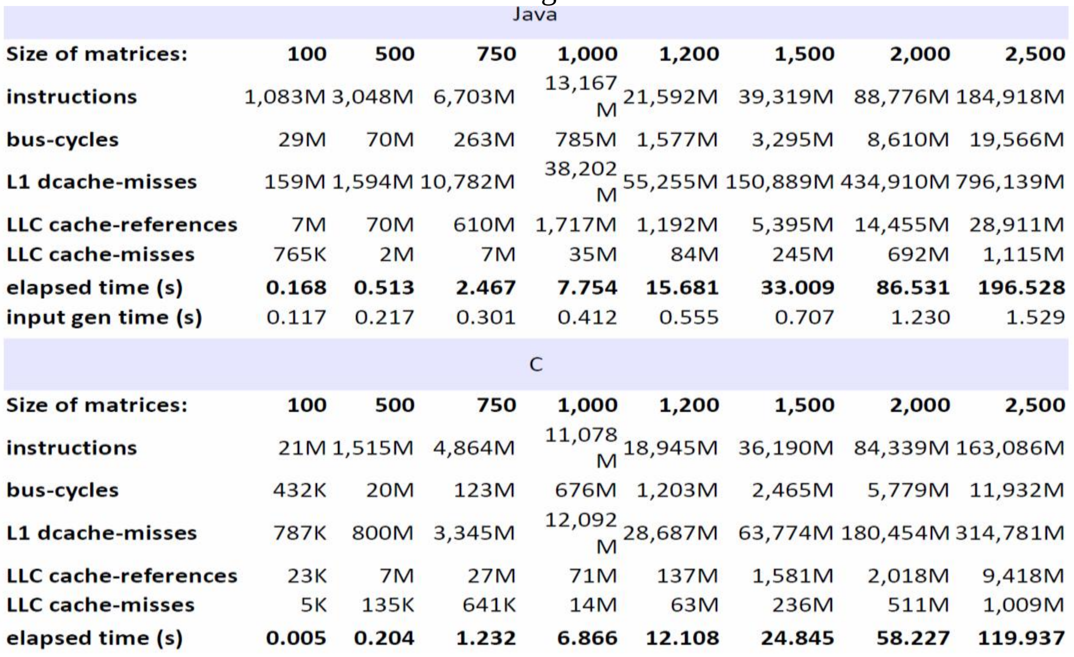
The signature of a Java method can be determined using the javap tool from SUN's Java SDK or developer can create it using the following table, Java VM Type Signatures.

To make things really simple when developing JNI code Google has released the Android Native Development Kit (NDK). It is easy to setup and use. In short, developer creates a folder named jni in the Android project. Here all the c-files together with an Android.mk file are put. In the Android.mk developer specifies which c-files are to be compiled. In the Android NDK/apps folder a directory named after the project (perhaps my-app) is created. In this directory developer adds an Application.mk file. In the Application.mk, variable APP_PROJECT_PATH is necessary to set to the path of the Android project.

After some simple benchmarking it is clear that the native declared method executed about 2–3 times faster than the original method executing within Dalvik. For larger images the improvement might be even bigger (Fig. 2 and table 1) since a call to a native declared method takes more time than calling a normal Java method.

Execution time was measured using Linux's *perf* tool. Each reported value is the average of 10 independent runs.

Table 1. Different data sizes benchmarking results



Java								
Size of matrices:	**100**	**500**	**750**	**1,000**	**1,200**	**1,500**	**2,000**	**2,500**
instructions	1,083M	3,048M	6,703M	13,167M	21,592M	39,319M	88,776M	184,918M
bus-cycles	29M	70M	263M	785M	1,577M	3,295M	8,610M	19,566M
L1 dcache-misses	159M	1,594M	10,782M	38,202M	55,255M	150,889M	434,910M	796,139M
LLC cache-references	7M	70M	610M	1,717M	1,192M	5,395M	14,455M	28,911M
LLC cache-misses	765K	2M	7M	35M	84M	245M	692M	1,115M
elapsed time (s)	**0.168**	**0.513**	**2.467**	**7.754**	**15.681**	**33.009**	**86.531**	**196.528**
input gen time (s)	0.117	0.217	0.301	0.412	0.555	0.707	1.230	1.529

C								
Size of matrices:	**100**	**500**	**750**	**1,000**	**1,200**	**1,500**	**2,000**	**2,500**
instructions	21M	1,515M	4,864M	11,078M	18,945M	36,190M	84,339M	163,086M
bus-cycles	432K	20M	123M	676M	1,203M	2,465M	5,779M	11,932M
L1 dcache-misses	787K	800M	3,345M	12,092M	28,687M	63,774M	180,454M	314,781M
LLC cache-references	23K	7M	27M	71M	137M	1,581M	2,018M	9,418M
LLC cache-misses	5K	135K	641K	14M	63M	236M	511M	1,009M
elapsed time (s)	**0.005**	**0.204**	**1.232**	**6.866**	**12.108**	**24.845**	**58.227**	**119.937**

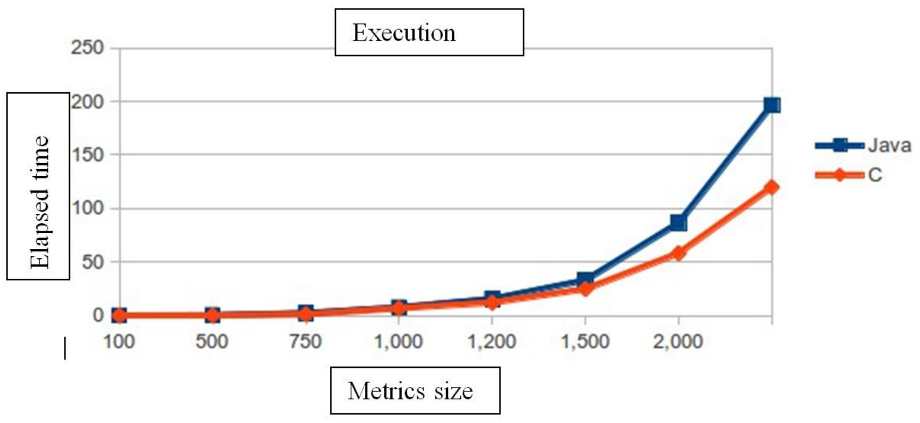


Fig. 2. Execution time and payload dependency when using JNI

4. JNI Performance overheads

The main disadvantage of using JNI is calling a native method can be slower than making a normal Java method call. There are some cases will be described below.

Native methods will not be inlined by the JVM. Nor will they be just-in-time compiled for this specific machine – they are already compiled [3].

A Java array may be copied for access in native code, and later copied back. The cost can be linear in the size of the array. It was measured JNI copying of a 100,000 array to average about 75 microseconds on Windows desktop, and 82 microseconds on Mac. Fortunately, direct access may be obtained via GetPrimitiveArrayCritical or NewDirectByteBuffer.

If the method is passed an object, or needs to make a callback, then the native method will likely be making its own calls to the JVM. Accessing Java fields, methods and types from the native code requires something similar to reflection. Signatures are specified in strings and queried from the JVM. This is both slow and error-prone [4].

Java Strings are objects, have length and are encoded. Accessing or creating a string may require an O(n) copy.

In Bresenham's algorithm c implementation, a number of different techniques are used to copy an array that contains 1,000 elements. The copy operations are performed using various native and nonnative methods. To make it easier to compare the performance of these different techniques, the program performs each copy 10,000 times.

The first two copy techniques don't use any native code; the rest call C functions. These C functions are written using different JNI usage patterns to illustrate the costs associated with different coding techniques. Note that some of these C functions don't perform the full array copy; they're included to highlight the costs of particular operations. Table 2 shows the benchmark results for the different methods.

Table 2. Array Copy Results

Copy Method	Time
Arraycopy	234 ms
Assign	984 ms
Dumbnativecopy	1,609 ms
Nativedonothing	1,125 ms
Nativedoabsolutelynothin g	63 ms
Nativecritical	578 ms
Nativecriticalmemcpy	422 ms
Nativepullonly	391 ms

5. Summary

Using information presented above it was defined, that replacing critical parts of code with JNI analog could show significant difference in performance. There are two main reasons for this: technical restrictions of JVM and fast nature of C code. In spite the fact that there is speed issue on the very low java level, especially on mobile devices, modern CPU power enough for coupe with most of the issues.

REFERENCES

1. Reinholtz K. Java will be faster than C++ / K. Reinholtz // ACM Sigplan Notices (ICDMA). – 2000. – P. 25 – 28.
2. Zorn B. The Measured Cost of Conservative Garbage Collection Software / B. Zorn // Software – Practice & Experience. – 1993. – Vol. 23, Issue 7. – P. 733 – 756.
3. Schanzer E. Performance Considerations for Run-Time Technologies in the NET Framework / E. Schanzer. – Germany: Microsoft Developer Network article, 2012. – P. 71 – 93.
4. Cowell-Shah C.W. Nine Language Performance Round-up: Benchmarking Math & File I/O [Електронний ресурс] / C.W. Cowell-Shah // OSnews.com. – 2004. – Режим доступу: http://www.osnews.com/story/5602.

Стаття надійшла до редакції 13.05.2016