2011 International Joint Conference of IEEE TrustCom-11/IEEE ICESS-11/FCST-11

Efficient Java Native Interface for Android Based Mobile Devices

Yann-Hang Lee, Preetham Chandrian, and Bo Li

School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, AZ, 85287

Abstract— **Java has been making its way into the embedded systems and mobile devices like Android. The Java platform specifies the Java Native Interface (JNI) which allows Java code that runs within a JVM to interoperate with applications or libraries that are written in other languages and compiled to the host CPU. JNI plays an important role in embedded system as it provides a mechanism to interact with libraries specific to the platform and to take the advantage of fast execution of native programs. To address the overhead incurred in the JNI due to reflection and serialization, this paper proposes to cache class, field, and method information obtained from reflection for subsequent usage. It also provides a function to pin objects to their memory locations such that they can be accesses through the known reference. The Android emulator is used to evaluate the performance of these techniques and we observed that there was 10- 30 % performance gain in the Java Native Interface for two Android applications.**

I. INTRODUCTION

With the introduction of JavaME (Micro Edition) by Sun Micro System for mobile and embedded devices, Java has applied widely to embedded systems and mobile devices. A notable application is the Android phone platform on which mobile computing applications are developed in Java and run on Dalvik virtual machine [1][2]. The applications like contact, email, browser settings, Bluetooth etc., come with the Android package. Additional applications can be acquired through Android Market [3] and installed in individual phones.

While the Java applications on Android platform are portable across various phone devices, there are several issues to be considered for this approach:

- Standard Java libraries do not support hardware-platform specific features that are needed by the applications, including user interface and graphic rendering.
- The developers may want to use libraries written in different languages because of the fact that these libraries are more efficient than their Java counterparts.

The second issue has a profound impact to the user experience not only in fast response time of application operations, but also in energy consumption and battery life.

To address these issues, JNI (Java Native Interface) has been specified and applied to integrate native code and Java program in a single application [4][5]. It has played a major role in developing application which needs to interact with native code (C/C++). Android uses JNI in its NDK (Native Development Kit) [6], which is a toolset that helps developers to interact with native code components in their applications. The NDK includes tools and builds files that can be used to generate native code libraries from C and C++ sources, and to embed native libraries into an application package file. Thus, with NDK, libraries can be developed to build activity, handle user inputs, manage hardware sensors, and invoke platform specific operations by calling the drivers of the underlined Linux system.

As more and more features and applications are being added to the mobile devices, there is a need for an efficient JNI. As illustrated in [7][8], the overhead incurred while making JNI calls can be costly during the data transfer between the JVM address space and the native address space. While it is claimed in [7] that using the JNI with the native code is faster than using Dalvik virtual machine, the communication delay in JNI is not negligible, e.g., it takes 0.15 micro-second to pass a string. In fact, the overhead can be significant when reflection and serialization operations are included for passing arguments. Note that serialization is to marshal and de-marshal the data passed between Java and native modules, and can create deep and multiple copies of objects as arguments of function calls. For serializing non-primitive data, reflection mechanism will be invoked to introspect the metatdata of binary class format to discover object fields and methods at runtime.

In this paper, we report an experiment of developing an efficient JNI mechanism. In the current JNI implementation, each call to the JNI is treated as a new call without the information of its previous history. Thus, a lot of information collected in previous operations is lost. If this information were available for the subsequent calls, the execution time of the JNI can be reduced. For instance, the overhead of searching the class structure can be avoided, if we can cache the class structure and the field identifiers of object attributes. Also, the transfer of data can be simplified if we can pin the memory address of the object or array that we intend to manipulate.

The proposed solution alters the JNI environment structure by including a hash map to store the recently accessed field identifiers and methods. It also caches the class structure in the JNI environment structure. A new method is introduced to pin objects to their current memory location so that the same addresses can be used in future accesses. This decreases the time of referencing the objects and avoid additional calls to JNI as we already know its location in the memory.

In the following, we briefly introduce the JNI mechanism and Android platform. A concise survey of the related work in

978-0-7695-4600-1/11 $26.00 © 2011 IEEE
DOI 10.1109/TrustCom.2011.162

JNI mechanisms and performance improvement is given in Section III. Our design and implementation follows in Section IV. In section V, we show the evaluation results with two real Android applications. Finally, we describe the future work in the conclusion section.

II. JNI AND ANDROID PLATFORM

A. *JNI*

The Java Native Interface is designed to allow the programmer to take advantage of the Java virtual machine, but still can use code written in other languages and libraries. With JNI, Java applications call native methods in a similar way as a simple Java method invocation. The only difference is that the method it invokes has the keyword *native* in its prototype and these Java calls are internally translated to invoke the native methods or library. On the other hand, the JNI supports an invocation interface that allows the invocation of Java virtual machine code from the native code. This is done via a native library that uses Java API.

When a native function is called, one of the arguments passed to the function is the JNIEnv interface pointer. The JNIEnv pointer links to a memory location containing a pointer to a function table. Each entry in the function table points to a JNI function. The JNIEnv interface pointers links to thread-local data and is organized like a C++ virtual function table. Native methods always access data structures in the Java virtual machine through one of the JNI functions. The other arguments provide instance or class information. If the native method is an instance method, the argument is a reference to the object on which the method is invoked, similar to this pointer in C++. For a static native method, it is a reference to the class in which the method is defined.

The Java native interface provides various functions that can be called to access the Java objects and methods [4]. For instance, to access arrays, it provides get<Type>ArrayRegion . If an array contains objects in Java, each element can be accessed individually, and whenever we access the data, it is copied from the Java heap to native region.

B. *Android Architecture*

Android is a software stack for mobile devices that includes an operating system, middleware, and key applications. The major components of the Android operating system include Linux kernel, Android runtime (including Dalvik VM [9][10] and core library), C/C++ libraries (as system components, e.g., OpenGL, media framework, SQLite, etc), and application framework (for phone functions, e.g., location manager, activity manager, etc.)

Android applications are written in the Java programming language. The Java code, all the necessary data and resource files of the application are bundled by the "aapt" tool into an *Android package.* This file can then be used for installing the application on devices. All the code in a single Android package file is considered to be one *application*.

Android application sandbox model uses the process separation provided by Linux kernel as the primary means of achieving isolation against other suspicious applications. Each application runs in its own Linux process. Furthermore, each managed piece of code executes in a virtual machine (DVM). As a result each application is sand-boxed from the other applications running at any given time. All IPC is achieved via the mechanisms provided by the Binder module. A second level of isolation builds upon the capability of underlying Linux to strongly isolate data/files of one user from the other. This is achieved by allocating a unique user-id to each installed application on a particular system. Android starts the process when any of the application code needs to be executed, and shuts down the process when it's no longer needed and other applications are in need of resources.

JNI (or NDK in Android) is a part of the Java virtual machine implementation. It enables the two-way communication between native and Java programs. Once JNI finds a method that needs to be executed, it transfers the control to the DVM interpreter. The interpreter checks weather the function is Java function or native. If the function is Java then the byte code is interpreted into the architecture specific code.

When a native function is called, JavaVM and JNIEnv arguments will become the function interfaces to the supported JNI functions. JNI function FindClass should be called from the native function to find the class by name. Similarly, to get an instance field or method, native program can invoke JNI functions getFieldId and getMethodId with class, field (or method) names. For these invocations, symbolic lookups based on the name and descriptor of the field or method is required. Symbolic lookups are relatively expensive and can be avoided by caching field and method IDs in static variables in native programs, as suggested in [4].

III. RELATED WORK

From the discussion above it is clear that if we could decrease the time required to retrieve an object in JNI and the time taken for the reflection and serialization of the data that is being transferred from the JVM to the native space, a better performance and a save of battery life of the device can be obtained. Arrays are the main area of concern as a huge amount of data needs to be moved to and from between JVM and native spaces. If the class object is huge, the time spent on finding the attributes that we are interested in becomes significant. In [11], Hirzel et. al., discussed the issues of runtime pointer analysis while using reflection, dynamic loading of libraries, and other JVM mechanisms, as well as some of the JNI APIs which rely on finding and loading of classes using reflection.

A. *JNI Bridge*

One reason to use Java is that it can be ported easily on different platform as the application runs inside the virtual machine. But if the application uses native calls to the libraries that are specific to the platform, the porting becomes difficult. This paper on JNI Bridge [12] describes the challenges and solution so that the JVM supports the native calls on different instruction set architectures and dynamic translators are used to translate native calls based on the underlying architecture. To handle the JNI up-calls and data marshaling, a simulated JNIEnv object in the IA-32 execution environment is used to enable 32-bit native libraries to call 64-bit function pointer. They also use marshaling tables to map 64-bit references to 32-bit references by intercepting the up calls and wrapping it with the reference and during the down call the corresponding 32-bit reference is used. To avoid the data movement when *GetPrimitiveArrayCritical* JNI API call is made, the reference is directly taken from the JVM internals. The JVM-independent implementation has to resort to Java reflection to obtain this information.

B. Interfacing Java to the Virtual Interface Architecture

This paper in [13] explores the use of user-level network interface for the communication between the Java heap and native buffer. It describes two approaches: the first approach manages the buffer between the Java heap and the native space which requires the data to be copied while the second approach uses a Java-level buffered abstraction and allocates space outside the Java heap. This allocated space can be accessed like array in the Java space. The second approach eliminates the use of copying the data but the native garbage collector has to be modified.

The first level of Javia (Javia-I) [13] manages the buffers used by VIA in native code (i.e. hides them from Java) and adds a copy on the transmission and reception paths to move the data into and out of Java arrays. Javia-I can be implemented for any Java VM or system that supports a JNI-like native interface. (Javia-II) introduces a special buffer class that, coupled with special features in the garbage collector, eliminates the need for the extra copies. In Javia-II [13], the application can allocate pinned regions of memory and use these regions as Java arrays. These arrays are genuine Java objects (i.e. can be accessed directly) but are not affected by garbage collection as long as they need to remain accessible by the network interface. This allows the application to manage the buffer and to send or receive Java array directly. The issues of memory management when application creates memory outside the Java heap are also discussed in the paper.

C. Jeannie

Instead of JNI, Jeannie, a new foreign functional interface is designed in [14]. Through Jeannie, programmers can write both the Java code and the native code in the same file. Jeannie compiles these files down to their respective JNI calls. This enables static error detection across the languages and simplifies the resource management. It addresses the issues of JNI being unsafe as it does not require dynamic checks. By integrating and analyzing both Java and C together, the compiler can produce error messages which can prevent many maintenance issues. The compiler is implemented using rats! [14]. To access string from C in Java, conversion from UTF-8 to UTF-16 is made, and vice versa is done while accessing strings from Java in C. The array region is still copied from the Java heap to C memory space when access is made.

D. Inlining Java Native Calls At Runtime

This technique inlines the native functions using JIT in java applications. The callbacks to the JNI are transformed into their equivalent lightweight operations [15]. IBM TR JIT [15] compiler is used as it supports multiple JVMs and class library implementation. The control flow for the TR JIT consists of phases for intermediate language (IL) generation, optimization and code generation. The inliner is enhanced so that it can synthesize the opaque call to the native function. In addition, a callback transformation mechanism is introduced that replaces the expensive callbacks with compile-time constants and byte code equivalents, while preserving the semantics of the original source language.

E. Janet

Janet [16] is the Java language extension which enables convenient development of Java to native code interfaces by completely hiding the JNI layer from the user. The source file is similar to ordinary Java source file except that it may contain embedded native code (in terms of native method implementations), and the native code can easily and directly access Java variables as Java code would. It enables efficient direct access to Java arrays from the native side. However, when the array is to be processed by external routine the array pointer has to be used. Java types are converted by Janet generally to native types having the same name. The array conversion introduces no performance reduction on platforms where appropriate Java and native types are equivalent, but it requires allocation and copying of the whole array in the case when they are different.

Other research work includes the analysis of the use of reflection and its internal working analyses [17]. The paper proposes a class named SmartMethod which transforms the calls made by the use of reflection to direct calls that will be carried out similar to the standard Java method invocation. SmartInvokeC [17] tool is used to generate the stub of a class from its byte code, so to invoke a method we no longer need to use the JNI. The call is made from a C stub that is generated. To retrieve and invoke method faster the necessary information is hashed. Tamar et al., provides a memory management scheme for thread local heap in [18]. This technique determines the objects that are local and global and uses this information to avoid unnecessary synchronization.

In addition, the technique used to profile native code that is a part of the application is discussed in [19]. Most of the profiling tool like 'hprof' do not segregate the time spent in native code if we have this information then we can find the parts of the code that can be improved further. The paper [19] introduces a profiling tool based on JVM tool interface. The technique involves introduction of a wrapper methods for the native function prototype in Java. It has the same method name and signature as that of the native method but not a native method. This wrapper function calls the *J2N_Begin* which recodes the time stamp and other profiling information. Then it calls the native method. Upon return the wrapper calls *J2N_End* is called which records the exit timestamp.

IV. Design and Implementation

To reduce the invocations to search operations and data copying operations, our design focuses in 5 methods of JNI interface, *findClass, getFieldID, getMethodID, pinObject* and *unpinObject*. The first 3 JNI functions are invoked to obtain accesses to object fields and methods. The modification is to keep the information obtained from the first invocation available for subsequent usage. For the last two methods, the objects in memory are pinned to their existing location via JNI, and would not be moved by garbage collection. Thus, object data can be accessed directly from the memory heaps.

A. *Hashing JNI fields*

There are several ways to store the data and use it in the future so the process of the next call made to the function can be expedited. Hashing is the technique used here to make the JNI calls efficient. As a JNI call accesses object fields in the Java domain it first needs to know the exact memory locations of the fields. Note that each function in the JNI API has the access to the pointer of the JVM environment that it is running in. This JVM environment pointer contains the pointer to the heap and the references to the objects. It can also access the structure of the classes that are loaded and that are present in the class path or in the jar file that is included in the class path. When we make a JNI request to access a field, the JNI API first checks if the class is loaded or not. Then it accesses the class structure and goes through the field list. If it finds the field that we are looking for it returns the fieldId to the caller and the caller can access this field's data using this Id.

To accelerate this search process, hashing technique is used. There are three hash tables. The first one, *refEntryTable*, is to hash and store the class structures. The second one, *methEntryTable,* is used to store the offset and the methodId of methods that are called through the JNI regardless of whether it is native method or Java method. The third one, *fieldEntryJni,* is for the fields that are accessed by the JNI in its native domain. These hash tables are initialized when the new JNI environment variable is created when *dvmCreateJNIEnv* is called during API invocation.

The hashing bypasses the search operations that need to be made every time when we need to access a method or field. Now due to hashing the symbolic, search operation is made during the first access and any future access will use the value obtained from the hash table. One problem with this approach is that we will not know the size of the hash table that we need to create. This issue is addressed by increasing the size by a set amount when the hash table reaches its limit.

To deal with possible collisions in hash tables, a probing technique is used where a newly inserted entry is saved in the next available free slot. This technique is chosen over the chaining for collision resolution as it uses the memory in an optimal way. During hash table accesses, we need not address the synchronization issue as JNI API is executed by a single thread and there is only one thread accessing the JVM environment variable.

The keys used in hash function are strings and are of UTF8 format. To compute the hash of the string we use the predefined function in the UtfString file and the formula to calculate the hash is

$$hash = key *31 + value\ of\ character$$

This hash is computed for the whole length of the string. Here we presume that add and look up happen more frequently than the remove. Whenever remove function is called there should be an explicit call to the free function as well as the remove function does not invoke it internally. If the free function is not called then there are high chances that we will be running out of space in the local reference table of the JNI environment. Similarly the free function also handles the global references that we have created for the values that we store.

B. *Find Class*

The find class in the JNI is used to load the class given the fully qualified class name in the JVM pointed by the JNI environment object. The find class takes in two arguments: the JNI environment object and the class name. The *findClass* first make a check by calling *dvmGetCurrentJNIMethod,* this method check if the current thread is executing a native method if so it returns the method by inspecting the interpreter stack. Then we get the class descriptor form the class name, load the class from its class loader, and add the local reference in the reference table. This is an important step as if the class does not have a reference then this class cannot be accessed in the native stack. Finally, the reference is returned to the caller who can use it to access fields and methods.

If the class was already loaded then we could reuse this instance of the class to get the fieldIds and methodIds. To accomplish this we store the reference of the class in the hash table *refEntryTable* in the JNI environment variable. Every time the *findClass* is called we check if the class is already loaded by looking up in the hash map. If not we proceed as usual and load the class. After loading the class we add this to the hash map. If the class we are looking for is present in the hash map we validate its reference as it may be garbage collected. Then this instance is returned to the caller. Note that the class descriptor, obtained by calling the method *dvmNameToDescriptor* is used as the key for hashing operation rather than the class name. This is because that the two classes can have the same name but their descriptors are different. The flowchart of *findClass* with hash table in illustrated in Figure 1.

C. *Get Field*

The JNI *getFieldId* method is used to get the instance field given the field name, its signature, and the Java class. To access any field in a Java object, we need to use this function. It returns the fieldId in the class structure through which we can get the offset from the object pointer. The Java class structure is obtained from the *findClass* method. Once the reference in the local reference table of the JNI is obtained, a call to *dvmDecodeIndirectRef* is made to receive the reference of the actual class. If the class is initialized, the steps to find the filedId are as follows

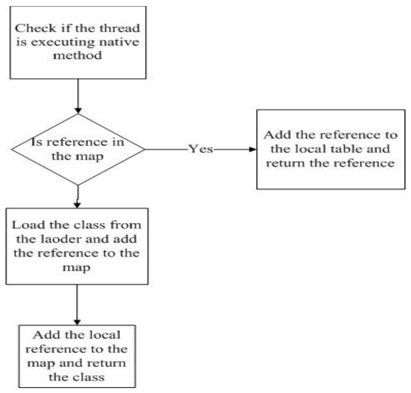
Check if the thread is executing native method
Is reference in the map
Yes
Add the reference to the local table and return the reference
Load the class from the laoder and add the reference to the map
Add the local reference to the map and return the class

Figure 1 *findClass* execution flow

- The class object is searched for the given field name and signature.
- This is done by walking through the ifields of the class that are provided. If found, the filedId is returned.
- If it is not found in the class, then it needs to check any super class. If yes, a search for the ifields of the super class is carried out. If it is found, the fieldId is returned. Otherwise, an exception is thrown.

The execution time of *getfieldId* method can be substantial due to the search operation of multiple lists. However, once a fieldId is found, it can be saved in a hash table to avoid following search for the same field as depicted in Figure 2. Note that multiple classes may have the same field name and signature. To make the key unique in hash table, a combination of field name, field signature, and class descriptor should be considered. In addition a check for the loaded class is required when we return the fieldID from the hash table. If the class that the field belongs to is not loaded, any further use of the Id will generate exception. This can be taken care of by looking into the hash table of findClass. If the class is not loaded then an exception is asserted.

D. Get Method

The g*etMethodId* JNI function is used to find the methodId for a given set of method name, signature, and class. If the class is loaded, the offset from the class pointer can be obtained based on the methodId. The method can then be invoked. The following steps are performed by this function.

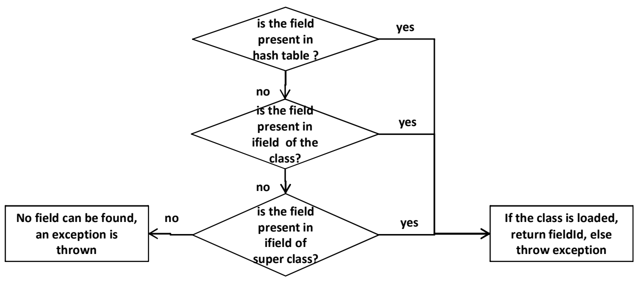
is the field present in hash table ?
yes
no
is the field present in ifield of the class?
yes
no
is the field present in ifield of super class?
yes
no
No field can be found, an exception is thrown
If the class is loaded, return fieldId, else throw exception

Figure 2 The operation of *getFieldId* Mothod

- It first checks if the class is loaded and initialized. If not an exception is thrown.
- It then goes through the list of methods in vtable to check if the method and the signature are present and checks if the method is static. If a virtual function is found and is not static, the methodId is returned.
- If it is not a virtual function then it goes through the directMethods list to check if it is present in the list. If present, it checks if the method is static or not. If static then it throws exception else returns the methodId.
- The method's class may not be the same as the class that is supplied. In this case, it must be a virtual method and the class must be a superclass which should be initialized.

To simplify the process of *getMethodId*, we add the method Id to the hash table represented by *methEntryTable*. When a call is made, we check if the method is present in the hash table. If the method is found, we check if the class and its super class are initialized or not. This is done by checking the hash table for the *refEntryTable*. If the class is not initialized, we initialize it and add it to the *refEntryTable* and to the local reference. The key used in the hash table is the class descriptor, the method name, and the signature. The complete operation of the *getMethodId* is illustrated in Figure 3 **Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.**. If we find that the method is static then we throw an exception as static methods cannot be accessed by the g*etMethodId*. Also check is made to see if there is a code segment to execute in the method while returning the method Id so that we do not catch any abstract method.

E. Pinning and Unpinning Object

To access objects, applications need to go through the several steps in JNI. This means that they need to do find class first then find the field offset and the get the value by invoking the get method for the object. This is because the garbage collector may move the object. However, if we pin the object, we are guaranteed that the object will be in same memory location. This is very helpful as we can access array of objects if we know the starting location in memory and size of the object instead of calling *getObjectArrayElement*.

Two new functions, *pinObject* and *unpinObject,* are implemented to avoid the complex process of accessing objects. If the memory location of an object is pinned and is known in the native space, its field can be accessed by knowing the offset. Pinning not only allows the access to the object but makes sure that the memory location of the object would not be changed. Obviously, it is important to unpin the object once we are done using it as these can be picked up by the garbage collector.

The *pinObject* method adds an entry to the global DVM pin reference table. This is done by making a call to

dvmAddToReferenceTable(&gDvm.jniPinRefTable, (Object)Obj)*

This assures that, once we add this object to this reference table, the garbage collector will no longer move or reclaim this memory location. The *unpinObject* function removes the reference from the global DVM reference table. We also remove the global reference that we added. To access the global DVM reference table, a mutex lock over the table must be obtained.

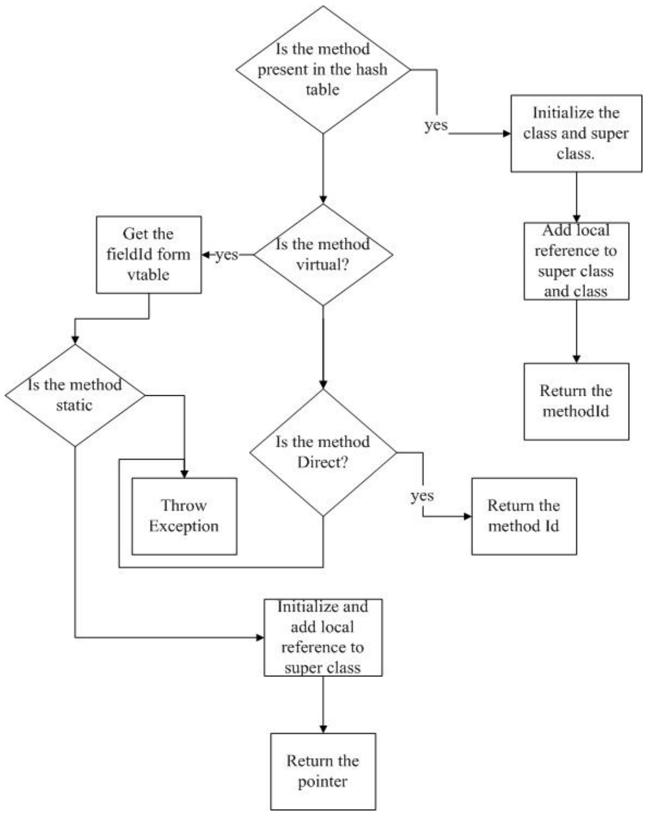
Is the method present in the hash table
yes
Initialize the class and super class.
Add local reference to super class and class
Return the methodId
Is the method virtual?
yes
Get the fieldId form vtable
Is the method static
Is the method Direct?
yes
Return the method Id
Throw Exception
Initialize and add local reference to super class
Return the pointer

Figure 3: The operation of getMethodId

V. EVALUATION

Experiment has been carried out for the comparisons between the JNI in the Dalvik virtual machine and the JNI with the proposed changes. To evaluate the execution time in the JNI we record the system clock in microseconds. This is accomplished by the following function is used to get the CPU time.

```
static inline u8 getClock()
{
#if defined(HAVE_POSIX_CLOCKS)
      struct timespec tm;
      clock_gettime(CLOCK_THREAD_CPUTIME_ID, &tm);
       if (!(tm.tv_nsec >= 0 && tm.tv_nsec < 1000000000)) {
            LOGE("bad nsec: %ld\n", tm.tv_nsec);
            dvmAbort();
      }
       return tm.tv_sec * 1000000LL + tm.tv_nsec / 1000;
#else
      struct timeval tv;
      gettimeofday(&tv, NULL);
      return tv.tv_sec * 1000000LL + tv.tv_usec;
#endif
}
```

The time is logged into the system log file using the command *LOGI(...)*. This is a predefined function in the Dalvik virtual machine which internally uses *printf* command to write strings into the system log buffer and then the buffer is flushed to Android Debug Bridge console. The log statements have been put in the JNI_ENTRY and JNI_EXIT macro. These macros are invoked each time there is a call to JNI API. In each of the JNI API we log the method name so that we can identify the time taken in each of these calls. In addition, these log messages are tagged with their corresponding processId and threadId.

A. *Profiling Results*

As explained earlier, we use the logging facility provided by the Android Operating System and the system clock to profile the JNI calls. In the first experiment, each of the applications Lunar Lander and JetBoy is executed 1000 times. The average execution time of each JNI call is listed in Table 3.

Time (ms)	Find Class	Get Method ID	Get Filed ID	Get Static FiedlID	Dvm Create JNIEnv	GetStatic Method ID
JNI	708.2	147.8	133.2	269.4	308	80.7
JNI with changes	606.4	117.1	114.9	198.4	281.6	82.2
% of change	-14.4%	-20.8%	-13.7%	-26.4%	-8.6%	1.8%

(a) Lunar Lander

Time (ms)	Find Class	Get Method ID	Get Filed ID	Get Static FiedlID	Dvm Create JNIEnv	GetStatic Method ID
JNI	638.4	206.3	133.2	362.3	308	80.7
JNI with changes	572.4	168.5	92.9	294.4	281.6	82.2
% of change	-10.3%	-18.3%	-30.3%	-18.7%	-8.6%	1.9%

(b) JetBoy

Table 1: The average execution time of JNI methods in Lunar Lander and JetBoy applications

The reduction in execution time of JNI method calls is due to the fact that the subsequent calls to these methods need not go through the symbolic search for the object to get the field or class. The calls still require when the object is initialized so that the reference is present in JVMs local reference. There are some differences of the percentage of reductions in the two applications. JetBoy implements an interactive music soundtrack adapted to user action. In Lunar Lander, users try to land on the moon by controlling the lander using keystrokes. It demonstrates the drawing resources and animation in Android. So, based on the classes, methods, and fields, defined in the applications, they experience various execution times of JNI operations and results in different reduction. The results on the last two JNI methods in the rightmost columns are the same as the methods are invoked to start the applications and don't depend upon application data.

While there are 10% to 30% reductions in the execution times of application-dependent JNI calls, it is interesting to look into the overall impact to application's execution. We consider JetBoy application and record its execution times in Java, native calls and Linux OS. For this we use *gprof* to profile the emulator. To address the issue of OS scheduling that is done by the kernel when multiple applications are running, only one application is loaded into the emulator so that all the resources are used by this application. The application changes the music based on the events that occur in the game. It uses the JET library that is provided by Android to play the music files. OpenGL is used to render the graphic content that is used in the game. Both music playing and graphic rendering use JNI by the means of NDK library. The measurement is taken for approximately 2 seconds after the application is started. The results are shown in the following figure.

Environment	with original JNI	with modified JNI
Java	1.49	1.48
JNI	0.37	0.31
OS and others	0.1	0.1

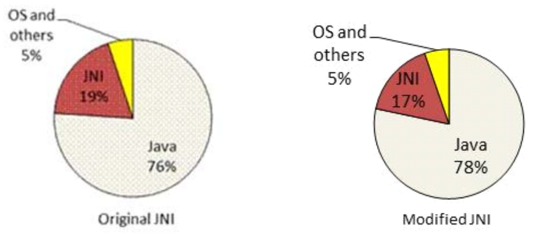

Figure 4 Execution times of JetBoy with the original modified JNIs

There are 1173 and 1184 calls made to the JNI interface during the program execution with the original and modified JNI. Time is captured in the JNI API using the system time stamp. The number of JNI calls is recorded in a static variable declared in the JNIEnv variable. Once the JNI destroy was called we printed it to the log. Then the execution times are summed up all the count. Though the number of calls remained almost same there is an improvement in the JNI time as the table shows. This is mainly due to the calls made to the *findClass* and *getFieldId* API methods. Note that more than 75% of JetBoy execution time is in Java space. This percentage should be reduced substantially if the Android JIT is applied in the experiments.

To investigate the effect of data movement, we test the running time of a heap sort algorithm on a data array. First, the heap sort is implemented in Java and recorded the execution time. Then, using JNI, the heap sort is implemented in the native domain. Then we ran the same code and the data on the modified JNI. In Figure 5**Error! Reference source not found.**, the average execution times of 1000 runs are shown. The improvements of native method and the one with the modified JNI can be noted when the array size increases and more data to be moved from the Java heap to the native space. Even with the JNI overheads, the use of native methods can be valued in mobile devices in terms of execution time and energy consumption.

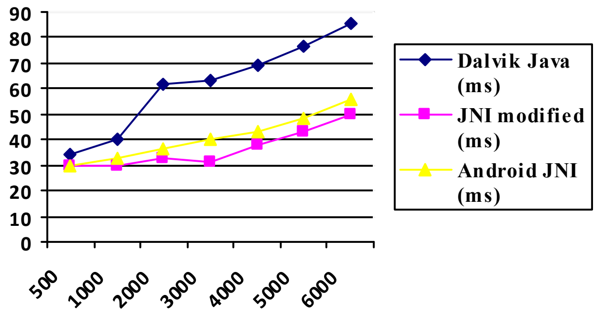

Figure 5 Execution times of heap sort of various size arrays

VI. CONCLUSION

JNI has been an important part of the Java virtual machine. It is widely used in embedded applications as an interface to architecture specific libraries in native code (C/C++) which can improve the performance of the applications and provide access to platform hardware devices. Our technique reduces the overhead of the JNI functions for accessing Java objects, fields, and methods. The pinning of objects helps the programmer to reference the data through its memory location rather than copying the object into the native space. There is a performance gain of 10-30% in the JNI functions achieved using the proposed technique.

Our analysis shows that there are still a significant overhead in JNI that should be consider further. This overhead is caused due to the separate executions spaces. Currently we are looking into the stack when the transfer is being made to the JNI and back. Also the profiling of the JNI is done using the execution time of the applications. To gain a deeper insight for the areas of improvement, the instruction profile in the JNI should be inspected.

REFERENCES

[1] "Android Open Source Project", http://source.android.com/index.html

[2] Damianos Gavalas and Daphne Economou "Development Platforms for Mobile Applications: Status and Trends" IEEE Software, Vol. 28, Issue 1, Jan.-Feb. 2011, page. 77-86.

[3] "App – Android Market", https://market.android.com/.

[4] S. Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, June 1999.

[5] Java Native Interface. http://download.oracle.com/javase/1.3/docs/guide/jni/index.html.

[6] "Android Native Development Tools – What is NDK," http://developer.android.com/sdk/ndk/overview.html.

[7] Sangchul Lee and Jae Wook Jeon "Evaluating Performance of Android Platform Using Native C for Embedded Systems" *2010 International Conference on Control Automation and Systems (ICCAS),* pp. 1160 - 1163.

[8] Dawid Kurzyniec and Vaidy Sunderam, "Efficient cooperation between Java and native codes – JNI performance benchmark*." 2001 International Conference on Parallel and Distributed Processing Techniques and Applications, (PDPDA), 2001*.

[9] Google, "Android 2.3 User Guide", www.google.com/googlephone/AndroidUsersGuide-2.3.pdf.

[10] David Ehringer, "The Dalvik Virtual Machaine Architecture," http://davidehringer.com/software/android/The_Dalvik_Virtual_Machine.pdf.

[11] Martin Hirzel, Daniel Von Dincklage, Amer Diwan, and Michael Hind, "Fast online pointer analysis", *ACM Trans. Programming Languages and Systems,* Vol. 29, Issue 2, April 2007.

[12] Miaobo Chen, Shalom Goldenberg, Suresh Srinivas, Valery Ushakov, Young Wang, Qi Zhang, Eric Lin, and Yoav Zach, "Java JNI Bridge: A Framework for Mixed Native ISA Execution", *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06),* pp. 65—75.

[13] Chi-Chao Chang and Thorsten von Eicken "Interfacing Java to the Virtual Interface Architecture" *JAVA '99 Proceedings of the ACM 1999 conference on Java Grande*, pp. 51-57.

[14] Martin Hirzel and Robert Grimm, "Jeannie: Granting Java Native Interface Developers Their Wishes" *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented Programming Systems and Applications (OOPSLA '07),* pp. 19 - 38

[15] Levon Stepanian, Angela Demke Brown, Allan Kielstra, Gita Koblents and Kevin Stoodley, "Inlining Java Native Calls At Runtime" *Proceedings of the 1st ACM/USENIX international conference on Virtual Execution Environments (VEE '05)*, pp. 121--131.

[16] Marian Bubak, Dawid Kurzyniec, and Piotr Luszczek "Creating Java to Native Code Interfaces with Janet Extension" *Proc. SGI Users's Conference,* Oct. 2000, pp. 283—294.

[17] Walter Cazzola "SmartMethod: an Efficient Replacement for Method" *Proceedings of the 2004 ACM symposium on Applied computing (SAC'04),* pp. 1305 - 1309,

[18] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald, "Thread-Local Heaps for Java" *Proceedings of the 3rd International Symposium On Memory Management (ISMM '02),* pp. 76-87.

[19] Walter Binder, Jarle Hulaas and Philippe Moret "A Quantitative Evaluation of the Contribution of Native Code to Java Workloads" *Proceedings of the 2006 IEEE International Symposium on Workload Characterization* (IISWC 2006), pp. 201-209.