Title Text

Subtitle Text, if any

Roussel Gilles
University Paris-Est Marne-la-Vallee
roussel@univ-mlv.fr

Forax Remi
University Paris-Est Marne-la-Vallee
forax@univ-mlv.fr

Pilliet Jerome
University Paris-Est Marne-la-Vallee
pilliet@univ-mlv.fr

Abstract

This is the text of the abstract.

bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla bla . . .

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Android

Android is the mobile operating system of Google. It's an open-source project called "Android Open-Source Project" (AOSP). In Q2 2013, Android occupies almost 80% of the market share with more than 187 million units shipped [1]. The success of Android can be explain by an open project and an open market which are very attractive for devices producers, service providers and application developers [5].

Android mainly run on embedded environments like smartphones and tablets. Therefore, strong constraints apply to Android and it have to support ARM architecture and now Intel architecture.

Even if hardware of devices which supports Android becomes more efficient, devices can't be as efficient as a desktop computer because of miniaturization of devices. Being a mobile device requires a minimal energetic consumption. Moreover, Android is portable and it must be adaptable to different devices. Forcing it to abstract itself from hardware.

The Android architecture is described like a "software stack" (fig. 1). It's composed to:

- a modified version of the Linux kernel. It offers an hardware abstraction, an existing memory and process managements and a security and networking models;
- native libraries (C/C++) which provide most of features of the Android system;

PPPJ '14, Month d–d, 20yy, City, ST, Country.
Copyright © 2014 ACM 978-1-nnnn-nnnn-n/14/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

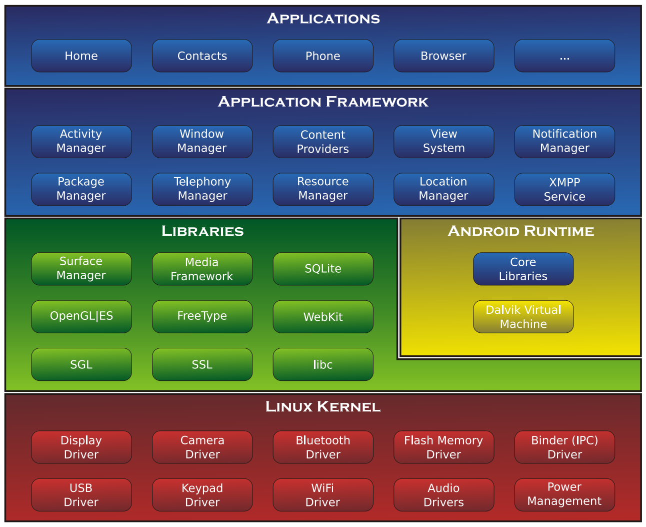

Figure 1. android system architecture [3]

- a virtual machine called "Dalvik". Dalvik is a important component of Android. It runs applications converted into the DEX format. Dalvik uses a core API written from scratch in Java which can be assimilate to the version 5 or 6 of the Java API;
- an application framework to enable the making of Android applications;
- and some default applications.

1.1 Differences between the JVM and Dalvik

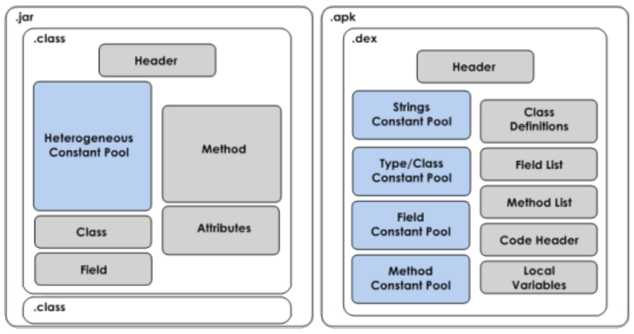

Figure 2. structure of JAR and APK files

The main difference between these two machines is that Dalvik is a register based machine while the JVM is stack based. A stack based VM uses more instructions to manipulate data and to implement Java code than a register based VM. But the register based VM instructions tends to be larger [6]

The bytecode Java is given in a ".class" file and only contains the code of one class (fig. 2). The file is made of a series of tables, which contains various information, the code of the class and some other things. The code contains references to these tables. Among these tables, one is the table of constants (constant-pool). This table stores most of the constant values of the class (numbers or texts) and more evolved elements (data types, class names, attribute names, . . .).

Whereas, DEX format contains all the code of the application (fig. 2). It will be loaded in one piece. It is a read-only format, so the code can't be changed at runtime. Duplicate constants such as strings used in multiple class files are included only once in the DEX file to save space. Unlike the JVM, Dalvik uses one constant-pool by type of constant except for the primitive types which are directly encoded with the opcode.

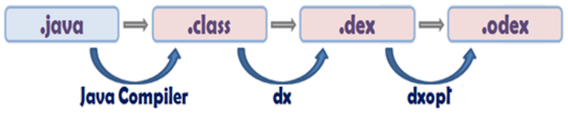

Figure 3. Android application tools-chain

Dalvik cannot directly execute Java bytecode. It have first to translate the Java byte code to a Dalvik specific bytecode called DEX and It cans optimize its bytecode (ODEX) to save space (fig. 3).

2. Java Specification Request 292

The JSR 292 is a request from JCP and led by Sun Microsystems in 2006. It had been integrated in 2011 in the version 7 of Java, modifying both the Java language specification and the JVM specification.

Since late 90's, a lot of language implementors have chosen the JVM as their target platform [4], for several reasons among the size of the ecosystem, the presence of mature GCs and Just In Time compilers.

The aim of this JSR is to ease the implementation of dynamically typed languages by providing a new way to do function calls that allows language implementors to specify a specific semantics, independantly of the semantics of the language Java.

This JSR is composed of two parts, the first part, specifies the semantics of new opcodes. The second part defined an API, java.lang.invoke that allows to create runtime typesafe function pointers (java.lang.invoke.MethodHandle) and methods to combine those function pointers to do thing like function composition, function argument permutation, etc.

The next sections will present the new API and new instructions but before we want to introduce an example that we will use for the rest of the paper.

2.1 Example

Let suppose we have the following code in Groovy [2]

```
def foo(a) {
  a + 2
}

println foo(9)
println foo("test")
```

It prints 11 and test2. For the fist call to foo, a is an integer so the function + refers to the function +(int,int) while for the second call to foo, a is a string, so the function + refers to the function +(String,String).

If one try to translate that code in Java bytecode, it will be something like

```
def foo(a) {
  aload 0   // 0 is offset of first variable, 'a' here
  iconst_2  // load the integer constant 2
  invokevirtual Object + (I)LObject;
  areturn
}
```

As you can see, the Java bytecode is typed, each instruction is prefixed by the type of the operand ('a' for object, 'i' for integer, etc) and the method call (invokevirtual) specify the type of the receiver and its parameter ; an object (java/lang/Object) followed by an int (I) ; and its return type which is also an object in the example.

The code above doesn't work because invokevirtual is an existing Java bytecode that call a virtual method using the Java semantics so the Virtual Machine or more precisely the bytecode verifier will check that there is a method '+' on java.lang.Object that takes an int and return an Object. If you have used Java, you alreadty know that this method doesn't exist. So the bytecode verification will fail.

Another possible translation is to use a cascade of if ... instanceof.

```
static Object plus(String v, Object v2) {
  return v.concat(String.valueOf(v2));
}

static Object plus(int v, int v2) {
  return Integer.sum(v, v2)
}

static Object foo(Object v, int i) {
  if (v instanceof Integer) {
    return plus((Integer)v, i);
  }
  if (v instanceof String) {
    return plus((String)v, i);
  }
  throw new Error();
}

public static void main(String[] args) {
  System.out.println(foo(9));
  System.out.println(foo("bar"));
}
```

While this code works, it supposes that all possible variations of + are known at compile time, something which is not true in most dynamic languages (Ruby, Groovy, Dart by example). This code can also be slow because for real languages, the number of if ... instanceof branches can be greater than a dozen.

To solve these issues, the JSR introduces a new bytecode named invokedynamic with no predefined linking semantics and a mechanism that allows to specify the linking semantics in Java or any languages that can be compiled to bytecodes. Invokedynamic is a function call so it can simulate every other existing Java calls that delegate its linking semantics to an external Java method, the bootstrap method. The bootstrap method is called the first time the invokedynamic opcode is encounter by the interpreter, with the context where the opcode is located i.e. the declaring class (encapsulated in a java/lang/invoke/Lookup object), a symbolic name, the declared parameter types as a java/lang/invoke/MethodType and return a CallSite object which is a box that contains a function pointer (a java/lang/MethodHandle). Any subsequent execution of the invokedynamic opcode will call the function pointer stored inside the callsite object returned by the bootstrap method.

For our running example, the code of foo with invokedynamic is

```
def foo(a) {
  aload 0
  iconst_2
  invokedynamic  + (LObject;I)LObject;
    bsm: invokestatic RT.bootstrap:(LMethodHandles$Lookup;LString;LMethodType;)LCallSite;
  areturn
}
```

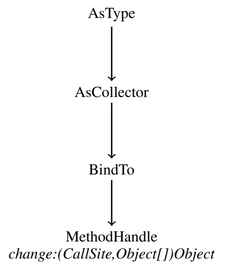

Figure 4.

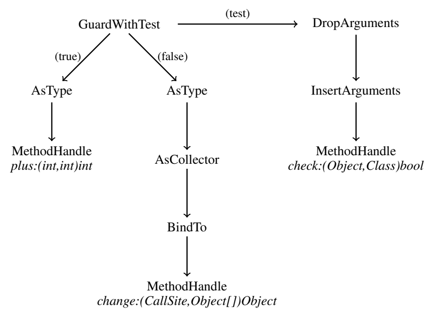

Figure 5.

The first time that foo is executed, the method bootstrap of the class RT will be called, with a lookup corresponding to the class containing foo, with "+" as string and a MethodType corresponding to the descriptor "(Ljava/lang/Object;I)Ljava/lang/Object;".

Moreover, invokedynamic comes with an API that allow to create several data pattern dynamically and lazily like by example a tree of if ... instanceof branches.

2.2 Package java.lang.invoke

The JSR 292 has created a new package named "java.lang.invoke". It contains several classes to ease the implementation of dynamic languages support. These classes are directly understood by the JVM. It is composed of classes MethodType, MethodHandle, CallSite and SwitchPoint. All classes answer to one or more problems coming from dynamic languages optimizations or implementations.

2.2.1 MethodType

A method type represents the arguments and return type accepted and returned by a method handle, or the arguments and return type passed and expected by a method handle caller. This class represents a method signature. A method type is immutable, so this type can be created only by factory methods.

```
  // "bar".concat(String.valueOf(2));
MethodType.methodtype(String.class,String.class);

  // Integer.sum(a, 2);
MethodType.methodtype(int.class,int.class,int.class);
```

2.2.2 MethodHandle / MethodHandles

A method handle is a typed, directly executable reference to an underlying method, constructor, field, or similar low-level operation, with optional transformations of arguments or return values. This class represents a function pointer.

A method handle can be create by different factory methods depending to the kind of method handle and accessibility.

1. **Kinds of MethodHandle**
   The MethodHandle class is composed of different types grouped into 4 kinds, "constant", "bound", "invokers" and "others". The three first have many specials paths in the VM for each instruction. While "others" have a unique path.
2. **Contents**
   It contains a type descriptor (methodtype). All instances of MethodHandle are immutable. It have a pair of special invoker methods called invokeExact and invoke. The invokeExact invoker accepts calls which exactly match the method handle's own type.

2.2.3 CallSite

A CallSite is a holder for a variable MethodHandle, which is called its target. An invokedynamic instruction linked to a CallSite delegates all calls to the site's current target. A CallSite may be associated with several invokedynamic instructions, or it may be "free floating", associated with none.

It has three immediate, concrete subclasses that may be either instantiated or subclassed.

1. **ConstantCallSite** : If a mutable target is not required.
2. **MutableCallSite** : If a mutable target is required
3. **VolatileCallSite** : If a mutable target is required which has volatile variable semantics.

A non-constant call site may be relinked by changing its target. The new target must have the same type as the previous target.

2.2.4 SwitchPoint

The Java bytecode is compiled and optimized on the fly by the JIT compiler. However, it may be that the code need to be deoptimize. The JVM contains some "flags" which indicate a need to degrade a code. So, it can "drop" a optimized code to optimize it again. it is essential that all threads were aware of changes. Using the safepoint (a mechanism of Garbage Collector) is recommended.

A SwitchPoint is an object allowing this mechanism. It indicates that a code must be degraded and synchronizes threads. The SavePoint is not necessarily available. An alternative is to synchronize all threads with a volatile variable. It contains a target method and a fallback method.

2.3 Instructions

Even if formally the JSR 292 adds only one instruction to the opcode set ("invokedynamic"), it also enhances the semantics of the ldc (load constant) instructions to load two new constants. The JSR 292 add new instructions to the VM opcode set like "invokedynamic". The "ldc" instruction has been enlarge to accept few new instructions. And some native methods are used as instructions.

2.3.1 ldc_methodtype

Like classes and strings, method types can also be represented directly in a class file's constant pool as constants. A method type may be loaded by an ldc instruction which refers to a suitable CONSTANT_MethodType constant pool entry.

2.3.2 ldc_methodhandle

Java code can create a constant native method with a polymorphic signature. A method handle that directly accesses any method, constructor, or field that is accessible to that code. Method handles that correspond to accessible fields, methods, and constructors can also be represented directly in a class file's constant pool as constants to be loaded by ldc bytecodes.

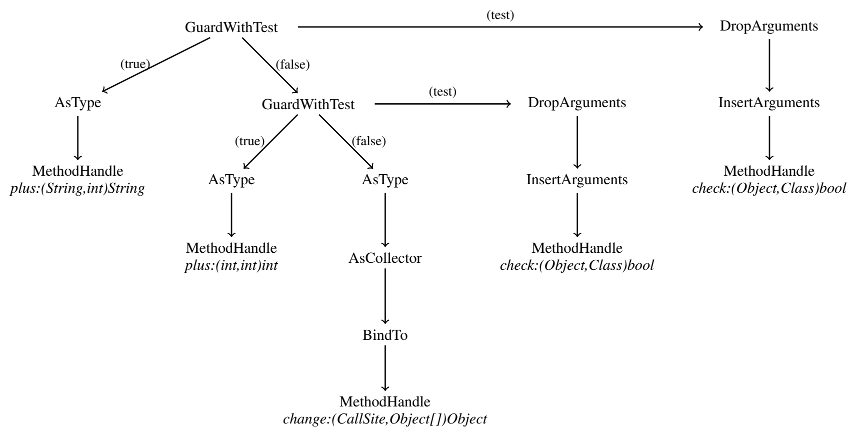

Figure 6.

A new type of constant pool entry, CONSTANT_MethodHandle, refers directly to an associated CONSTANT_Methodref, CONSTANT_InterfaceMethodref or CONSTANT_Fieldref constant pool entry.

2.3.3 invokedynamic

An invokedynamic instruction allows to call a method dynamically.

Before the first call, It must be initialized with a bootstrap method. This bootstrap method can take 0 to 251 arguments. Then, a call site is produced and replaces the first call. Moreover, a bootstrap method may be only an invokeStatic or an newInvokeSpecial.

It needs a descriptor of the resolved type of call, the target's name and a bootstrap method.

2.3.4 invokeVirtual on MH.invokeExact and MH.invoke

These instructions are here to make simple method calls *MethodHandle.invokeExact* and *MethodHandle.invoke*. Indeed, they are methods with a polymorphic signatures, so, it's difficult or impossible to implement it in Java. They use variables in the stack. The code cannot be more optimized than with a VM instruction.

2.4 Problème

2.4.1 2 implémentations (J9/HotSpot)

2.4.2 Dalivk = VM spécifique

2.4.3 non réutilisation des techniques

2.4.4 pas le même format DEX

2.5 Solutions

2.5.1 nouveau format DEX

2.5.2 nouvelles façons d'implémentations

3. Nouveau DEX

3.1 opcodes / LDC

3.2 DEX header (constant-pools)

3.3 Class header

3.4 rétro-compatibilité

4. Implémentations d'invoke-kind et des LDC

4.1 LDC

4.1.1 MethodType (interné ou pas)

4.1.2 MethodHandle

4.2 invoke-generic

4.3 invoke-exact

4.4 invoke-dynamic

4.4.1 tableau des MethodHandle (volatile)

4.4.2 CAS (bootstrap)

5. Implémentations des Combinateurs

5.1 Implémentation générale

5.2 Quelques exemples (GWT, fold, ...)

6. Benchmark

7. Related Work

8. Conclusion

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

[1] IDC website (2013) - `www.idc.com/getdoc.jsp?containerId=prUS24257413`

[2] Groovy website - `http://groovy.codehaus.org/`

[3] Android (Wikipedia) - `en.wikipedia.org/wiki/Android_(operating_system)#Linux`

[4] JVM languages (Wikipedia) - `https://en.wikipedia.org/wiki/List_of_JVM_languages`

[5] Butler, M., ”Android: Changing the Mobile Landscape,” Pervasive Computing, IEEE , vol.10, no.1, pp.4,7, Jan.-March 2011

[6] Paul, K.; Kundu, T.K., ”Android on Mobile Devices: An Energy Perspective,” Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on , vol., no., pp.2421,2426, June 29 2010-July 1 2010

[7] Paulson, L.D., ”Developers shift to dynamic programming languages,” Computer , vol.40, no.2, pp.12,15, Feb. 2007