UNIVERSITY OF WATERLOO
Faculty of Mathematics

A Project to Connect Two Oceans of Programming Languages

IBM Canada
Ottawa, Ontario

Table of Contents

List of Figures

Executive Summary

The purpose of this report is to compare how a program runs in Java to how it runs in native languages such as C and C++, and to show how native methods handles in Project Panama bridges the gap between these two types of languages. This involves solving problems caused by the compilation, the data types and the calling conventions of native languages, and comparing Project Panama with existing ways to call native methods. This report is intended for readers who have knowledge in program execution and object-oriented programming concepts.

Native method handles are able to call C or C++ methods in Java by obtaining the address from a native library using the method name. However, C++ method names are mangled during compile-time. To solve this problem, the native library is grovelled so that it contains metadata used by the native method handle when calling a native method.

There are data types in C and C++ that do not have direct Java equivalents, and vice versa. This means certain argument or return types could not be mapped between Java and native methods. Project Panama solves this problem by providing additional Java classes to represent data types such as pointers and structures.

Arguments are passed to the native method through the stack. However, integer values and floating point values are saved in different sets of registers on certain machines. The interface that Project Panama uses to call the method would only know which register set contains the values if it is given the argument types.

The major conclusion is that Project Panama is a safe and user-friendly way for Java programs to access native methods.

1.0 Introduction

The goal of Project Panama is to allow Java programmers to use the same libraries that non-Java programmers can use. Just like how the Panama Canal connects two oceans, Project Panama aims to connect two groups of programming languages: Java and native languages, namely C and C++. However, the compile-time and runtime differences between these languages make this task challenging.

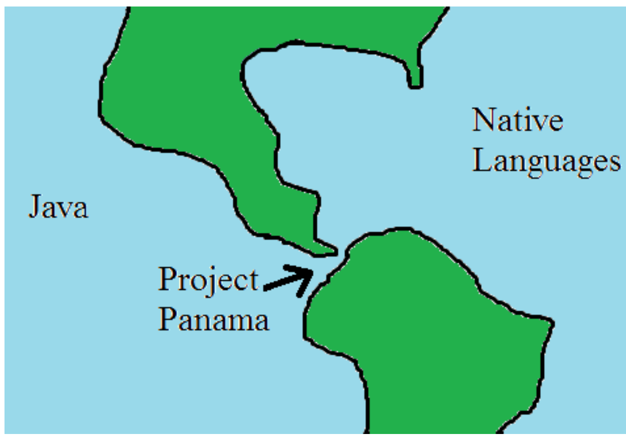

Figure 1: Project Panama connects two languages

Project Panama provides an alternative to the existing Java Native Interface (JNI), with application program interfaces (APIs) that are safer and easier to use. The programmer would not have to write JNI binding code, such as a JNI function for each native method. Project Panama also allows data types such as pointers and structures to be used safely, while JNI only supports primitive types and objects.

Programs written in native languages are compiled into object code and run on the machine itself, while Java programs are compiled into byte code and run on the Java Virtual Machine (JVM). This means a Java program can not call a native method directly like a Java method, but instead it must call the native method through a Foreign Function Interface (FFI). This call is triggered when invoking a native method handle, a new Java feature provided by Project Panama. The native program must be compiled into a native library. The native method handle then does a lookup in the native library to find the address of the method. Therefore, information about the native library, the method name, and the argument and return types are required.

```java
import java.lang.invoke.*;
import java.nicl.*;

public class NativeCallout {
    public static void main(String[] args) throws Throwable {

        Library lib = NativeLibrary.loadLibraryFile("/team/yuamy/NativeTestPrograms/libpanama.so");

        MethodHandle mh = MethodHandles.lookup().findNative(lib, "addTwoInt",
                 MethodType.methodType(int.class, int.class, int.class));

        int result = (int)mh.invoke(6,2);

        System.out.println(result);
    }
}
```

Figure 2: Creating and invoking a native method handle

Primitive data types, such as integers and floating point values, can be mapped directly between Java and native languages, but more complex data types, such as pointers and structures, exist in native languages but not in Java. Thus, there needs to be a way to express these data types in Java when calling methods where they are the argument or return types.

Argument values are copied from the Java stack to an internal buffer, which is then later copied to the native stack or passed by register. The native method handle has an interface for invoking the method, and the JVM interpreter puts the arguments onto the stack and triggers the FFI call on the method. However, different machines have different calling conventions, such as which set of registers store the argument values. Thus, the FFI call needs to be provided with enough information to resolves these differences.

```
0x006E5CD0 :  0000000000000002 0000000000000006 [ ................ ]
0x006E5CE0 :  00000000fffe2320 0000000000000000 [  #.............. ]
0x006E5CF0 :  00007ffff690a17a 00000000006e5d40 [ z.......@]n..... ]
0x006E5D00 :  0000000084fdc8e8 00000000fffe2320 [ ........ #...... ]
```

Figure 3: Two arguments on the Java stack

2.0 Analysis

This report will compare Project Panama with JNI and analyze the challenges that Project Panama faces for a Java program to be able to call a native method. There are many differences between Java and native languages, but this report will focus on three of them: how programs are compiled, how values are stored, and how arguments are passed.

2.1 Compilation

A native method gets compiled into object code, which can not run on the JVM, but a Java program can trigger a call-out on a native method with a native method handle. The native method handle does a FFI call with the address of the native method. This address can be obtained by doing a symbol lookup on the native library using the method name.

In C, the symbol lookup uses the name of the native method, so the programmer is expected to know it. However, C++ is object-oriented, which means methods can have the same name through overloading or if they belong to different classes. This is why name mangling happens when a C++ program is compiled, where the information about the argument types and parent classes are added onto the name of the method. This poses a problem for the symbol lookup, since it requires the mangled name that the programmer would not know.

```cpp
#include "stdlib.h"
#include "stdio.h"

class CppTest {
    public:
    int a;
    CppTest(int n);
    int test();
    int abc(int a, char b);
};

int CppTest::abc(int a, char b){
    return 1;
}

CppTest::CppTest(int n) {
    a = n;
    printf("constructor int\n");
}

int CppTest::test() {
    a++;
}

int addTwoInt(int val1, int val2) {
    return val1 + val2;
}
```

```
$ nm cpptest.so
0000000000200b30 d DW.ref.__gxx_personal
0000000000200938 a _DYNAMIC
0000000000200b00 a _GLOBAL_OFFSET_TABLE_
                 w _Jv_RegisterClasses
0000000000000773 T _Z9addTwoIntii
000000000000071c T _ZN7CppTest3abcEic
000000000000075a T _ZN7CppTest4testEv
0000000000000734 T _ZN7CppTestC1Ei
0000000000000734 T _ZN7CppTestC2Ei
0000000000200910 d __CTOR_END__
0000000000200908 d __CTOR_LIST__
0000000000200920 d __DTOR_END__
0000000000200918 d __DTOR_LIST__
0000000000000900 r __FRAME_END__
0000000000200928 d __JCR_END__
0000000000200928 d __JCR_LIST__
0000000000200b38 A __bss_start
                 w __cxa_finalize@@GLIBC
00000000000007b0 t __do_global_ctors_aux
0000000000000670 t __do_global_dtors_aux
0000000000200930 d __dso_handle
                 w __gmon_start__
                 U __gxx_personality_v0@
0000000000200b38 A _edata
0000000000200b48 A _end
00000000000007e8 T _fini
00000000000005f0 T _init
```

Figure 4: The method names are mangled

One solution is where the native method handle generates the mangled name programmatically using the given argument types before doing the symbol lookup. However, this would not be suitable, because the mangled name includes the class the method belongs to, which the native method handle would not know, and because the name mangling conventions are different for each compiler.

Project Panama solves this problem by grovelling the native library to create a Java interface that contains each native method as an abstract Java method. Each method contains metadata about the mangled name, and the argument and return types.

```
@C(file="/team/yuamy/NativeTestPrograms/cpptest.so", line=0, column=0, USR="c:@F@addTwoInt")
@NativeType(layout="(ii)i", ctype="int (int , int )", name="_Z9addTwoIntii", size=1L)
@CallingConvention(1)
public abstract int addTwoInt(int val1, int val2);
```

Figure 5: A grovelled native method with metadata about the mangled name

Instead of creating a method handle and invoking it, the programmer can do a higher level native method call through the Java interface created by the groveller. Project Panama is able to obtain the necessary information from the metadata to create a private native method handle for the higher level call. It then automatically invokes that method handle to trigger the FFI call.

This is the preferred way of calling native methods in Project Panama and the only way that will be provided to users. The low level way of invoking a native method handle will be hidden or made hard to access. The final product will require users to run the groveller to generate the interfaces, while mechanisms for invoking native method handles directly will be used mostly by JVM developers to support the high level functionality.

```
import java.lang.invoke.*;
import java.nicl.*;

import panama.test.cpptestPanama;

public class NativeCallout {
    public static void main(String[] args) throws Throwable {

        Library lib = NativeLibrary.loadLibraryFile("/team/yuamy/NativeTestPrograms/libpanama.so");

        cpptestPanama test = (cpptestPanama) NativeLibrary.bindRaw(cpptestPanama.class, lib);

        int result = test.addTwoInt(2, 6);

        System.out.println(result);
    }
}
```

Figure 6: Higher level native method call on a grovelled method

2.2 Data Types

Pointers and structures exist in native languages, but not in Java, so how would Project Panama deal with native methods that take these data types as argument or return values?

One way to store a pointer in Java would be simply as an integer or long variable containing the address, as it is done in JNI. However, this would lose all type safety. The data type that the pointer points to may not be the right type for the operation.

In addition, directly accessing to the value that the pointer makes reference to can cause the JVM to crash. A Java program can use an address to access native memory directly through the sun.misc.Unsafe API. However, as the name suggests, this is unsafe. It is platform dependent and allows users to access memory that shouldn't be accessed. One of the main goals of Project Panama is to provide a safer alternative to limit the use of Unsafe APIs.

Project Panama provides a *Pointer* class, which stores the address as one of its fields, but also has an function called *lvalue* for obtaining the value that the address refers to.

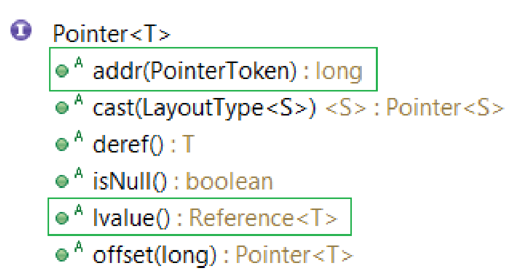

Figure 7: The *Pointer* class

The native side can not use an object of type *Pointer*, which is a Java class. To pass an argument of type *Pointer* from Java to the native side, it would have to be converted into a long value containing the address, before saving that value onto the stack. A pointer return value is passed from the native method to the Java side as a long value which can be used to create an object of type *Pointer*.

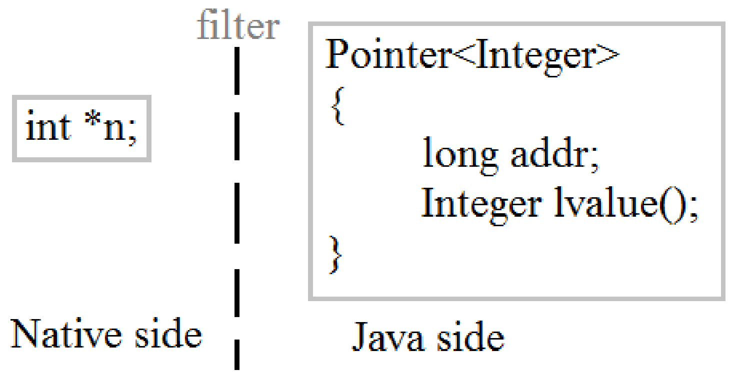

Figure 8: The MethodHandle API provides filters to convert pointers between native and Java

For structure data types, JNI uses DirectByteBuffer or sun.misc.Unsafe, and the programmer has to manually keep track of memory offsets. This can cause safety issues. For example, if the offset is wrong, the program will try to access memory that is not part of the structure. Project Panama represents structures as interfaces in a grovelled native library, which is safer and more user-friendly, since the programmer will not have to worry about the memory locations.

A structure can then get passed into or out from a native method as a consecutive slots on the stack. However, the FFI call needs to know where the structure ends, in other words, the size of the structure. Since the size of the structure depends on its field types, there needs to be information about the fields of the structure.

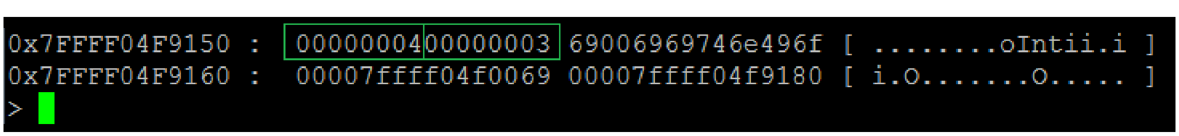

Figure 9: A structure with two fields on the stack

Information about a structure's fields can be provided in the metadata of a grovelled native library. The layout metadata describes the argument and return types of a native method, or the field types of a structure. When the native method is invoked, the higher level native method handle parses the layout to obtain the structure's field types for the FFI call.

```
@java.nicl.metadata.C(file="", line=0, column=0, USR="c:@SA@Point")
@java.nicl.metadata.NativeType(layout="[ii]", ctype="Point", size=8L, isRecordType=true)
public abstract static interface Point extends java.nicl.types.Reference {
    @java.nicl.metadata.C(file="/team/yuamy/NativeTestPrograms/cpptest.so", line=0, column=0,
    @java.nicl.metadata.NativeType(layout="i", ctype="int", size=4L, name="x")
    @java.nicl.metadata.Offset(offset=0L)
    public abstract int x$get();
    public abstract void x$set(int arg0);
    public abstract java.nicl.types.Reference x$ref();

    @java.nicl.metadata.C(file="/team/yuamy/NativeTestPrograms/cpptest.so", line=0, column=0,
    @java.nicl.metadata.NativeType(layout="i", ctype="int", size=4L, name="y")
    @java.nicl.metadata.Offset(offset=32L)
    public abstract int y$get();
    public abstract void y$set(int arg0);
    public abstract java.nicl.types.Reference y$ref();

}
@C(file="/team/yuamy/NativeTestPrograms/cpptest.so", line=0, column=0, USR="c:@F@createPoint")
@NativeType(layout="(ii)[ii]", ctype="Point (int , int )", name="_Z11createPointii", size=1L)
@CallingConvention(1)
public abstract Point createPoint(int x, int y);
```

Figure 10: A grovelled structure and native method with metadata about the layout

2.3 Calling Conventions

Since Java programs run on the JVM and native programs run on the machine itself, native programs can not access values from the Java side directly, and Java programs do not have access to values from the native side. This poses a challenge when passing argument and return values between Java programs and native methods.

To solve this problem, the JVM provides an interpreter containing two stacks, the Java stack which stores values from the Java side, and the native stack which stores values from the native side. The interpreter can move values from one stack to another. It also has information about the pointer to the first slot of the stack frame, and the stack pointer which points to the top of the current stack.

An argument is passed into a native method by moving the value from the Java stack to the native stack so that the native method can access it. Similarly, a return value is passed from the native method to Java by moving it from the native stack to the Java stack.

When passing a structure into or out of a native method, the structure's size is required, as explained in the previous section. However, it is not enough to only provide the size. The FFI call requires information about the data type of each individual field of the structure. This is because calling conventions require that values of different types are not passed in the same way during a method call.

For example, the x86 calling conventions state that integers are passed to registers RCX, RDX, R8, R9 and floating point numbers are passed to registers XMM0, XMM1, XMM2, XMM3. This means the FFI call needs to know which fields are integer and which fields are floating point, so that it would know which set of registers to find the values.

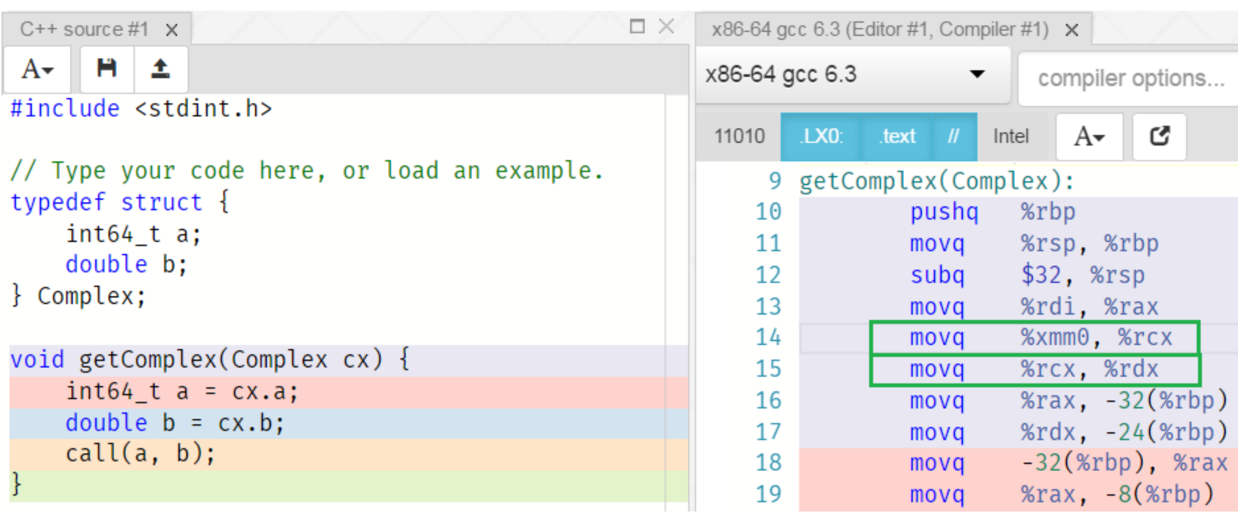

Figure 11: Field *a* is moved to register RCX and field *b* is moved to XMM0

This is why the FFI call needs to be provided with the argument and return types by the programmer, and a structure's field types from the layout metadata.

3.0 Conclusion

In conclusion, Project Panama allows Java programs to call native methods by using the correct name of the method to do symbol lookup, providing classes to represent pointers and structures in Java, and following calling conventions when passing argument and return values. It provides a more safety and usability than existing APIs for calling native methods, such as JNI.

A C++ method undergoes name mangling when compiled, which poses a challenge when doing symbol lookup, since the programmer isn't expected to know the mangled name. Project Panama solves this problem by obtaining the mangled name from the metadata of a grovelled native library.

Pointers and structures are data types that exist in native languages, but not in Java. To pass and receive such values from native methods, Project Panama uses classes that contain the address of the data and an interface to obtain the value. This is a safer alternative to accessing memory directly with the Unsafe API.

Calling conventions differ by machine. Integer and floating point values may be passed into different sets of registers. Project Panama resolves problems caused by calling conventions by providing information about the argument and return types to the FFI call, including the field types of structures.

References

Project Panama: Interconnecting JVM and native code [Online].
"http://openjdk.java.net/projects/panama/"

Project Panama Layout Prototype [Online].
"https://developer.ibm.com/open/openprojects/project-panama-layout-prototype/"

The isthmus in the VM [Online].
"https://blogs.oracle.com/jrose/entry/the_isthmus_in_the_vm"

Parameter Passing [Online].
"https://msdn.microsoft.com/en-us/library/zthk2dkh.aspx"

Name mangling demystified [Online].
"http://www.int0x80.gr/papers/name_mangling.pdf"

The Structure of the Java Virtual Machine [Online].
"https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html"