Project Report

Qiqi Zhou(4901286)   Zimu Yang(5781489)

# Call C from Java

*Java Native Interface*

The most common way for developers using C function in Java is Java Native Interface (JNI). It is a framework that allows programmer's Java code to call native applications or libraries written in languages C. One of  advantage for using JNI is that it can reuse existing C code. It is well known that native method have better performance in low level O/S and H/W routines. The JNI framework lets a native method use Java objects in the same way that Java code uses these objects. A native method can create Java objects 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. 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 [1]. Here are simple six steps to use JNI:

1) Writing Java classes to declare the native method and load the shared library which includes  the native code. Then call the native method.
2) Use Java compiler, `javac`, to compile the Java code down to bytecode.
3) Define native function signatures by creating a C header file. In the Java SDK, there is a native method C stub generator tool, `javah`, can help us create the a C header file, which defines C-style functions.
4) Implement the C function according to the C header.
5) Create a shared library file which contains the native code.
6) Run the Java program.

*JNI in Android , Android runtime and Android NDK*

After researching, we find that JNI is widely used in Android. Android is a Java-based platform and it is necessary to think about how to overcome the limitations of Java in Android, like memory management and performance. The runtime system on Android is called Android runtime (ART). Before ART introduced, Android also used Dalvik as runtime system for some system services on Android [2]. ART and Dalvik can convert Java class into DEX bytecode and then translate DEX bytecode to native machine. The difference between ART and Dalvik is than Dalvik is Just-in-time compilation based engine and ART is ahead-of-time compilation. One of advantages of introducing ART is that it can improve garbage collection on Android. ART uses one GC pause instead of two and parallels processing during the remaining GC pause. ART also compact GC to reduce background memory usage and fragmentation. To implement parts of Android in native code, the Android Native Development Kit (NDK) is an useful tool for developers to get better performance. The NDK provides all the tools, like compilers, libraries and header filers, to build apps that access device natively. JNI is core of NDK.

*Implementation and Test*

To test the performance of JNI on Android, we build a simple app on Android. The test algorithm we used is Gaussian Blur Algorithm. This is because Java is not good at image processing compared to C. The compiled program has less overhead from different libraries in C and there it does not get interpreted by the VM like Java programs. The core algorithm we used is Stack Blur [3]. It is a faster blur algorithm based on Gaussian Blur algorithm. It creates a kind of moving stack of colors scanning through the image. The Fig. 1 shows the user interface for our application.

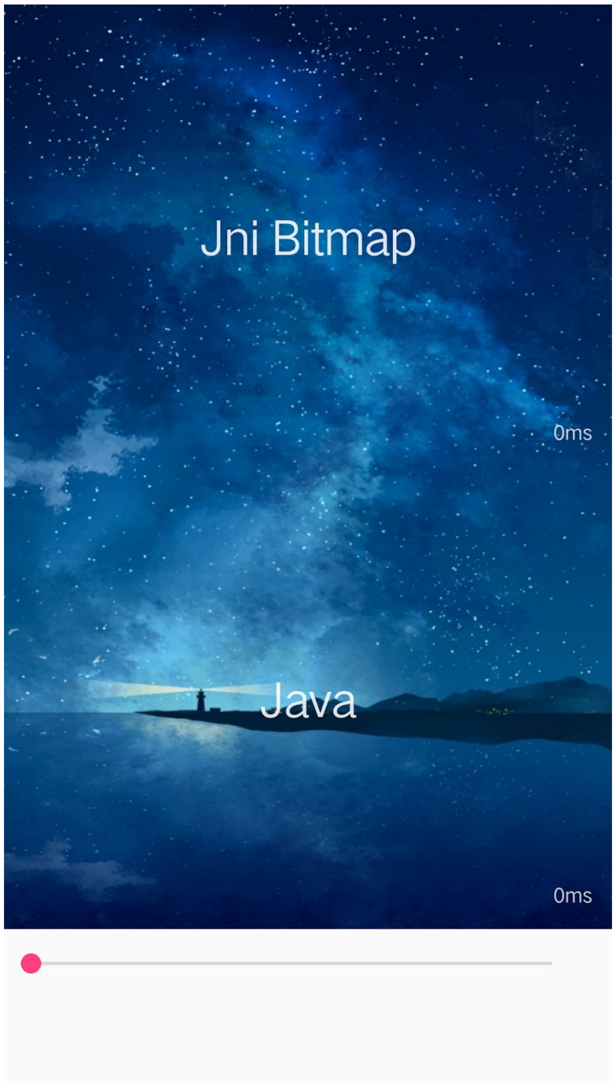


Fig. 1. User Interface

The two main areas in our app are "JNI" part and "Java" part. The "JNI" part uses Stack Blur algorithm written in C language and the "Java" part is written in Java. The seek bar on the bottom is to control the radius of Gaussian function in Stack blur algorithm. When the radius is increased, the image will be harder to see. The numbers shown on the bottom of two main areas are response time for program to blur image. The Fig. 2 shows the results of images when we applied our algorithm and choosed different radius.

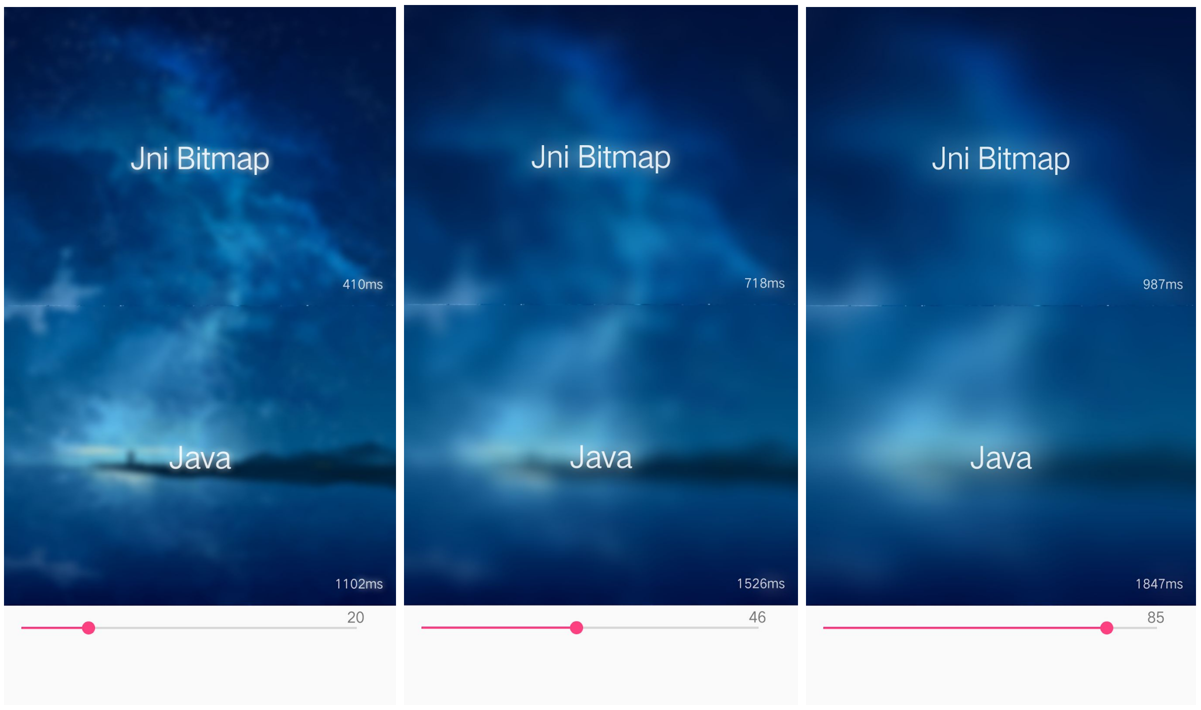
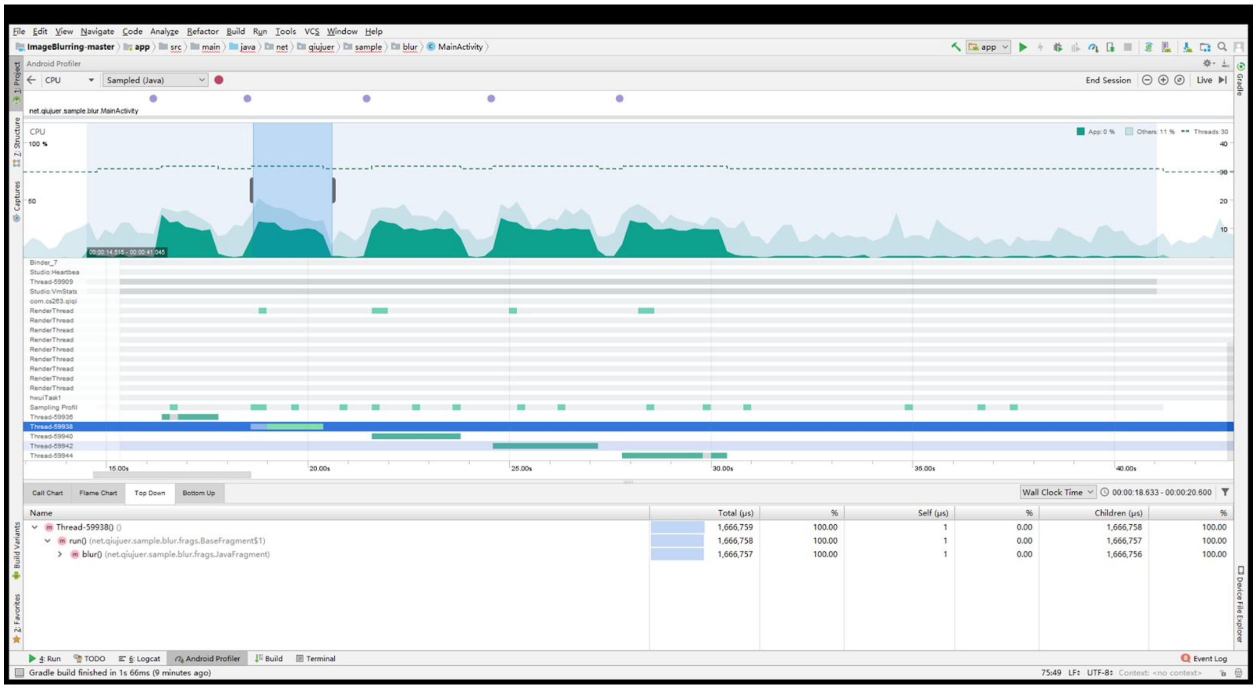


Fig. 2. The result of Image Blurring

*Result*

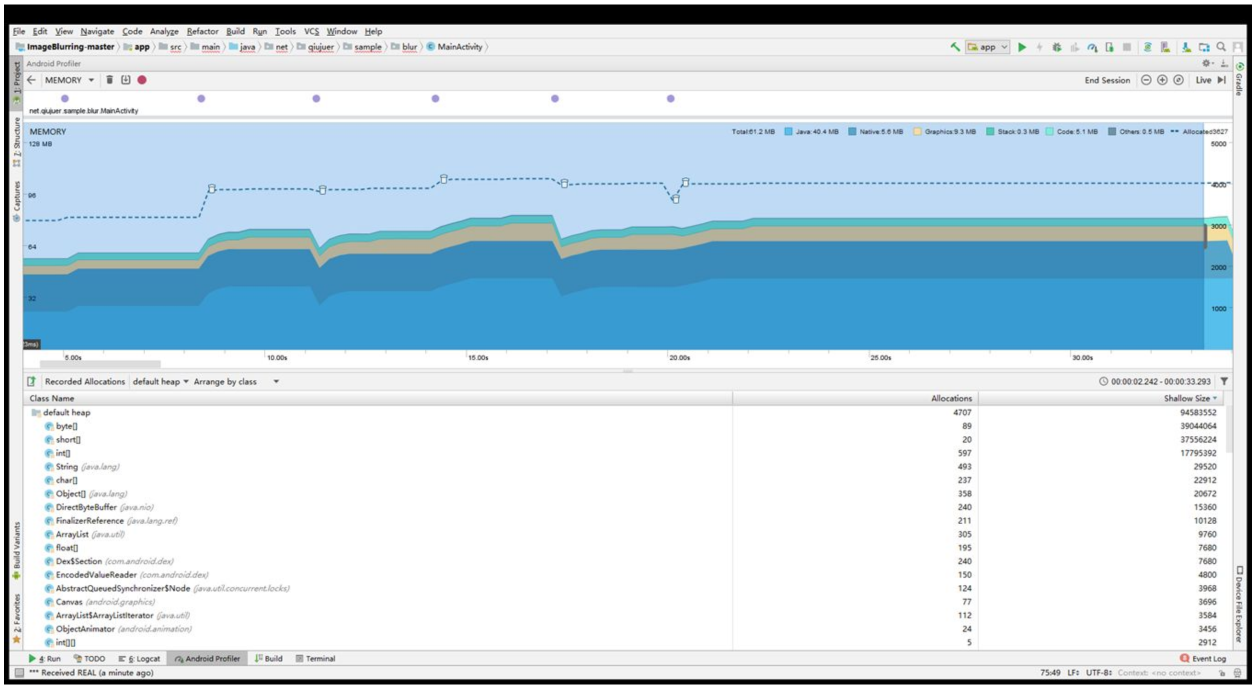
The Android device we used is OPPO R9 with CPU MT6755 and RAM 4GB. The version is Android 6.0. According to Fig. 2, we can clearly see that the response time for using JNI method is shorter than Java method. JNI is almost twice faster than Java in our application. For better analysis, we use Android profiler to profile our application. We collected five data samples when the radii are chosen as 10, 30, 50, 70 and 90. The results show in Fig. 3 to Fig. 6. The purple dots on the top of each graph in all figures represent actions when we increased the radius. From Fig. 3 and Fig. 5, we can see that JNI method has much lower CPU usage and shorter execution time of threads compared to Java method. According to Fig. 4 and Fig. 6, the memory usage are similar but we find there five garbage collections (trash icons) in Java method and there is only one garbage collection in JNI method. We believe this is one of reasons why Java method is slower than JNI method. In conclusion, the JNI method has better performance than Java method according to our test.



Wall Clock Time 00:00:18.633 - 00:00:20.600

| Name | Total (μs) | % | Self (μs) | % | Children (μs) | % |
|---|---|---|---|---|---|---|
| Thread-59938() () | 1,666,759 | 100.00 | 1 | 0.00 | 1,666,758 | 100.00 |
| run() (net.qiujuer.sample.blur.frags.BaseFragment$1) | 1,666,758 | 100.00 | 1 | 0.00 | 1,666,757 | 100.00 |
| blur() (net.qiujuer.sample.blur.frags.JavaFragment) | 1,666,757 | 100.00 | 1 | 0.00 | 1,666,756 | 100.00 |

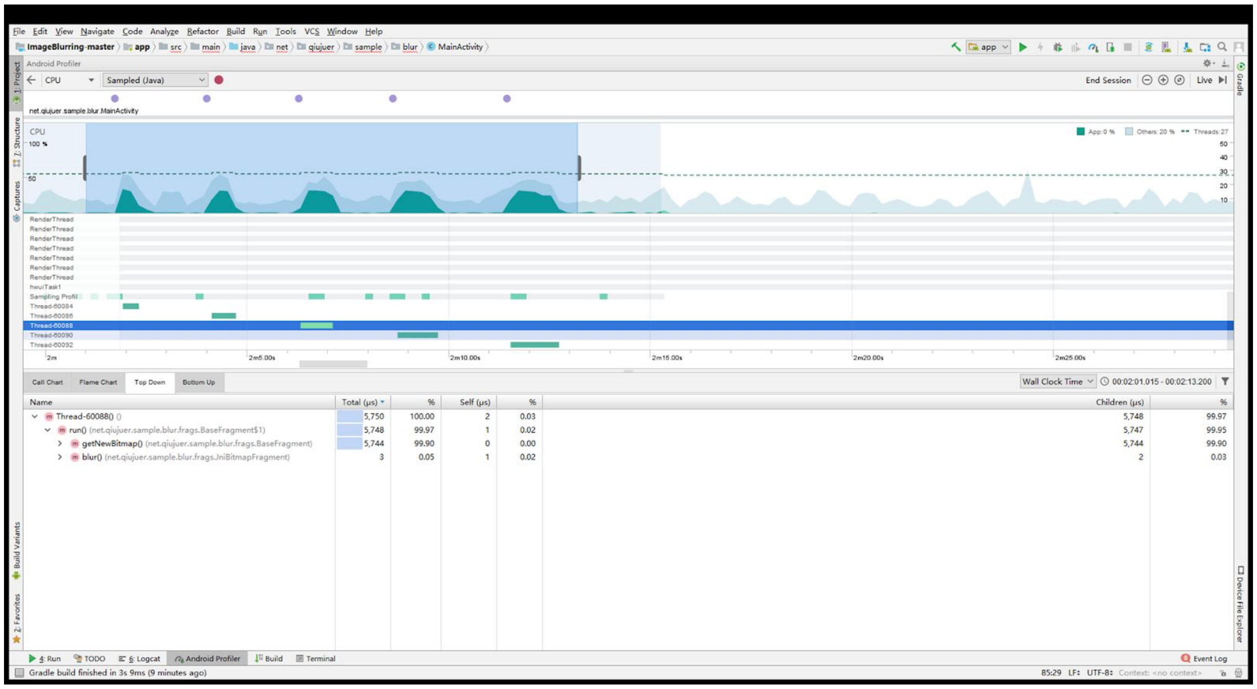
Fig. 3 CPU Usage for Java Method



Recorded Allocations default heap Arrange by class 00:00:02.242 - 00:00:33.293

| Class Name | Allocations | Shallow Size |
|---|---|---|
| default heap | 4707 | 94583552 |
| byte[] | 89 | 39044064 |
| short[] | 20 | 37556224 |
| int[] | 597 | 17795392 |
| String (java.lang) | 493 | 29520 |
| char[] | 237 | 22912 |
| Object[] (java.lang) | 358 | 20672 |
| DirectByteBuffer (java.nio) | 240 | 15360 |
| FinalizerReference (java.lang.ref) | 211 | 10128 |
| ArrayList (java.util) | 305 | 9760 |
| float[] | 195 | 7680 |
| Dex$Section (com.android.dex) | 240 | 7680 |
| EncodedValueReader (com.android.dex) | 150 | 4800 |
| AbstractQueuedSynchronizer$Node (java.util.concurrent.locks) | 124 | 3968 |
| Canvas (android.graphics) | 77 | 3696 |
| ArrayList$ArrayListIterator (java.util) | 112 | 3584 |
| ObjectAnimator (android.animation) | 24 | 3456 |
| int[][] | 5 | 2912 |

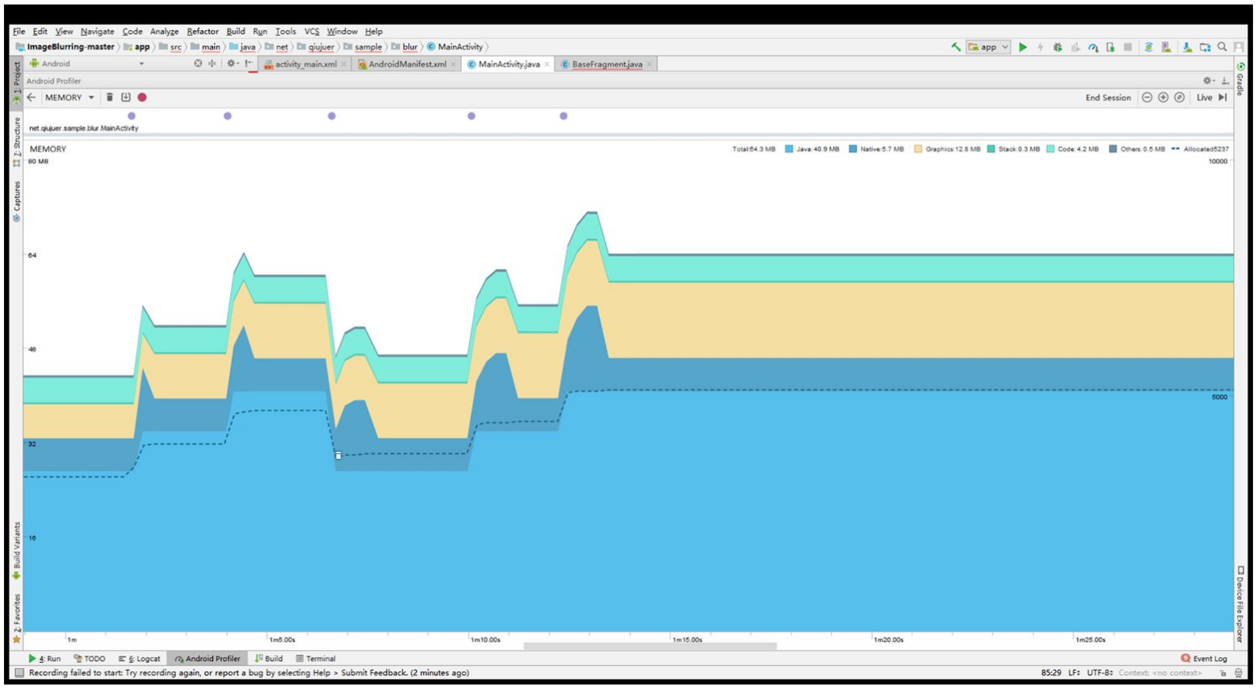
Fig. 4 Memory Usage for Java Method



| Name | Total (μs) | % | Self (μs) | % | Children (μs) | % |
|---|---|---|---|---|---|---|
| Thread-60088() () | 5,750 | 100.00 | 2 | 0.03 | 5,748 | 99.97 |
| run() (net.qiujuer.sample.blur.frags.BaseFragment$1) | 5,748 | 99.97 | 1 | 0.02 | 5,747 | 99.95 |
| getNewBitmap() (net.qiujuer.sample.blur.frags.BaseFragment) | 5,744 | 99.90 | 0 | 0.00 | 5,744 | 99.90 |
| blur() (net.qiujuer.sample.blur.frags.JniBitmapFragment) | 3 | 0.05 | 1 | 0.02 | 2 | 0.03 |

```
Gradle build finished in 3s 9ms (9 minutes ago)
```

Fig. 5 CPU Usage for JNI Method



```
Recording failed to start: Try recording again, or report a bug by selecting Help > Submit Feedback. (2 minutes ago)
```

Fig. 6 Memory Usage for JNI Method

## Reference

1. "Java Native Interface Specification Contents"
https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html

2. "Android NDK Guides " https://developer.android.com/ndk/guides/
3. "Stack Blur: Fast But Goodlooking" http://incubator.quasimondo.com/processing/fast_blur_deluxe.php

# Call java from C

## Main Components

To call java from pure C environment, we need to embed the JVM into our C program. Also, we need other components to communicate with JVM during runtime. JNI is a common interface to operate on JVM dynamically and JVMTI is the programmable interface we choose to retrieve profiling information of the application such as memory usage situation and stack trace. Below is the complete code to integrate these three components into our C program.

## Embed JVM and JNI:

```
JNIEnv* create_vm(JavaVM **jvm)
{
   JNIEnv* env;
   JavaVMInitArgs args;
   JavaVMOption options;
   args.version = JNI_VERSION_1_8;
   args.nOptions = 1;
   options.optionString = "-Djava.class.path=./";
   args.options = &options;
   args.ignoreUnrecognized = 0;
   jint rv;
   rv = JNI_CreateJavaVM(jvm, (void**)&env, &args);
   if (rv < 0 || !env)
      printf("Unable to Launch JVM %d\n",rv);
   else
      printf("Launched JVM! :)\n");
   return env;
}
```

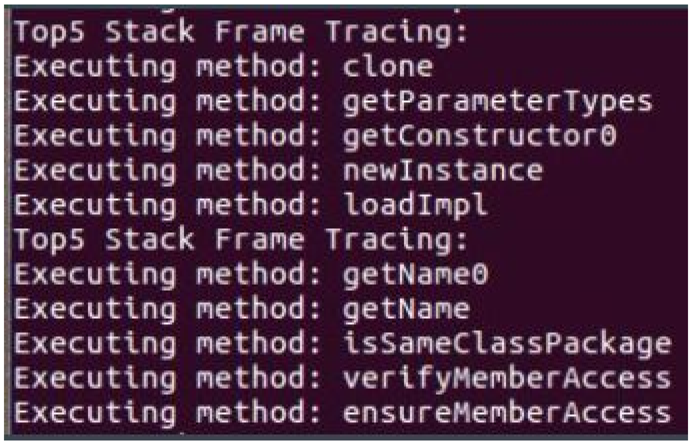
This function will return the JNI environment poiter and store the JVM pointer into the jvm parameter.

## Use JVMTI:

```
#include <jvmti.h>
```

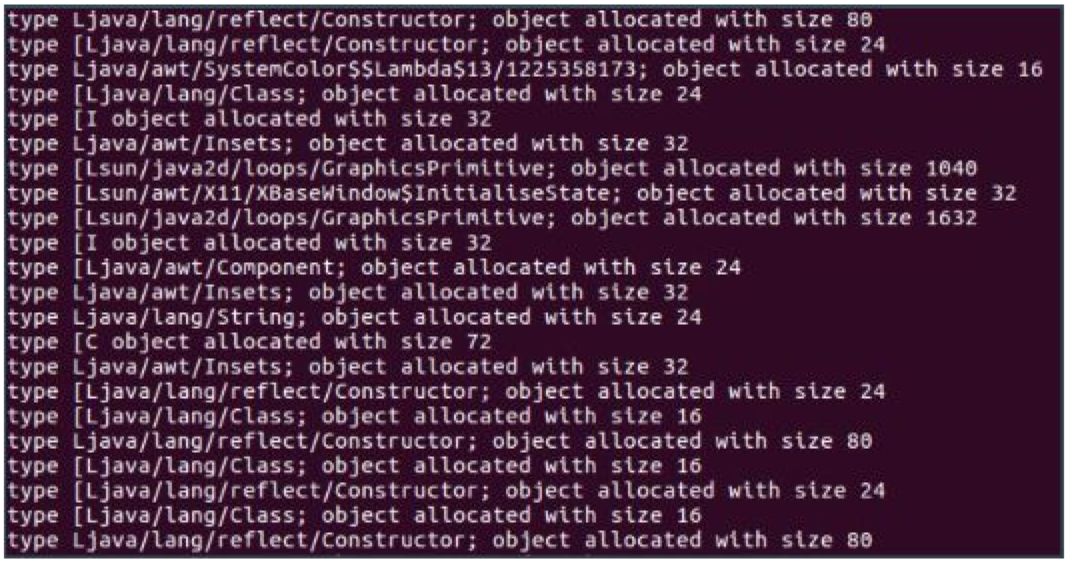
```
jint result = (*jvm)->GetEnv(jvm,(void** )&jvmti, JVMTI_VERSION_1_1);
        if (result != JNI_OK) {
                printf("ERROR: Unable to access JVMTI!\n");
        }
        jvmtiError err = (jvmtiError) 0;
        jclass *classes;
        jint count;
        memset(&capa,0,sizeof(jvmtiCapabilities));
        capa.can_get_thread_cpu_time=1;
        capa.can_tag_objects=1;
        capa.can_generate_vm_object_alloc_events=1;
        err = (*jvmti)->AddCapabilities(jvmti, &capa);
        if(err!=JVMTI_ERROR_NONE)
                printf("error add capabilities");
        Ecallbacks.VMObjectAlloc = &callbackVMObjectAlloc;
        //Ecallbacks.ObjectFree=;
        err=(*jvmti)->SetEventCallbacks(jvmti, &Ecallbacks,(jint)sizeof(Ecallbacks));
        if(err!=JVMTI_ERROR_NONE)
           printf("error add event call backs");

err=(*jvmti)->SetEventNotificationMode(jvmti,JVMTI_ENABLE,JVMTI_EVENT_VM_OBJE
CT_ALLOC,NULL);
     if(err!=JVMTI_ERROR_NONE)
           printf("error SET event mode");
```
Apart from including the header jvmti.h, we also need to do some setup work of JVMTI. Everytime the programmer wants to communicate with JVM, he has to get a JVMTI environment pointer and each JVMTI environment pointer is independent of each other. But to retrieve different profiling information, the JVMTI environment pointer has to register a set of callback functions into a special structure and the capabilities of this pointer has to be set by setting the value of another special structure. So it is a better choice to make this pointer a global pointer because the characteristics will not be inherited from one pointer to another one.

**Some examples of what we can do:**
Trace the stack frame and see what method is being called during runtime



```
Top5 Stack Frame Tracing:
Executing method: clone
Executing method: getParameterTypes
Executing method: getConstructor0
Executing method: newInstance
Executing method: loadImpl
Top5 Stack Frame Tracing:
Executing method: getName0
Executing method: getName
Executing method: isSameClassPackage
Executing method: verifyMemberAccess
Executing method: ensureMemberAccess
```

Trace the usage of Heap allocation dynamically:



```
type Ljava/lang/reflect/Constructor; object allocated with size 80
type [Ljava/lang/reflect/Constructor; object allocated with size 24
type Ljava/awt/SystemColor$$Lambda$13/1225358173; object allocated with size 16
type [Ljava/lang/Class; object allocated with size 24
type [I object allocated with size 32
type Ljava/awt/Insets; object allocated with size 32
type [Lsun/java2d/loops/GraphicsPrimitive; object allocated with size 1040
type [Lsun/awt/X11/XBaseWindow$InitialiseState; object allocated with size 32
type [Lsun/java2d/loops/GraphicsPrimitive; object allocated with size 1632
type [I object allocated with size 32
type [Ljava/awt/Component; object allocated with size 24
type Ljava/awt/Insets; object allocated with size 32
type Ljava/lang/String; object allocated with size 24
type [C object allocated with size 72
type Ljava/awt/Insets; object allocated with size 32
type [Ljava/lang/reflect/Constructor; object allocated with size 24
type [Ljava/lang/Class; object allocated with size 16
type Ljava/lang/reflect/Constructor; object allocated with size 80
type [Ljava/lang/Class; object allocated with size 16
type [Ljava/lang/reflect/Constructor; object allocated with size 24
type [Ljava/lang/Class; object allocated with size 16
type Ljava/lang/reflect/Constructor; object allocated with size 80
```

## Some problems

Even though there is not a very clear diagram showing the comparison of performance between the situation of calling Java from C and pure Java environment, what we observed in our experiments is that calling Java from C is a little bit slower than pure Java, which is very natural to understand. For most of the cases, they are about to cost almost the same time. But when encountering situation where the Java program has to iterate through an array of very large objects, pure Java outperforms calling Java from C apparently.

Besides, calling Java from C has another drawback that the programmer has to manage all variables in C himself. Like shown in the slides, even an object is not referred to by any other objects and is ready for garbage collection, if one of its previous references  lies in C program, the JVM will not know about it and will not automatically garbage collect it. The automation and convenience of JVM is not an advantage anymore in this situation.