Project Panama
Status update

March, 2019
A rising tide

GPUs and deep learning

• Linear algebra computations critical for machine learning
  – E.g. matrix multiplications (dot products) and additions

• Matrix computations are *embarrassingly parallel*!
  – GPUs provide acceleration for common computations (e.g. cuBLAS)

• Deep learning frameworks support GPUs as execution backend of choice
  – Theano, Tensorflow, Spark, Torch, …

• But wait, all these frameworks rely on *native* libraries!
Going native

• Sometimes you just have to “go native”
  – Off-CPU computing (Cuda, OpenCL)
  – Deep learning (Blas, cuBlas, cuDNN, Tensorflow, ...)
  – Graphics processing (OpenGL, Vulkan, DirectX)
  – Others (OpenSSL, SQLite, V8, ...)

• Languages/platforms must lower the *activation energy* required to do so!
Java Native Interface
Getpid in JNI

//Getpid.java
public class Getpid {
    native int getpid();
}

//Client.java
class Client {
    public static void main(String[] args) {
        new Getpid().getpid();
    }
}
Getpid in JNI

Workflow

Getpid_{java} \rightarrow \text{javac} \rightarrow \text{Getpid}_{class} \rightarrow \text{libc}_{so}

\text{unistd}_{h} \rightarrow \text{getpid}_{h} \rightarrow \text{getpid}_{c} \rightarrow \text{gcc/clang} \rightarrow \text{getpid}_{so}

user-written generated
Getpid in JNI
Gluing all the framents

//Getpid.java
public class Getpid {
    static {
        System.loadLibrary("getpid");
    }

    native int getpid();
}

//Client.java
class Client {
    public static void main(String[] args) {
        new Getpid().getpid();
    }
}

//getpid.h
#include <jni.h>
#include <stdlib.h>
#ifdef __Included_GetPid
#define __Included_GetPid
#endif

JNIEXPORT jint JNICALL Java_GetPid_getpid(JNIEnv *, jobject);

#endif

//getpid.c
#include <unistd.h>
#include "GetPid.h"

JNIEXPORT jint JNICALL Java_GetPid_getpid(JNIEnv * env, jobject recv) {
    return getpid();
}

#ifdef __cplusplus
}
#endif
Java Native Interface

Works, but...

• Good: Rich, bidirectional interop between Java and native code

• Bad: No support for modelling off-heap data
  – DIY solutions: Unsafe, ByteBuffer, ...

• Ugly: Convoluted workflow
  – (Java) users must know how to write (and build!) native code

• Result: writing native bindings in Java is hard!
  – Many things can go out of sync as native libraries are updated
When JNI fails

Java native bindings fall behind

Install TensorFlow for Java

TensorFlow provides a Java API—particularly useful for loading models created with Python and running them within a Java application.

Caution: The TensorFlow Java API is not covered by the TensorFlow API stability guarantees.

Supported Platforms

TensorFlow for Java is supported on the following systems:

- Ubuntu 16.04 or higher; 64-bit, x86
- macOS 10.12.6 (Sierra) or higher
- Windows 7 or higher; 64-bit, x86

To install TensorFlow on Android, see Android TensorFlow support and the TensorFlow Android Camera Demo.
"If non-Java programmers find some library useful and easy to access, it should be similarly accessible to Java programmers"

John Rose, JVM Architect
Panama

The approach

• Idea: model foreign libraries as ordinary Java interfaces
  – Foreign interfaces can be generated by tools
  – Implementations generated on-the-fly (*binding*)

• Rich API to model **off-heap** data
  – Layout, Pointer, Array, Scope, ...

• Result: no more native methods!
Getpid in Panama

Library as interfaces

- Foreign functions are *just* methods calls on some *library* object

_lib.getpid();
Getpid in Panama

Library as interfaces

- Foreign functions are just methods calls on some library object
- Library objects are obtained by binding a library interface

```javascript
var _lib = Libraries.bind(
    MethodHandles.lookup(),
    Getpid.class);

_lib.getpid();
```
Getpid in Panama
Library as interfaces

```java
@NativeHeader
interface Getpid {
   @NativeFunction("()i32")
   int getpid();
}

var _lib = Libraries.bind(
   MethodHandles.lookup(),
   Getpid.class);
_lib.getpid();
```

- Foreign functions are *just* methods calls on some *library* object
- Library objects are obtained by *binding* a library interface
- Library interfaces contain *metadata*
  - E.g. to describe native layouts
Getpid in JNI

Workflow

Getpid_{java} \rightarrow javac \rightarrow Getpid_{class} 

user-written generated

unistd_{h} \rightarrow getpid_{h} 

getpid_{c} \rightarrow gcc/clang \rightarrow getpid_{so} 

Copyright © 2019, Oracle and/or its affiliates. All rights reserved.
Getpid in Panama

Workflow

user-written

generated
Off-heap access
pointers and arrays

• Native pointers are modelled with generic class Pointer<X>
  – \text{Pointer}<X> = \text{address} + \text{layout}_{\text{pointee}} + \text{carrier}_{X}

• Basic operations
  – Offset, cast, dereference (get/set), iteration

• Pointers lifecycle managed by Scope
  – Cannot dereference a pointer whose owning scope has been closed!

• Native arrays are modelled with generic class Array<X>
  – \text{Array}<X> = \text{Pointer}<X> + \text{size}
Off-heap access

Pointers and arrays

```java
@NativeHeader
interface Strings {
    @NativeFunction("u64:u8)i32")
    int strlen(Pointer<Byte> buf);
}
...
var _lib = Libraries.bind(
    MethodHandles.lookup(),
    Strings.class);
try (var scope = Scope.newNativeScope()) {
    var strPtr = scope.allocateCString("Hello");
    _lib.strlen(strPtr);
}
```
Off-heap access

Pointers and arrays

```java
@NativeHeader
interface Strings {
    @NativeFunction("u64: u8) i32")
    int strlen(Pointer<Byte> buf);
}

... var _lib = Libraries.bind(
    MethodHandles.lookup(),
    Strings.class);
try (var scope = Scope.newNativeScope()) {
    var strPtr = scope.allocateCString("Hello");
    _lib,strlen(strPtr);
}
```

- **Scope + try-with-resources**
  - delimit code blocks which can safely access off-heap memory
Off-heap access

Pointers and arrays

@NativeHeader
interface Strings {
   @NativeFunction("u64:u8)i32")
   int strlen(Pointer<Byte> buf);
}

• Scope + try-with-resources
   – delimit code blocks which can safely access off-heap memory

• Scope provides many useful allocation helpers
   – allocateCString, allocateArray, ...

var _lib = Libraries.bind(
   MethodHandles.lookup(),
   Strings.class);
try (var scope = Scope.newNativeScope()) {
   var strPtr = scope.allocateCString("Hello");
   _lib.strlen(strPtr);
}
Panama

Scorecard so far

• Panama interfaces to access foreign functions/data w/o native code!
• But, writing annotated interfaces is (still) relatively hard and error prone!
  – Interface metadata contains platform-specific layout descriptions
• Real world example (Tensorflow)
  – 161 functions, 23 structs, 50 constants, 2 callbacks
  – Total: 26 annotated interfaces!
• Can we do better?
This Photo is licensed under CC BY-SA
Jextract

Tools sweet tools

- Goal: auto-generate bundles of annotated interfaces from a C header file
  - The generated jar bundle contains headers, structs, callbacks interfaces

- Jextract parses headers (clang), infers layouts, picks Java carrier types
  - The generated bundle is platform dependent!

- Tested with many real world libraries
  - Tensorflow, BLAS/LAPACK, OpenCL, Clang, OpenGL, Sqlite, Python, ...
Getpid in Panama

Workflow

Getpid_{java} \rightarrow javac \rightarrow \text{Getpid}_{\text{class}} \rightarrow \text{libc}_{\text{so}}
Getpid in Panama
Workflow w/ jextract

\[ \text{unistd}_h \rightarrow jextract \rightarrow \text{unistd}_\text{jar} \rightarrow \text{libc}_\text{so} \]
Getpid in Panama
Workflow w/ jextract

\[ \text{unistd} \_h \rightarrow jextract \rightarrow \text{unistd} \_\text{jar} \rightarrow \text{libc} \_\text{so} \]

- \text{unistd.class}
- \text{unistd$gid\_t.class}
- \text{unistd$intptr\_t.class}
- \text{unistd$off\_t.class}
- \text{unistd$pid\_t.class}
- \text{unistd$socklen\_t.class}
- \text{unistd$ssize\_t.class}
- \text{unistd$uid\_t.class}
- \text{unistd$useconds\_t.class}

user-written

generated
Getpid in Panama
Workflow w/ jextract

\[ \text{unistd}_h \rightarrow jextract \rightarrow \text{unistd}_\text{jar} \rightarrow \text{libc}_\text{so} \]

```c
... int getpid(); int getppid(); int getpgid(); int __getpgid(int); int getpgrp(); int setpgid(int, int); int setpgrp(); ...
```

```c
unistd.class
unistd$gid_t.class
unistd$intptr_t.class
unistd$off_t.class
unistd$pid_t.class
unistd$socklen_t.class
unistd$ssize_t.class
unistd$uid_t.class
unistd$useconds_t.class
```

user-written
generated
Getpid in Panama

```java
@NativeHeader(declarations=
   "getpid=()i32")
interface Getpid {
   int getpid();
}
...
var _lib = Libraries.bind(
   MethodHandles.lookup(),
   Getpid.class);
_lib.getpid();
```
Getpid in Panama
Closing the loop w/ jextract

import static stdlib.unistd_h.*;
...
getpid();
Performances
Performances

Getpid

Intel(R) Xeon(R) CPU E5-2665 @ 2.40GHz, 16 cores, 32G RAM

![Graph showing Performances of JNI and Panama with Intel(R) Xeon(R) CPU E5-2665 @ 2.40GHz, 16 cores, 32G RAM](https://example.com/graph.png)
Performances
getpid reloaded (don’t try this at home... yet!)

Intel(R) Xeon(R) CPU E5-2665 @ 2.40GHz, 16 cores, 32G RAM

Mop/sec (throughput)

- JNI
- Panama
- Panama (linkToNative - EXPERIMENTAL)
Performances

qsort

Intel(R) Xeon(R) CPU E5-2665 @ 2.40GHz, 16 cores, 32G RAM
Performances

qsort reloaded (upcalls are still expensive)

Intel(R) Xeon(R) CPU E5-2665 @ 2.40GHz, 16 cores, 32G RAM
Panama

Scorecard

• Ease of use: from header files to native bundles with jextract
• Rich API provides seamless integration with native code
  – much of the JNI boilerplate can now be expressed in Java!
• A safer alternative to JNI
  – Scope API manages resource lifecycles (pointers, structs, callbacks, ...)
• Room for performance improvement is huge
  – Reduce latency of native calls, hoist native transitions out of loops, ...
• Not just for C!
Panama status

• Early access binaries (macOS/Linux/Windows x64)
  – https://jdk.java.net/panama/
• Many community-extracted bindings
  – Vulkan, FFTW, Wayland, Cuda, ...
• Community-led ARM port effort is in the works
• Extensive talks with Intel (Steve Dohrmann) to support NVM
Panama Roadmap
Version 2.0

• Step 1: Low-level, foreign data support
  – MemoryAddress, MemoryScope, Layout API, VarHandle changes

• Step 2: Low-level foreign function support
  – SystemABI, VM changes to support “native” method handles (aka LinkToNative)

• Step 3: High level C interop support
  – Pointer<X>, Array<X>, Struct<X>, binder, jextract tool