# Precise and Efficient Garbage Collection in VMKit with MMTk

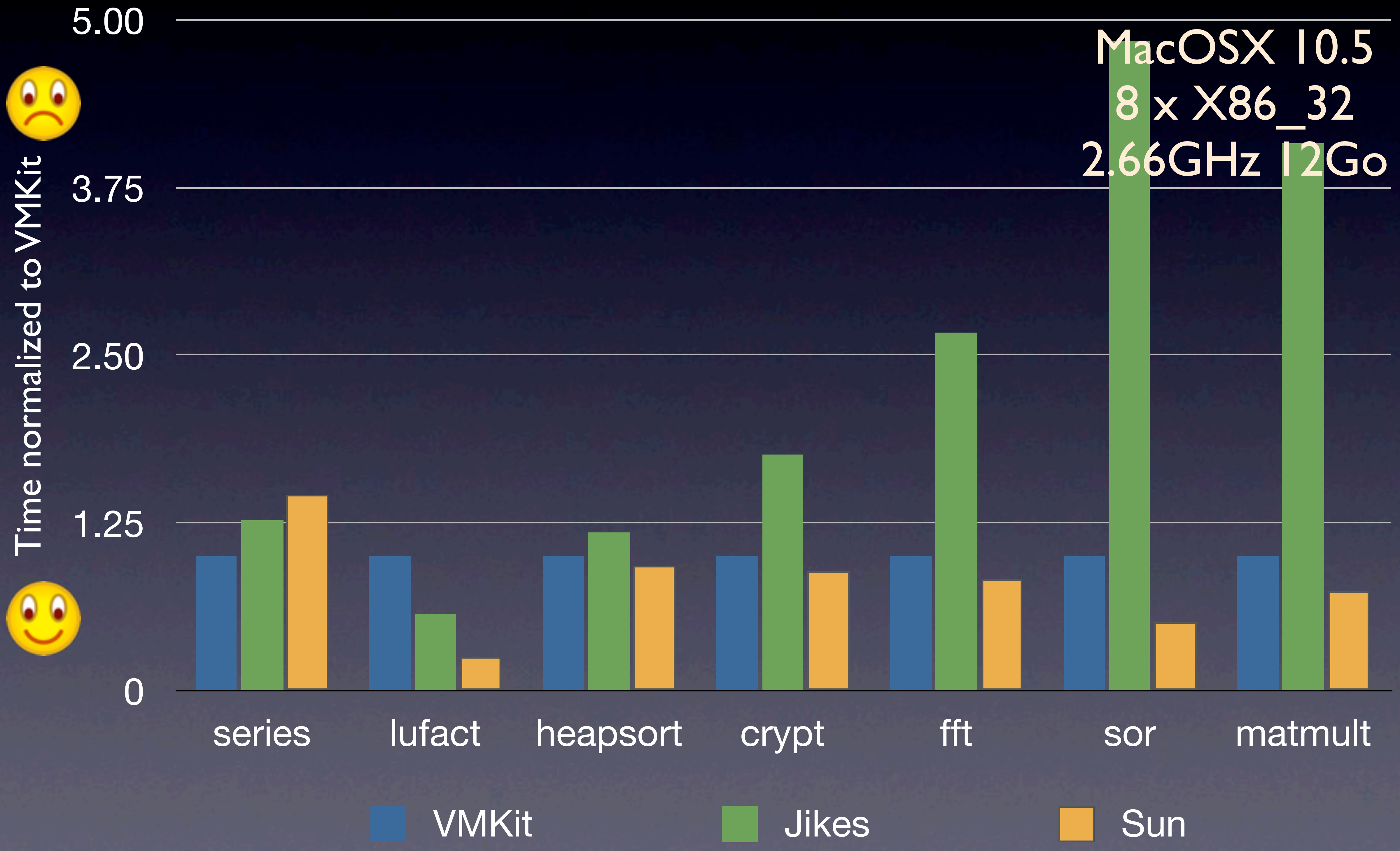
LLVM Developer's Meeting
Nicolas Geoffray

nicolas.geoffray@lip6.fr

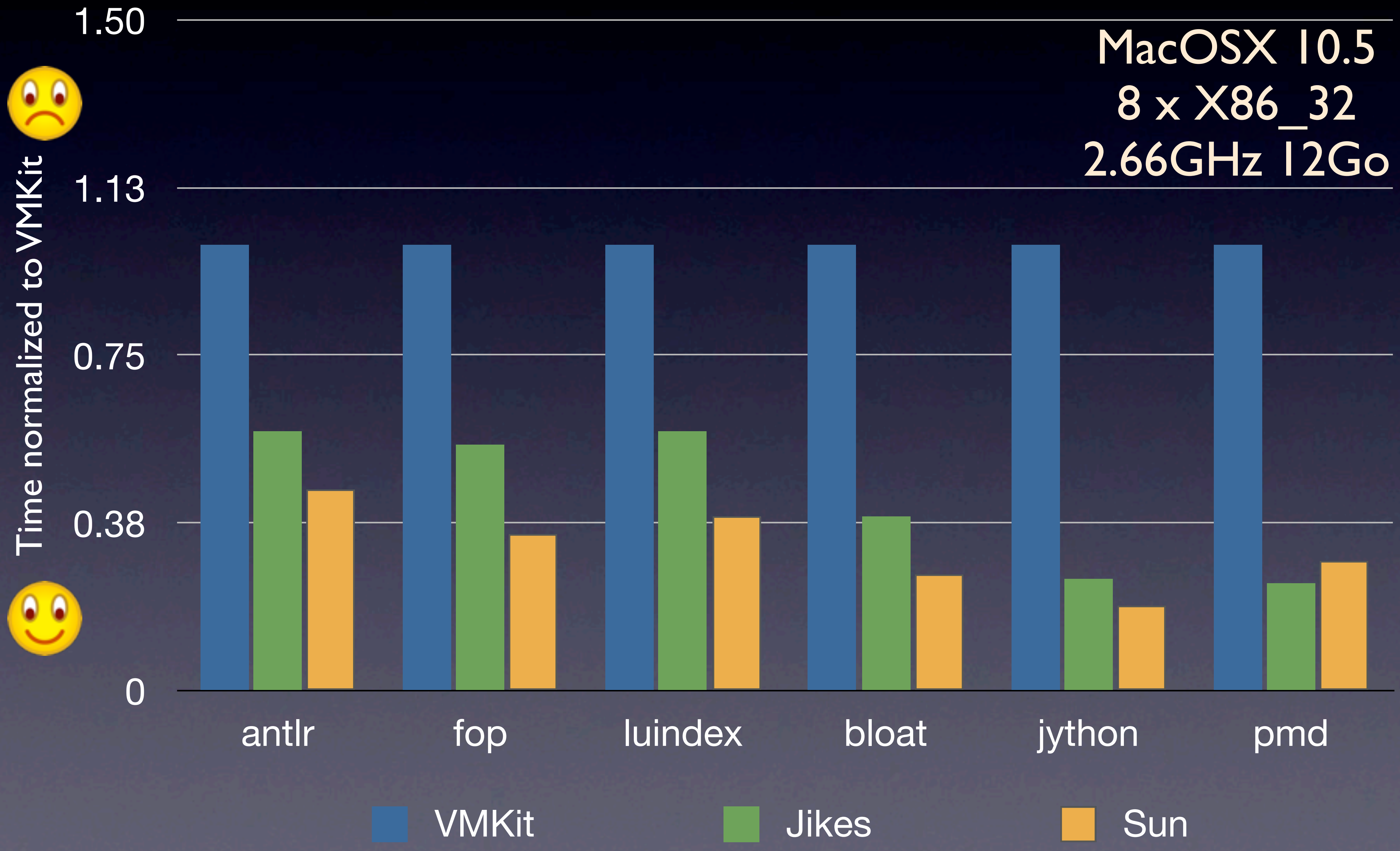
# Background

- VMKit: Java and .Net on top of LLVM
  - Uses LLVM's JIT for executing code
  - Uses Boehm for GC
- Performance bottlenecks
  - No dynamic optimization
  - Conservative GC

# CPU-intensive Benchmarks (JGF)

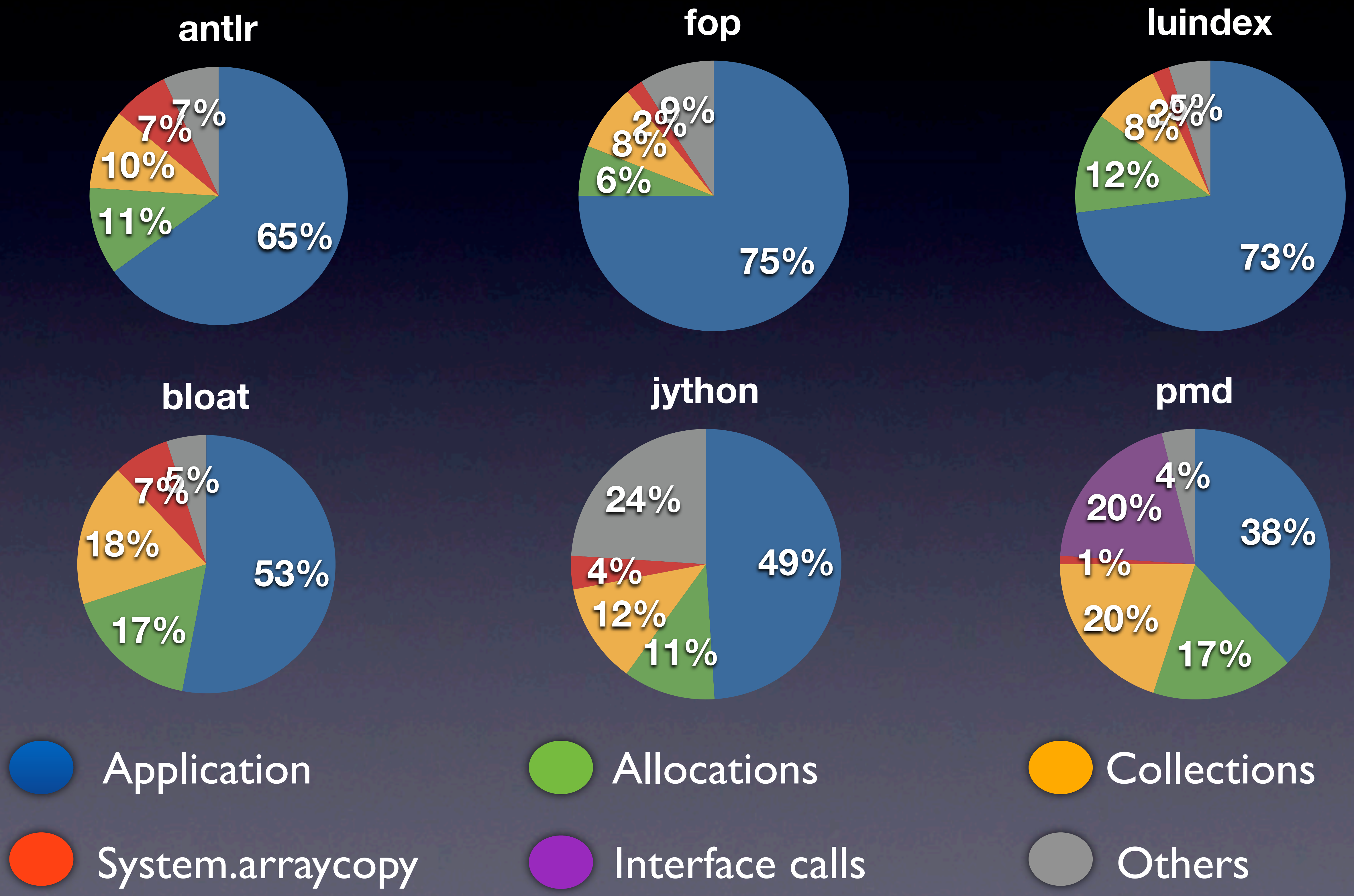


# VM-intensive Benchmarks (Dacapo)

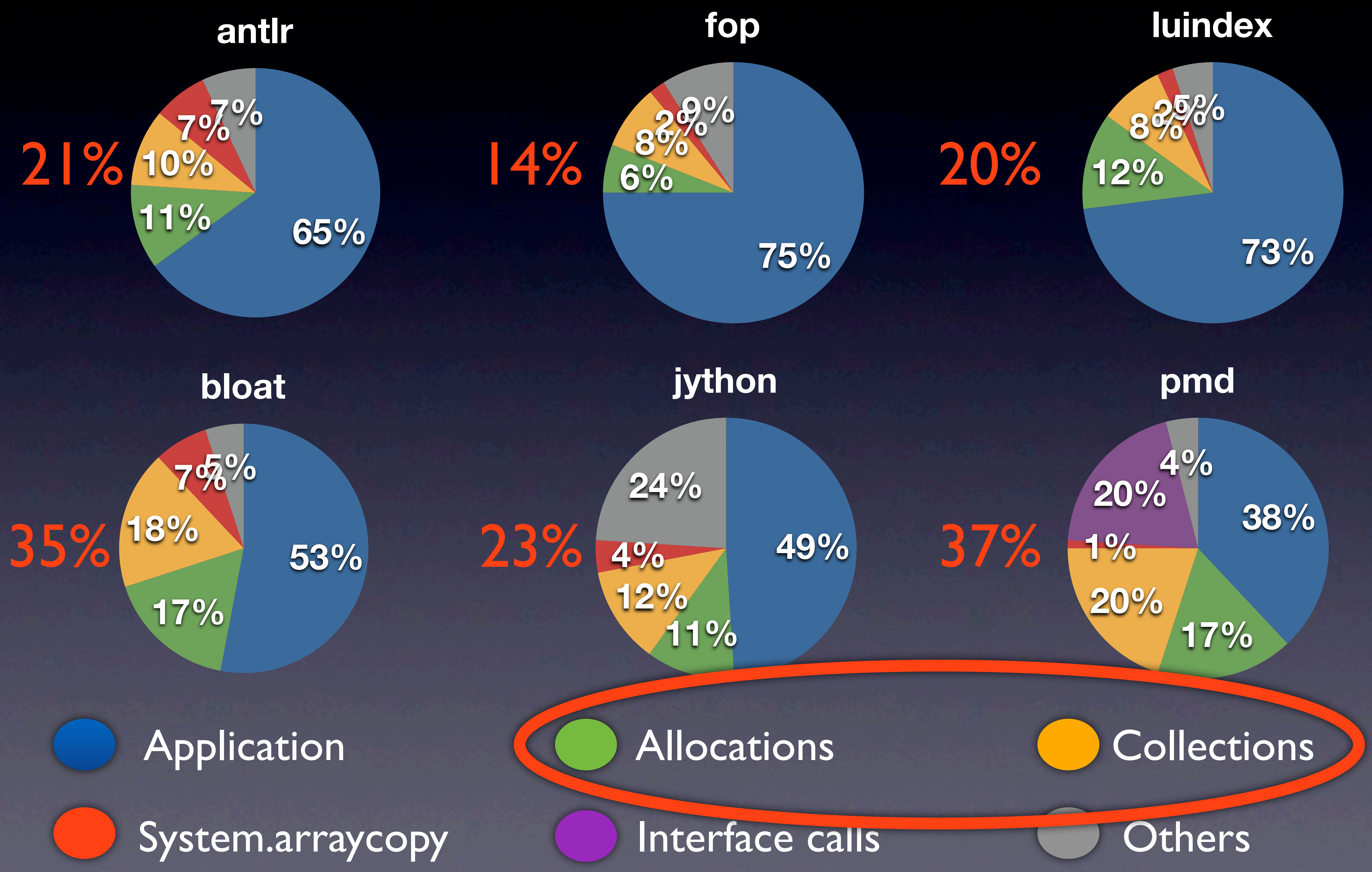


MacOSX 10.5
8 x X86_32
2.66GHz 12Go

# Execution Overheads



# Execution Overheads



# Goal: replace Boehm with MMTk

- MMTk is JikesRVM's GC
  - Framework for writing GCs
  - Multiple GC Implementations (Copying, Mark and trace, Immix)
- Copying collectors require precise stack scanning
  - Locate pointers on the stack

# But... it's in Java?

- Yes, but nothing to be afraid of:
  - Use of Magic tricks
  - No use of runtime features (exceptions, inheritance)
  - No use of standard library
- Use VMKit's AOT compiler
  - Transform MMTk into a .bc file

# Outline

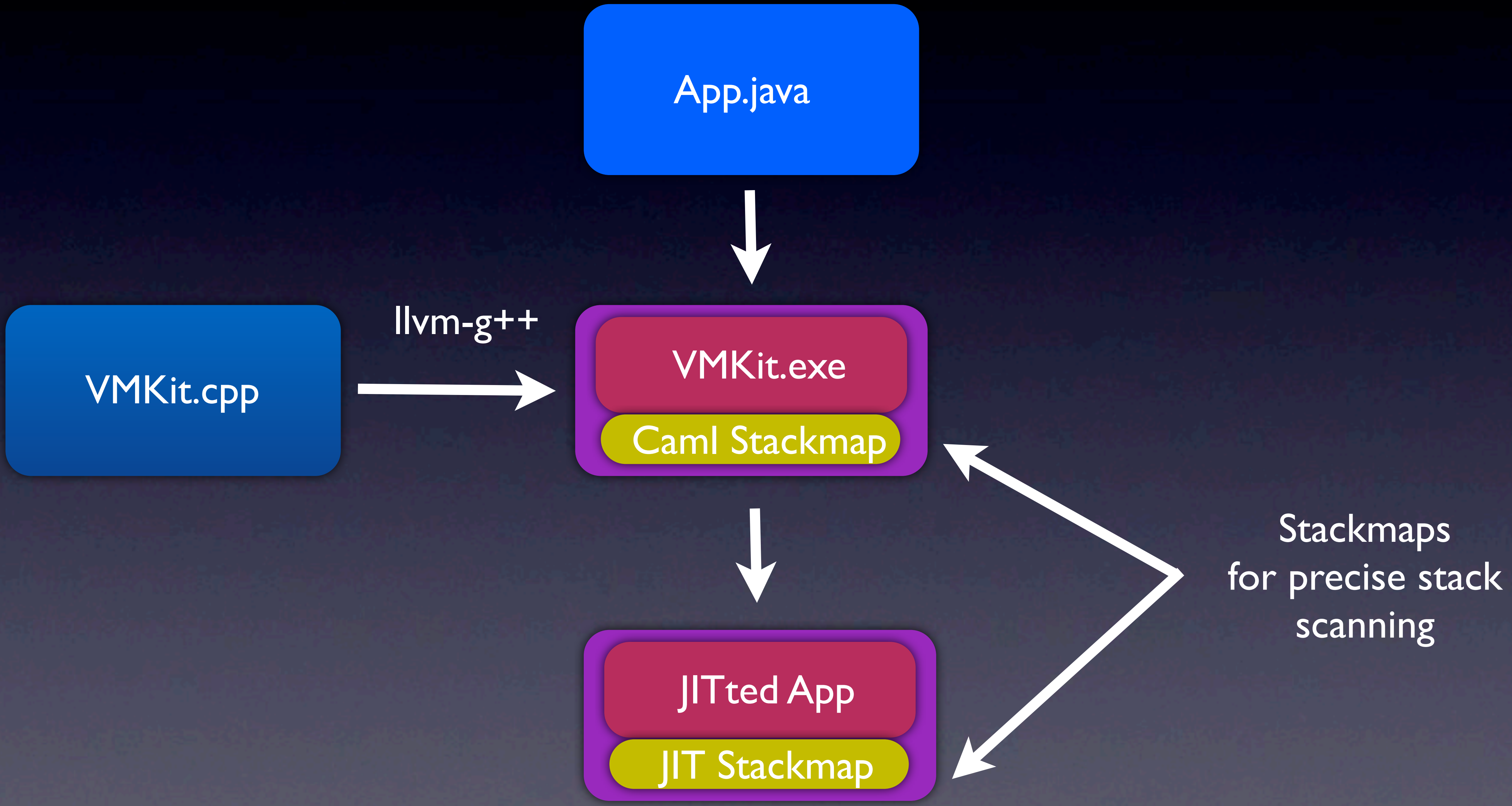
- Introduction
- Precise garbage collection
- Compiling MMTk with VMJC
- Putting it all together
- What's left

- Introduction
- Precise garbage collection
- Compiling MMTk with VMJC
- Putting it all together
- What's left

# Precise Garbage Collection

- Write code that locates pointers in the stack
  - llvm.gcroot in JIT-generated code
  - llvm.gcroot in VMKit's runtime written in C++
- Use LLVM's GC framework to generate stack maps
  - Caml stack maps for llvm-g++ generated code
  - JIT stack maps for JIT-generated code

# Precise Garbage Collection



Stackmaps
for precise stack
scanning

# Stack Scanning

- Problem: interweaving of different kinds of functions
  - Application's managed (Java or C#) functions: trusted
  - VMKit's C++ functions: trusted
  - Application's JNI functions: untrusted
- Solution: create a side-stack for frame addresses
  - Updated upon entry of a kind of method
  - VMKit knows the kind of each frame on the thread stack

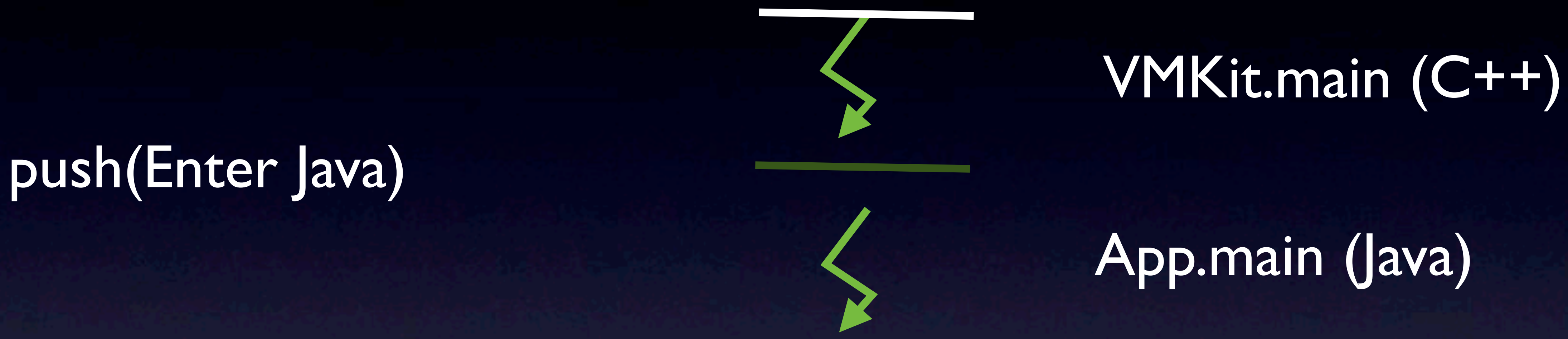
# Type of methods

- Trusted
  - Has a stack map, so can manipulate objects (llvm.gcroot)
  - Saves frame pointer (llvm::NoFramePointerElim)
- Untrusted
  - Has no stack map, so should not manipulate objects
  - May not save the frame pointer

# Stack Scanning Example (1)

VMKit.main (C++)

# Stack Scanning Example (1)

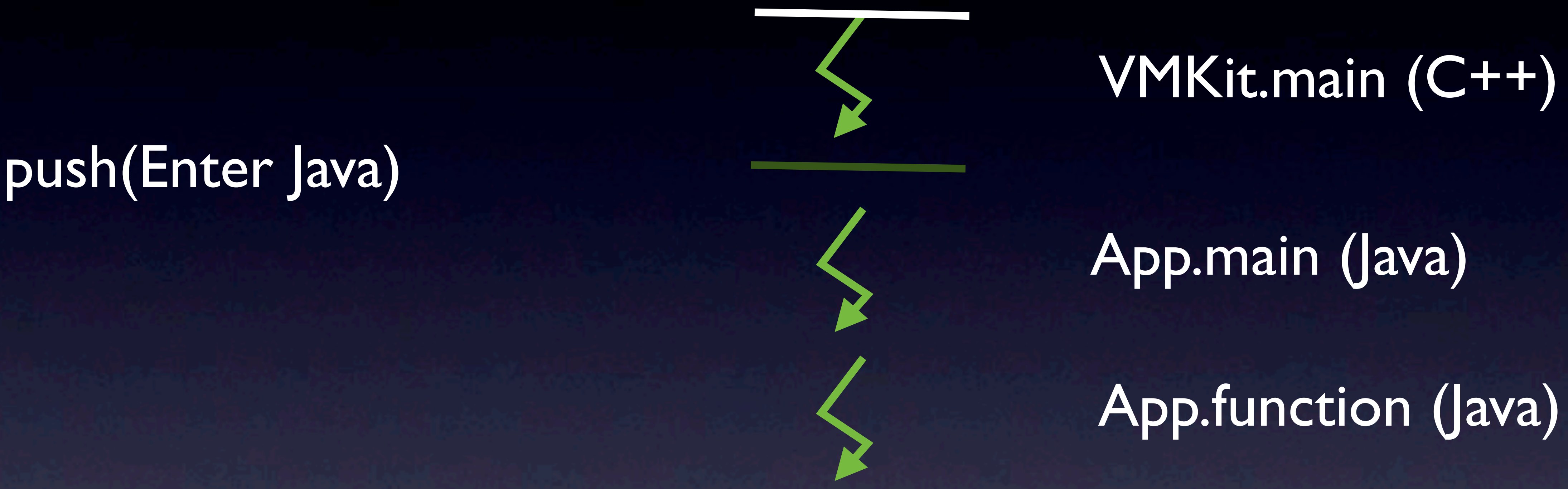


VMKit.main (C++)

push(Enter Java)

App.main (Java)

# Stack Scanning Example (1)



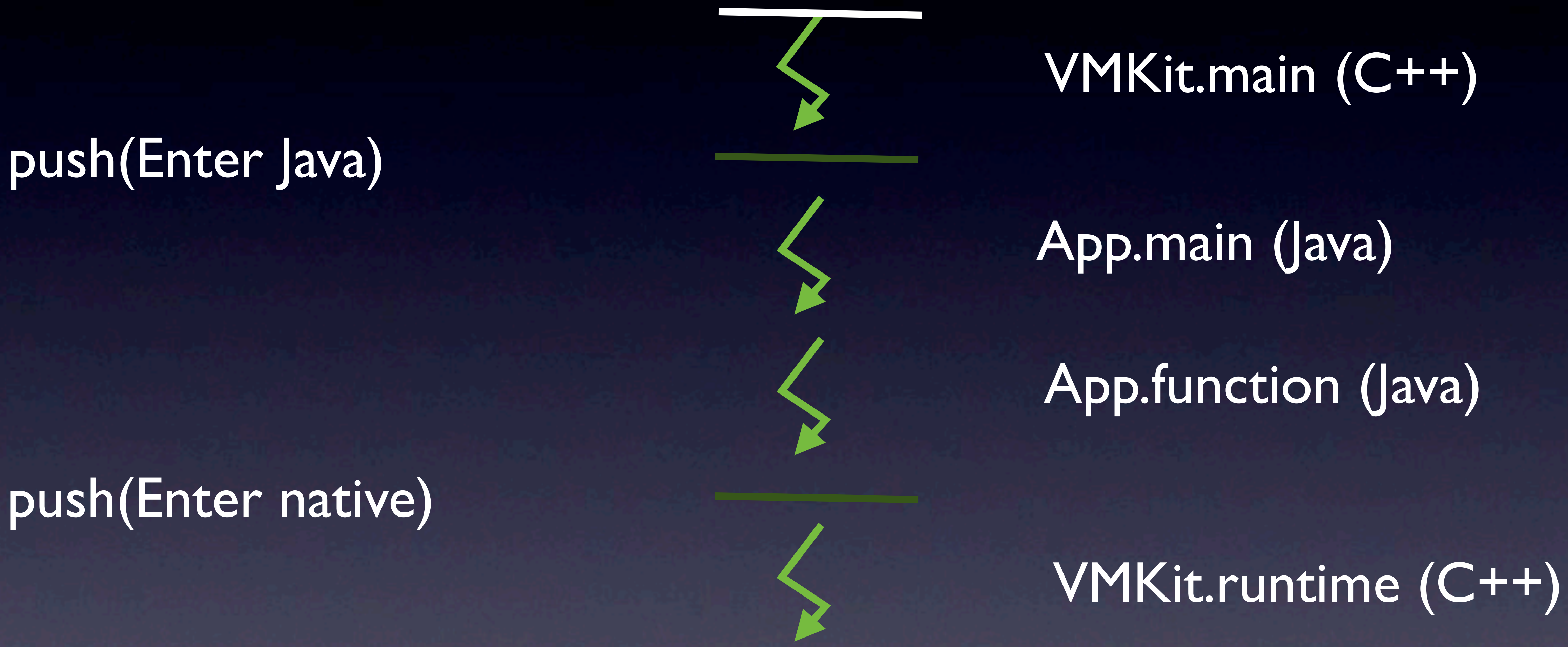
VMKit.main (C++)

push(Enter Java)

App.main (Java)

App.function (Java)

# Stack Scanning Example (1)



VMKit.main (C++)

push(Enter Java)

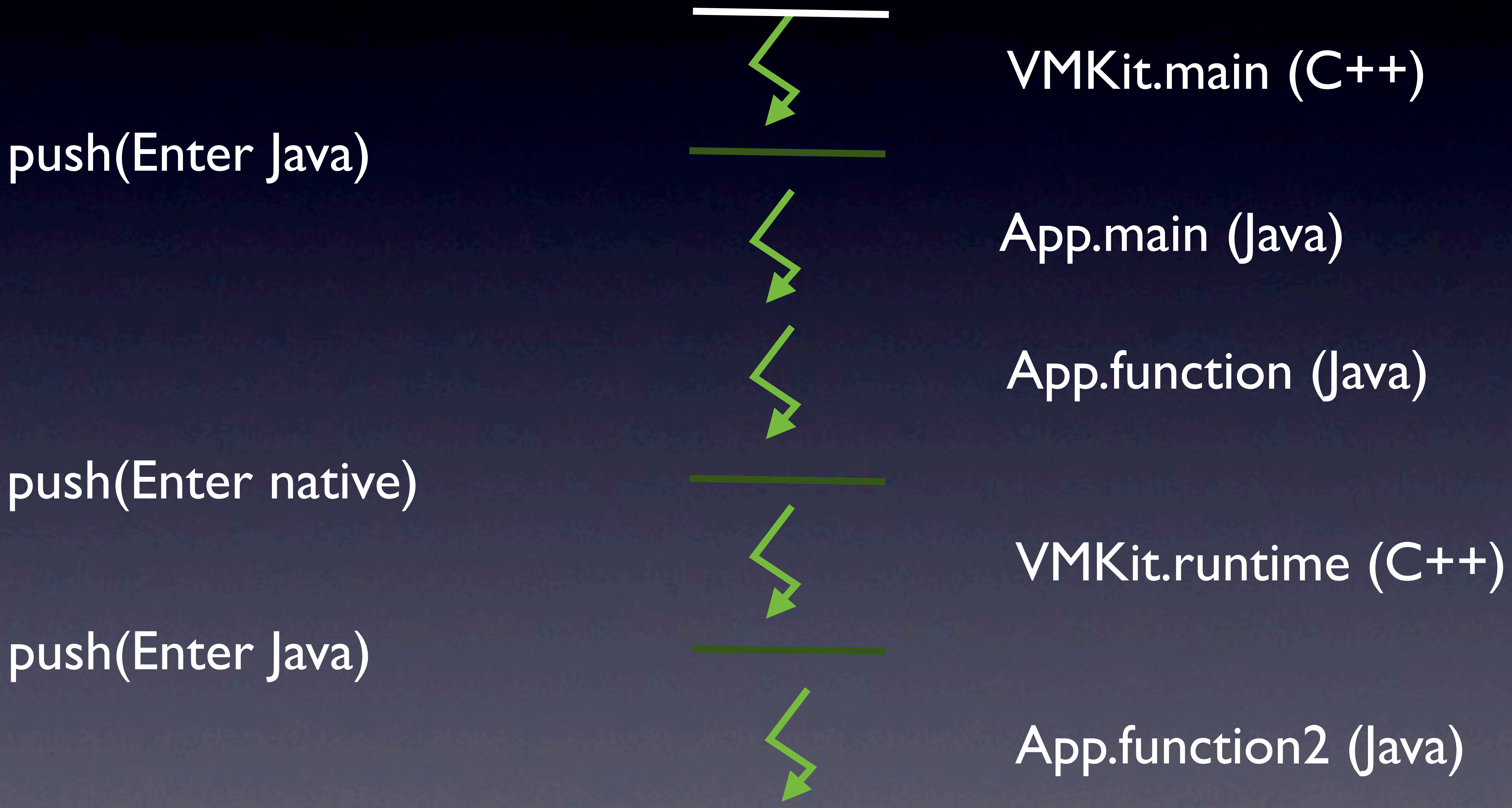
App.main (Java)

App.function (Java)

push(Enter native)

VMKit.runtime (C++)

# Stack Scanning Example (1)



VMKit.main (C++)

push(Enter Java)

App.main (Java)

App.function (Java)

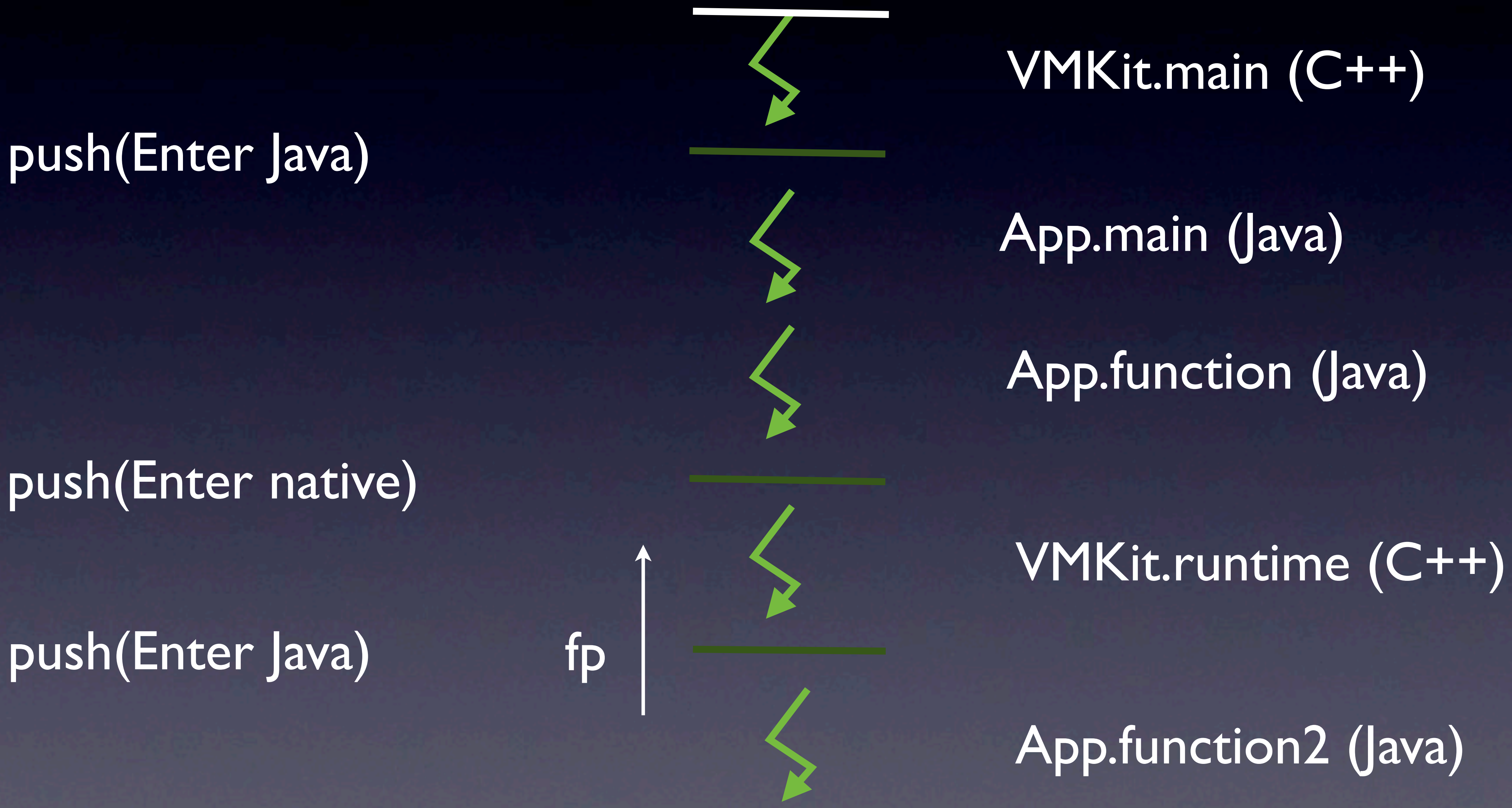
push(Enter native)

VMKit.runtime (C++)

push(Enter Java)

App.function2 (Java)

# Stack Scanning Example (1)



VMKit.main (C++)

push(Enter Java)

App.main (Java)

App.function (Java)

push(Enter native)

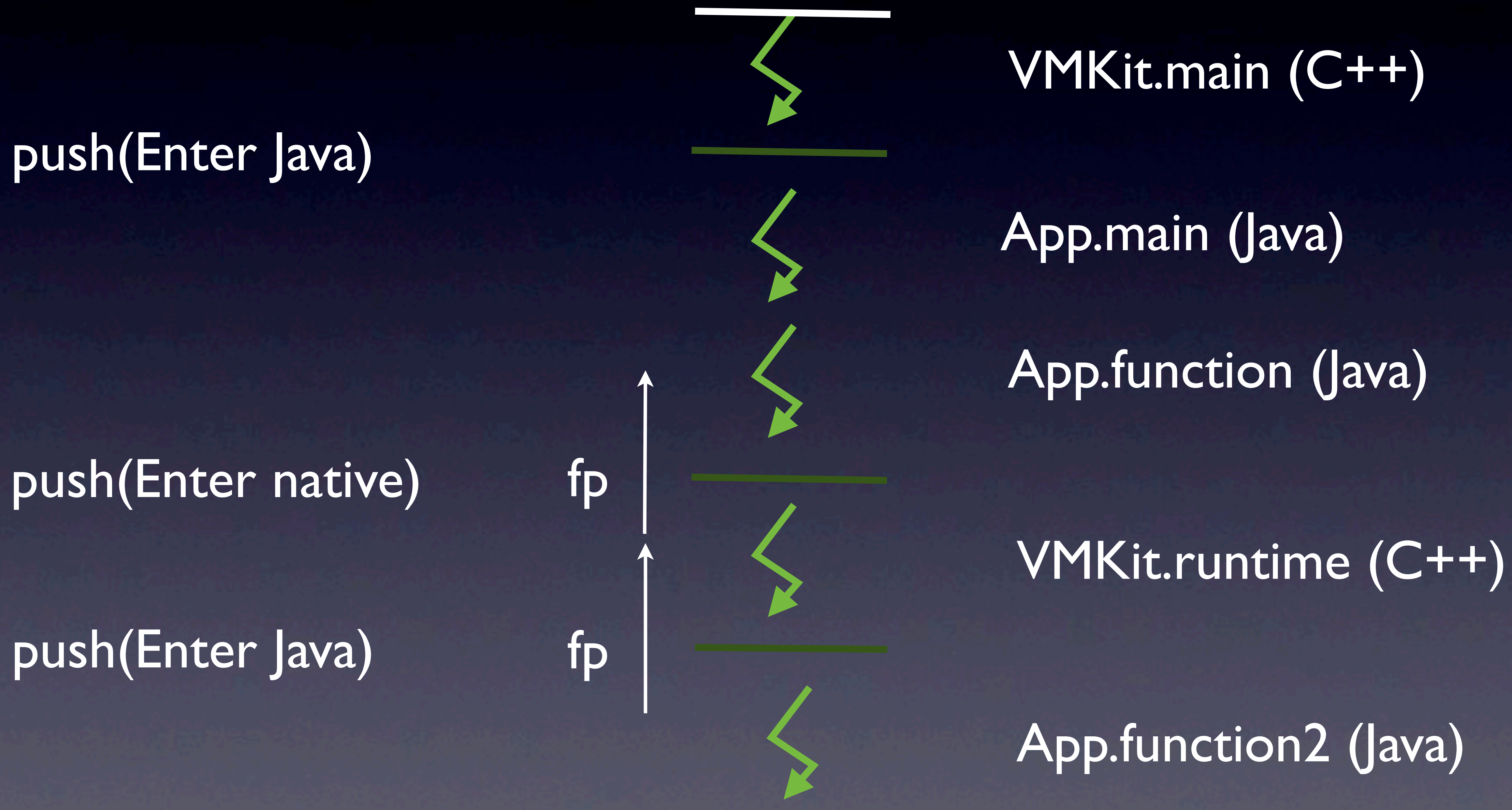
VMKit.runtime (C++)

push(Enter Java)

fp

App.function2 (Java)

# Stack Scanning Example (1)



VMKit.main (C++)

push(Enter Java)

App.main (Java)

App.function (Java)

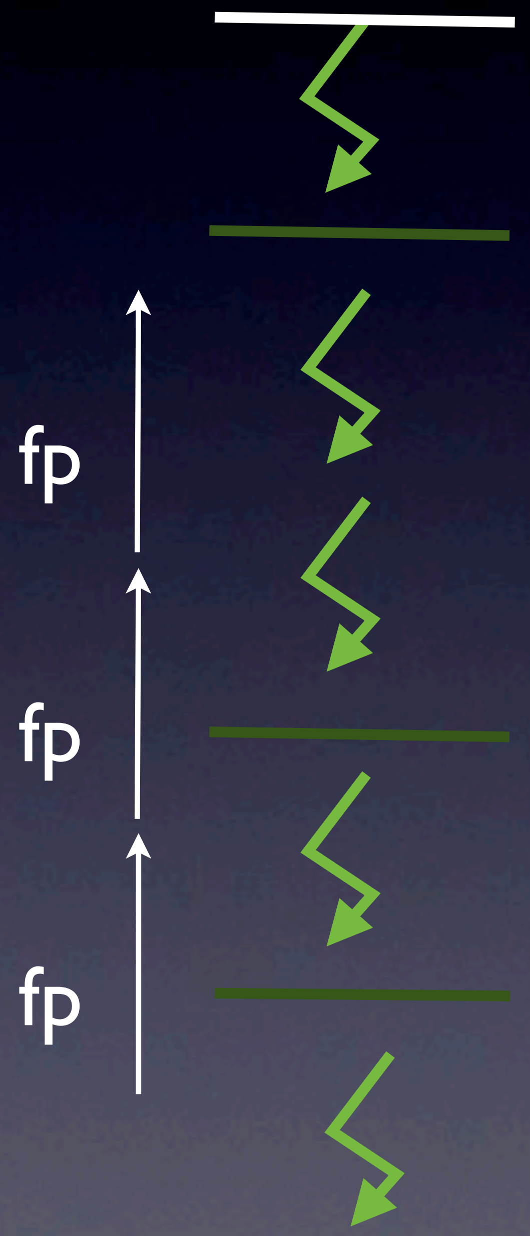
push(Enter native) fp

VMKit.runtime (C++)

push(Enter Java) fp

App.function2 (Java)

# Stack Scanning Example (1)



VMKit.main (C++)

push(Enter Java)

App.main (Java)

App.function (Java)

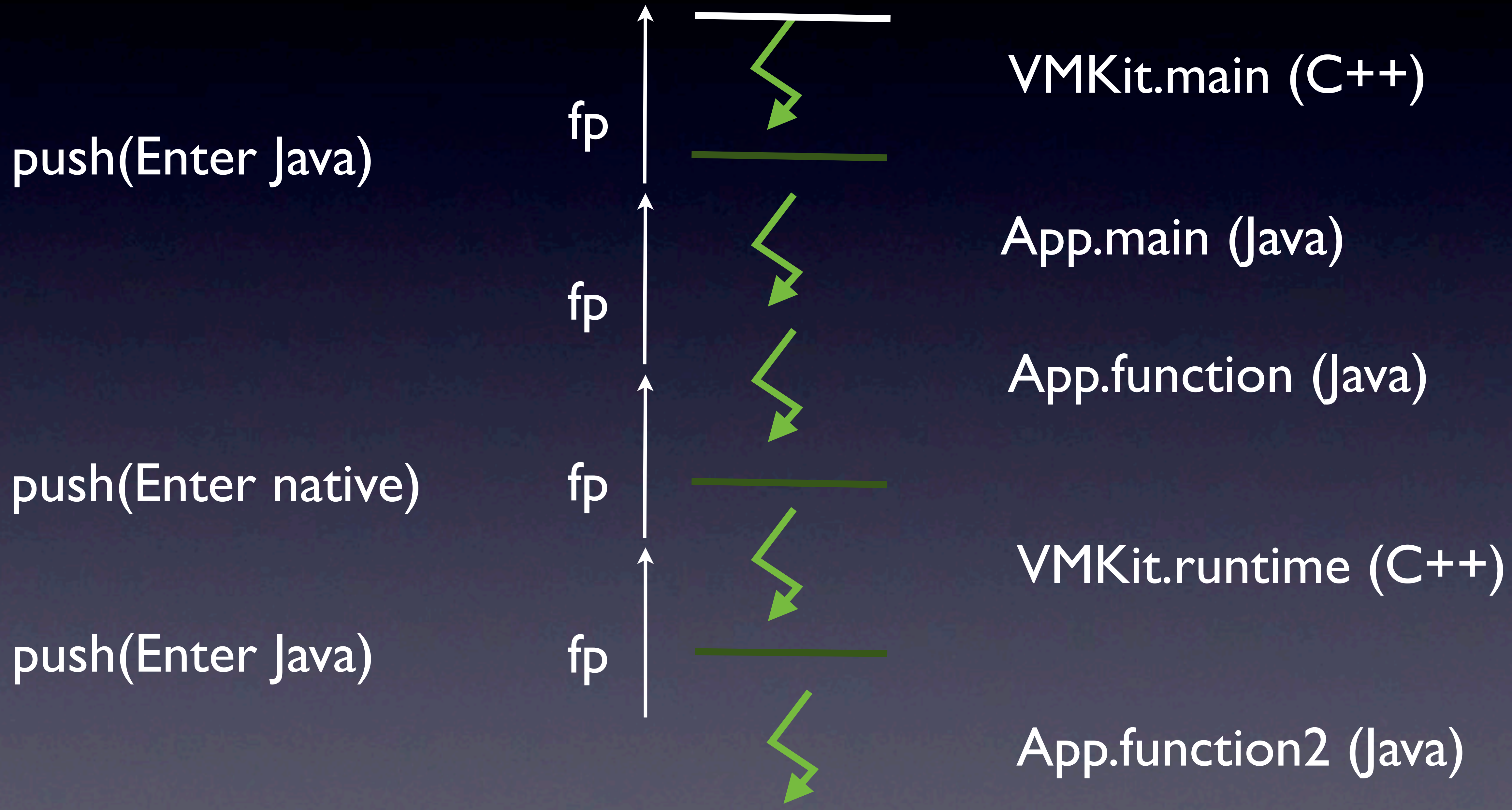
push(Enter native)

VMKit.runtime (C++)

push(Enter Java)

App.function2 (Java)

# Stack Scanning Example (1)



VMKit.main (C++)

push(Enter Java) fp

App.main (Java)

fp

App.function (Java)

push(Enter native) fp

VMKit.runtime (C++)

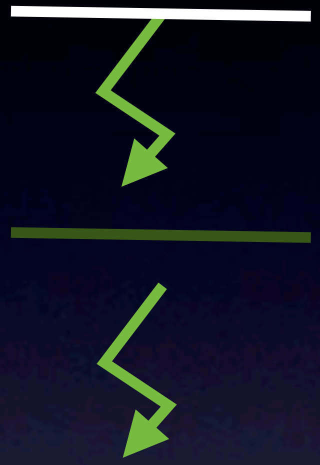
push(Enter Java) fp

App.function2 (Java)

# Stack Scanning Example (2)

VMKit.main (C++)

# Stack Scanning Example (2)

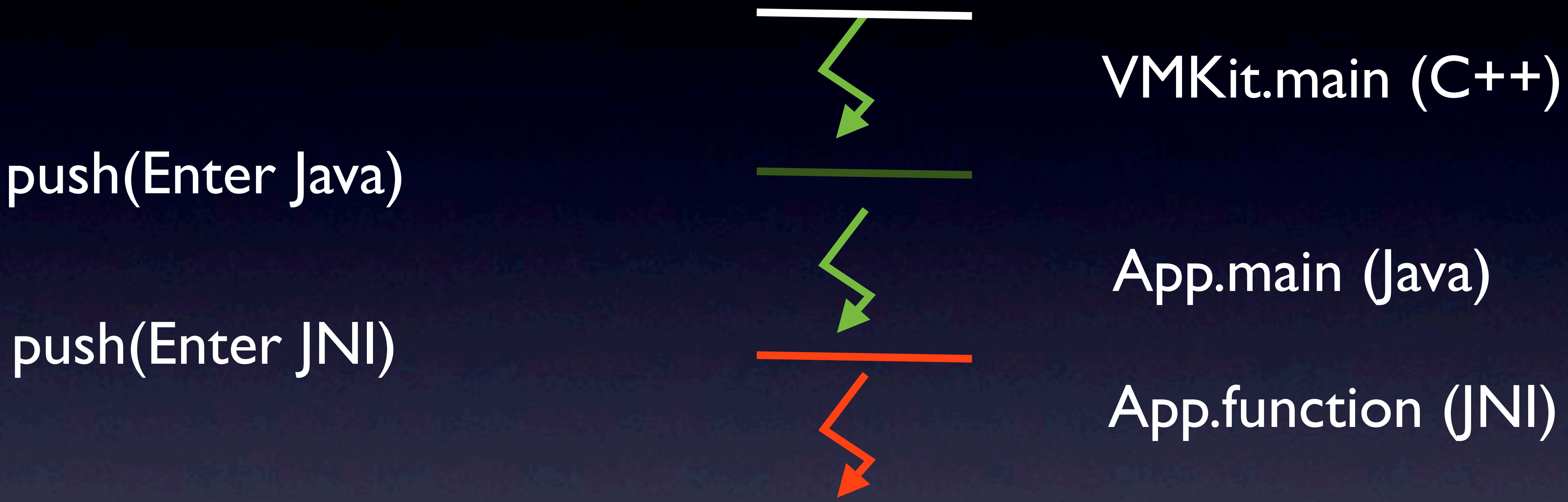


VMKit.main (C++)

push(Enter Java)

App.main (Java)

# Stack Scanning Example (2)



VMKit.main (C++)

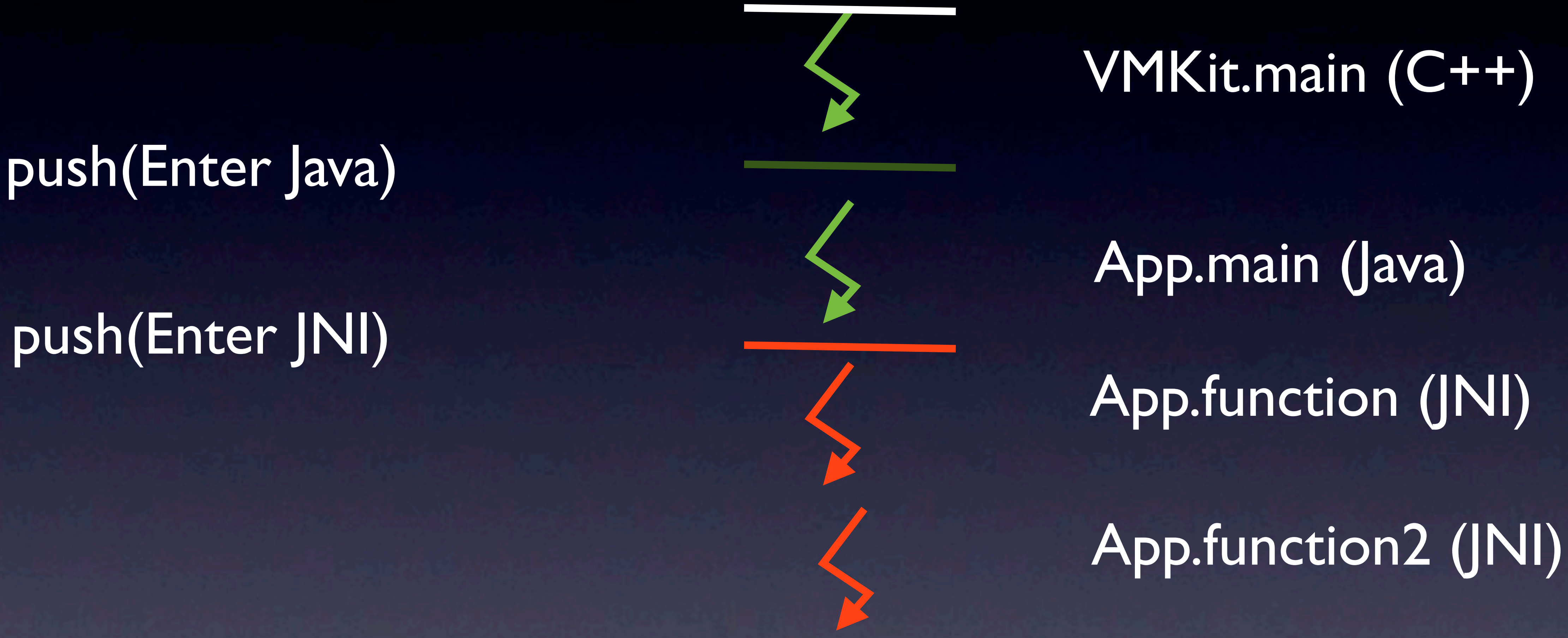
push(Enter Java)

App.main (Java)

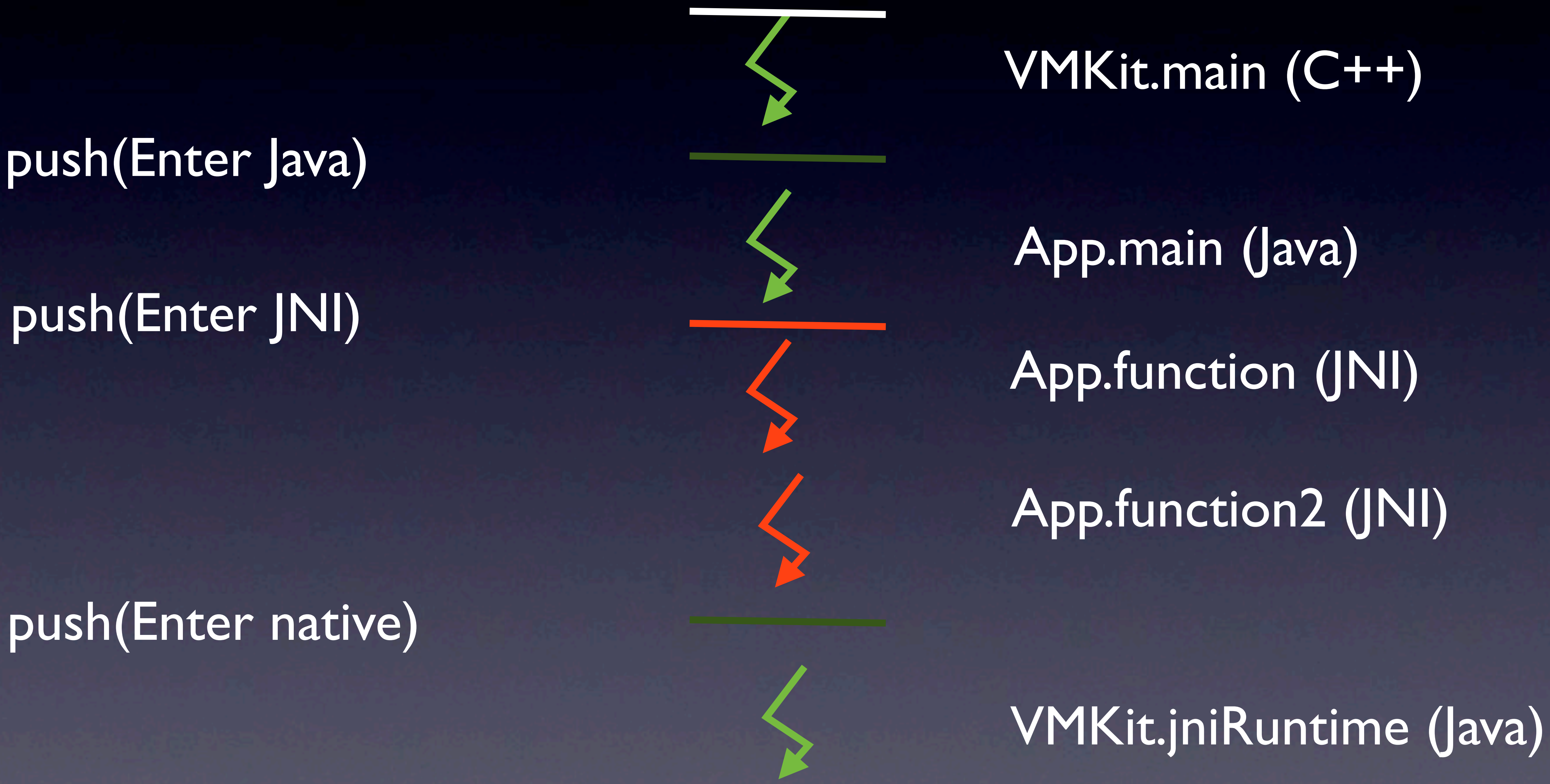
push(Enter JNI)

App.function (JNI)

# Stack Scanning Example (2)



# Stack Scanning Example (2)



VMKit.main (C++)

push(Enter Java)

App.main (Java)

push(Enter JNI)

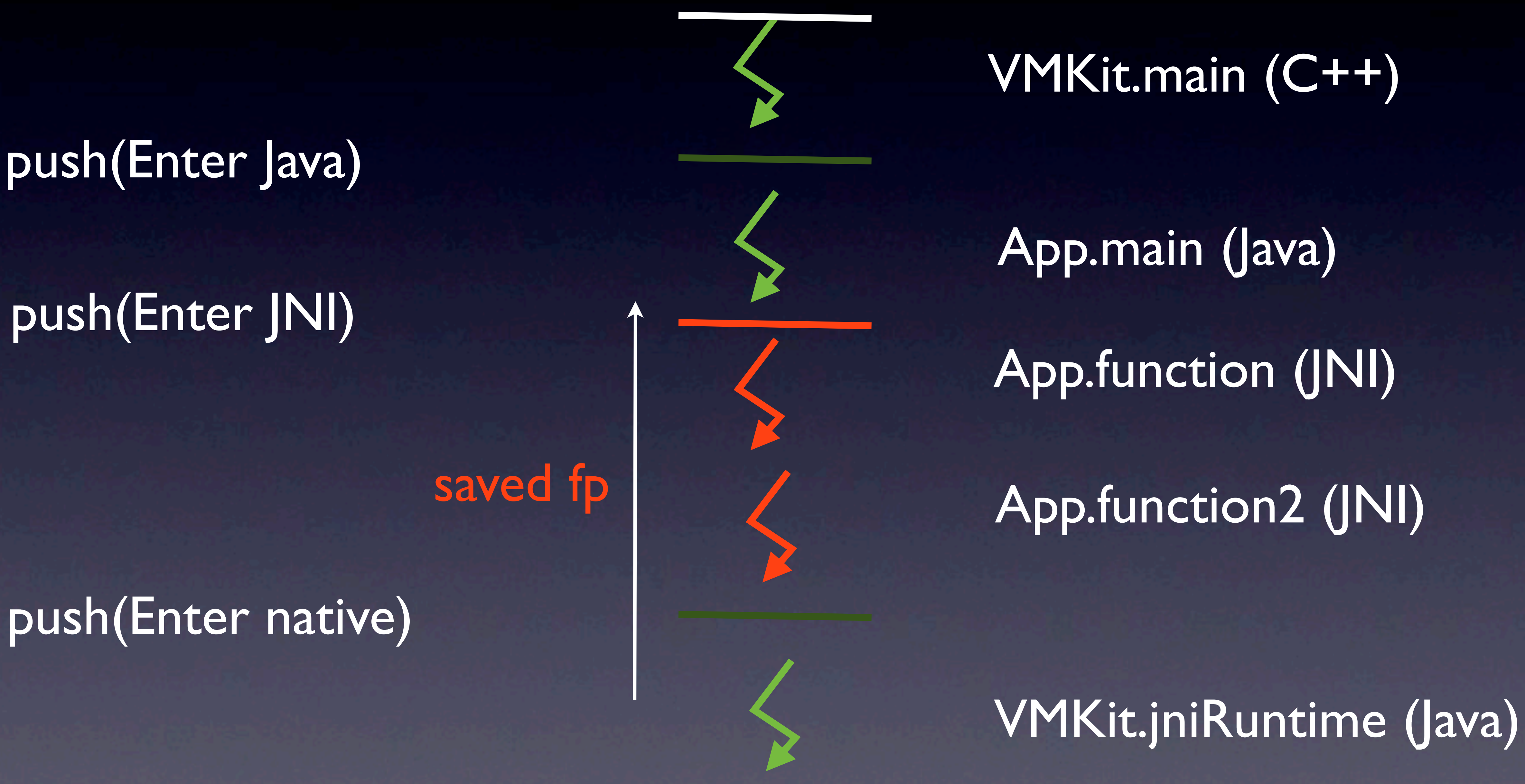
App.function (JNI)

App.function2 (JNI)

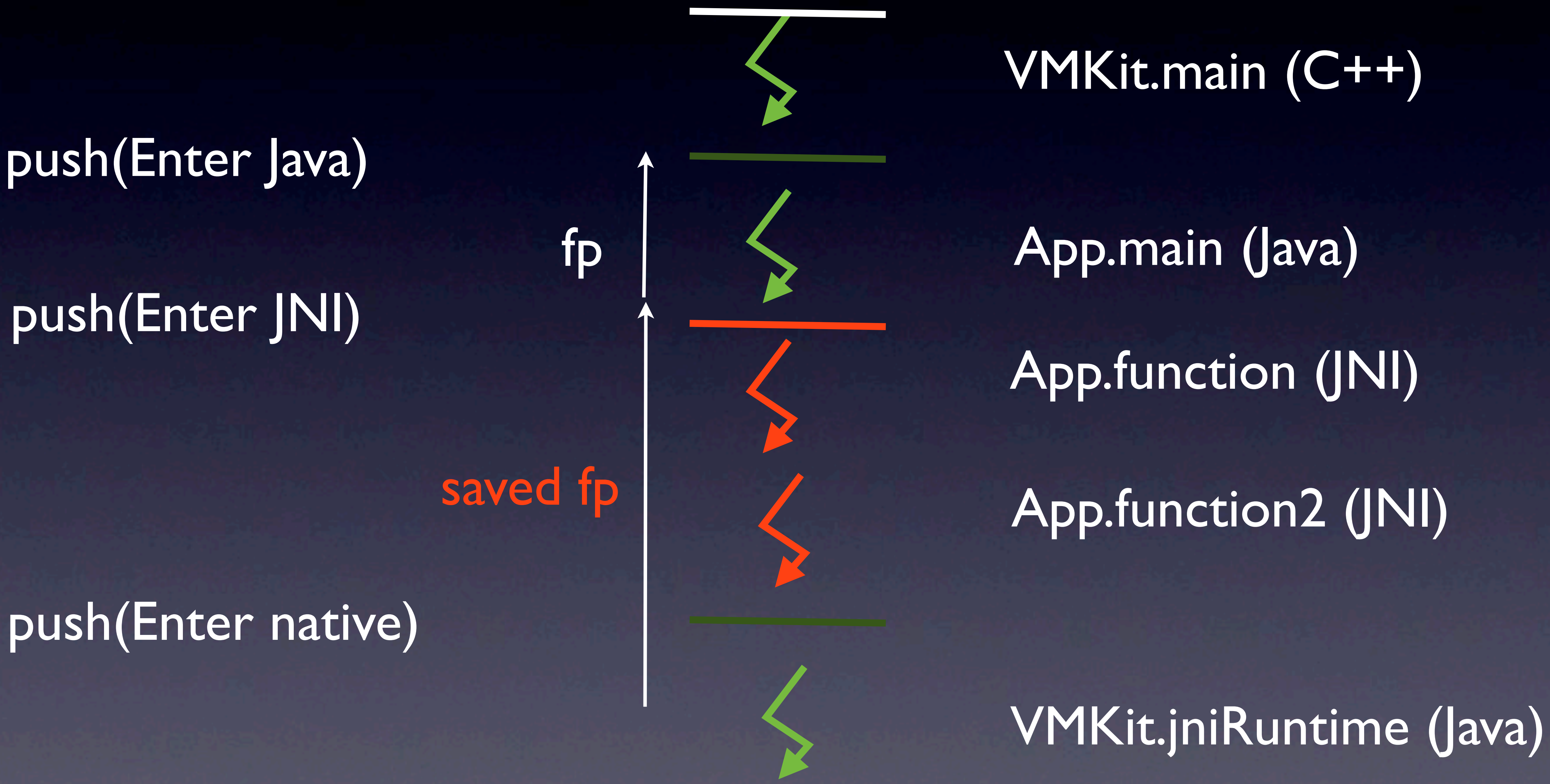
push(Enter native)

VMKit.jniRuntime (Java)

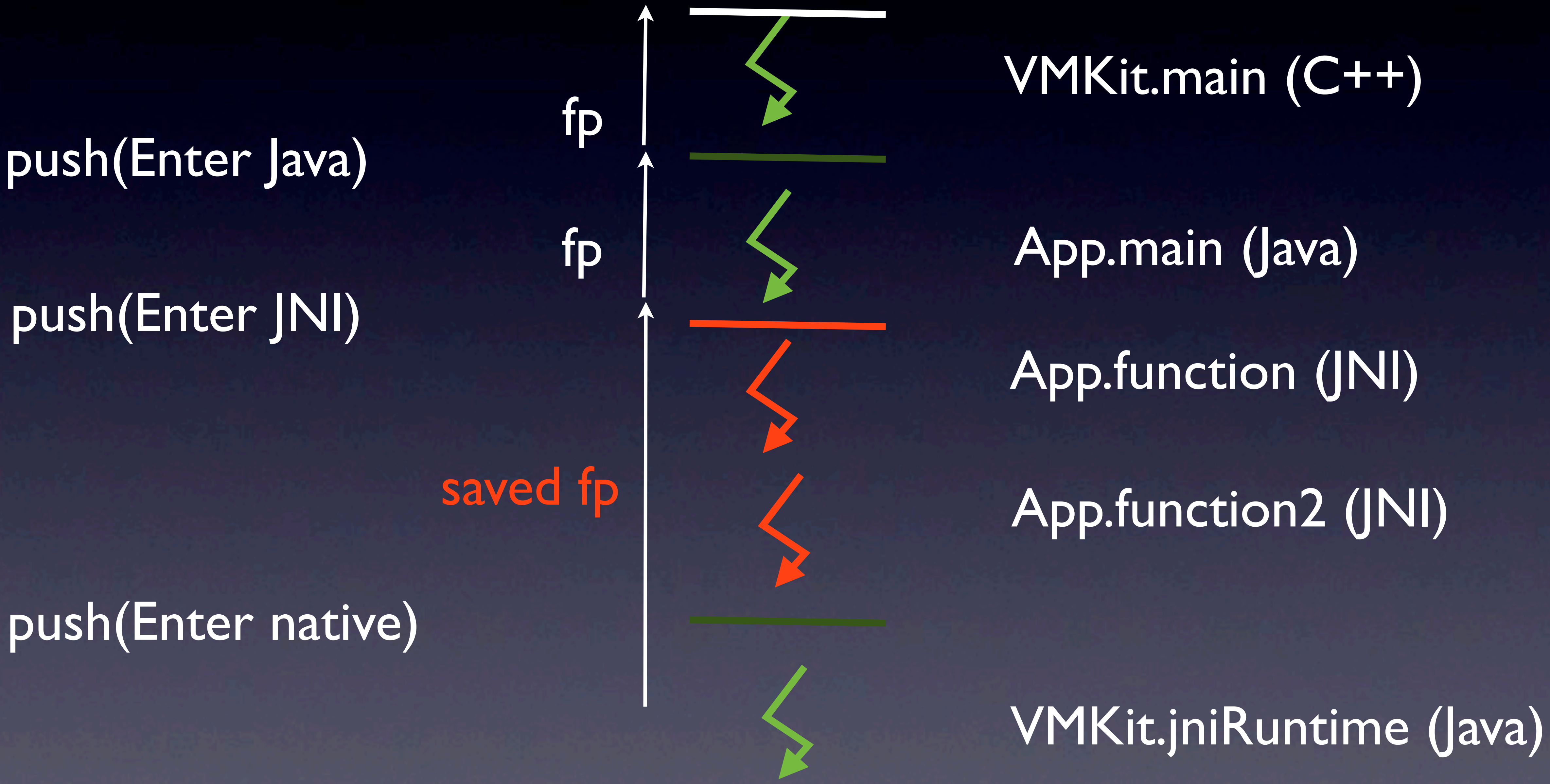
# Stack Scanning Example (2)



# Stack Scanning Example (2)



# Stack Scanning Example (2)



VMKit.main (C++)

push(Enter Java)

fp

fp

App.main (Java)

push(Enter JNI)

App.function (JNI)

saved fp

App.function2 (JNI)

push(Enter native)

VMKit.jniRuntime (Java)

# Running the GC

A precise GC scans the stacks at *safe points*: point during execution where the GC can know the type of each value on the stack

# Single-threaded Application

- GC always triggered at *safe points*
    - gcmalloc instrunctions
    - Collector::collect()

# Multi-threaded Application

- When entering a GC, must wait for all threads to join
  - Don’t use signals! or no safe point
  - Use a thread-local variable to poll on method entry and backward branches
  - Scan stacks of threads blocked in JNI or system calls

# Application changes for GC

```
public static void runLoop(int a) {
        while (a--) System.out.println(“Hello World”);
}
```

# Application changes for GC

```
public static void runLoop(int a) {
    if (getThreadID().doGC) GC()
    while (a--) {
        System.out.println(“Hello World”);
        if (getThreadID().doGC) GC()
    }
}
```

- Introduction
- Precise garbage collection
- Compiling MMTk with VMJC
- Putting it all together
- What's left

# What is VMJC?

- An Ahead of Time compiler (AOT)
    - Generates .bc files from .class files
- Use of llvm tools to generate platform-dependant files
    - shared library: llc -relocation-model=pic + gcc
    - executable: llc + ld vmkit + gcc

# Goal: compile MMTk with VMJC

- Generate a .bc file that can be linked with VMKit
  - Interface MMTK → VMKit (e.g. threads synchronization, stack scanning)
  - Interface VMKit → MMTk (e.g. gcmalloc)

# Why MMTk does not need a Java runtime?

- No use of runtime features
  - synchronizations, exceptions, inheritance
- No use of standard library
  - HashMap, LinkedList, ArrayList

# How MMTk is manipulating pointers?

- Definition of Magic classes and methods
  - Address, Word, Offset
  - Word Address.loadWord(Offset)
- Magic classes and methods translated by the compiler [VEE'09]
  - Similar mechanism than Inline ASM for C

# Example (Frampton [VEE’09])

## Inline ASM in C

```
void prefetchObjects(
  OOP *buffer,
  int size) {
  for(int i=0;i < size;i++){
    OOP o = buffer[i];
    asm volatile(
      "prefetchnta (%0)" ::
      "r" (o));
  }
}
```

## Magic in Java

```
@NoBoundsCheck
void prefetchObjects(
  ObjectReference[] buffer) {
  for(int i=0;i<buffer.length;i++) {
    ObjectReference current
      = buffer[i];
    current.prefetch();
  }
}
```

- Introduction
- Precise garbage collection
- Compiling MMTk with VMJC
- Putting it all together
- What’s left

# Option 1: Object File

- Create a .o file of MMTk
  - gcc mmtk.o vmkit.o -o vmkit
- But...
  - No inlining in application code

# Option 2: LLVM Bitcode File

- Create a .bc file of MMTk
  - vmkit (-load mmtk.bc) -java HelloWorld
- Late binding of allocations in VMKit code
  - gcmalloc in C++ are linked at runtime
- Inlining in Java code
  - new in applications are inlined with MMTk's malloc

# Option 3: Everything is Bitcode

- Create a .bc file of MMTk
- Create a .bc file of VMKit
- Link, optimize and run

- Introduction
- Precise garbage collection
- Compiling MMTk with VMJC
- Putting it all together
- What’s left

# What’s left

- Implementing the MMTK → VMKit interface
  - Interactions between the GC and the VM
- Finish implementation with read/write barriers
  - In VMKit code, in managed code
- Run benchmarks!
  - Benchmark with different GCs from MMTk

http://vmkit.llvm.org