Zero

Gary Benson

Senior Software Engineer

redhat

HotSpot

- HotSpot is the Java virtual machine of OpenJDK *et al*
- One of two interpreters selected at build time:
  - Template interpreter
  - C++ interpreter
- Optionally, one of two JITs selected at runtime:
  - Client (aka C1)
  - Server (aka C2)

redhat

Method dispatch

- Each method is represented by a `methodOop`
- Each method has an *entry point*:
  - Initialized to address of one of fourteen interpreter method entries
  - Replaced with address of compiled code by JIT
- To call a method:
  - Process the arguments as per the *calling convention*
  - Jump to the entry point
- To call a method from C++, use `StubRoutines::call_stub()`

redhat

Interpreter method entries

- In reality there are fourteen
- We only need to consider two:
  - `normal_entry`, the method entry for non-native methods
  - `native_entry`, the method entry for native methods
- Comparing the template and C++ interpreters:
  - The entry points for non-native methods are very different
  - All other entry points are substantially the same

redhat

A bare minimum port

- So what is the bare minimum you need for a port?
  - An assembler
  - The call stub
  - Two method entries
  - The signature handler generator
  - A stack walker

redhat

The normal entry point

- Set up the stack frame
- Loop:
  - Call `BytecodeInterpreter::run()`
  - Do what it asked
- Tear down the stack frame

redhat

The problem

- The Java stack is interleaved with the ABI stack
- We can't access the ABI stack in any meaningful way
- Interpreter frames need to be resizable
- Locking code relies on pointers being within the ABI stack

redhat

The solution

- `JavaStack` objects wrap a block of memory with stack-like accessors
- Each Java thread is initialized with an empty `JavaStack` object
- First call stub in a thread allocates half the ABI stack with `alloca()` and gives it to the thread's `JavaStack` to manage

redhat

The native entry point

- Set up the stack frame
- Call the native function:
  - Convert arguments from interpreter calling convention to native ABI
  - Jump to the function's start address
  - Convert result from native ABI to interpreter result convention
- Tear down the stack frame

redhat

libffi

- Two stage process:
  - Call `ffi_prep_cif()`
    - Takes a list of argument types and a result type
    - Returns an `ffi_cif` object
  - Call `ffi_call()`
    - Takes an `ffi_cif` object and a list of argument types
- The `ffi_cif` objects are reusable

redhat

Future work

- More platforms:
  - s390, s390x, ia64, arm
- Performance:
  - Profiling
  - Precompilation: GCJ
  - Some kind of JIT: LLVM, libjit

redhat

Questions

Gary Benson

gbenson@redhat.com

http://gbenson.livejournal.com/