Substrate VM
Safe Harbor Statement

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.
GraalVM Stack

Java

Scala

Truffle

HotSpot VM

Graal

JVMCI

JS

C/C++

Python

Ruby

R

InterOp

node

python

R

JS

Sulong

TruffleRuby

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.
GraalVM Stack
Substrate VM is ...

... an **embeddable** VM

with **fast startup** and **low footprint**

for, and written in, a **subset of Java**

optimized to **execute Truffle** languages

**ahead-of-time compiled** using Graal

integrating with **native development tools**.
Substrate VM: Execution Model

Points-To Analysis

- Truffle Language
- JDK
- Substrate VM

Reachable methods, fields, and classes

Ahead-of-Time Compilation

- Machine Code
- Initial Heap
- DWARF Info
- ELF / MachO Binary

All Java classes from Truffle language (or any application), JDK, and Substrate VM

Application running without dependency on JDK and without Java class loading
Features of Substrate VM

• Type safety and memory safety of Java
  – Type checks, array bounds checks, null pointer checks

• Garbage collection
  – All Java memory is managed automatically

• JDK support
  – Most core and utility classes

• C code integration
  – SystemJava: access C functions and C data structures without performance overhead

• Multithreading (optional feature)
  – Everything in java.util.concurrent package

• Native tool support for debugging, profiling, ...
  – Standard DWARF debug information for ahead-of-time compiled code and dynamically compiled code
"Hello World" in C, Java, Ruby

<table>
<thead>
<tr>
<th>Language</th>
<th>Virtual Machine</th>
<th>Instructions</th>
<th>Time</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>C hello</td>
<td></td>
<td>147,000</td>
<td>&lt; 1 ms</td>
<td>puts(&quot;Hello!&quot;);</td>
</tr>
<tr>
<td>Java</td>
<td>Java HotSpot VM 1.8.0_121-b13</td>
<td>140,000,000</td>
<td>50 ms</td>
<td>System.out.println(&quot;Hello!&quot;);</td>
</tr>
<tr>
<td>Java</td>
<td>GraalVM 0.21 Substrate VM</td>
<td>332,000</td>
<td>&lt; 1 ms</td>
<td>System.out.println(&quot;Hello!&quot;);</td>
</tr>
<tr>
<td>Ruby</td>
<td>MRI 2.3.3p222</td>
<td>125,000,000</td>
<td>35 ms</td>
<td>ruby -e 'puts &quot;Hello!&quot;'</td>
</tr>
<tr>
<td>Ruby</td>
<td>GraalVM 0.21 truffleruby</td>
<td>40,754,085,818</td>
<td>4955 ms</td>
<td>ruby -e 'puts &quot;Hello!&quot;'</td>
</tr>
<tr>
<td>Ruby</td>
<td>GraalVM 0.21 Substrate VM truffleruby image</td>
<td>345,160,332</td>
<td>90 ms</td>
<td>ruby -e 'puts &quot;Hello!&quot;'</td>
</tr>
</tbody>
</table>

Substrate VM gets Ruby startup close to MRI

Operating system: Linux x86_64
Instructions: perf stat -e instructions ...
Time: time (bash builtin) ...
SystemJava: Integrate Java and C Code

Implementation of `System.nanoTime()` using SystemJava:

```java
static long nanoTime() {
    timespec tp = StackValue.get(SizeOf.get(timespec.class));
    clock_gettime(CLOCK_MONOTONIC(), tp);
    return tp.tv_sec() * 1_000_000_000L + tp.tv_nsec();
}
```
Dependencies

• **Standard C library**
  – Allocate and free memory
  – Low-level file access
  – Optional: pthread

• **No dependency on**
  – Signal handling
  – Memory protection
  – Standard C++ library
  – Any Java virtual machine, Java bytecodes
  – Any native library of the JDK

• **Example: all dependencies of the JavaScript executable**
  – Memory mapping
    • mmap64, munmap
  – C Memory allocation
    • malloc, calloc, free
  – Core library
    • memmove, exit, gettimeofday, clock_gettime, strerror, __errno_location, getuid, getpwnam, sysconf
  – File access
    • open, close, read, write, pread64, opendir, closedir, readdir64_r, lseek, realpath, getcwd, dup2, fsync, ioctl, fcntl, unlink, __xstat64, __fxstat64
  – Thread
    • pthread_create, pthread_join, sched_yield, pthread_key_create, pthread_mutex_init, pthread_cond_init, pthread_mutex_lock, pthread_mutex_unlock, pthread_cond_wait, pthread_cond_timedwait, pthread_cond_signal, pthread_cond_broadcast, pthread_getspecific, pthread_setspecific, pthread_cancel, pthread_testcancel, pthread_attr_init, pthread_attr_setguardsize, pthread_attr_setstacksize, pthread_attr_destroy
Try It Yourself:

Integrated Cloud
Applications & Platform Services