Cross-ISA Testing of the Pharo VM

Lessons learned while porting to ARMv8 64bits

Tool Paper — MPLR’21

Guille Polito, Stéphane Ducasse, Pablo Tesone, Théo Rogliano, Pierre Misse-Chanabier, Carolina Hernandez, Luc Fabresse

RMoD Team — Inria Lille Nord Europe — UMR9189 CRIStAL — CNRS
Context
The Pharo VM
Some Numbers

- 255 bytecodes (77 different) + ~340 primitives/native methods
- 146 different IR instructions
- polymorphic inline caches
- threaded code interpreter
- generational scavenger GC

Lots of combinations!
Objective: Implementing an ARM64 Backend

- ARM64 is now pervasive:
  - New Apple M1
  - Raspberry Pi 4
  - Microsoft Surface Pro X
  - PineBook Pro
  - ...

```
move r1 #1
move r2 #17
checkSmallInt r1
checkSmallInt r2
add r3 r1 r2
checkSmallInt r3
move r1 r3
ret
```

JIT compiler IR
Targeting Real Hardware
Challenges

• How to do a partial implementation, in an iterative way?
• Hardware availability: did not have access to an Apple M1 yet
• Slow Change-Compile-Test cycle
• Bug reproduction is a demanding task
### Execution Mode Comparison

<table>
<thead>
<tr>
<th></th>
<th>Real Hardware Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Feedback-cycle speed</strong></td>
<td>Very low</td>
</tr>
<tr>
<td><strong>Availability</strong></td>
<td>Low</td>
</tr>
<tr>
<td><strong>Reproducibility</strong></td>
<td>Low</td>
</tr>
<tr>
<td><strong>Precision</strong></td>
<td>High</td>
</tr>
<tr>
<td><strong>Debuggability</strong></td>
<td>Low</td>
</tr>
</tbody>
</table>
Simulation Environment

Simulation Environment (Pharo)

VM
- Interpreter
- GC
- JIT Compiler

Heap

Native Code Cache

Production VM (C)

Transpiled to

Unicorn

LLVM Disassembler

Two decades of smalltalk vm development: live vm development through simulation tools.

VMIL’18
# Simulation Environment Comparison

<table>
<thead>
<tr>
<th></th>
<th>Real Hardware Execution</th>
<th>Full-System Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feedback-cycle speed</td>
<td>Very low</td>
<td>Low</td>
</tr>
<tr>
<td>Availability</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Reproducibility</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Precision</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Debuggability</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>
Unit Testing Infrastructure
Extending the simulation environment

Simulation Environment (Pharo)

- VM
  - Interpreter
  - GC
  - JIT Compiler
- Heap
- Native Code Cache

Transpiled to

Production VM (C)

Unicorn

LLVM Disassembler
Our testing infrastructure by example

`testPushConstantZeroBytecodePushesASmallIntegerZero`

```smalltalk
self compile: [ compiler genPushConstantZeroBytecode ].
self runGeneratedCode.

self assert: self popAddress equals: (memory integerObjectOf: 0)
```
Our testing infrastructure by example

Reusable test fixtures covering e.g.,
- trampoline and stub compilation
- heap initialization

testPushConstantZeroBytecodePushesASmallIntegerZero

```smalltalk
self compile: [ compiler genPushConstantZeroBytecode ].
self runGeneratedCode.

self assert: self popAddress equals: (memory integerObjectOf: 0)
```
Our testing infrastructure by example

Reusable test fixtures covering e.g.,
- trampoline and stub compilation
- heap initialization

Compiler internal DSL

testPushConstantZeroBytecodePushesASmallIntegerZero

self compile: [ compiler genPushConstantZeroBytecode ].
self runGeneratedCode.

self assert: self popAddress equals: (memory integerObjectOf: 0)
Our testing infrastructure by example

Reusable test fixtures covering e.g.,
- trampoline and stub compilation
- heap initialization

```
testPushConstantZeroBytecodePushesASmallIntegerZero

self compile: [ compiler genPushConstantZeroBytecode ].
self runGeneratedCode.

self assert: self popAddress equals: (memory integerObjectOf: 0).
```

Compiler internal DSL

JIT Execution helpers such as e.g.,
- run all code between two addresses
- run until the PC hits an address
VM Unit Testing Lessons

Insights: Black box testing

testPushConstantZeroBytecodePushesASmallIntegerZero

self compile: [ compiler genPushConstantZeroBytecode ].
self runGeneratedCode.

self assert: self popAddress equals: (memory integerObjectOf: 0).

- Depend only on observable behaviour
- Reusable on different backends / architectures
- Resistant to changes in the implementation
VM Unit Testing Lessons
Insights: Cross-compile / Cross-execute

testPushConstantZeroBytecodePushesASmallIntegerZero

```smalltalk
self compile: [ compiler genPushConstantZeroBytecode ].
self runGeneratedCode.

self assert: self popAddress equals: (memory integerObjectOf: 0)
```

Hardware independent

Parametrizable tests
VM Unit Testing Lessons
Insights: Start Small

testPushConstantZeroBytecodePushesASmallIntegerZero

self compile: [ compiler genPushConstantZeroBytecode ].
self runGeneratedCode.

self assert: self popAddress equals: (memory integerObjectOf: 0)

First: The simplest test, the simplest feature
Second: the next simplest test
Focus on enhancing the testing infrastructure
# Unit Testing Infrastructure Comparison

<table>
<thead>
<tr>
<th></th>
<th>Real Hardware Execution</th>
<th>Full-System Simulation</th>
<th>Unit-Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feedback-cycle speed</td>
<td>Very low</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Availability</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Reproducibility</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Precision</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Debuggability</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>
There is no silver bullet

• Simulators are cheap, but not 100% trustworthy
• Full execution (simulated or on real HW)
  • more expensive to run
  • cannot unit-test it (less controllable)
• Unit tests only exercise specific scenarios
• Full executions exercise not yet covered scenarios
Our testing Workflow

- Simulate the execution, less than you run tests
- Run the real app, less than you simulate

- Go back and forth:
  - Turn full execution failures into tests
  - **Fix with the aid of the test:**
    - => unit test are faster to run
    - => easier to debug
    - => detect regressions
Case Study 1
Porting the Cogit JIT Compiler to ARM64

• Started with no tests and no hardware (main target Apple M1)
• Incremental test development: bytecode, native methods, PICs, code patching
• All tests run from the beginning on our four targets: x86, x86-64, ARM32 and ARM64
• Test allowed safe modifications in the IR to support e.g., ARM64 Multiplication overflow
• ARM64 specific tests covered stack alignment, W+X …
Case Study 2
Ongoing Port to RISCV64

• Currently under development
• Is our harness test suite enough to develop a new backend?
• Are our tests general enough?

• Collaboration with Q. Ducasse, P. Cortret, L. Lagadec from ENSTA Bretagne
• Future work on: Hardware-based security enforcement
Case Study 3
Debugging and Testing Memory Corruptions

- Bug report using Ephemerons
  https://github.com/pharo-project/pharo/issues/8153

- Starting the other way around
  - First reproducing the bug in real-hardware
    => long to execute (even longer in simulation)
    => required manual developer intervention
  - Then building a unit test from observations
  - Test becomes a part of the regression test suite
Future Perspectives

Automatic VM Validation

• Automatic (Unit?) Test Case Generation
• Interpreter vs Compiler Differential Testing
• VM Tailored Multi-level Debugging
Cross-ISA Testing of the Pharo VM  Lessons learned while porting to ARMv8 64bits

<table>
<thead>
<tr>
<th></th>
<th>Real Hardware Execution</th>
<th>Full-System Simulation</th>
<th>Unit-Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feedback-cycle speed</td>
<td>Very low</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Availability</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Reproducibility</td>
<td>Low</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Precision</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Debuggability</td>
<td>Low</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

Lessons learned while porting to ARMv8 64bits

Test x1000  Beliefs Infrastructure
Simulate x100  Real Execution
Debugging a compiler
Insights: build your own tools, based on needs, not desires

Examples:
- Machine code debugger
- Bytecode-IR visualization
- Disassembler DSL