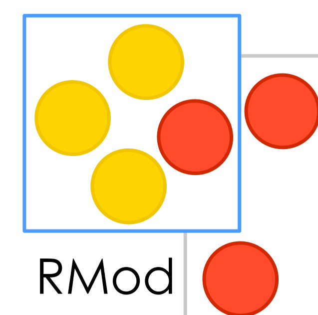
Cross-ISA Testing of the Pharo VM

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

Heap
07
18
25
bytecode methods
objects
Bytecode Interpreter
DoIt
foo
fum
bar:
Pharo execution Stack
Jit Compiler
method
method
Native Code Zone

+ GC
+ FFI
...

Some Numbers

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

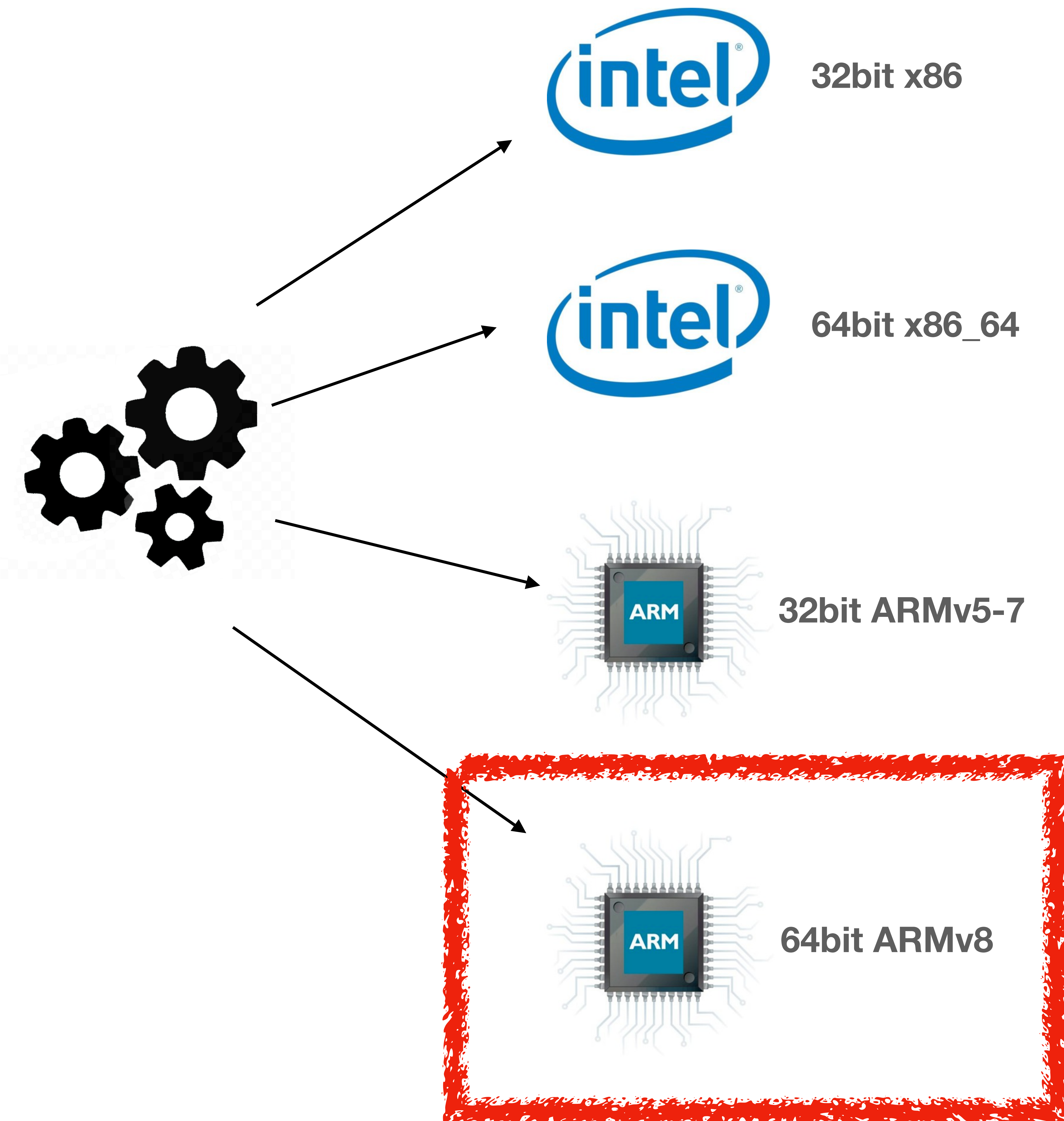
Lots of combinations!

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

	Real Hardware Execution		
Feedback-cycle speed	Very low		
Availability	Low		
Reproducibility	Low		
Precision	High		
Debuggability	Low		

Simulation Environment

Simulation Environment (Pharo)

VM

Interpreter

GC

JIT Compiler

Transpiled to

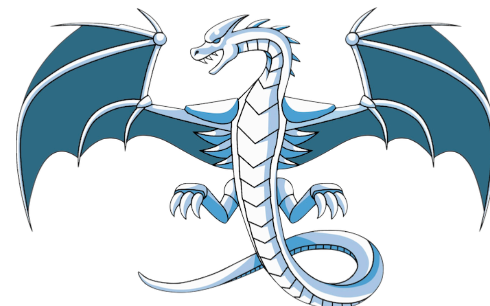
Heap

Native Code Cache

Production VM (C)

Unicorn

LLVM Disassembler



Miranda et al.

Two decades of smalltalk vm development: live vm development through simulation tools.
VMIL'18

Simulation Environment Comparison

	Real Hardware Execution	Full-System Simulation	
Feedback-cycle speed	Very low	Low	
Availability	Low	High	
Reproducibility	Low	Low	
Precision	High	Low	
Debuggability	Low	High	

Unit Testing Infrastructure

Extending the simulation environment

Simulation Environment (Pharo)

Testing infrastructure

VM

Interpreter

GC

JIT Compiler

Heap

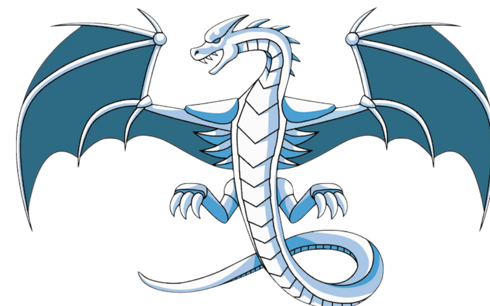
Native Code Cache

Transpiled to

Production VM (C)

Unicorn

LLVM Disassembler



Our testing infrastructure by example

```
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)
```

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: se                                   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

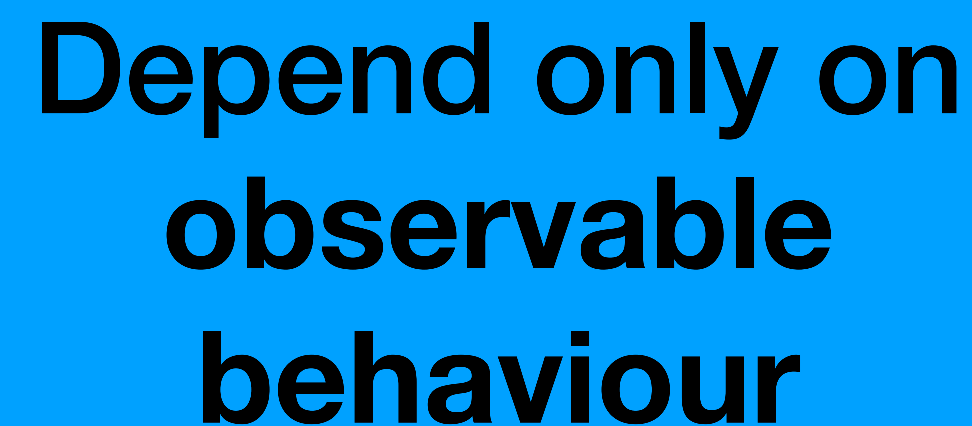
VM Unit Testing Lessons

Insights: Black box testing

```
testPushConstantZeroBytecodePushesASmallIntegerZero

  self compile: [ compiler genPushConstantZeroBytecode ].
  self runGeneratedCode.

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



Reusable on different backends / architectures

Resistant to changes in the implementation

VM Unit Testing Lessons

Insights: Cross-compile / Cross-execute

```
testPushConstantZeroBytecodePushesASmallIntegerZero

  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)
```

Second: the next simplest test

Focus on enhancing the testing infrastructure

Unit Testing Infrastructure Comparison

	Real Hardware Execution	Full-System Simulation	Unit-Testing
Feedback-cycle speed	Very low	Low	High
Availability	Low	High	High
Reproducibility	Low	Low	High
Precision	High	Low	Low
Debuggability	Low	High	High

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

Test x1000

Better Infrastructure

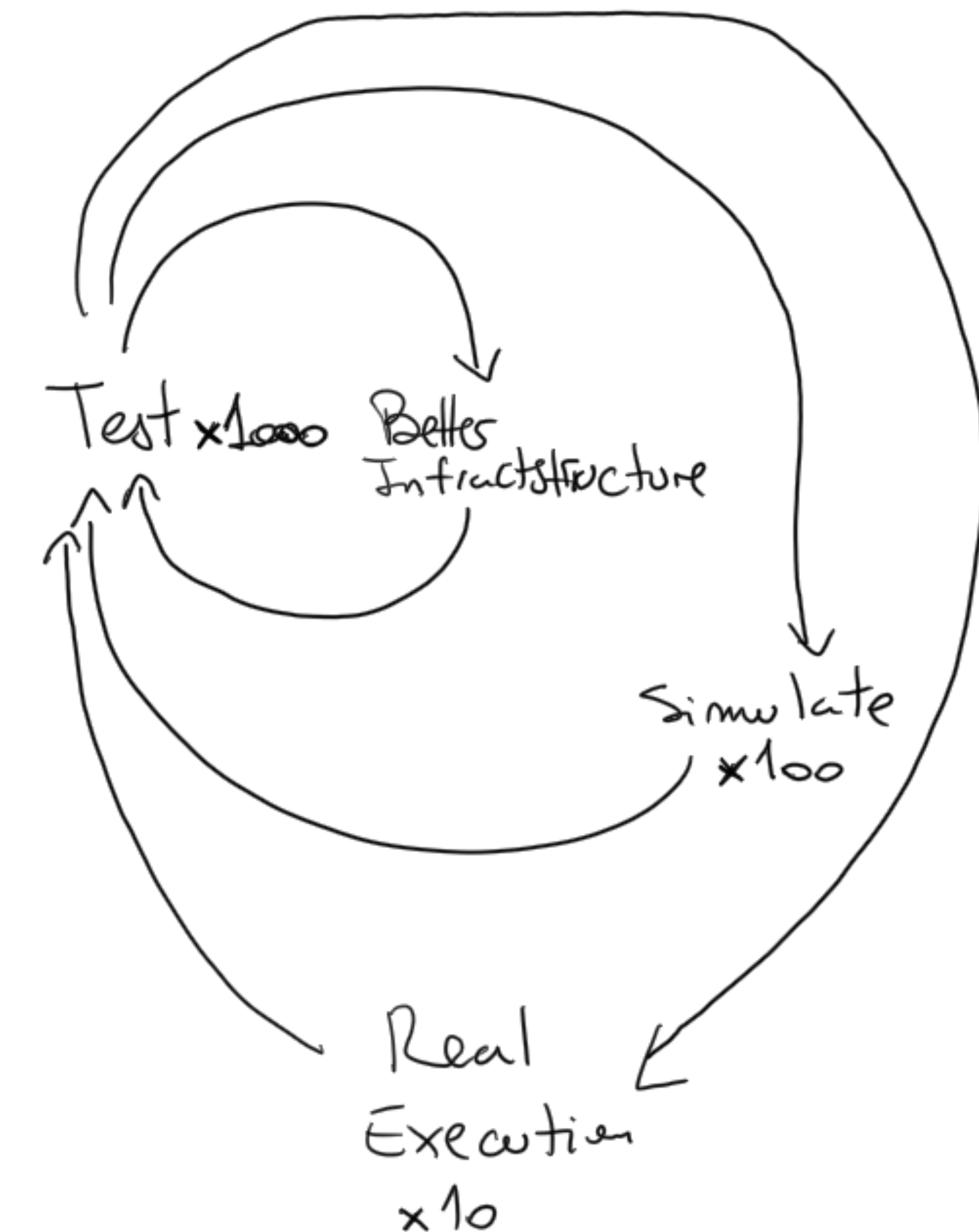
Simulate x100

Real Execution x10

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

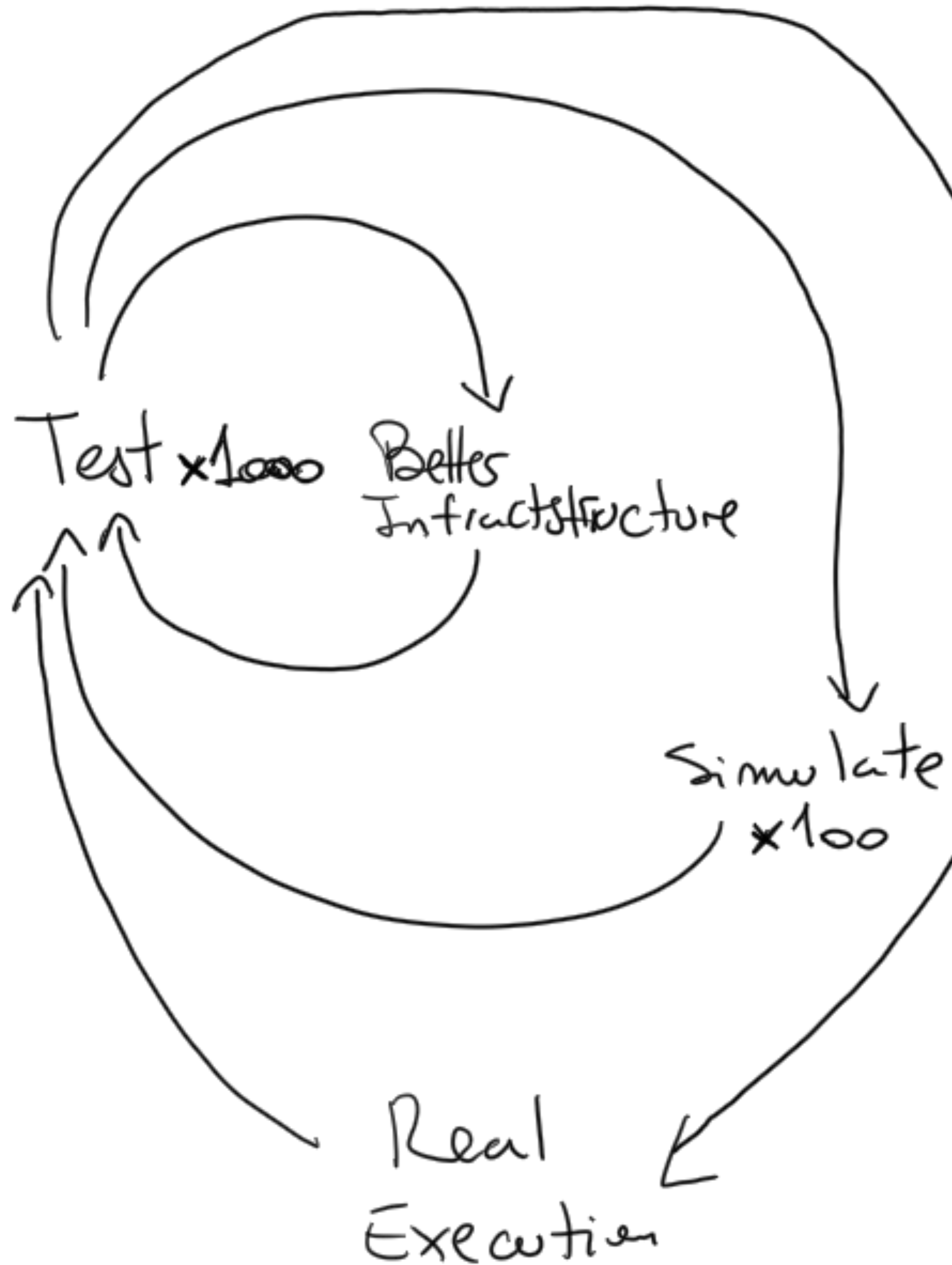
Automatic VM Validation

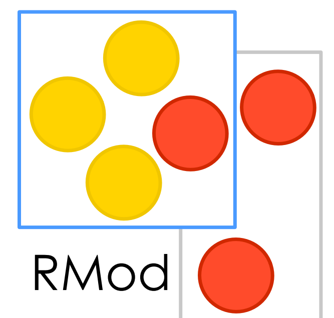
- Automatic (Unit?) Test Case Generation
- Interpreter vs Compiler Differential Testing
- VM Tailored Multi-level Debugging

Cross-ISA Testing of the Pharo VM

Lessons learned while porting to ARMv8 64bits

	Real Hardware Execution	Full-System Simulation	Unit-Testing
Feedback-cycle speed	Very low	Low	High
Availability	Low	High	High
Reproducibility	Low	Low	High
Precision	High	Low	Low
Debuggability	Low	High	High





Debugging a compiler

Insights: build your own tools, based on needs, not desires

Examples:

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

VM Debugger

IR Instructions
' (PopR 10 13503 810113)'
' (Label 1)'
' (TstCqR 7 10 757D93)'
' (JumpNonZero (Label 2) 20D9063)'
' (MoveMwrR 0 10 22/16 53B03)'
' (AndCqR 4194295/3FFFF7 22/16 no mcode)'
' (JumpNonZero (Label 2) D9663)'
' (MoveMwrR 8 10 10 853503)'
' (Jump (Label 1) FE1FF06F)'
' (Label 2)'
' (MoveMwrR 0 2 23/17 13B83)'
' (Label 3)'
' (TstCqR 7 23/17 7BFD93)'
' (JumpNonZero (Label 4) 20D9063)'
' (MoveMwrR 0 23/17 22/16 BBB03)'
' (AndCqR 4194295/3FFFF7 22/16 no mcode)'
' (JumpNonZero (Label 4) D9663)'
' (MoveMwrR 8 23/17 23/17 8BBB83)'
' (Jump (Label 3) FE1FF06F)'
' (Label 4)'
' (CmpRR 10 23/17 41750DB3)'
' (JumpNonZero (MoveCqR 16856080/1013410 10
' (MoveCqR 16856096/1013420 10 1013537 42050
' (Jump (MoveRMwr 10 0 2 A13023) C0006F)'

Address		ASM	Bytes
16r1000000		ld a0, 0(sp)	#[3 53 1 0]
16r1000004		addi sp, sp, 8	#[19 1 129 0]
16r1000008		andi s11, a0,	#[147 125 11
16r100000C		bnez s11, 32	#[99 144 13
16r1000010		ld s6, 0(a0)	#[3 59 5 0]
16r1000014		lui t0, 1024	#[183 2 64 0]
16r1000018		addiw t0, t0, -	#[155 130 11
16r100001C		and s6, s6, t0	#[51 123 91
16r1000020		bnez s11, 12	#[99 150 13
16r1000024		ld a0, 8(a0)	#[3 53 133 0]
16r1000028		j -32	#[111 240 31
16r100002C		ld s7, 0(sp)	#[131 59 1 0]
16r1000030		andi s11, s7,	#[147 253 12
16r1000034		bnez s11, 32	#[99 144 13
16r1000038		ld s6, 0(s7)	#[3 187 11 0]
16r100003C		lui t0, 1024	#[183 2 64 0]
16r1000040		addiw t0, t0, -	#[155 130 11
16r1000044		and s6, s6, t0	#[51 123 91
16r1000048		bnez s11, 12	#[99 150 13
16r100004C		ld s7, 8(s7)	#[131 187 13
16r1000050		j -32	#[111 240 31
16r1000054		sub s11, a0, s	#[179 13 117
16r1000058		bnez s11, 16	#[99 152 13
16r100005C		lui a0, 4115	#[55 53 1 1]

lr		'16r1001000'
pc		'16r1000'
sp		'16r1002FE8'
fp		'16r1003000'
x0	zero	'16r0'
x1	ra	'16r1001000'
x2	sp | sp	'16r1002FE8'
x3	gp	'16r0'
x4	tp	'16r0'
x5	t0 | ip1	'16r0'
x6	t1 | ip2	'16r0'
x7	t2	'16r0'
x8	s0(fp) | fp	'16r1003000'
x9	s1	'16r0'
x10	a0 | arg0	'16r0'
x11	a1 | arg1	'16r0'
x12	a2 | carg0	'16r0'
x13	a3 | carg1	'16r0'
x14	a4 | carg2	'16r0'
x15	a5 | carg3	'16r0'
x16	a6	'16r0'
x19	s3 | extra1	'16r0'
x20	s4 | extra2	'16r0'
x22	s6 | temp	'16r0'

SP	16r1002FE8	16r1013400
	16r1002FF0	16r1013400
	16r1002FF8	16r1013400
FP	16r1003000	16r0
	16r1003008	16r0
	16r1003010	16r0
	16r1003018	16r0
	16r1003020	16r0
	16r1003028	16r0
	16r1003030	16r0
	16r1003038	16r0
	16r1003040	16r0
	16r1003048	16r0
	16r1003050	16r0
	16r1003058	16r0
	16r1003060	16r0
	16r1003068	16r0
	16r1003070	16r0
	16r1003078	16r0
	16r1003080	16r0
	16r1003088	16r0
	16r1003090	16r0
	16r1003098	16r0
	16r10030A0	16r0

Jump to | Step | Disassemble at PC