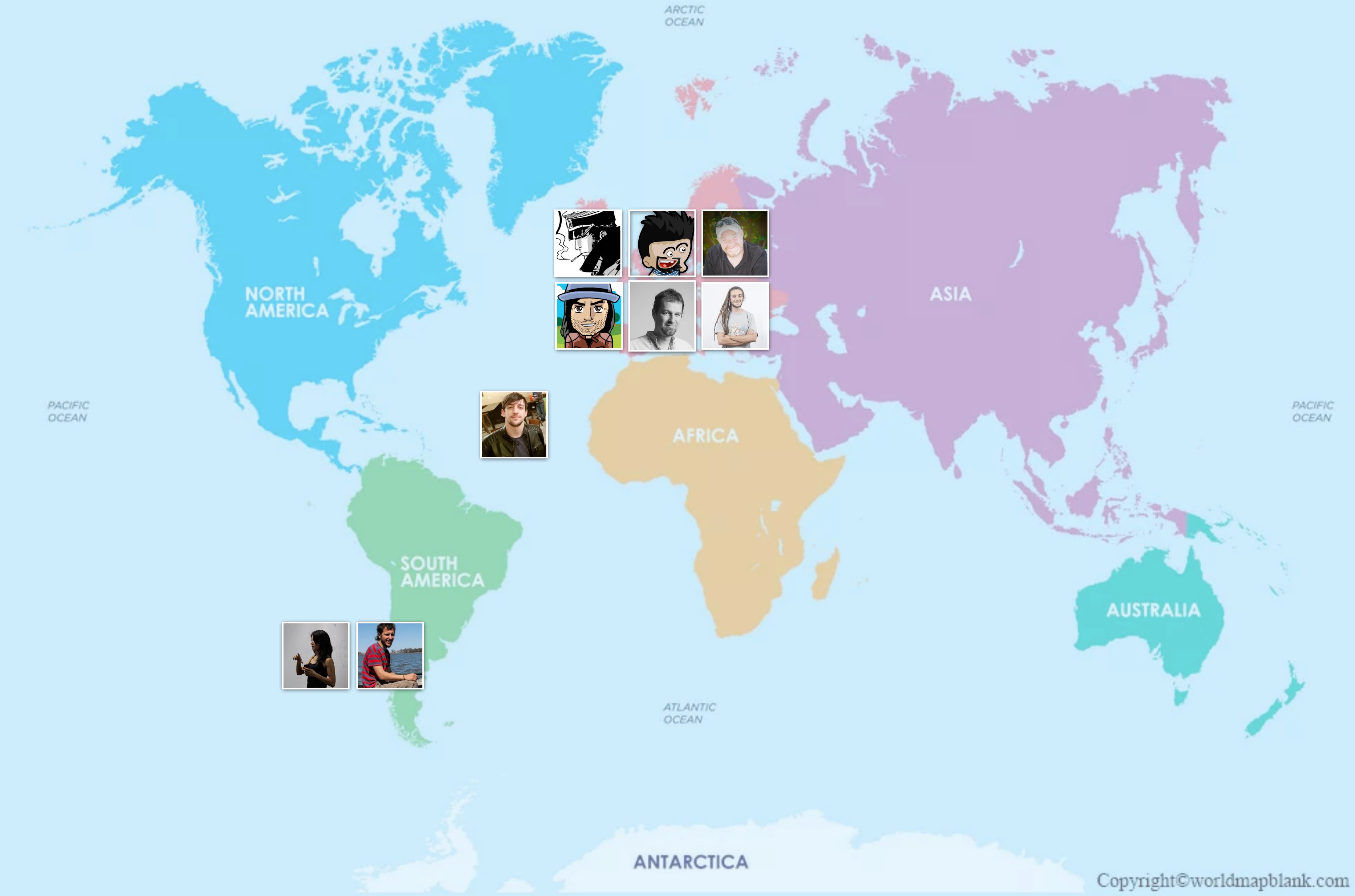
Pharo Virtual Machine

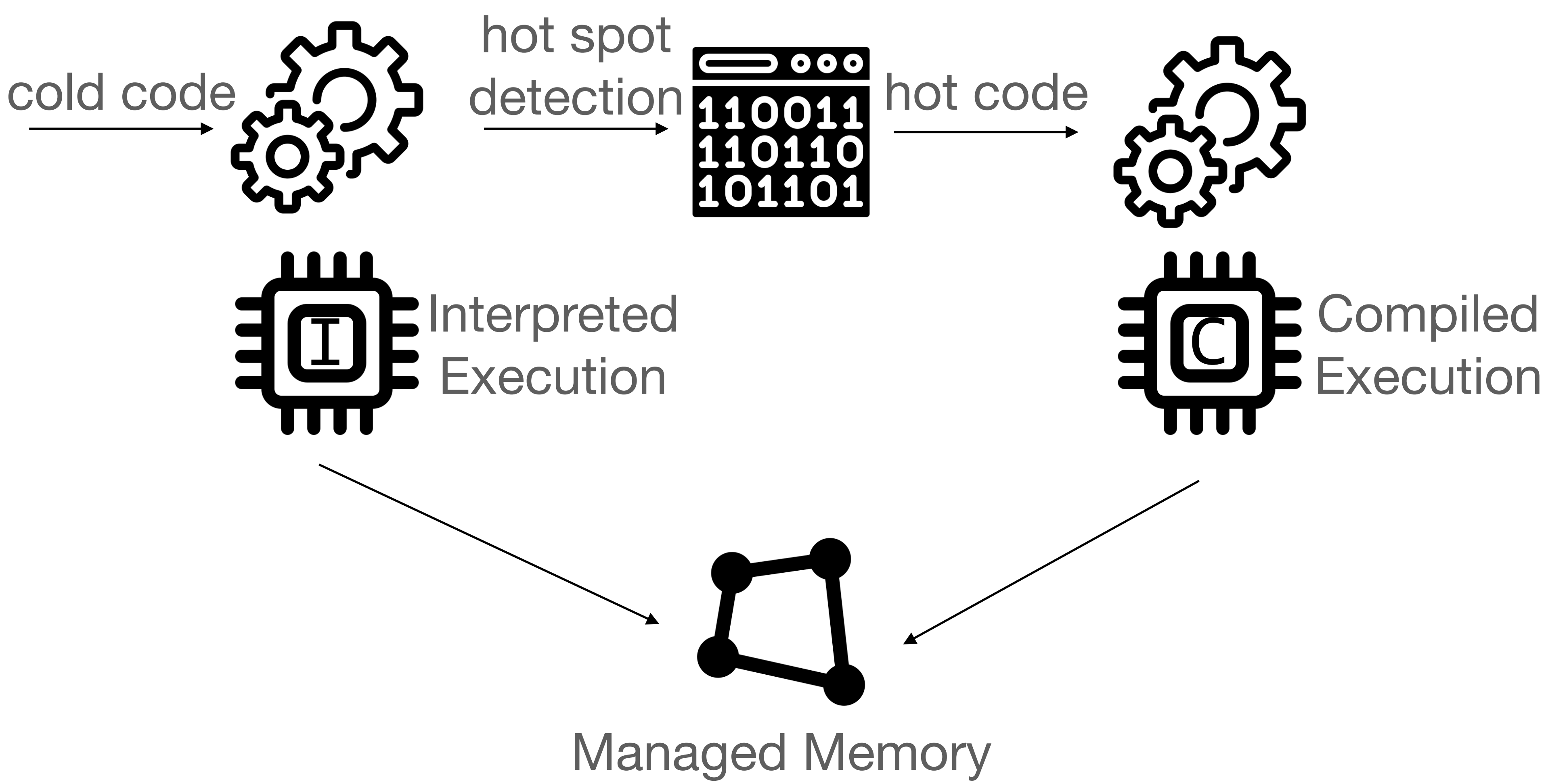
News from the Front

P. Tesone - G. Polito - ESUG'22
@tesonep pablo.tesone@inria.fr
@guillep guillermo.polito@univ-lille.fr

2022 VM+ Team


ARCTIC OCEAN
NORTH AMERICA
ASIA
PACIFIC OCEAN
AFRICA
PACIFIC OCEAN
SOUTH AMERICA
AUSTRALIA
ATLANTIC OCEAN
ANTARCTICA
Copyright©worldmapblank.com

Virtual Machine Execution Engine

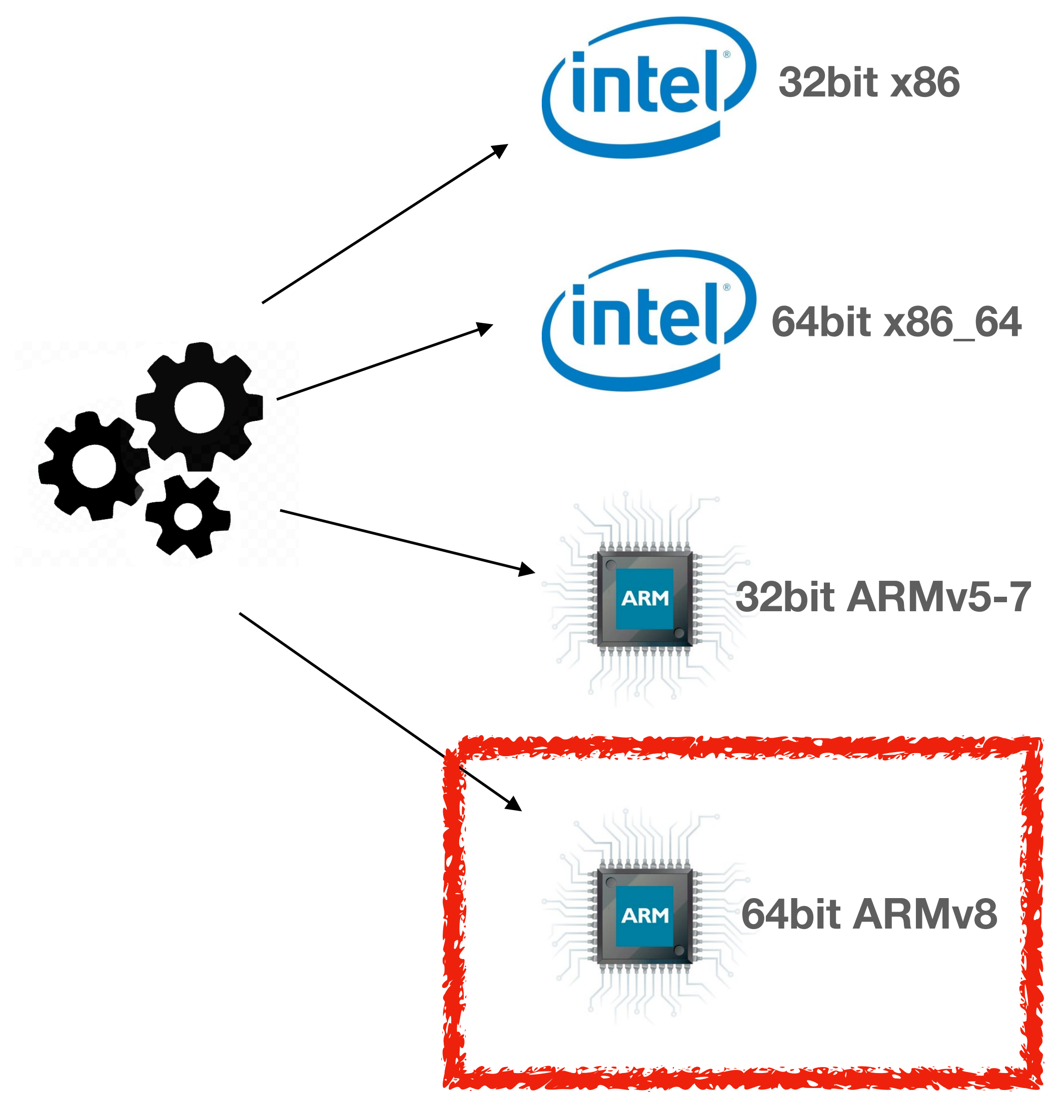


ARM64 Backend

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

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

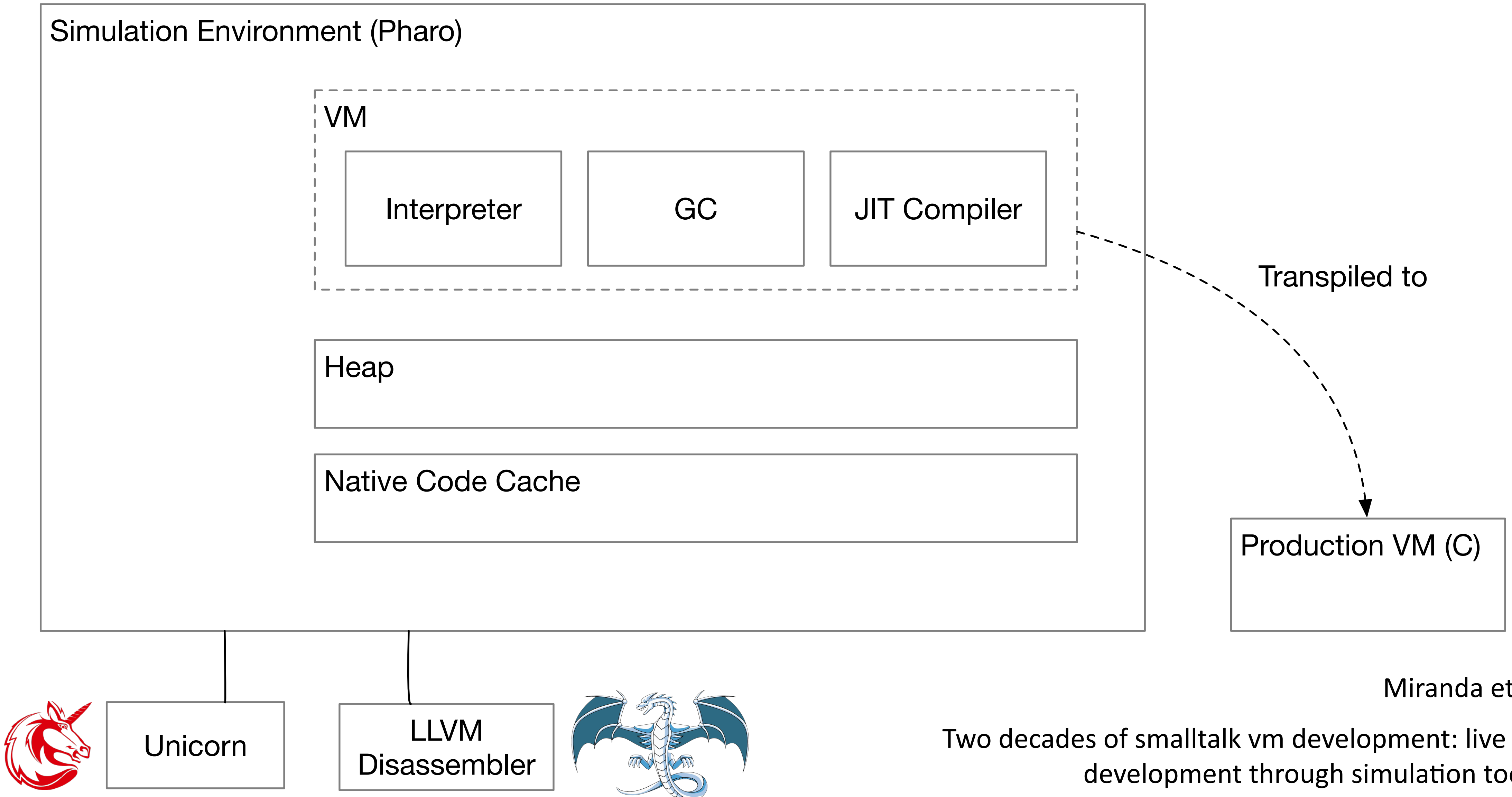
JIT compiler IR



Working Directly on Real Hardware

- 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

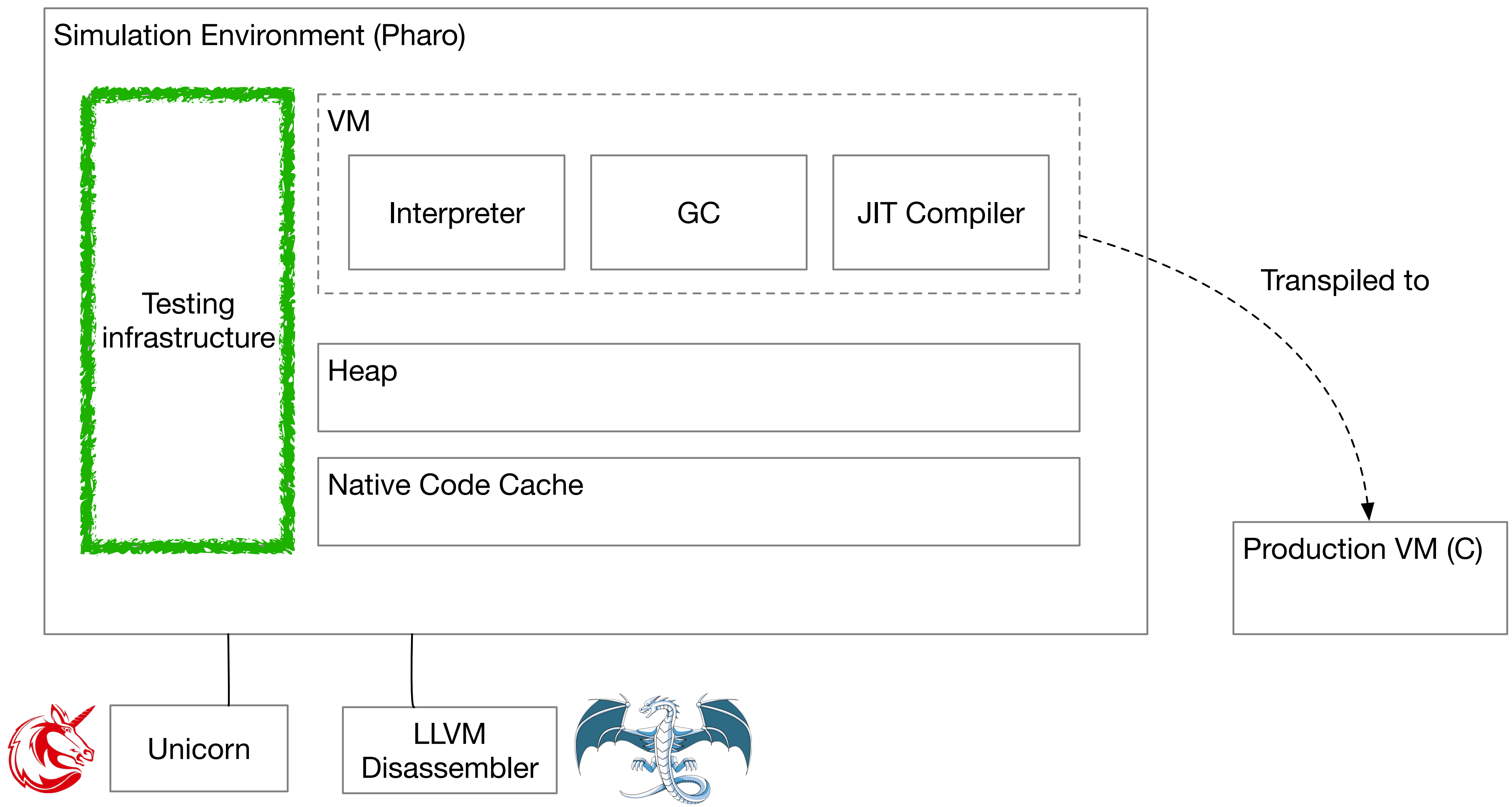
Simulation Environment



Miranda et al.

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

Extending Simulation with Unit Tests



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

```
testPushConstantZeroBytecodePushesASmallIntegerZero

    self compile: [ compiler genPushConstantZeroBytecode
    self runGeneratedCode.

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

Compiler internal DSL

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

Blackbox 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 /

Resistant to changes in the implementation

Cross-Compilation, Cross-Execution

http://www.unicorn-engine

```
testPushConstantZeroBytecodePushesASmallIntegerZero

    self compile: [ compiler genPushConstantZeroBytecode ].
    self runGeneratedCode.

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

Hardware independent

Parametrizable tests

ARM

intel

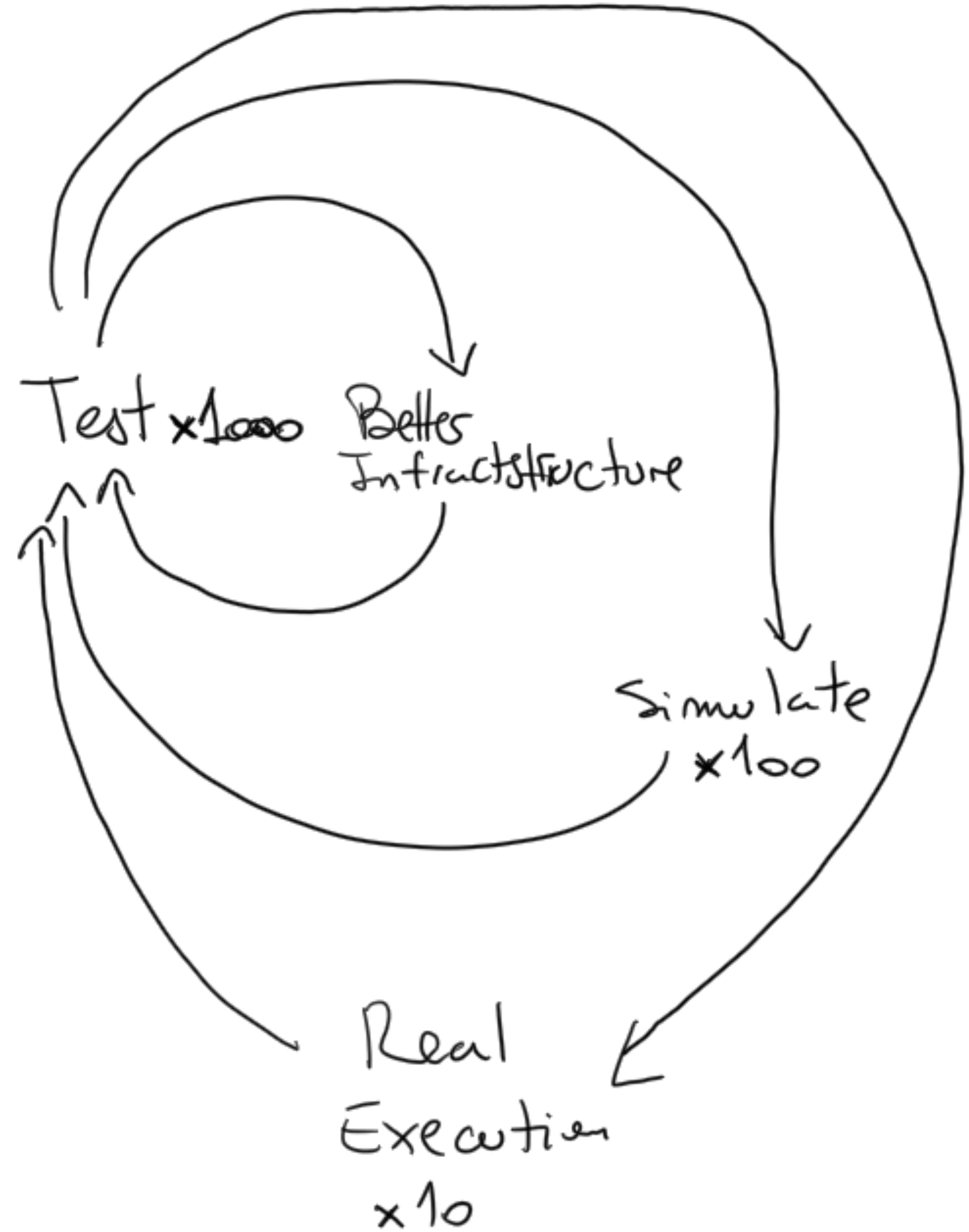
RISC-V

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



Testing & TDDing the VM

- No useful unit tests by ~06/2020
- Large manual testing effort during 2020 while porting to ARM64bits
  - Extended VM simulation with a (TDD compatible) unit testing infrastructure
  - **450+** written tests on the interpreter and the garbage collector*
  - **580+** written tests on the JIT compiler*
  - Parametrisable for 32 and 64bits, ARM32, ARM64, x86, x86-64

* Numbers by 05/2021

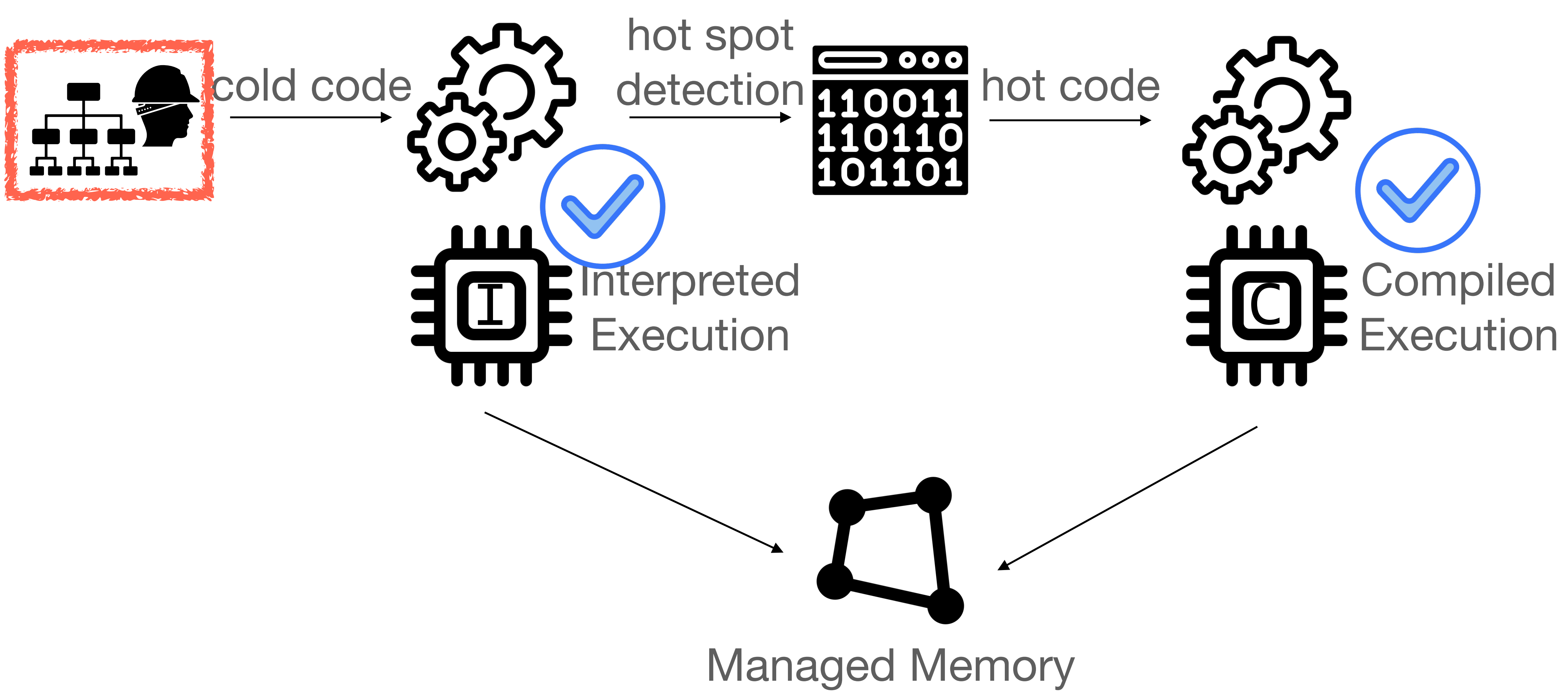
Testing & TDDing the VM

1040+ tests, are they enough?

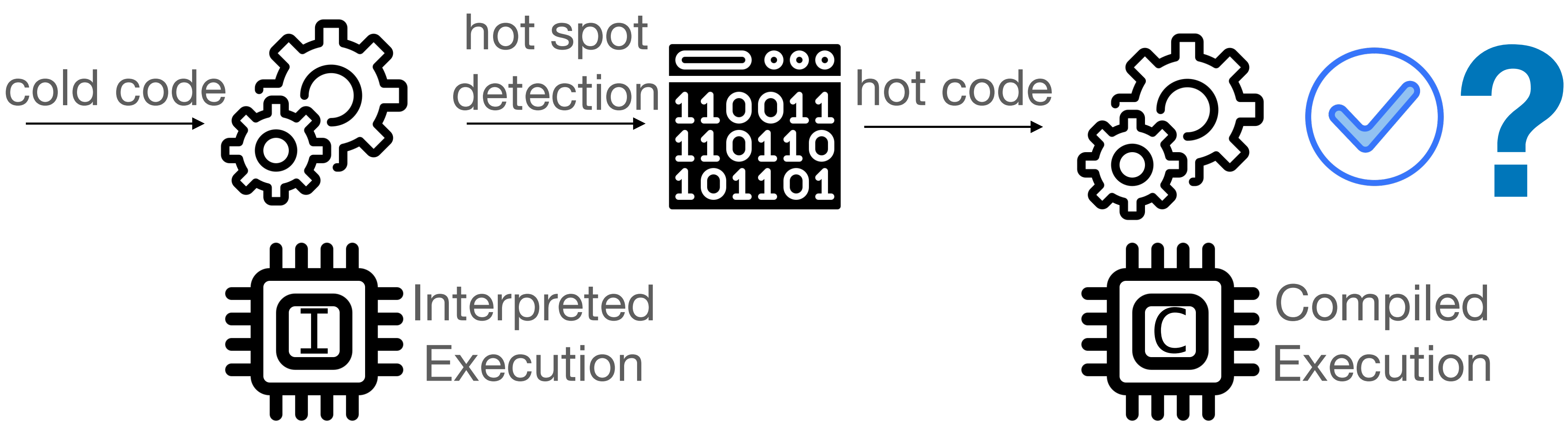
- No useful unit tests by ~06/2020
- Large manual testing effort during 2020 while porting to ARM64bits
  - Extended VM simulation with a (TDD compatible) unit testing infrastructure
  - **450+** written tests on the interpreter and the garbage collector*
  - **580+** written tests on the JIT compiler*
  - Parametrisable for 32 and 64bits, ARM32, ARM64, x86, x86-64

* Numbers by 05/2021

How can we automatically test VMs?



Challenges of VM Test Generation



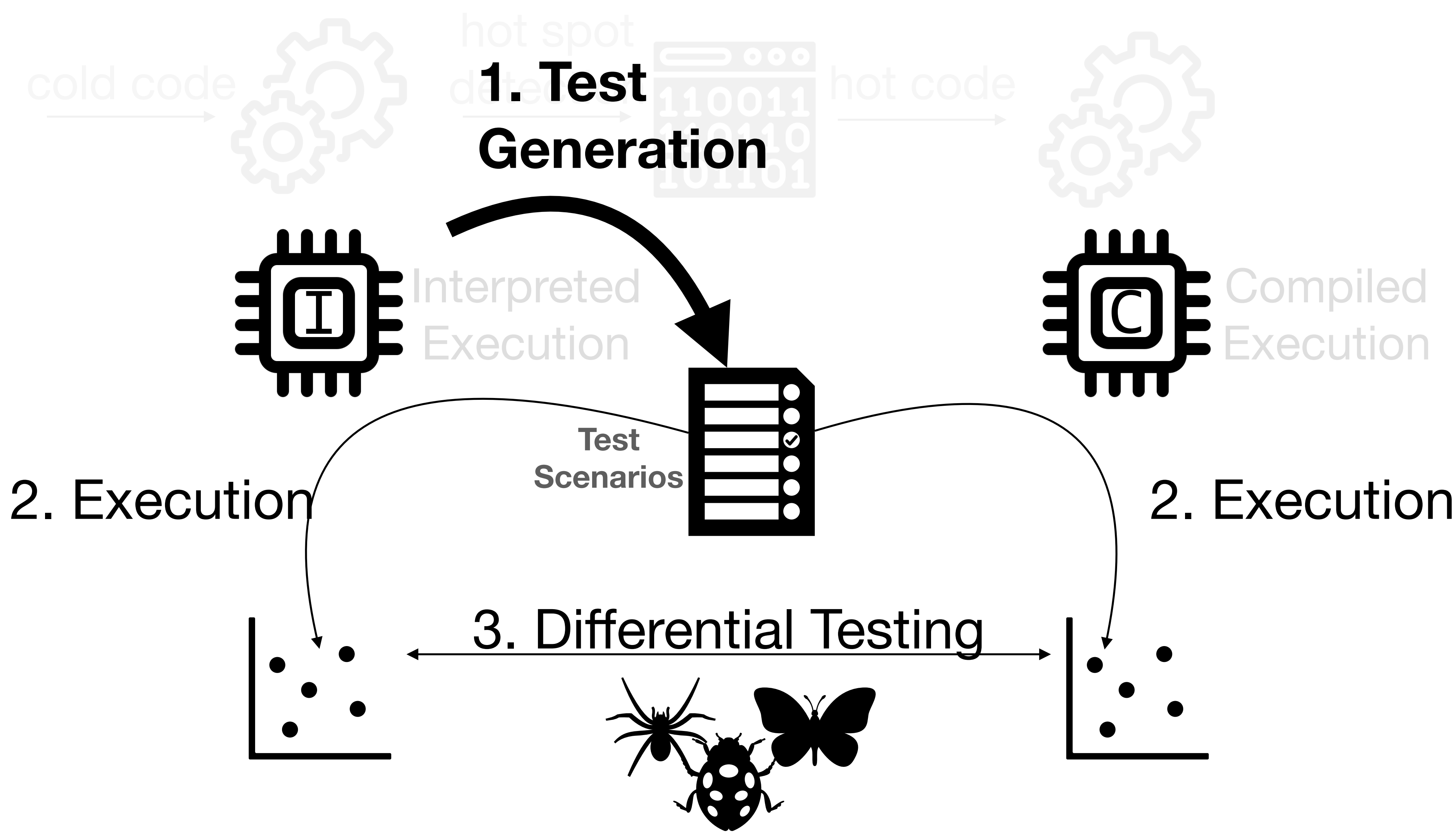
Challenge 1: Test

- Do they cover different code ***regions/branches/paths***?

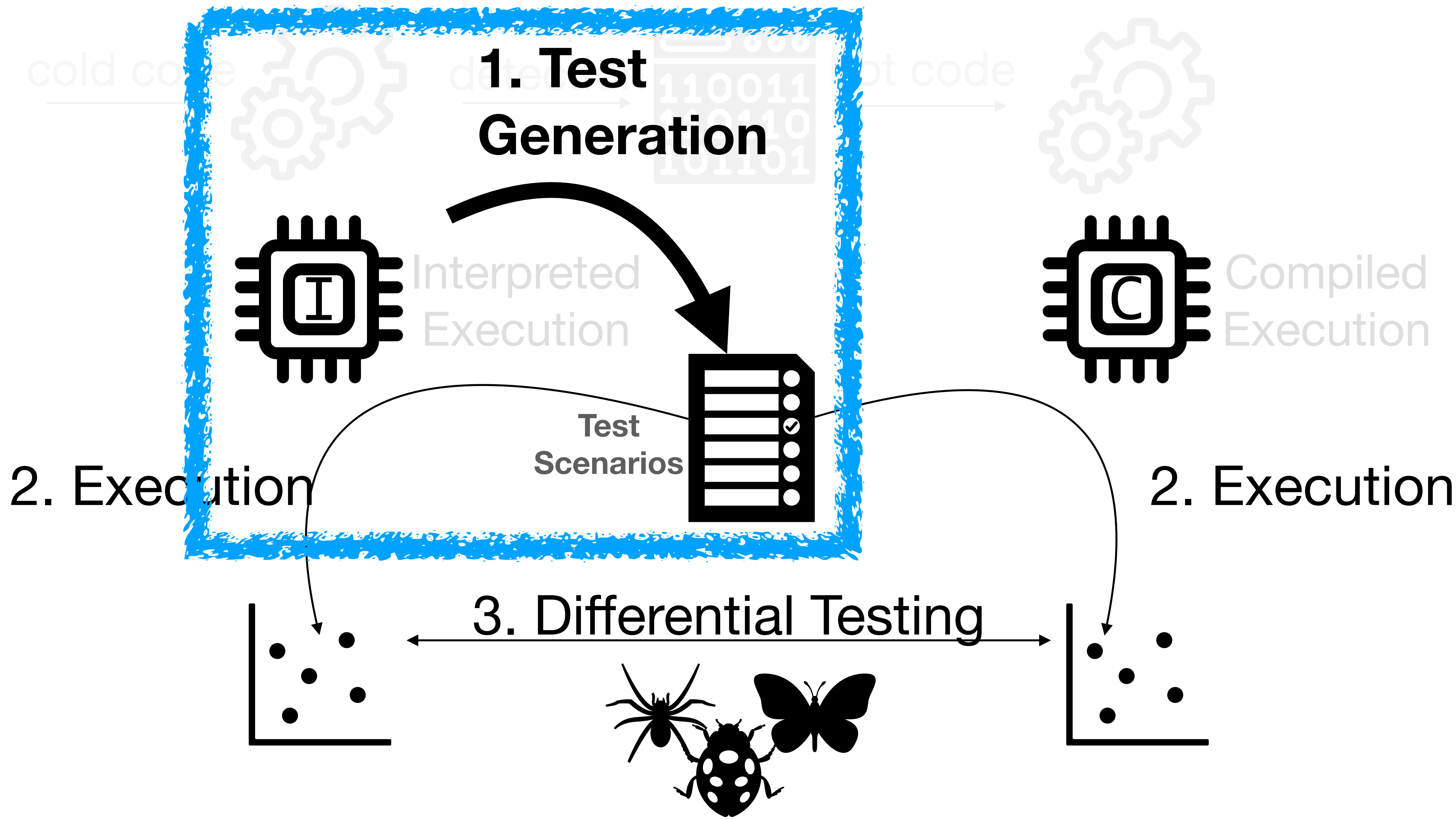
Challenge 2: Test

- How do we determine what is the ***expected output*** of a generated test?

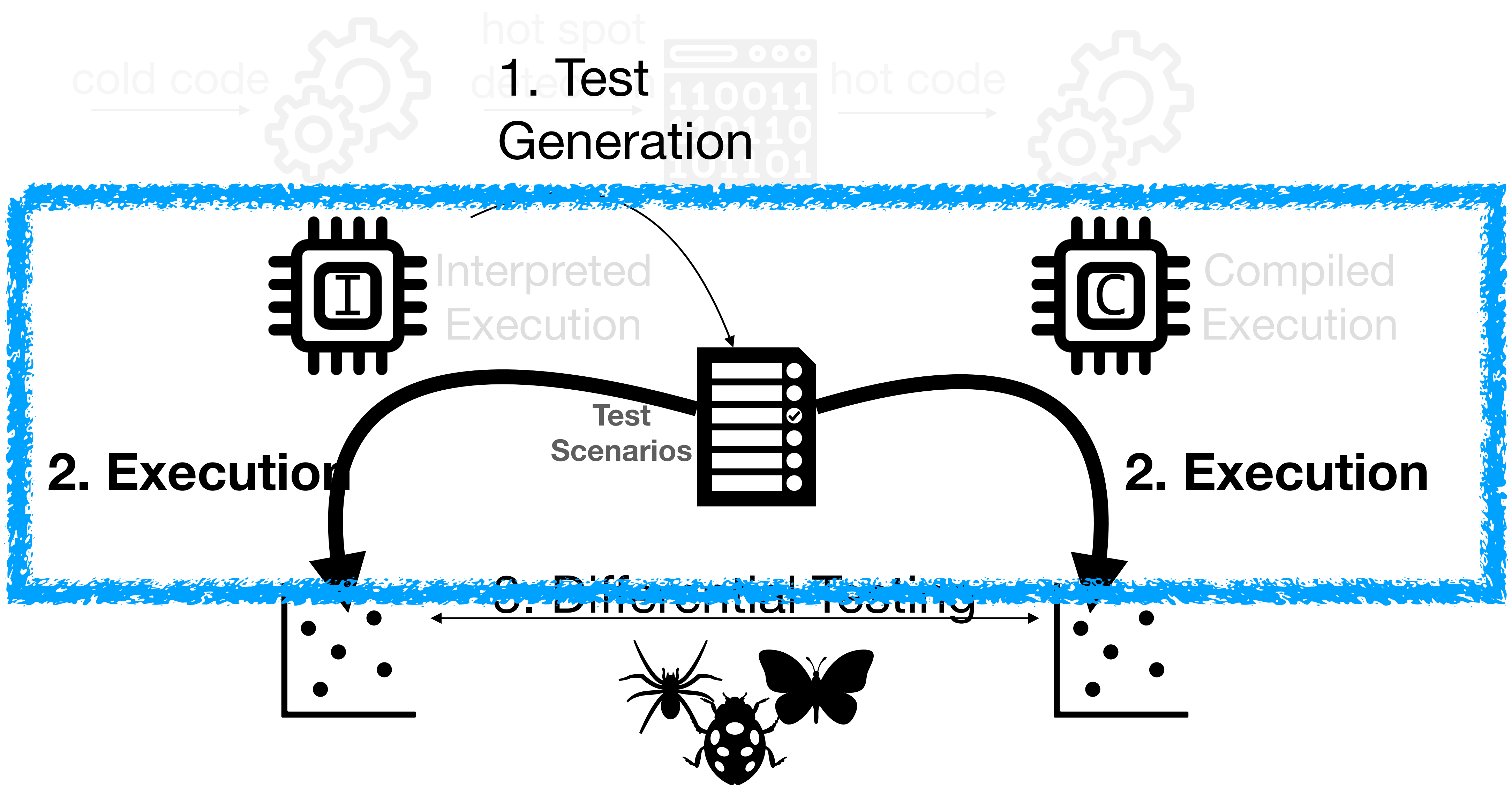
Interpreter-Guided Automatic JIT Compiler Unit Testing



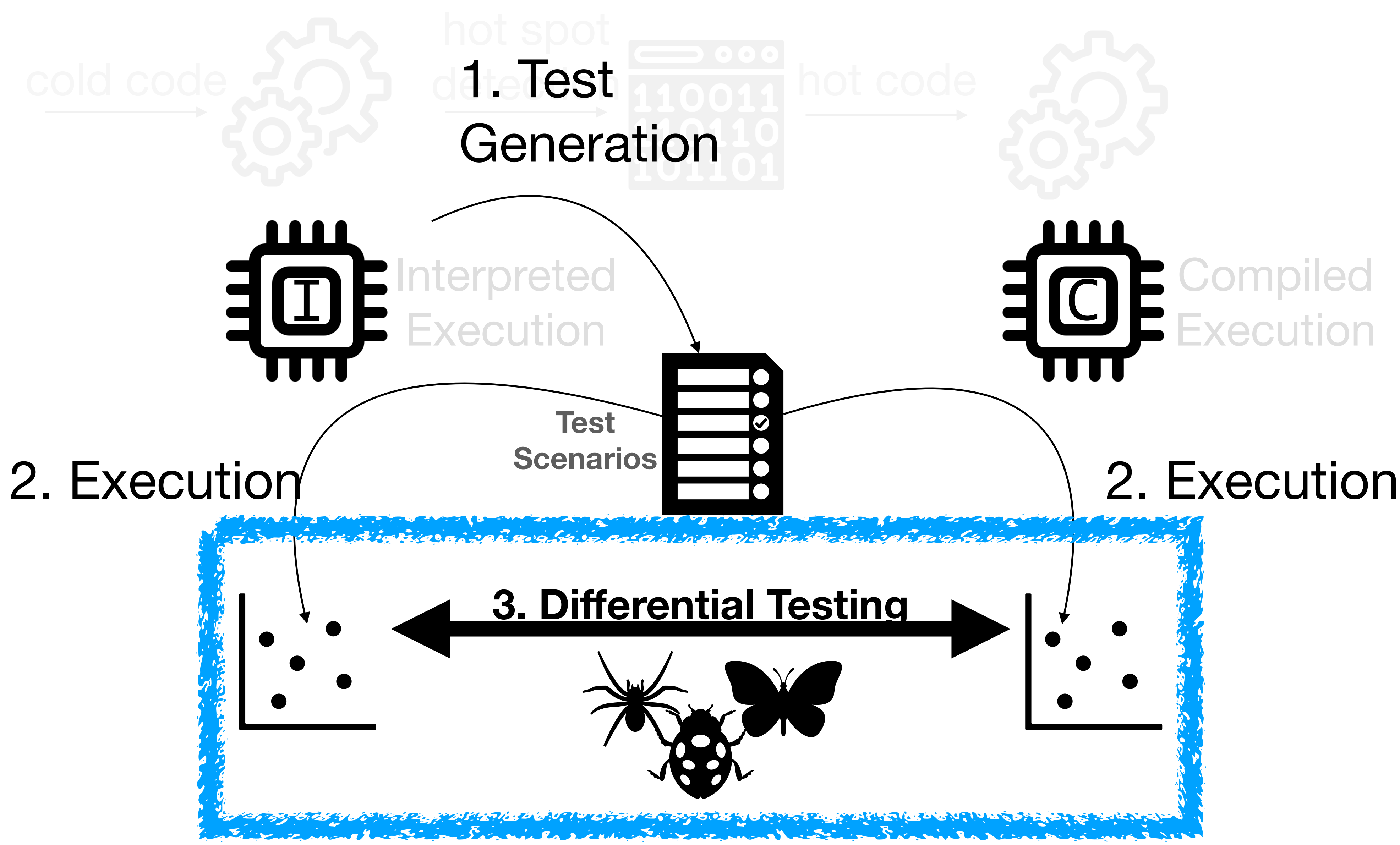
Interpreter-Guided Automatic JIT Compiler Unit Testing



Interpreter-Guided Automatic JIT Compiler Unit Testing



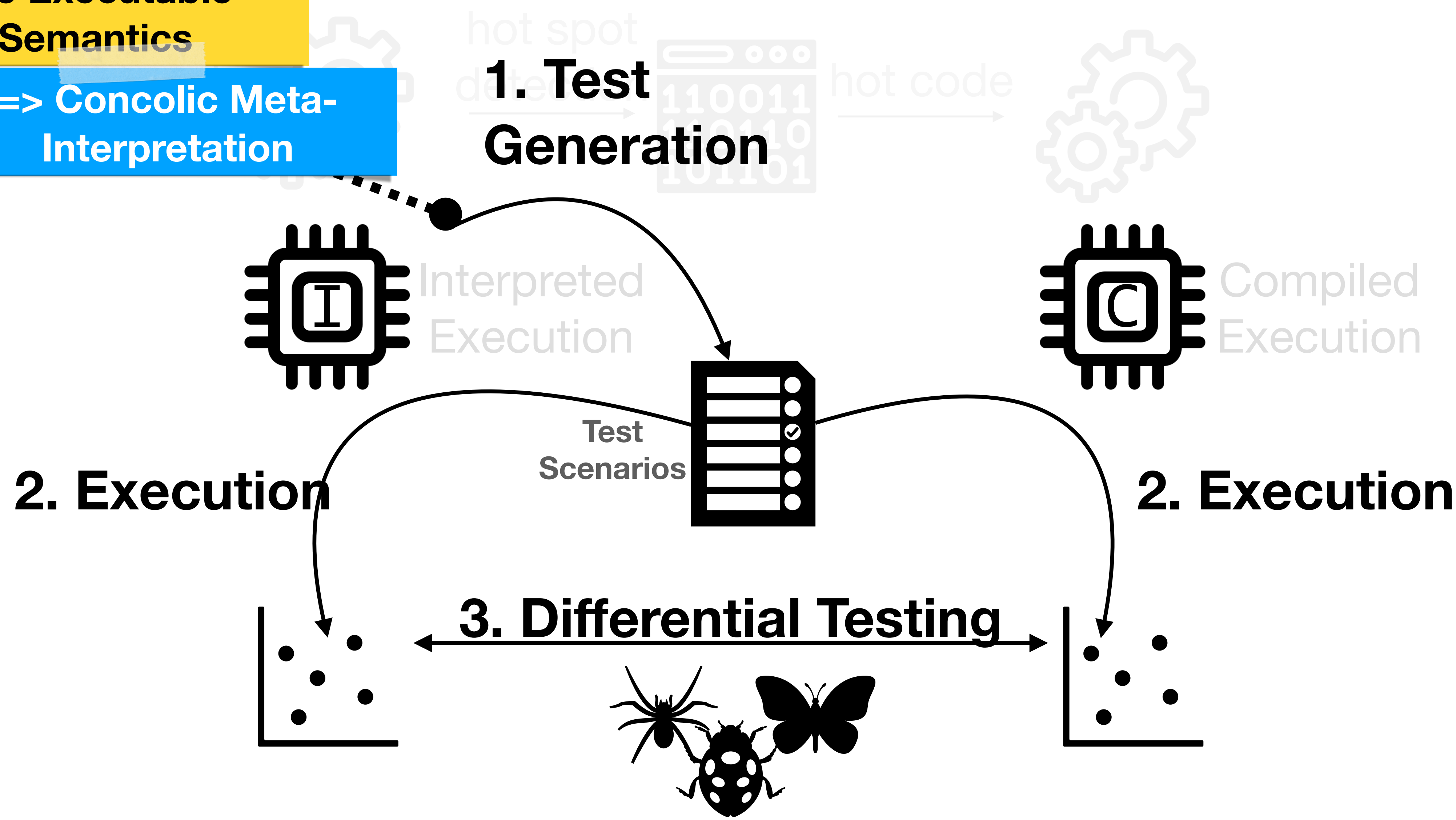
Interpreter-Guided Automatic JIT Compiler Unit Testing



Interpreter-Guided Automatic JIT Compiler Unit Testing

Insight 1: Interpreters are Executable Semantics

=> Concolic Meta-Interpretation



Interpreter are Executable Semantics

Pharo VM Example

```
Interpreter >> bytecodePrimAdd
  | rcvr arg result |
  rcvr := self internalStackValue: 1.
  arg := self internalStackValue: 0.
  (objectMemory areIntegers: rcvr and: arg) ifTrue: [
    result := (objectMemory integerValueOf: rcvr) + (
      objectMemory integerValueOf: arg).
    "Check for overflow"
    (objectMemory isIntegerValue: result) ifTrue: [
      self
        internalPop: 2
        thenPush: (objectMemory integerObjectOf: result).
      ^ self fetchNextBytecode "success"]].
  "Slow path, message send"
  self normalSend
```

Interpreter are Executable Semantics

Pharo VM Example

```
Interpreter >> bytecodePrimAdd
  | rcvr arg result |
  rcvr := self internalStackValue: 1.
  arg := self internalStackValue: 0.
  (objectMemory areIntegers: rcvr and: arg) ifTrue: [
    result := (objectMemory integerValueOf: rcvr) + (
      objectMemory integerValueOf: arg).
    "Check for overflow"
    (objectMemory isIntegerValue: result) ifTrue: [
      self
        internalPop: 2
        thenPush: (objectMemory integerObjectOf: result).
      ^ self fetchNextBytecode "success"]].
  "Slow path, message send"
  self normalSend
```

If both operands are integers

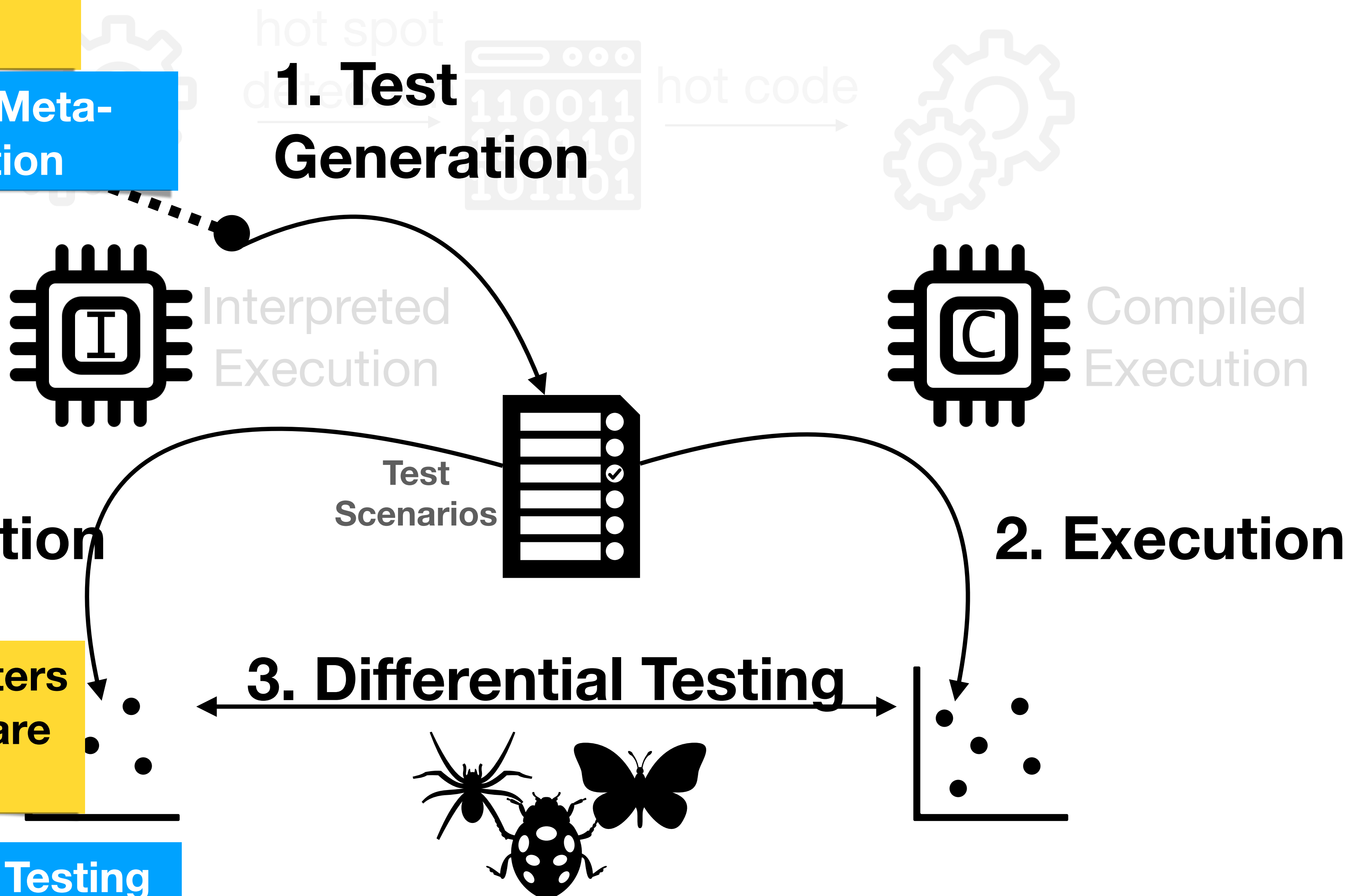
If their sum does not overflow

Else, slow path => message send

Interpreter-Guided Automatic JIT Compiler Unit Testing

Insight 1: Interpreters are Executable Semantics

=> Concolic Meta-Interpretation



Insight 2: Interpreters and Compiler Share Semantics

=> Differential Testing

Interpreter VS Compiled Code

Pharo VM Example

```
Interpreter >> bytecodePrimAdd
  | rcvr arg result |
  rcvr := self internalStackValue: 1.
  arg := self internalStackValue: 0.
  (objectMemory areIntegers: rcvr and: arg) ifTrue: [
    result := (objectMemory integerValueOf: rcvr) + (
      objectMemory integerValueOf: arg).
    "Check for overflow"
    (objectMemory isIntegerValue: result) ifTrue: [
      self
        internalPop: 2
        thenPush: (objectMemory integerObjectOf: result).
      ^ self fetchNextBytecode "success"]].
  "Slow path, message send"
  self normalSend
```

```
  ... # previous bytecode IR
      checkSmallInteger t0
      jumpzero notsmi
      checkSmallInteger t1
      jumpzero notsmi
      t2 := t0 + t1
      jumpIfNotOverflow continue
notsmi: #slow case first send
      t2 := send #+ t0 t1
continue:
  ... # following bytecode IR
```

Interpreter VS Compiled Code

Pharo VM Example

```
Interpreter >> bytecodePrimAdd
  | rcvr arg result |
  rcvr := self internalStackValue: 1.
  arg := self internalStackValue: 0.
  (objectMemory areIntegers: rcvr and: arg) ifTrue: [
    result := (objectMemory integerValueOf: rcvr) + (
      objectMemory integerValueOf: arg).
    "Check for overflow"
    (objectMemory isIntegerValue: result) ifTrue: [
      self
        internalPop: 2
        thenPush: (objectMemory integerObjectOf: result).
      ^ self fetchNextBytecode "success"]].
  "Slow path, message send"
  self normalSend
```

```
  ... # previous bytecode IR
  checkSmallInteger t0
  jumpzero notsmi
  checkSmallInteger t1
  jumpzero notsmi
  t2 := t0 + t1
  jumpIfNotOverflow continue
notsmi: #slow case first send
  t2 := send #+ t0 t1
continue:
  ... # following bytecode IR
```

Concolic Testing through Meta-interpretation

- Idea: Guide test generation by looking at the implementation

```
int f(int x, int y){
  if (x > 100){
    if (y == 1023){
      segfault(!!)
    } } }
```

Different cases
if x > 100 or <= 100!!

Different cases
if x = 1023 or != 1023

Godefroid et al. DART: Directed Automated Random Testing. PLDI’05

Set et al. CUTE: a concolic unit testing engine for C. FSE’05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
  if (x > 100){
    if (y == 1023){
      segfault(!!)
    } } }
```

x	y	constraints	next?

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
  if (x > 100){
    if (y == 1023){
      segfault(!!)
    } } }
```

x	y	constraints	next?
0	0	x <= 100	

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
  if (x > 100){
    if (y == 1023){
      segfault(!!)
    } } }
```

x	y	constraints	next?
0	0	x <= 100	x > 100

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
  if (x > 100){
    if (y == 1023){
      segfault(!!)
    } } }
```

x	y	constraints	next?
0	0	x <= 100	x > 100
101	0		

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
  if (x > 100){
    if (y == 1023){
      segfault(!!)
    } } }
```

x	y	constraints	next?
0	0	x <= 100	x > 100
101	0	x > 100, y != 1023	

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Concr**ete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
  if (x > 100){
    if (y == 1023){
      segfault(!!)
    } } }
```

x	y	constraints	next?
0	0	x <= 100	x > 100
101	0	x > 100, y != 1023	x > 100, y == 1023

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
  if (x > 100){
    if (y == 1023){
      segfault(!!)
  } } }
```

x	y	constraints	next?
0	0	x <= 100	x > 100
101	0	x > 100, y != 1023	x > 100, y == 1023
101	1023		

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Concolic Testing by Example

- **Conc**rete + Symb**olic** execution
- Goal: automatically discover *all* execution *paths*

```
int f(int x, int y){
  if (x > 100){
    if (y == 1023){
      segfault(!!)
    } } }
```

x	y	constraints	next?
0	0	x <= 100	x > 100
101	0	x > 100, y != 1023	x > 100, y == 1023
101	1023	x > 100, y != 1023	finished!

Godefroid et al. DART: Directed Automated Random Testing. PLDI' 05

Set et al. CUTE: a concolic unit testing engine for C. FSE'05

Some Numbers

- 3 bytecode compilers + 1 native method compiler
- *4928* tests generated
- **478 differences**

Compiler	# Tested Instructions	# Interpreter Paths	# Curated Paths	# Differences (%)
Native Methods (primitives)	112	2024	1520	440 (28,95%)
Simple Stack BC Compiler	175	1308	1136	18 (1,59%)
Stack-to-Register BC Compiler	175	1308	1136	10 (0,88%)
Linear-Scan Allocator BC Compiler	175	1308	1136	10 (0,88%)
Total	637	5948	4928	478 (9,7%)

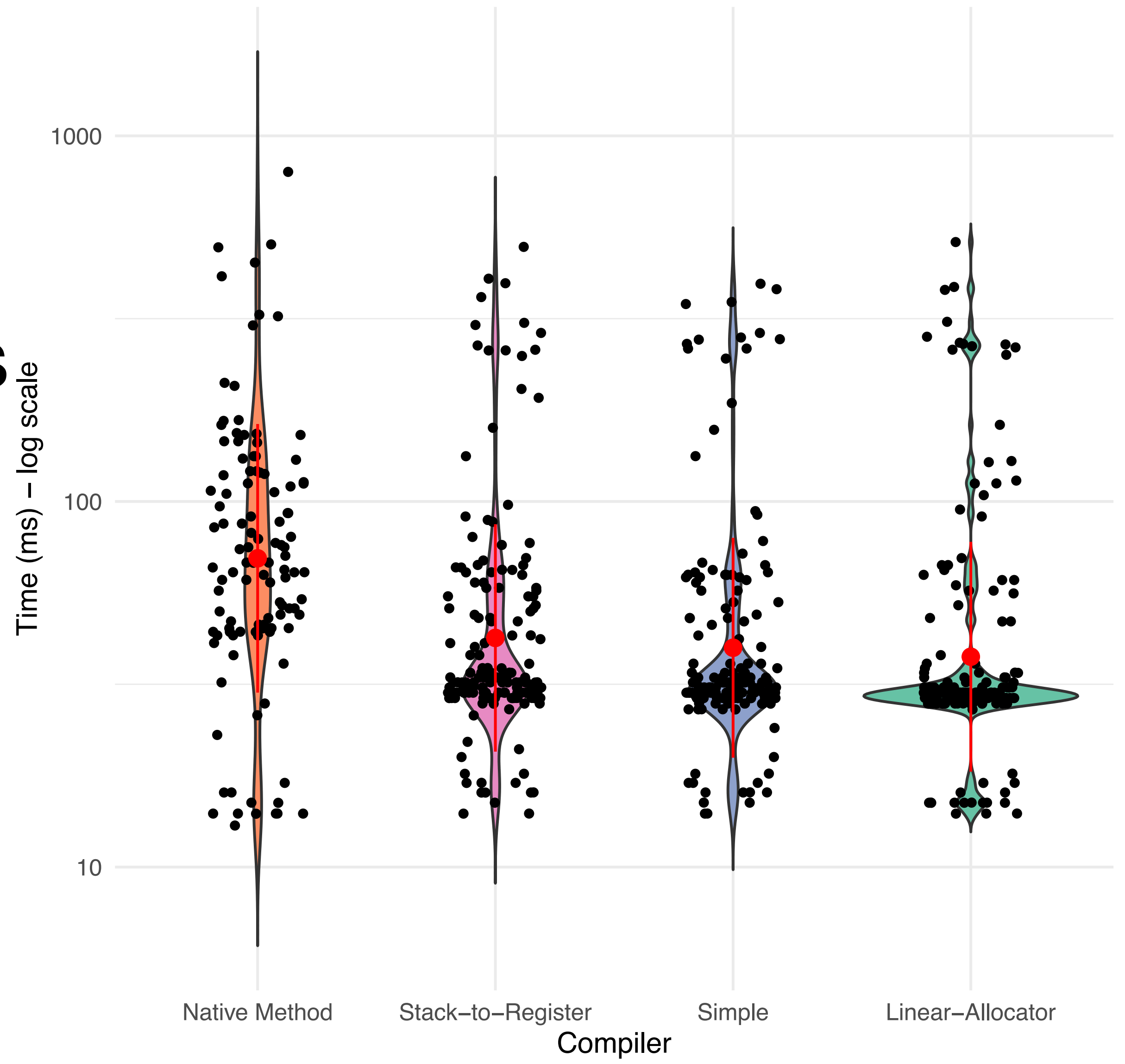
Analysis of Differences through Manual Inspection

- 91 causes, *6 different categories*
- Errors both in the interpreter AND the compilers
- 14 causes of ***segmentation faults***!

Family	# Cases
Missing interpreter type check	1
Missing compiled type check	13
Optimisation difference	10
Behavioral difference	5
Missing Functionality	60
Simulation Error	2

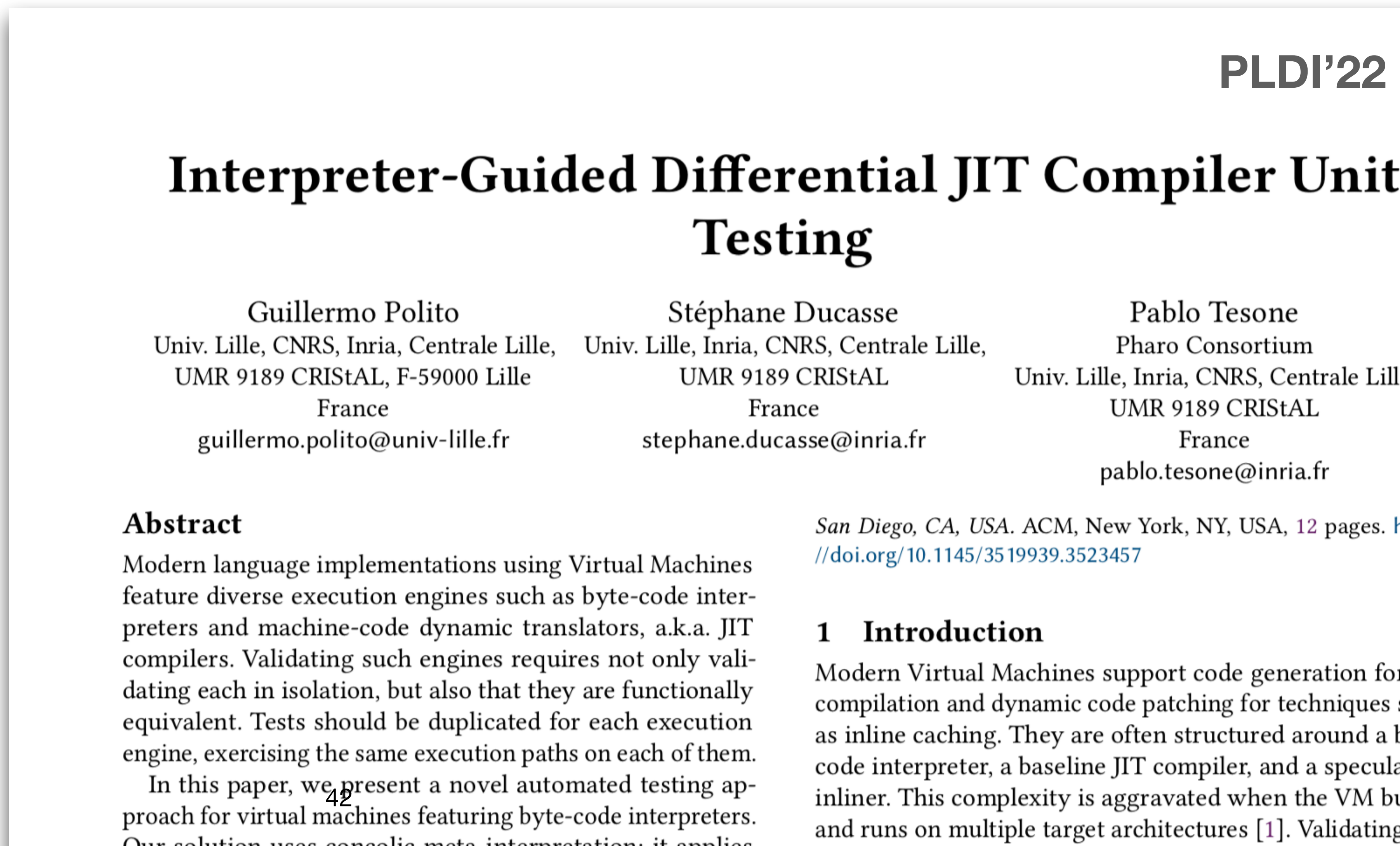
Practical and Cheap

- Test generation ~5 minutes
- Total run time of ~10 seconds
    - *Avg* 30ms per instruction



More in the PLDI article!

- Discovered Bugs
- Concolic Model
- Testing Infrastructure


PLDI'22

Interpreter-Guided Differential JIT Compiler Unit Testing

Guillermo Polito
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille
France
guillermo.polito@univ-lille.fr

Stéphane Ducasse
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
France
stephane.ducasse@inria.fr

Pablo Tesone
Pharo Consortium
Univ. Lille, Inria, CNRS, Centrale Lill
UMR 9189 CRIStAL
France
pablo.tesone@inria.fr

Abstract

Modern language implementations using Virtual Machines feature diverse execution engines such as byte-code interpreters and machine-code dynamic translators, a.k.a. JIT compilers. Validating such engines requires not only validating each in isolation, but also that they are functionally equivalent. Tests should be duplicated for each execution engine, exercising the same execution paths on each of them.

In this paper, we present a novel automated testing approach for virtual machines featuring byte-code interpreters.

San Diego, CA, USA. ACM, New York, NY, USA, 12 pages. h
//doi.org/10.1145/3519939.3523457

1 Introduction

Modern Virtual Machines support code generation fo
compilation and dynamic code patching for techniques
as inline caching. They are often structured around a b
code interpreter, a baseline JIT compiler, and a specula
inliner. This complexity is aggravated when the VM bu
and runs on multiple target architectures [1]. Validating

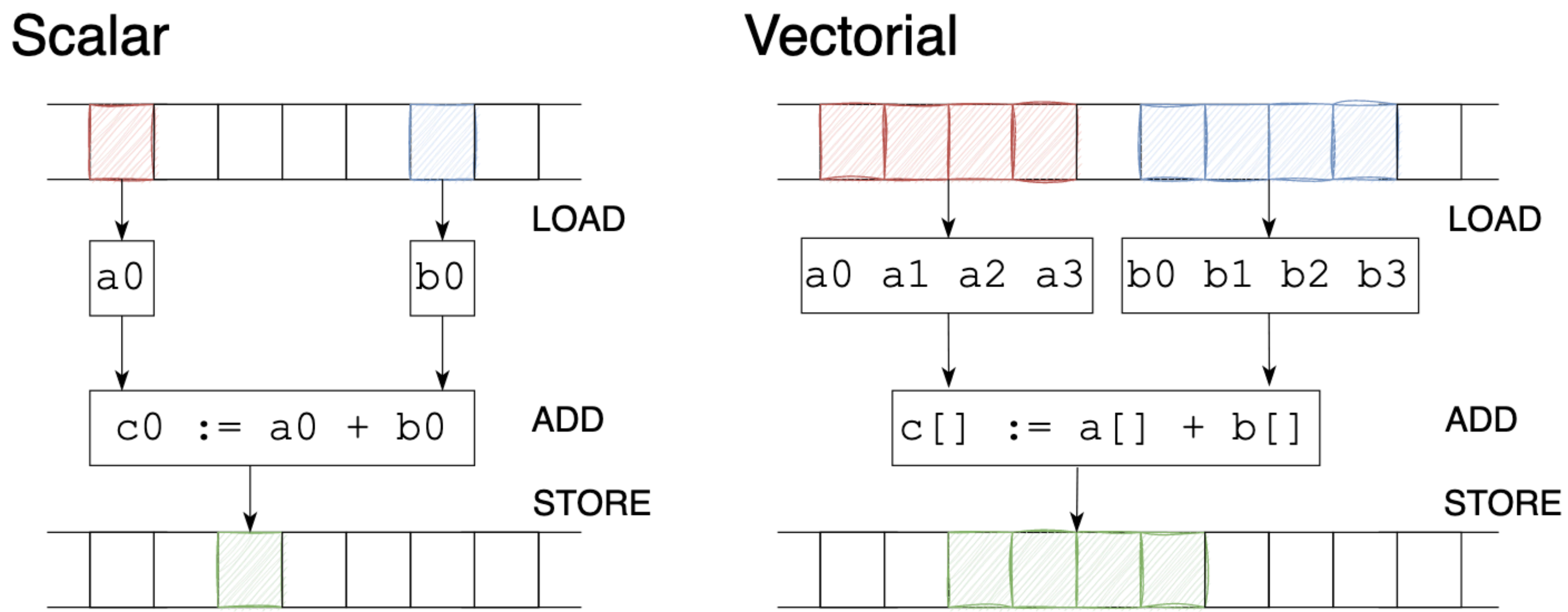
Is that all?

Ongoing RISCV64 Port

- Currently under development: **Real HW testing** stage
- Taking advantage of our harness test suite
- Improving tests and scenarios

- Collaboration with Q. Ducasse, P. Cortret, L. Lagadec from ENSTA Bretagne
- Future work on: *Hardware-based security enforcement*

Single Instruction Multiple Data Extensions

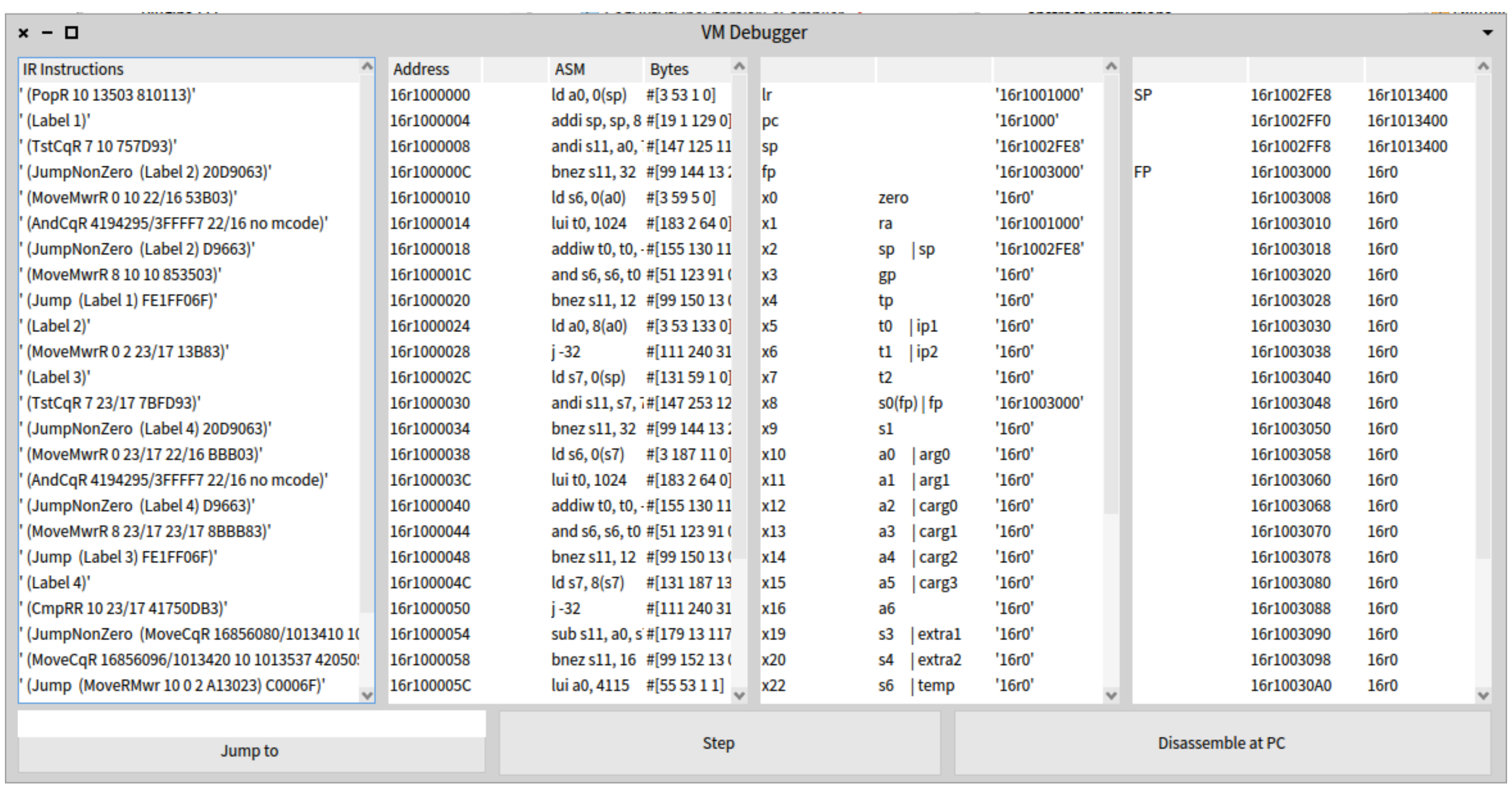


SIMD Design Space

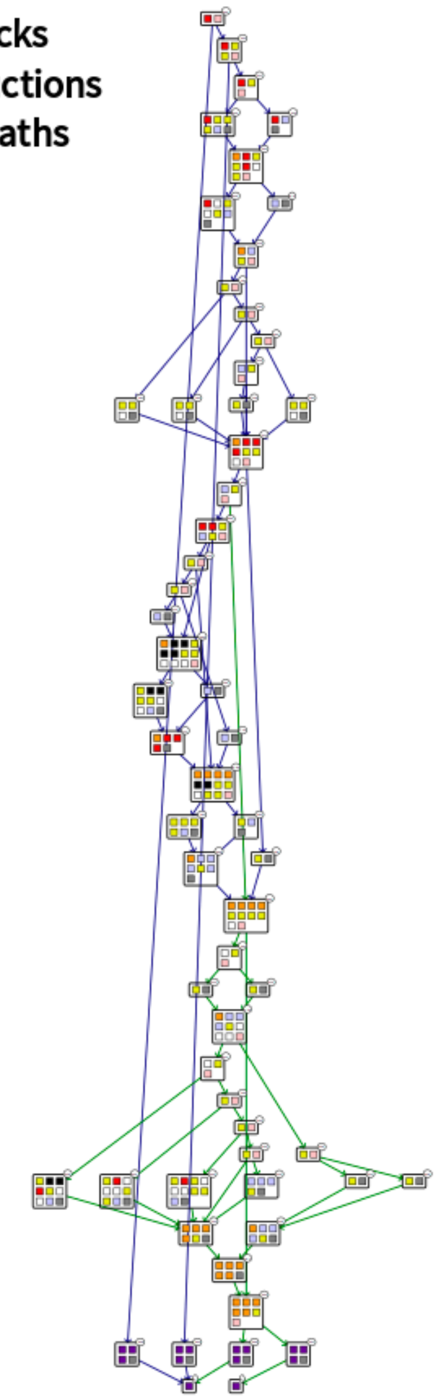
- VM Primitives
  - **Specialised**
  - Faster, less checks
- Vectorised Bytecode
  - **Composable**
  - Safe at the expense of speed

Tools for Debugging

- Machine Code Debugger
- Compiler IR Visualisations
- Disassembler DSL
- …



59 blocks
262 instructions
10082 paths



Pharo VM Manual Variable Localisation

```
interpret
 self fetchNextBytecode.
  [ true ] whileTrue: [
    self
      dispatchOn: currentBytecode
      in: BytecodeTable ].
  ^ nil
```

```
pushReceiverBytecode
  self fetchNextBytecode.
  self internalPush: self receiver
```

```
pushBool: trueOrFalse
   <inline: true>
  self push: (objectMemory booleanObjectOf: trueOrFalse)
```

```
internalAboutToReturn: resultOop through: aContext
   <inline: true>
   […]
  self internalPush: resultOop
  […]
```

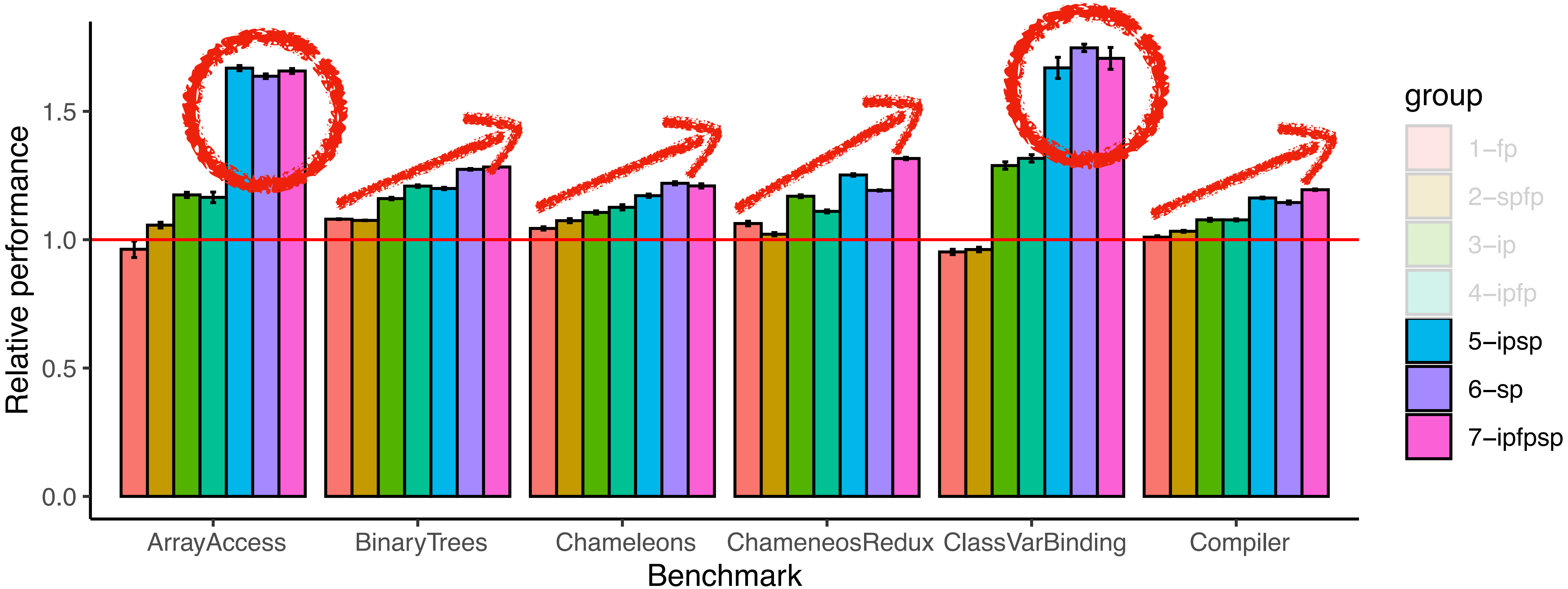
```
internalPush: aValue
  localSP := localSP - bytesPerWord.
  self longAt: localSP put: aValue
```

```
push: aValue
  stackPointer := stackPointer - byte
  self longAt: stackPointer put: aVal
```

Developer should know C generation semantics

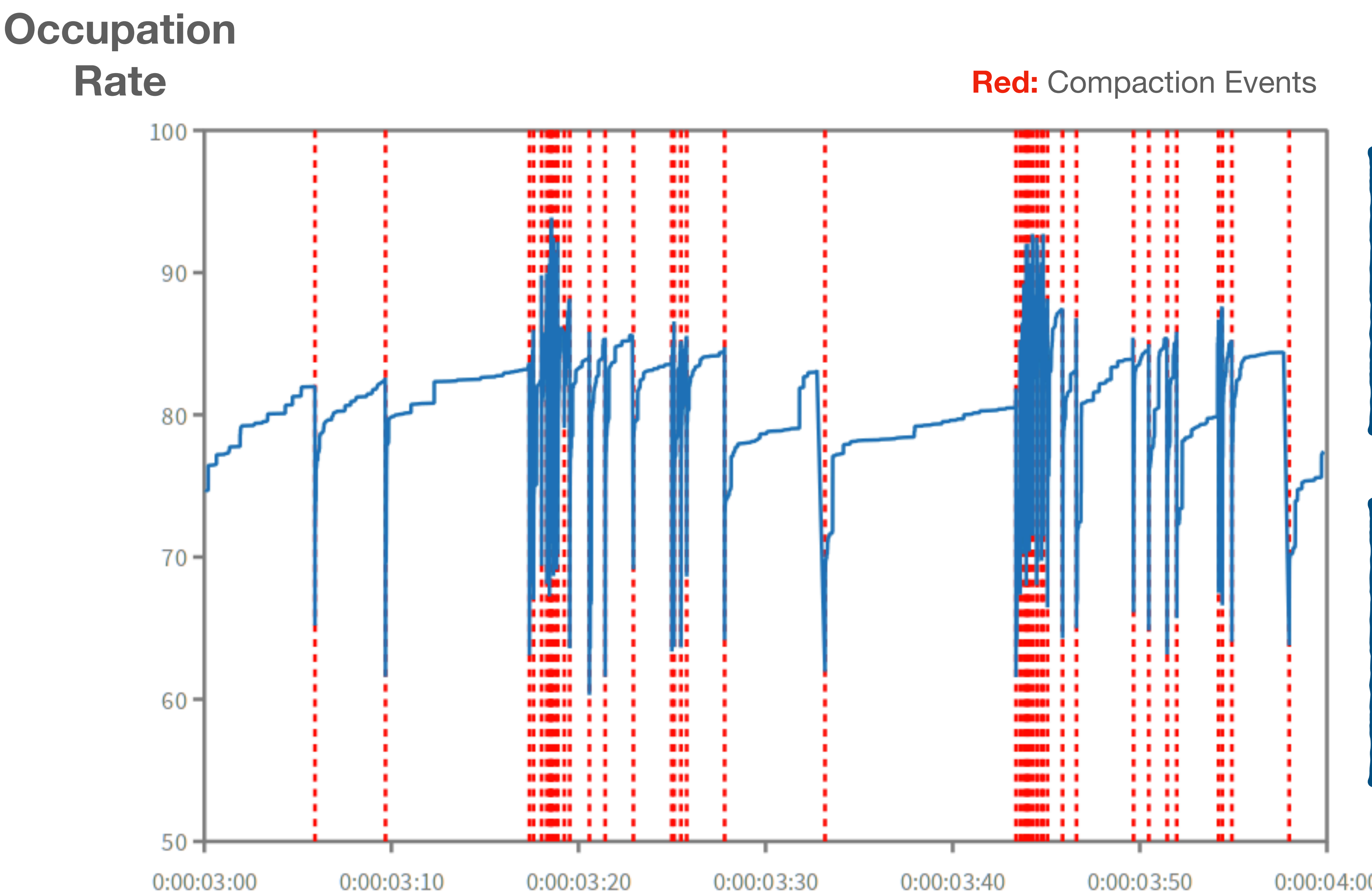
Automatic Variable localisation!

Intel x86-64



Averages of 100 iterations + stdev. Relative to baseline (no optimisation). Higher

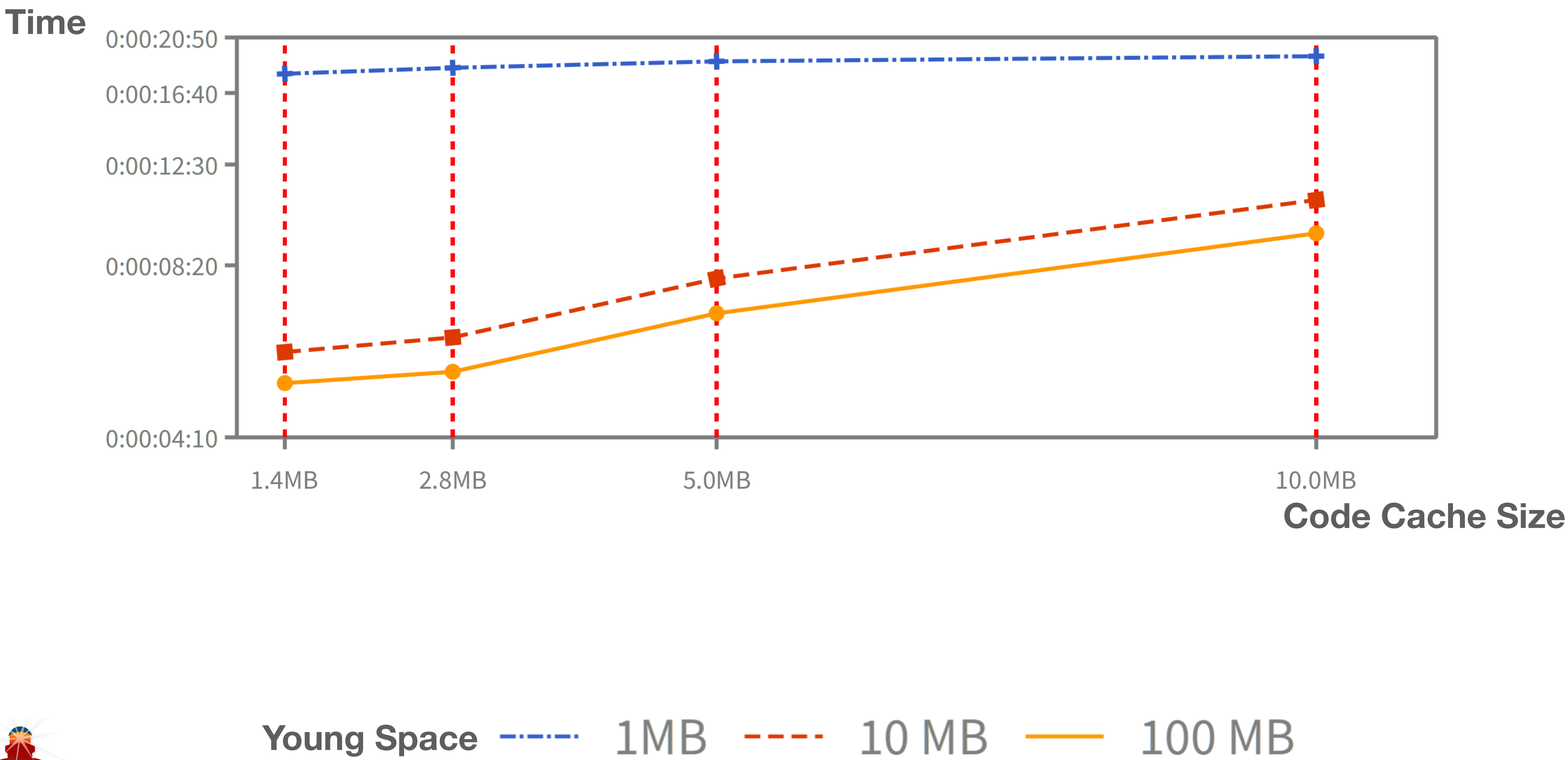
Analysing Code Cache Behavior



Analysing Events
We see trashing
in the code

We need to
increase the size
of the code

Code Cache Unexpected results



Loading Moose

We are hiring!

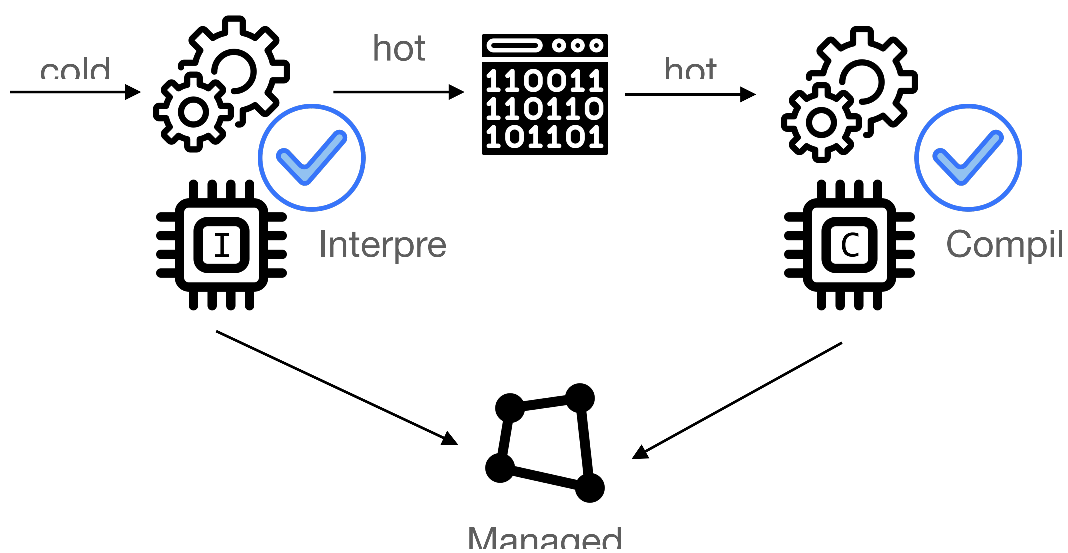
- We have
  - Engineer Positions
  - Phd Positions

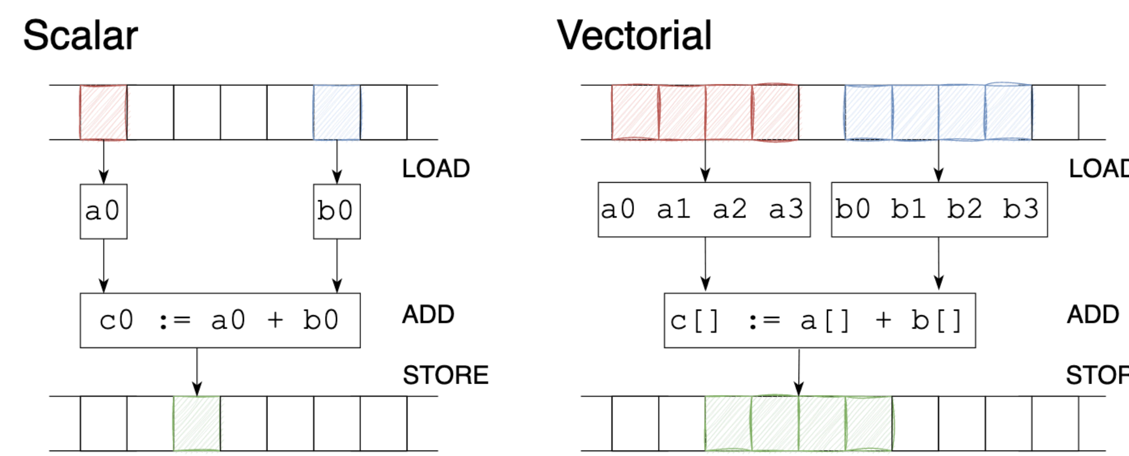
- Keywords: *Compilers, Interpreters, Memory Management, Security*

- **Come talk to us!**

Pharo VM - News from the Front

Université de Lille





RMod

Stay in Sync!

Permanent Space

Ephemerons

New Image Format

Lifeware

Faster Startup / Saving

Speculative Compilation

guillermo.polito@univ-lille.fr

@pharoproject

pharo.org

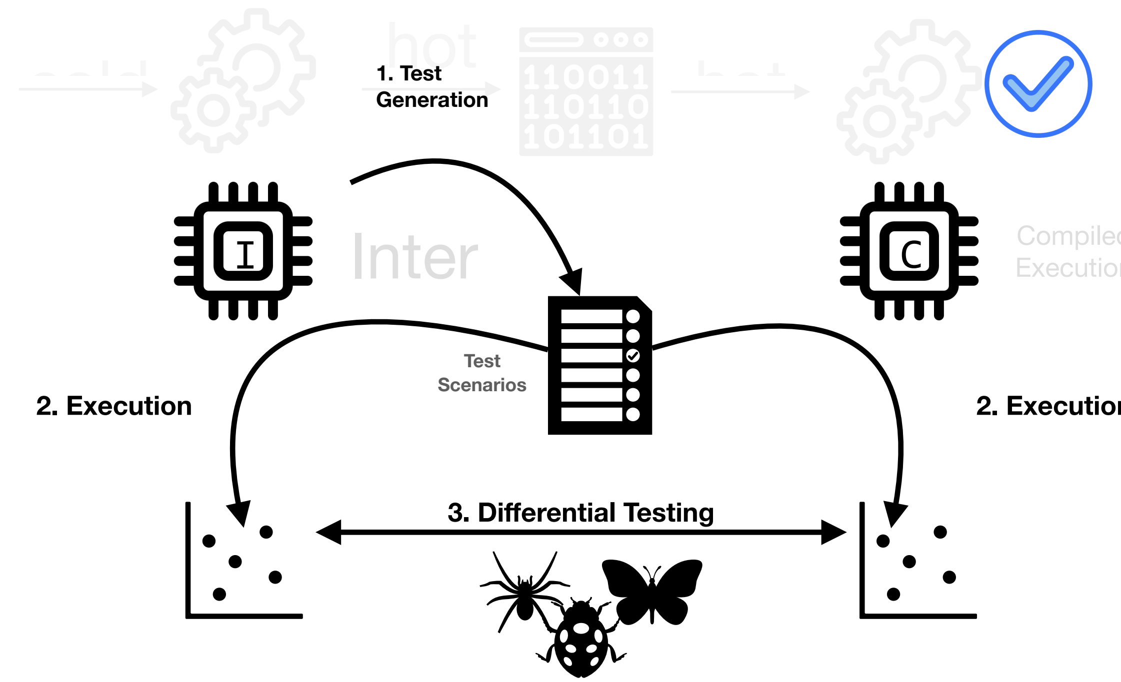
consortium-adm@pharo.org

discord.gg/QewZMZa

thepharo.dev

Conclusion

- 478 differences found, 91 causes, 6 categories
- Practical:
    - 4928 tests generated in ~8 minutes
    - 4928 tests run in ~40 seconds



Guille Polito - Pablo Tesone - Stéphane Ducasse
guillermo.polito@univ-lille.fr
@guillep

Improvements: Clean Up

- V3 Support
- Old Memory Format
- Old Block Closures
- Dead Code
- ~ 65KLOC

Improvements: Sockets

- Unified Implementation in all Platforms
- Better Async Support
- Unix Sockets (Under Work)
- IPv6 Addresses (Under Work)

Improvements: Serial Port FFI

- Pure FFI implementation
- Working in all Platforms (Unix / Windows / OSX)
- Migrating Plugins to FFI

Improvements: RISCV64

Ongoing Port

- Currently under development: Real HW testing stage
- Taking advantage of our harness test suite.
- Improving tests and scenarios

- Collaboration with Q. Ducasse, P. Cortret, L. Lagadec from ENSTA Bretagne
- Future work on: Hardware-based security enforcement

Improvements: Open Build Service

Better Support for Linux Distributions

	Arch	Debian_10	Debian_9.0	Debian_Testing	Fedora_31	Fedora_32	Fedora_33	Raspbian_10		Raspbian_9.0	
	x86_64	x86_64	x86_64	x86_64	x86_64	x86_64	x86_64	aarch64	x86_64	aarch64	x86_64
libffi7		succeeded	succeeded		succeeded	succeeded	succeeded	succeeded	succeeded	succeeded	succeeded
libgit2-1		succeeded		failed							
pharo9	failed	succeeded	failed	failed	failed	failed	failed	succeeded	succeeded	failed	failed
pharo9-ui	succeeded	succeeded	succeeded	failed	succeeded	succeeded	succeeded		succeeded		succeeded

	Raspbian_9.0		openSUSE_Leap_15.1	openSUSE_Leap_15.2	openSUSE_Tumbleweed	xUbuntu_18.04	xUbuntu_19.04	xUbuntu_20.04	
	arch64	x86_64	x86_64	x86_64	x86_64	x86_64	x86_64	aarch64	x86_64
libffi7	:ceeded	succeeded	succeeded	succeeded	succeeded	succeeded	succeeded	succeeded	succeeded
libgit2-1			succeeded	succeeded		succeeded	succeeded		succeeded
pharo9	failed	failed	failed	failed	failed	failed	succeeded	succeeded	succeeded
pharo9-ui		succeeded	succeeded	succeeded	succeeded	succeeded	succeeded		succeeded

Initial targets:

- Arch / Manjaro
- Debian
- Fedora
- Raspbian
- Ubuntu
- openSuse

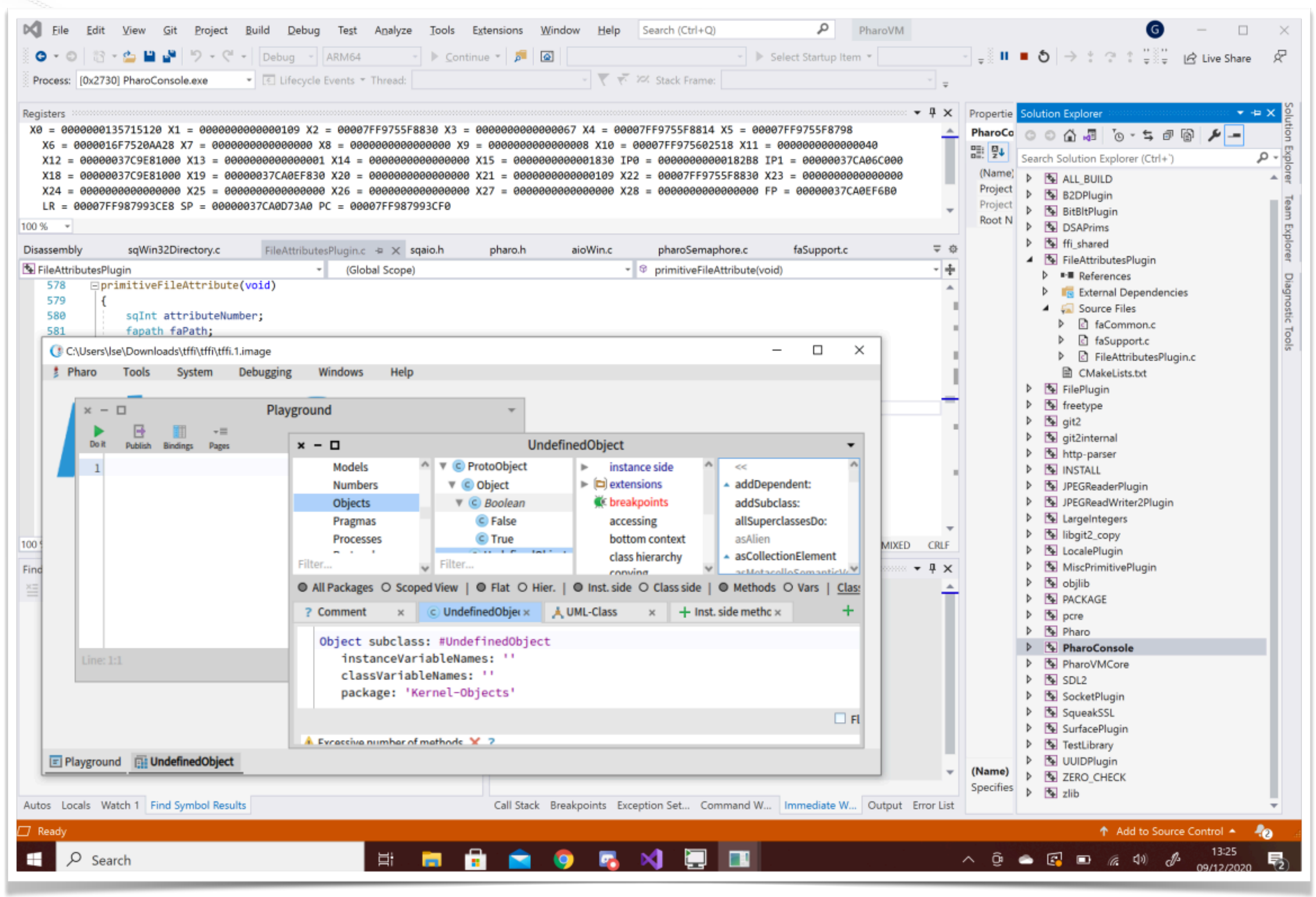
Multiple Architectures

Supporting system

Building using existing system

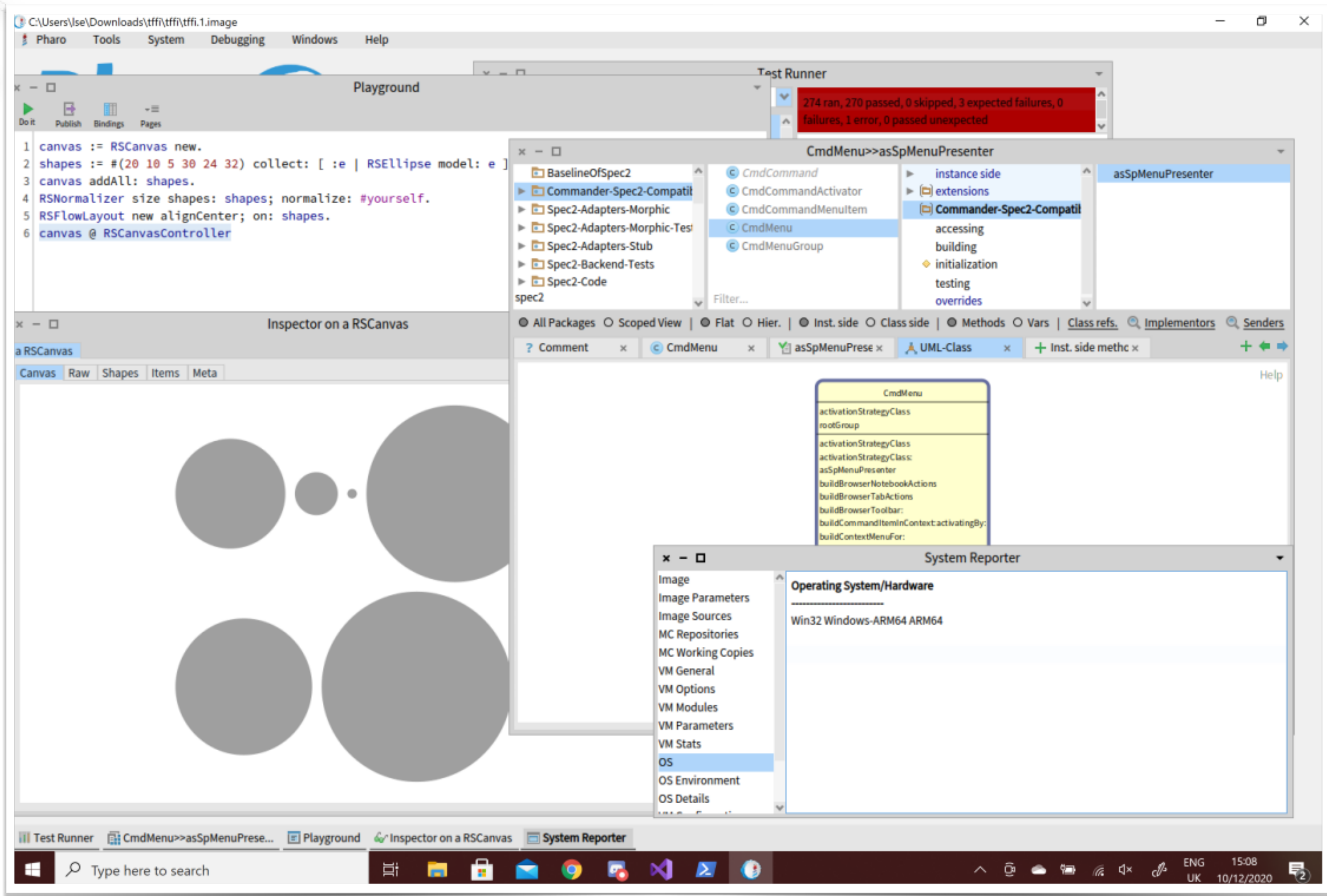
Improvements: Visual Studio Support

Building & Debugging



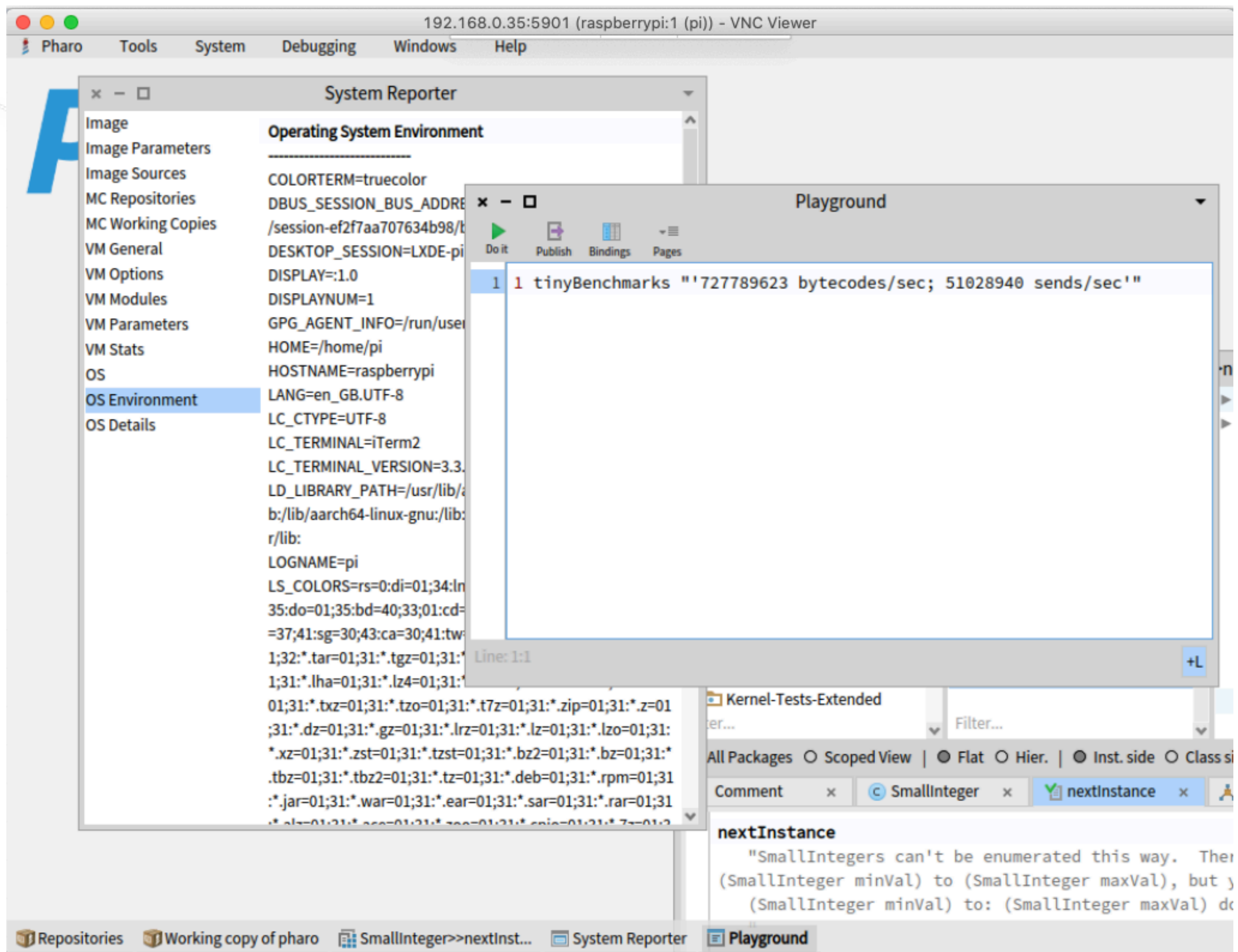
MSVC - No cygwin

Improvements: Windows ARM



MSVC - No cygwin

Improvements: Raspbian 32/64 bits



Back to the Future

Objectives for 2022

Permanent Space

Problem

Lifeware

Generates Long Pauses

- Many permanent objects
- They have references from/to other objects
- We are traversing them to GC
- E.g., Classes, Methods, Literals, Resources

Permanent Space

Our Solution

- New Object Space for permanent Objects
- Minimise or Eliminate GC passes
- Persisting them through executions

Permanent Space

Our Solution

- New Object Space for permanent Objects
- Minimise or Eliminate GC passes
- Persisting them through executions

We need to put them in a

New Image Format

Problem

- Current Image format only support a single object space
- No extensible: not new metadata nor new data
- Cannot be Memory Mapped (it is modified before save/load)
- Requires to discard all state of the running VM (slow saves)

New Image Format

Problem

- Current Image format only support a single object space
- No extensible: not new metadata nor new data
- Cannot be Memory Mapped (it is modified before save/load)
- Requires to discard all state of the running VM (slow saves)

Slow and Restricting

New Image Format

Our Solution

- New Image format based in directories / bundles
- Many Elements of data and metadata
- Metadata en User & Machine readable format (STON?)
- Extensible format

Fast Snapshots / Loading

Based on PermSpace & Image Format

- Memory Mapped Image
- Shareable State
- Saving / Loading Warm State of the VM

Next Objectives

Permanent Space

Ephemerons

New Image Format

Speculative Compilation

Faster Startup / Saving

- ARM64, RISCV64, Slang…
- Lots of Tests!
- Integration: Sockets, serial
- Visual Studio, Open Build Service

@pharoproject
pharo.org
consortium-admin@pharo.org
discord.gg/QewZMZa
thepharo.dev

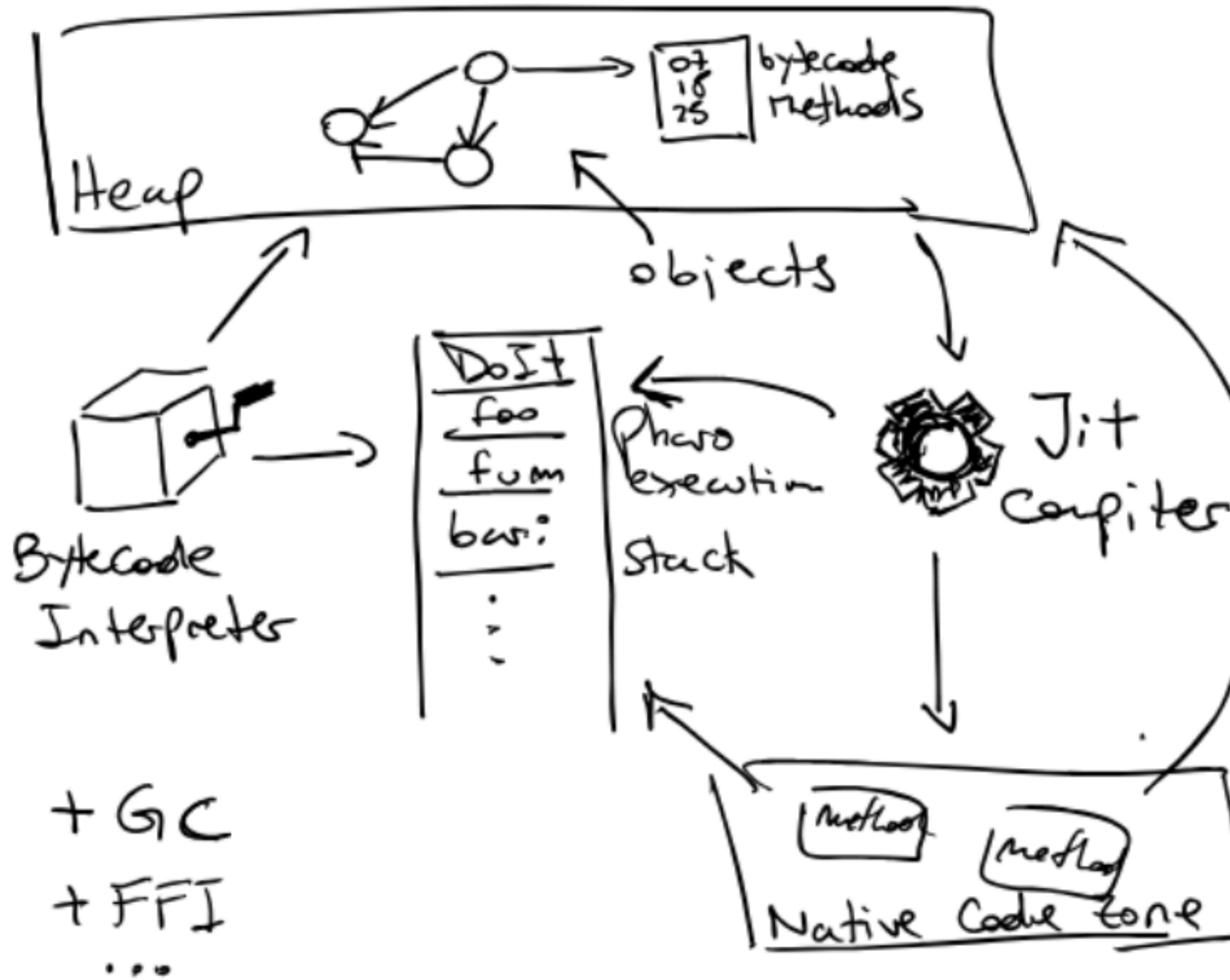
Cross-ISA Testing of the Pharo VM

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 stack based bytecodes (77 different) + ~340 primitives/ native methods

Lots of combinations!

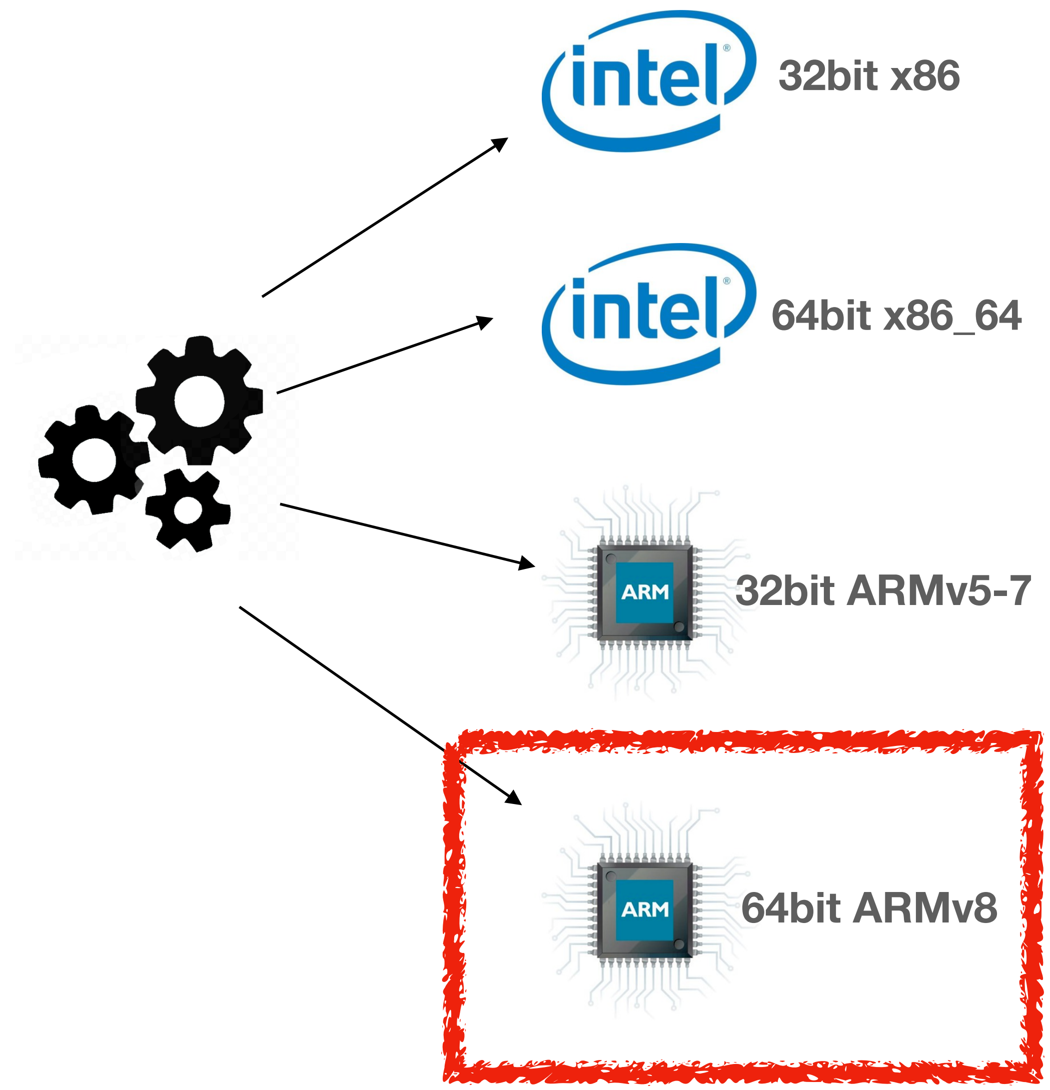
- 146 different IR instructions
- polymorphic inline caches
- threaded code interpreter
- generational scavenger GC

Objective: Implementing an ARM64 Backend

- ARM64 is now pervasive
  - New Apple M1
  - Raspberry Pi 4
  - Microsoft Surface Pro
  - PineBook Pro
  - …

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

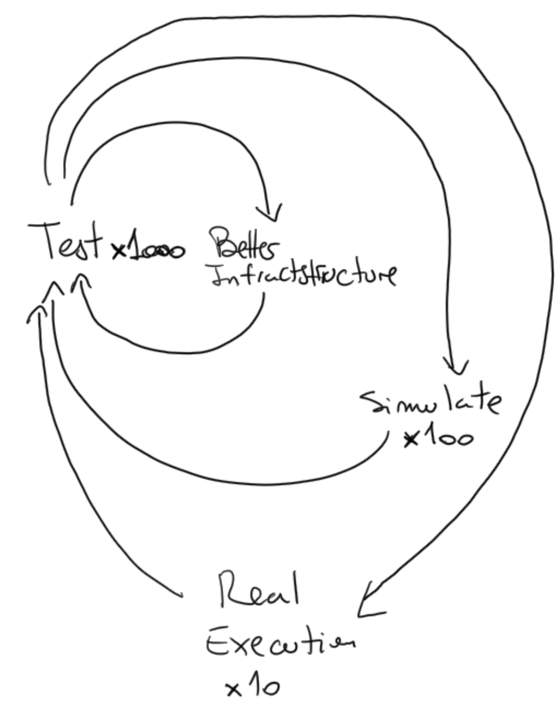
JIT compiler IR



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

ENSTA BRETAGNE

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

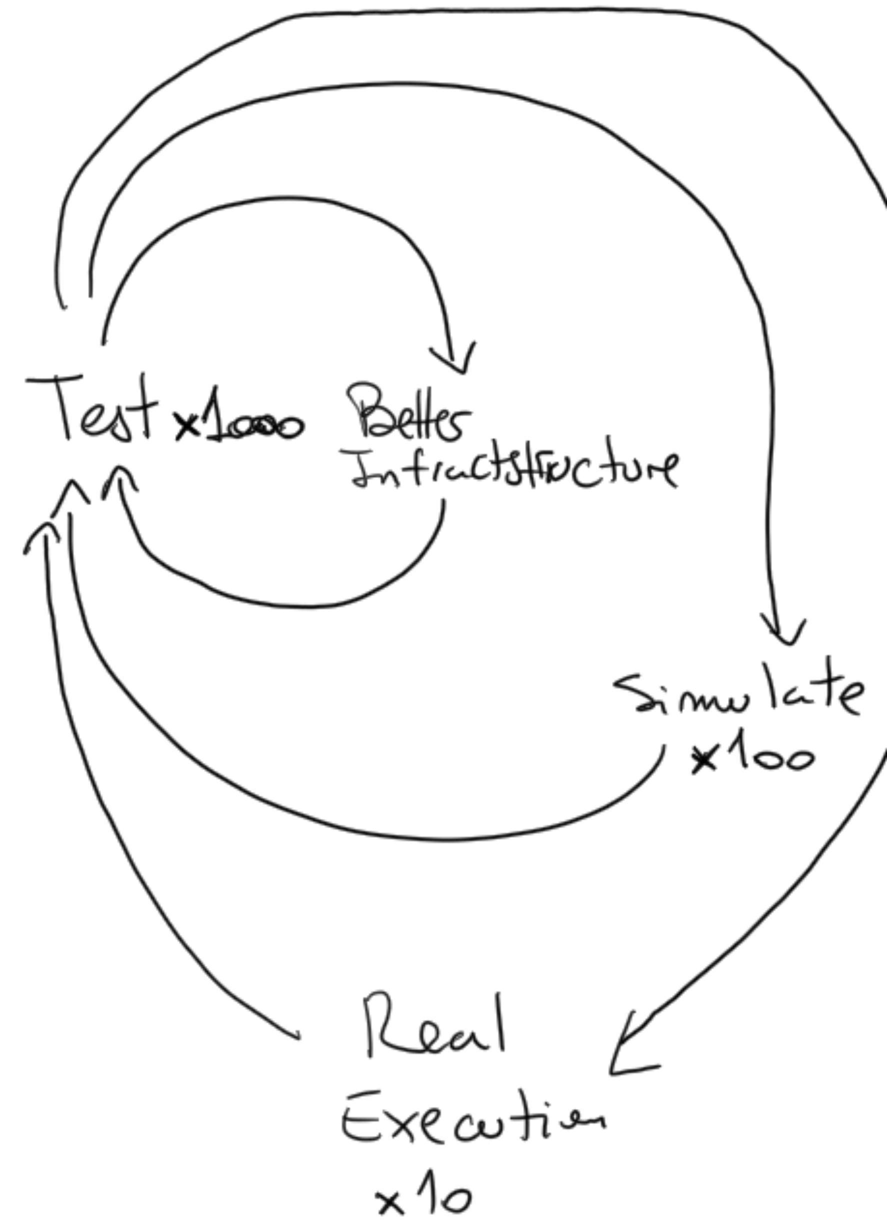
Automatic VM Validation

- 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

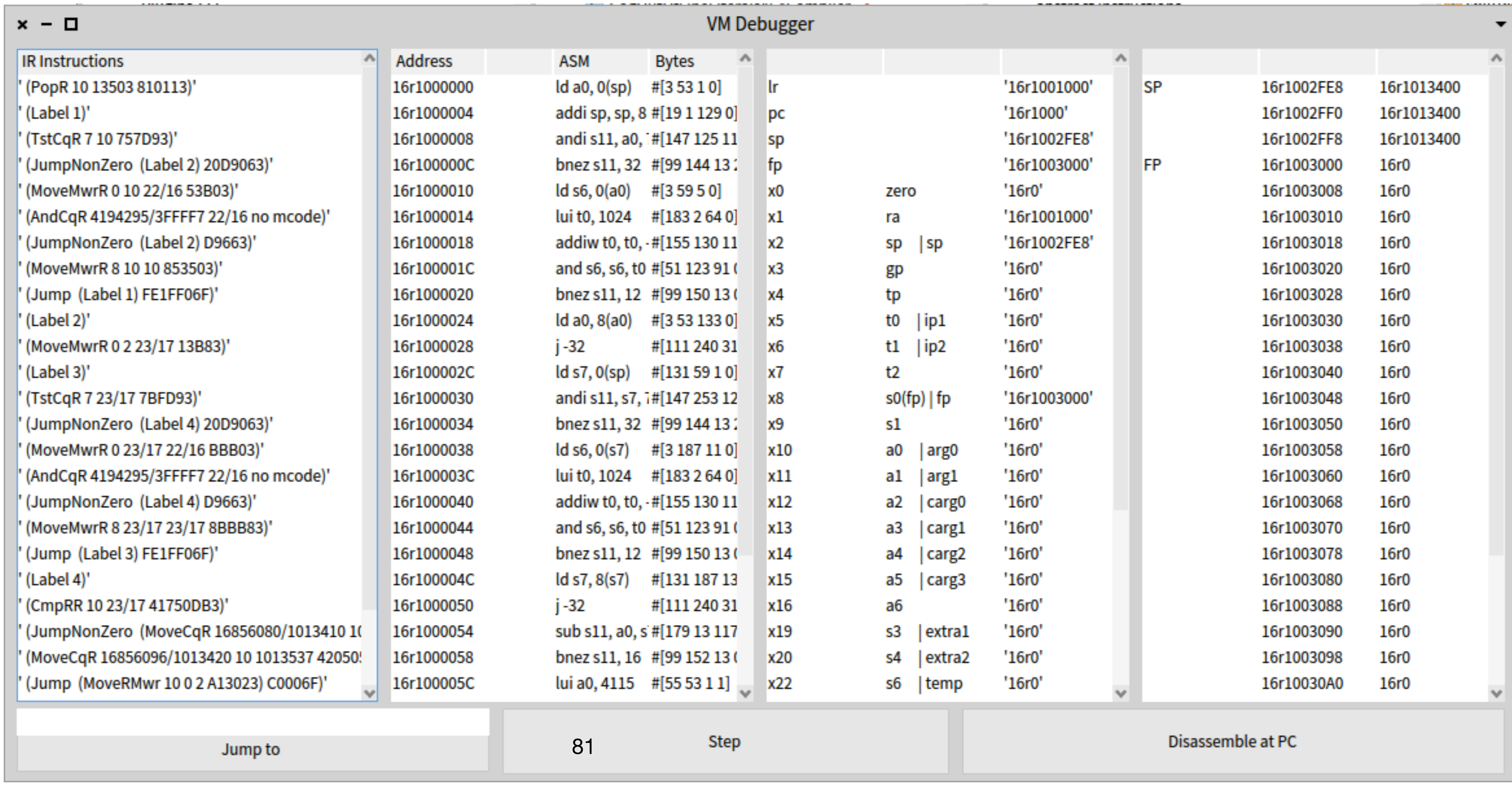


Debugging a compiler

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

Examples:

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



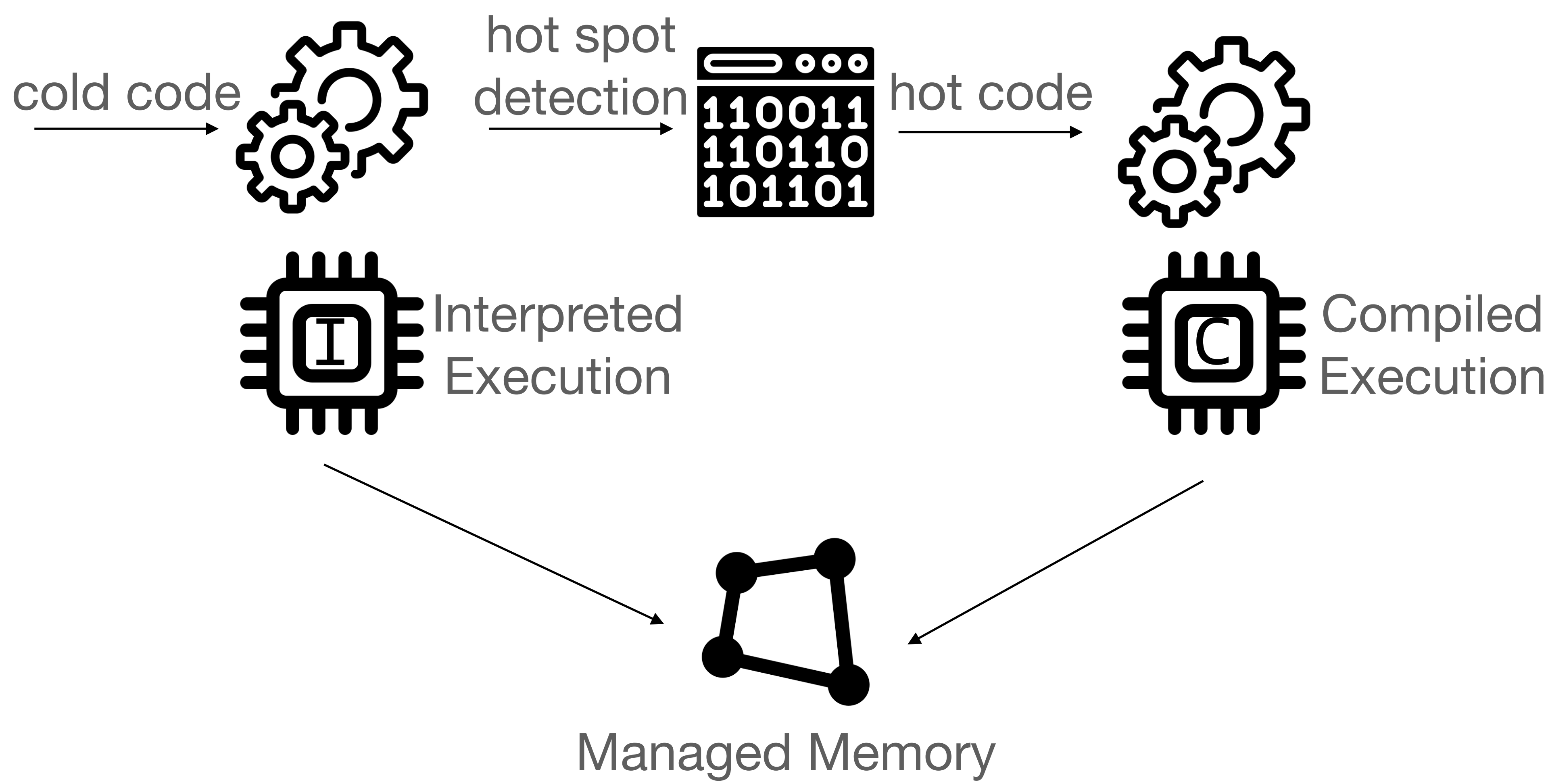
Interpreter-Guided JIT Compiler Test Generation

Validating the Pharo JIT compiler through **concolic** execution and **differential** testing

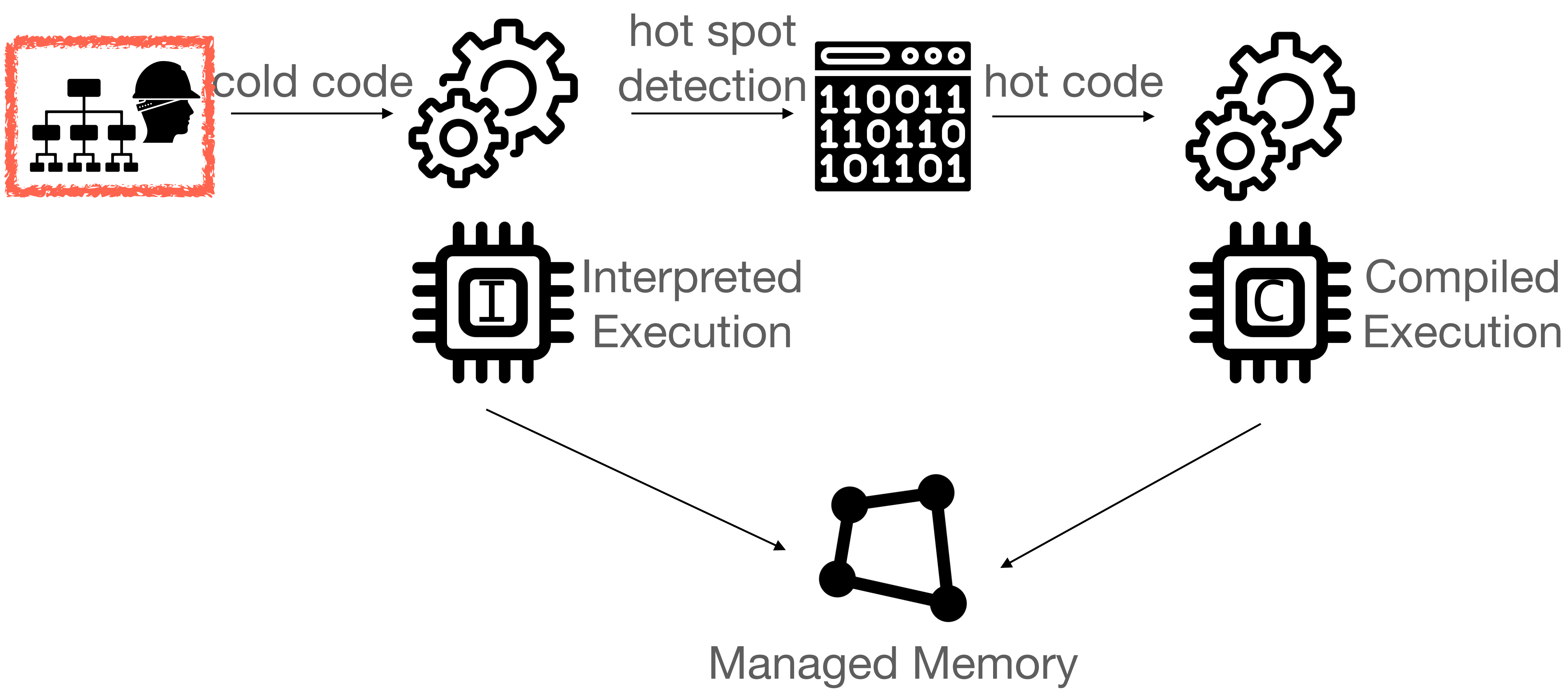
Guille Polito - Pablo Tesone - Stéphane Ducasse
guillermo.polito@univ-lille.fr
@guillep

PLDI'22 — San Diego

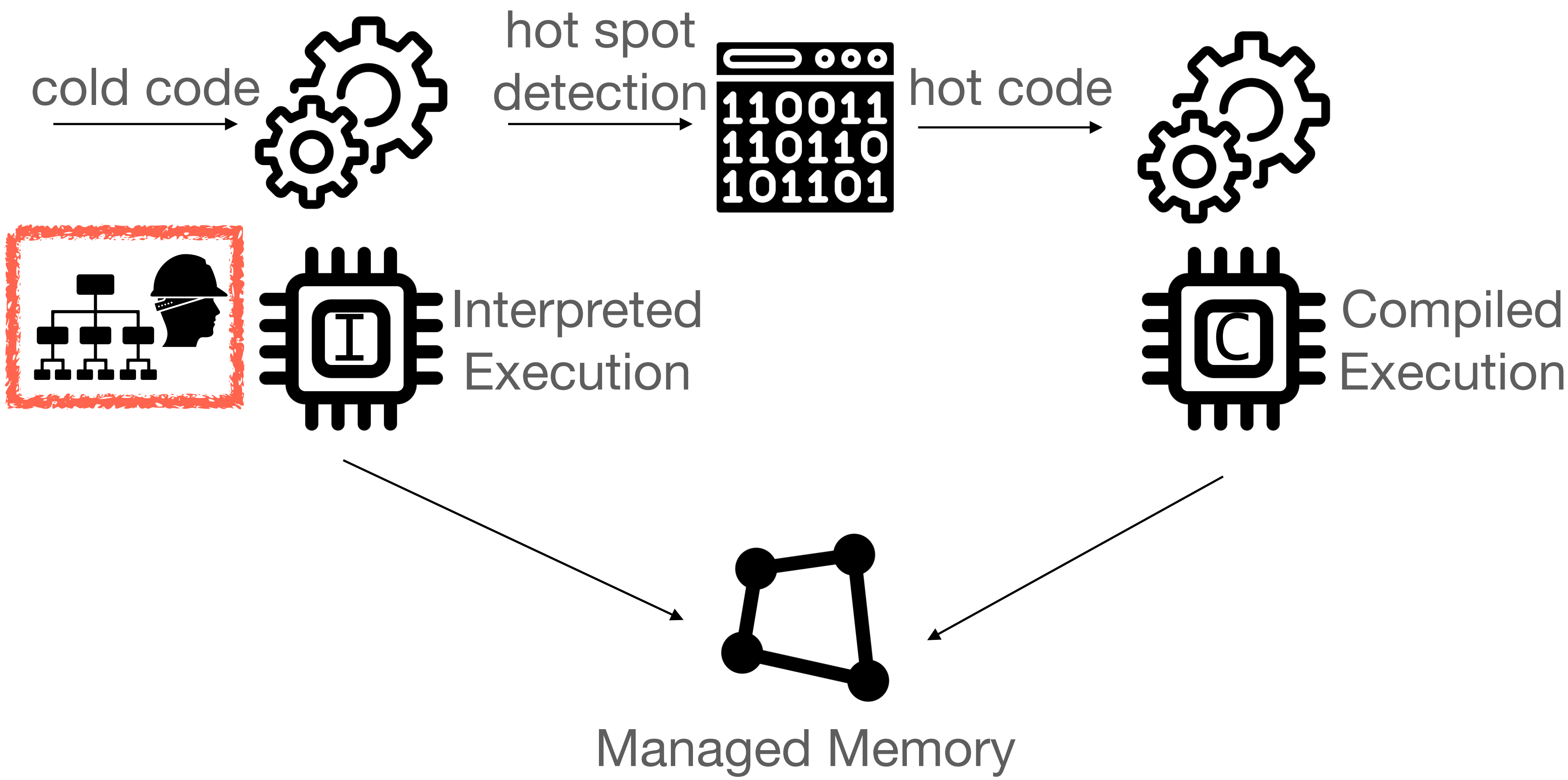
Virtual Machine Execution Engine



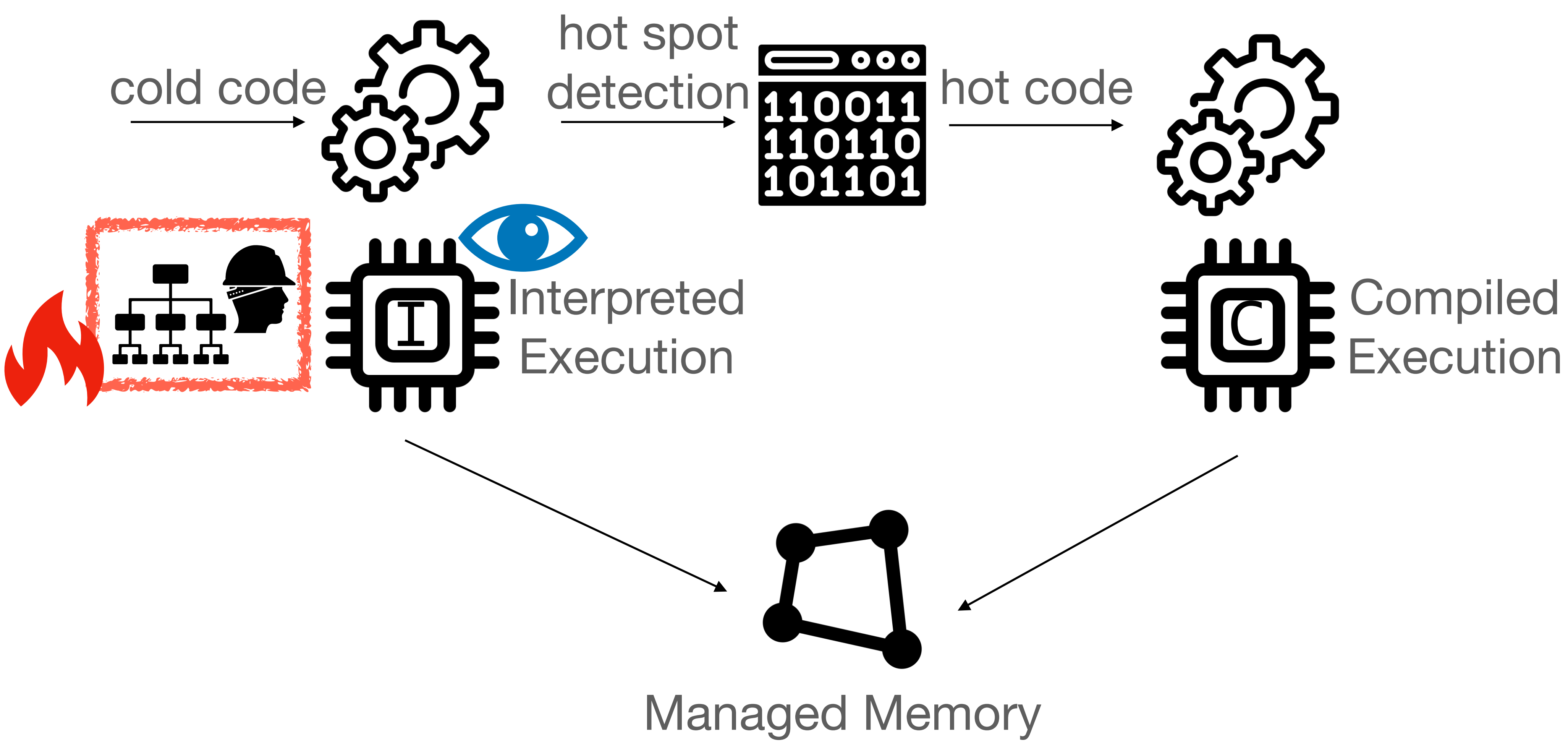
Virtual Machine Execution Engine



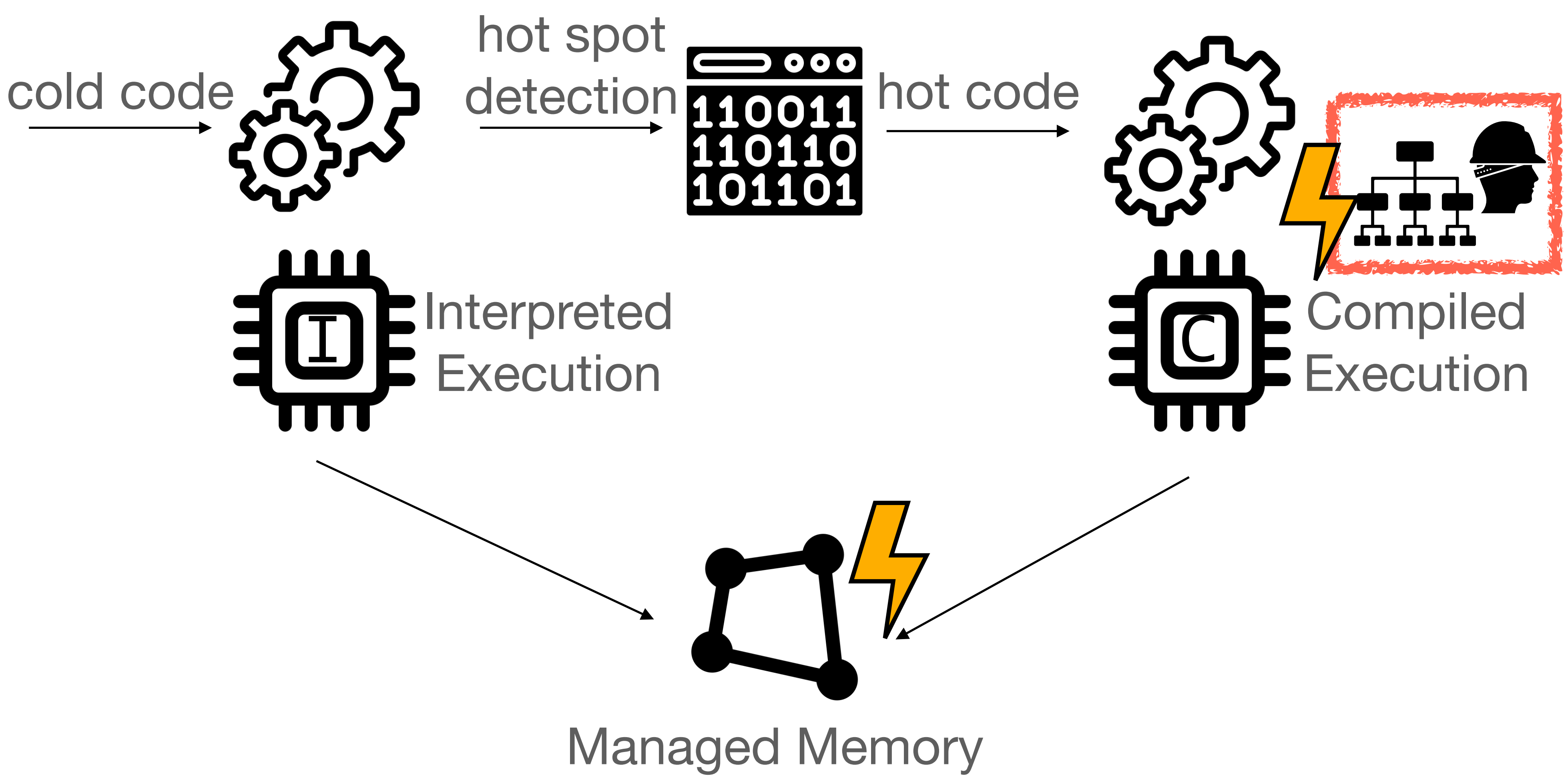
Virtual Machine Execution Engine



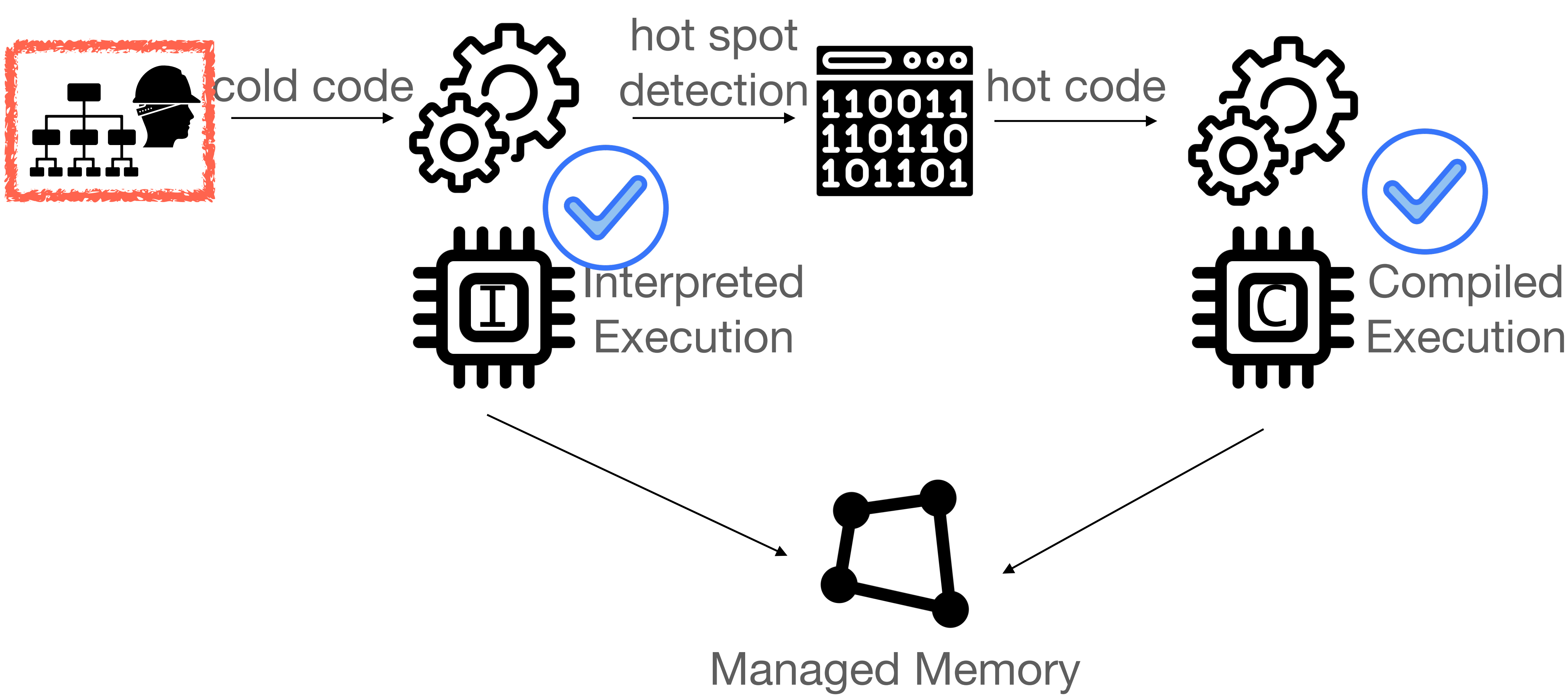
Virtual Machine Execution Engine



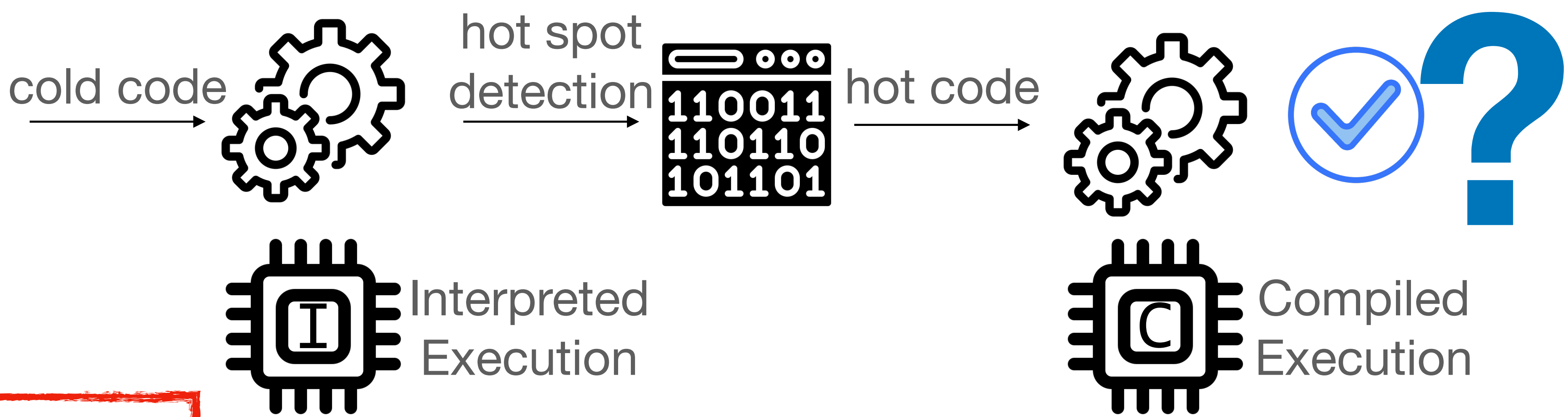
Virtual Machine Execution Engine



How can we automatically test VMs?



Challenges of VM Test Generation

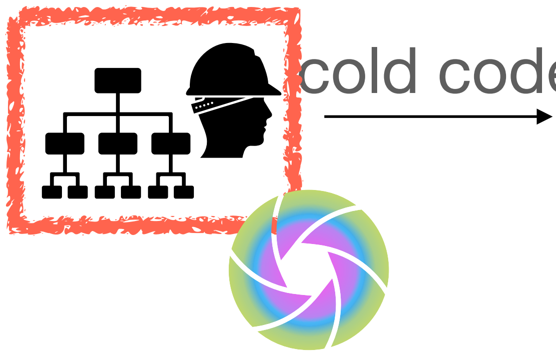


Challenge 1: Test

Challenge 2: Test

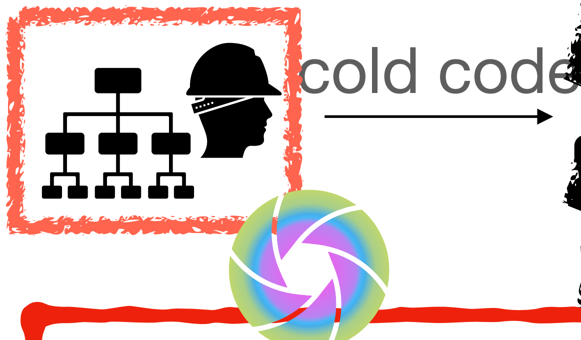
- Do they cover different code ***regions/branches/ paths***?
- How do we determine what is the ***expected output*** of a generated test?

Black Box Testing + Fuzzing



[redacted]

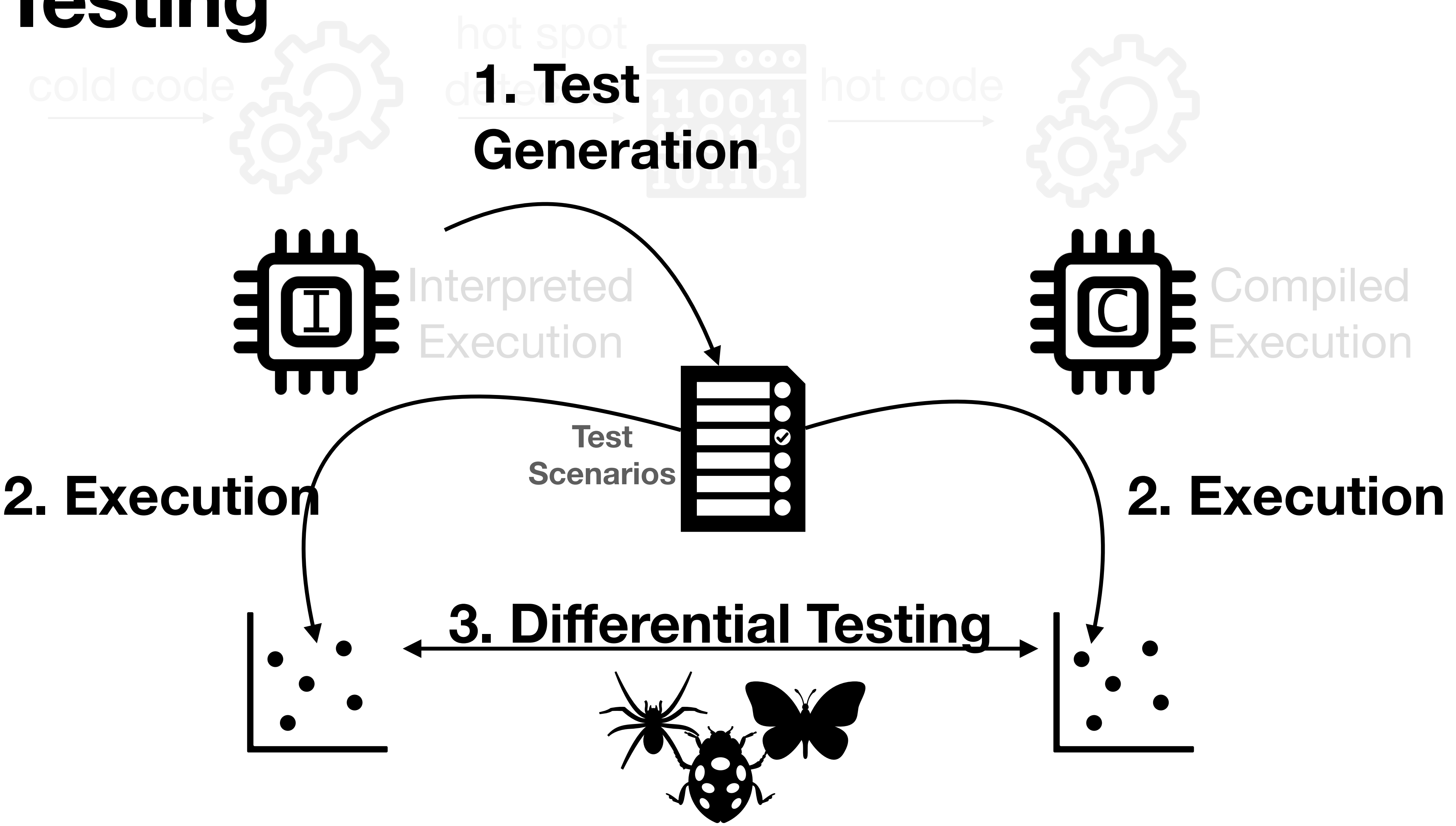
Black Box Testing + Fuzzing



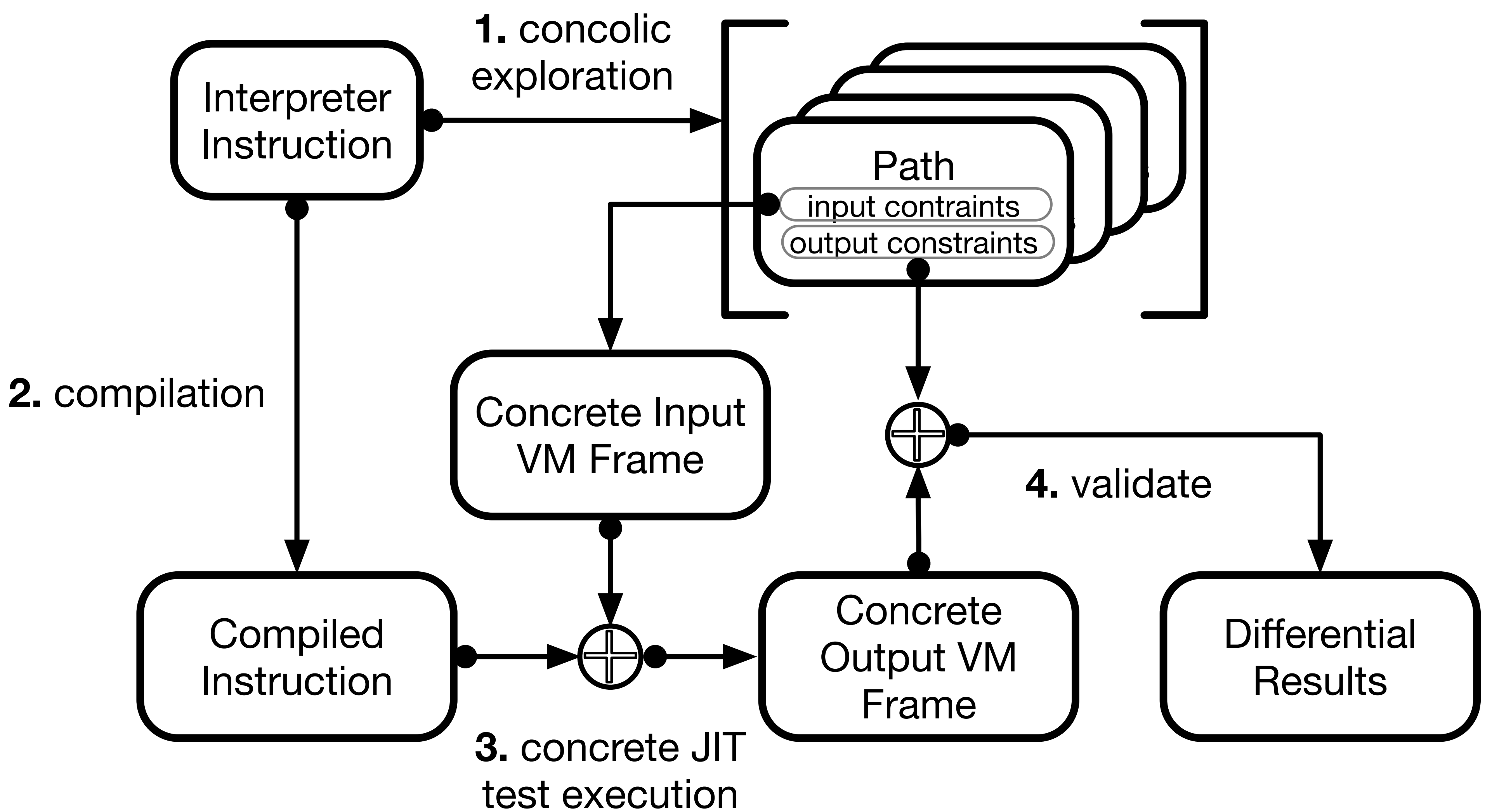
[redacted]

- ✗ Slow
- ✗ Coarse Grained
- ✗ Non Determinism
- ✗ Require Multiple Reference Implementations

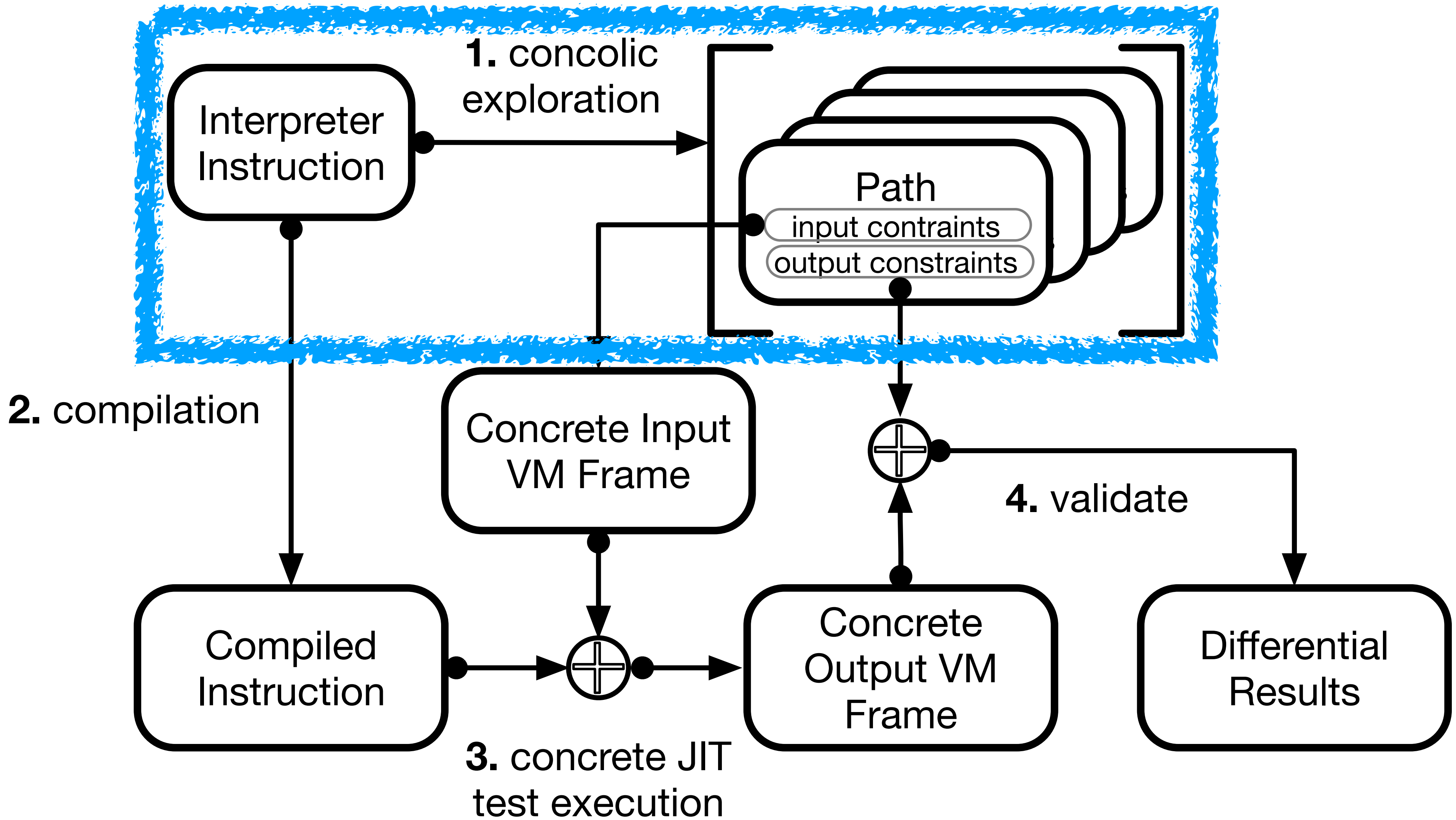
Interpreter-Guided Automatic JIT Compiler Unit Testing



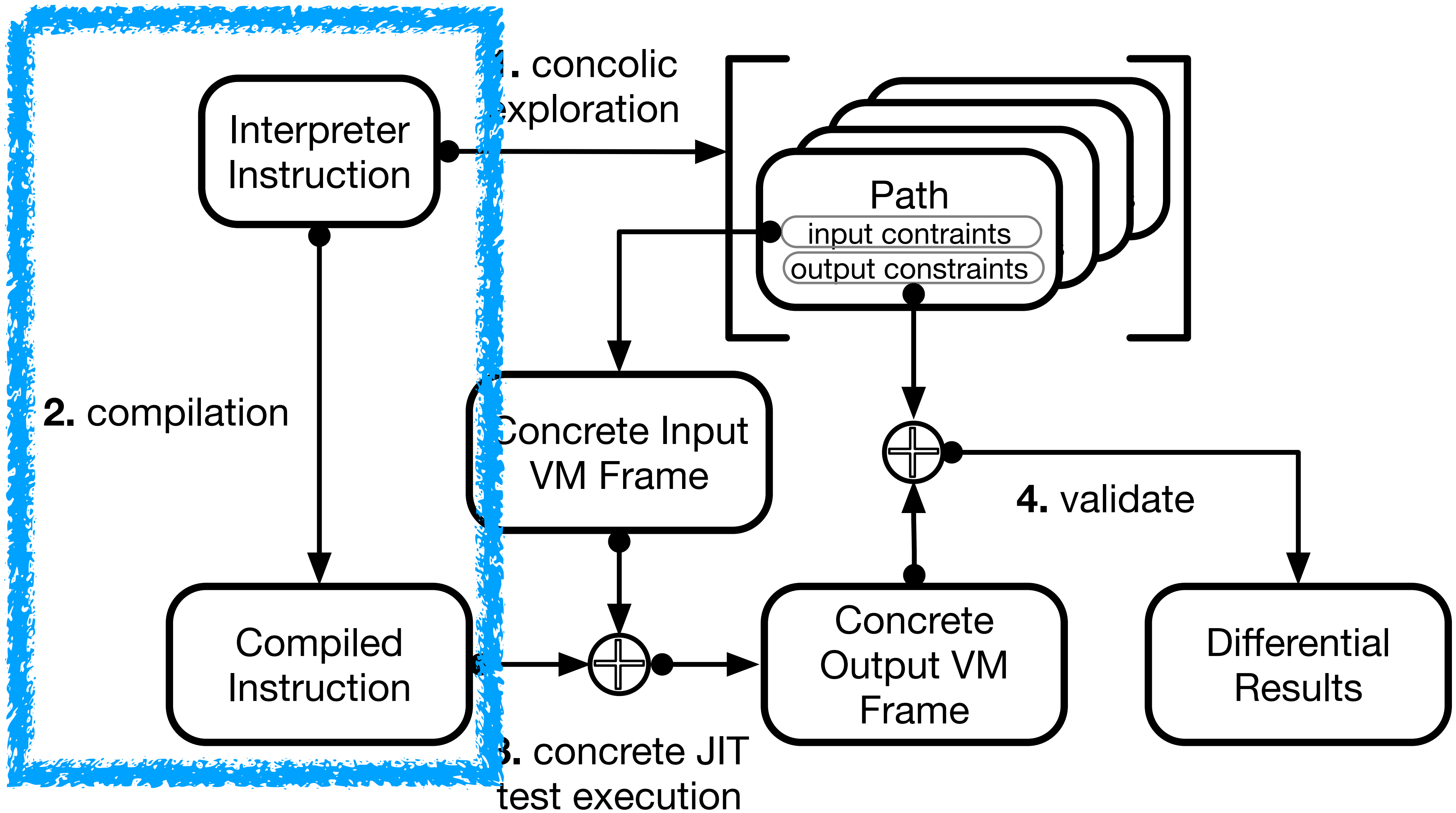
Implementation View



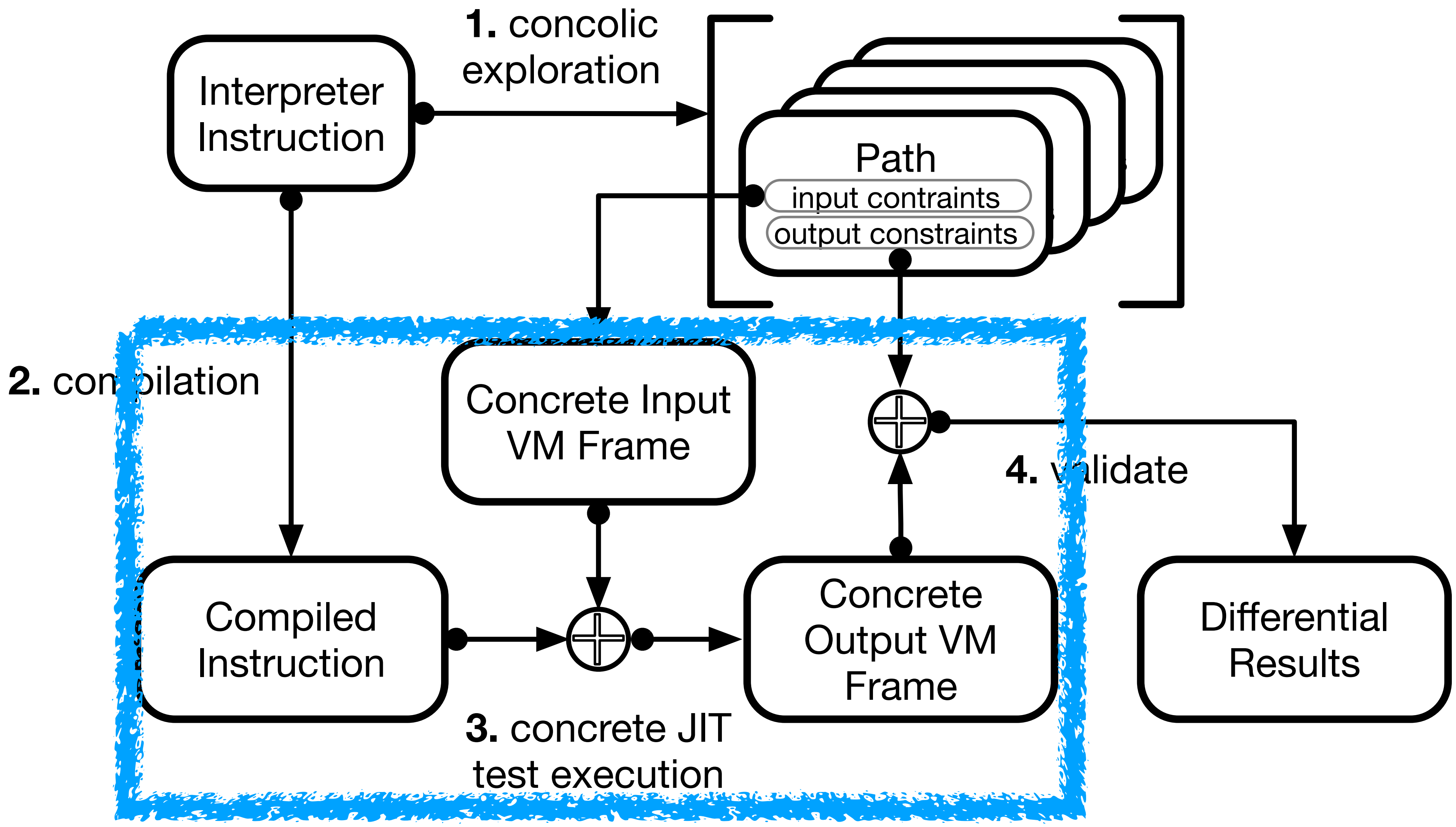
Implementation View



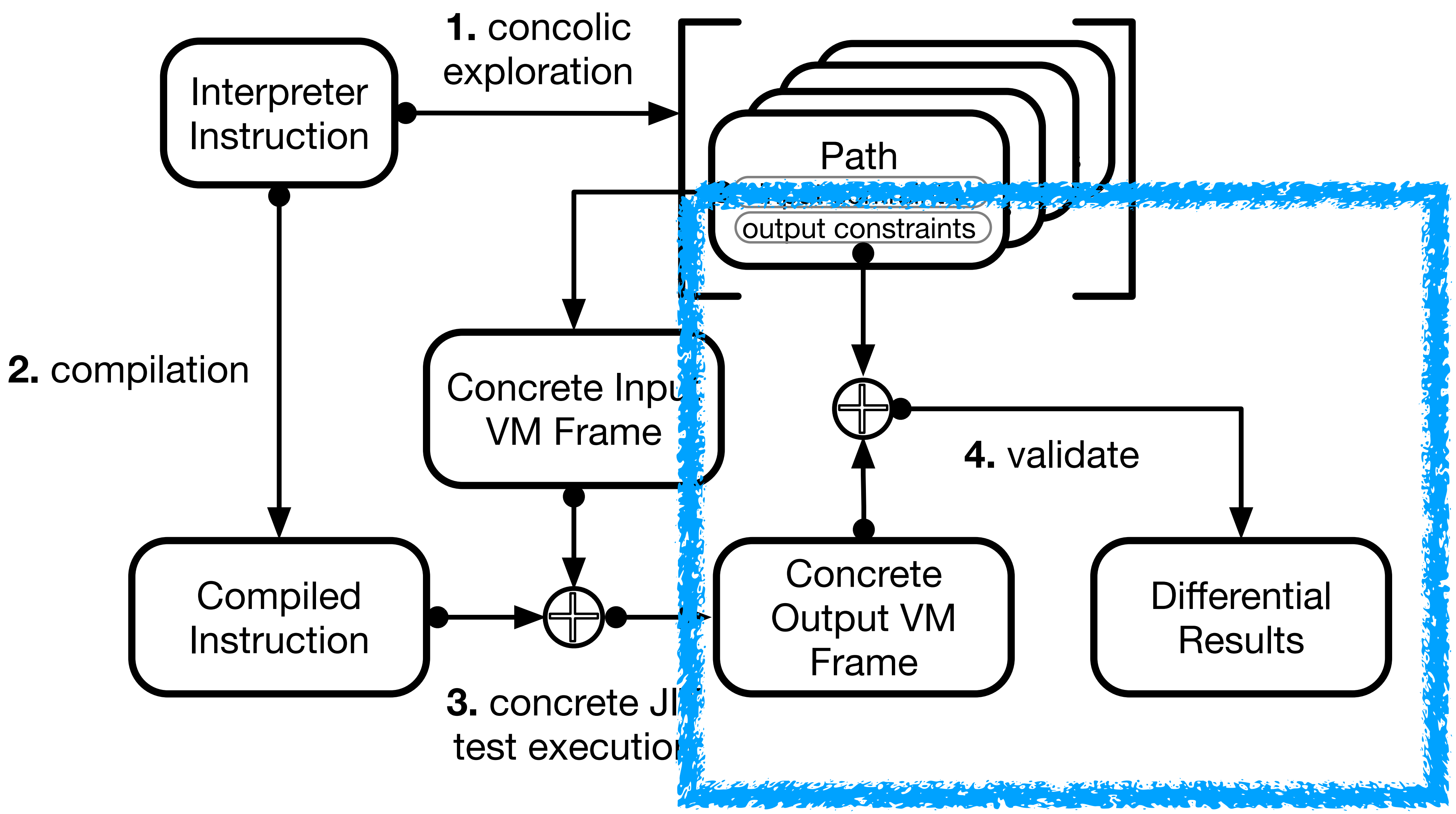
Implementation View



Implementation View

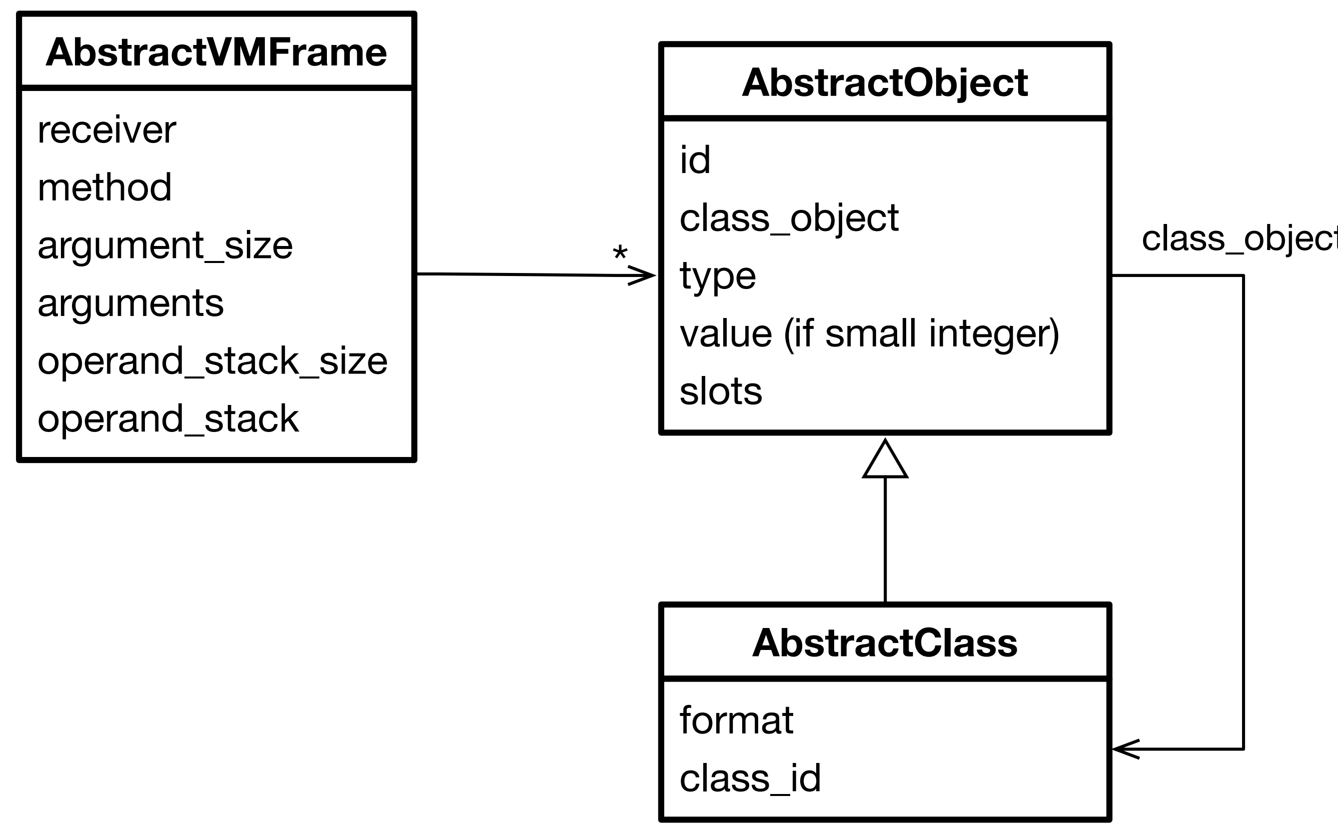


Implementation View



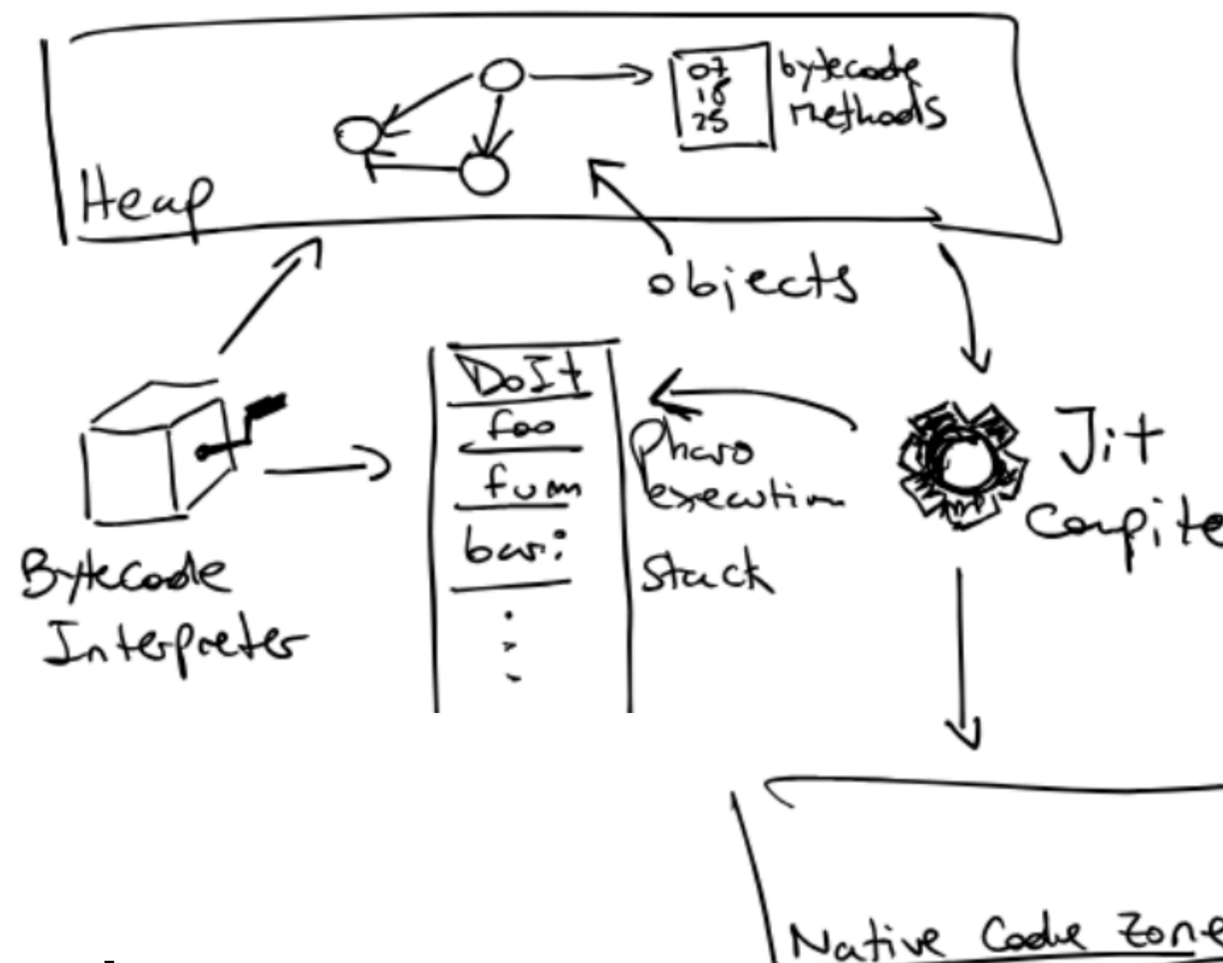
Concolic Meta-Interpretation Model

- Models VM behaviour during concolic execution
    - Frame
    - Objects + types
    - Classes
- Then flattened into SAT solver equations



Experimental Context: The Pharo VM

- Interpreted-compiled mixed execution
- Some numbers:
  - 255 stack based bytecodes
  - ~340 primitives/native methods
  - 146 different IR instructions
  - x86, x86-64, ARMv7, ARMv8, RISC-V
- Industrial consortium:
  - **28 International companies, 26 academic partners**



Previous Manual Testing Effort

- No useful unit tests by ~06/2020
- Large manual testing effort during 2020 while porting to ARM64bits
  - Extended VM simulation with a (TDD compatible) unit testing infrastructure
  - **450+** written tests on the interpreter and the garbage collector*
  - **580+** written tests on the JIT compiler*
  - Parametrisable for 32 and 64bits, **ARM32, ARM64, x86, x86-64**

*Numbers by 05/20

Evaluation

- 3 bytecode compilers + 1 native method compiler
- 4928 tests generated

Compiler	# Tested Instructions	# Interpreter Paths	# Curated Paths	# Differences (%)
Native Methods (primitives)	112	2024	1520	440 (28,95%)
Simple Stack BC Compiler	175	1308	1136	18 (1,59%)
Stack-to-Register BC Compiler	175	1308	1136	10 (0,88%)
Linear-Scan Allocator BC Compiler	175	1308	1136	10 (0,88%)
Total	637	5948	4928	478 (9,7%)

Analysis of Differences through Manual Inspection

- 91 causes, *6 different categories*
- Errors both in the interpreter AND the compilers
- 14 causes of ***segmentatior***

Family	# Cases
Missing interpreter type check	1
Missing compiled type check	13
Optimisation difference	10
Behavioral difference	5
Missing Functionality	60
Simulation Error	2

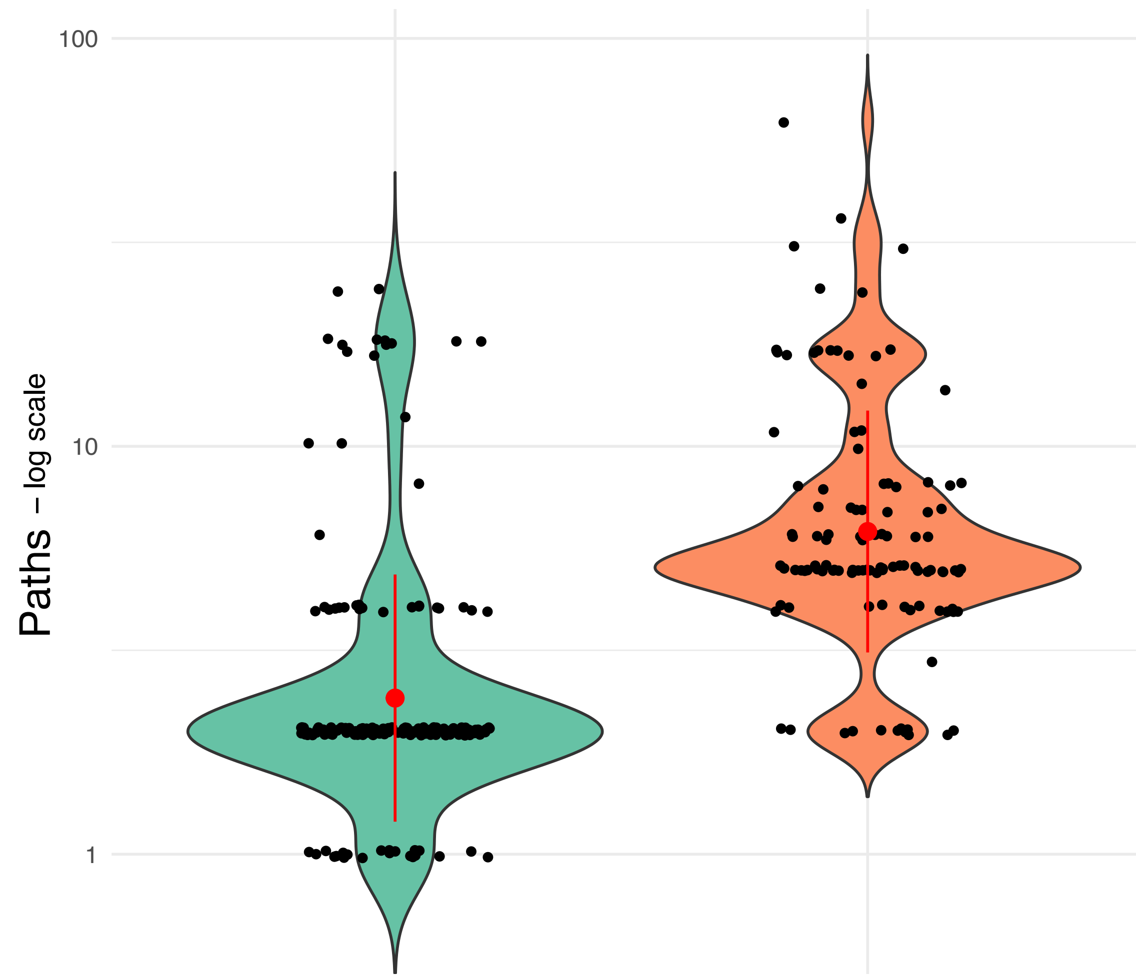
Characterising Concolic Execution

Paths per instruction

- Native methods present in average more paths than bytecode instructions

  => longer time to explore

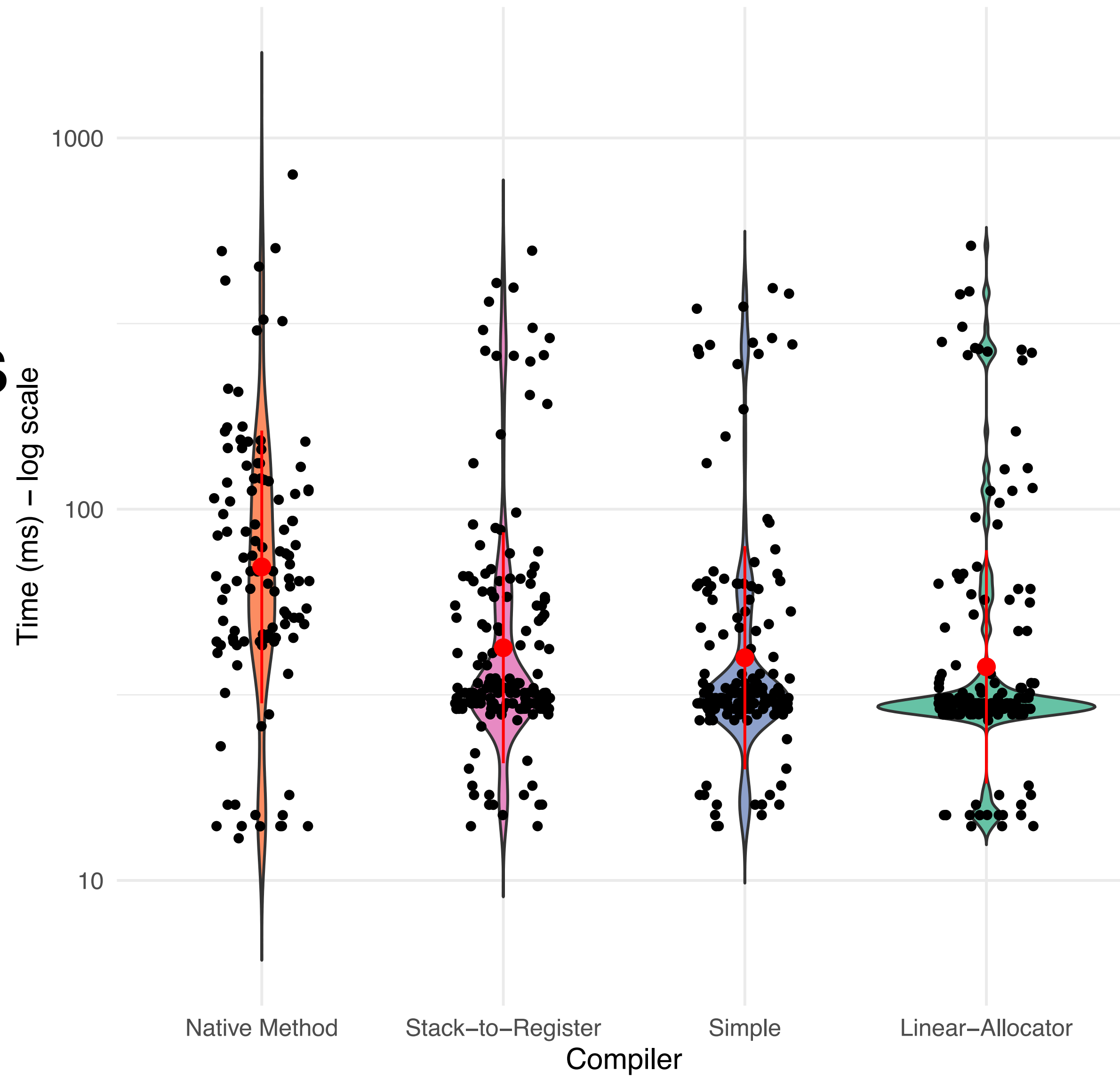
  => potentially more bugs



Paths per Type of Instruction

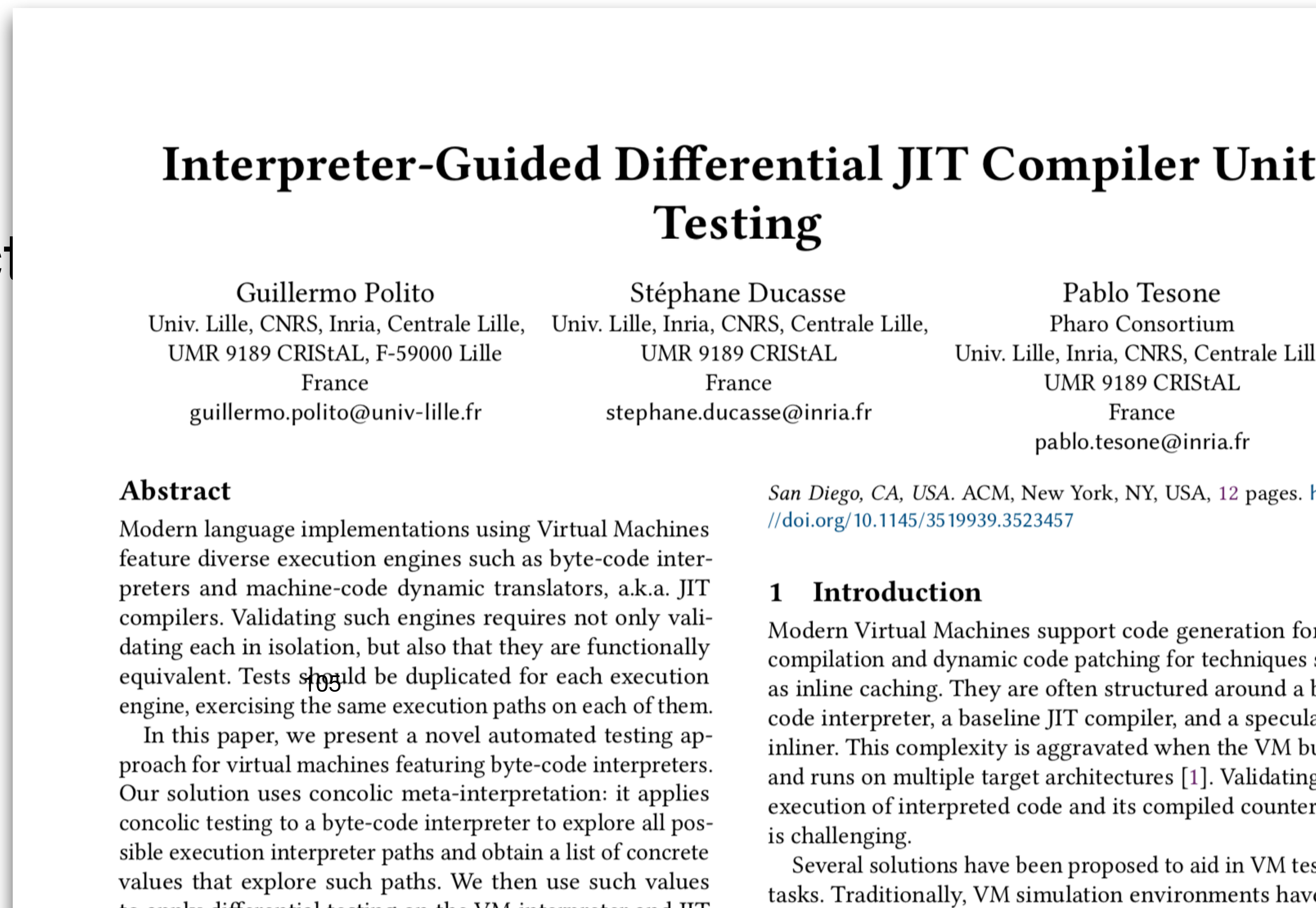
Practical and Cheap

- Test generation ~5 minutes
- Total run time of ~10 seconds
  - *Avg* 30ms per instruction



More in the article!

- Discovered Bugs
- Concolic Model
- Testing Infrastruct



Interpreter-Guided Differential JIT Compiler Unit Testing

Guillermo Polito
Univ. Lille, CNRS, Inria, Centrale Lille,
UMR 9189 CRIStAL, F-59000 Lille
France
guillermo.polito@univ-lille.fr

Stéphane Ducasse
Univ. Lille, Inria, CNRS, Centrale Lille,
UMR 9189 CRIStAL
France
stephane.ducasse@inria.fr

Pablo Tesone
Pharo Consortium
Univ. Lille, Inria, CNRS, Centrale Lill
UMR 9189 CRIStAL
France
pablo.tesone@inria.fr

Abstract

Modern language implementations using Virtual Machines feature diverse execution engines such as byte-code interpreters and machine-code dynamic translators, a.k.a. JIT compilers. Validating such engines requires not only validating each in isolation, but also that they are functionally equivalent. Tests should be duplicated for each execution engine, exercising the same execution paths on each of them.

In this paper, we present a novel automated testing approach for virtual machines featuring byte-code interpreters. Our solution uses concolic meta-interpretation: it applies concolic testing to a byte-code interpreter to explore all possible execution interpreter paths and obtain a list of concrete values that explore such paths. We then use such values

San Diego, CA, USA. ACM, New York, NY, USA, 12 pages. h
//doi.org/10.1145/3519939.3523457

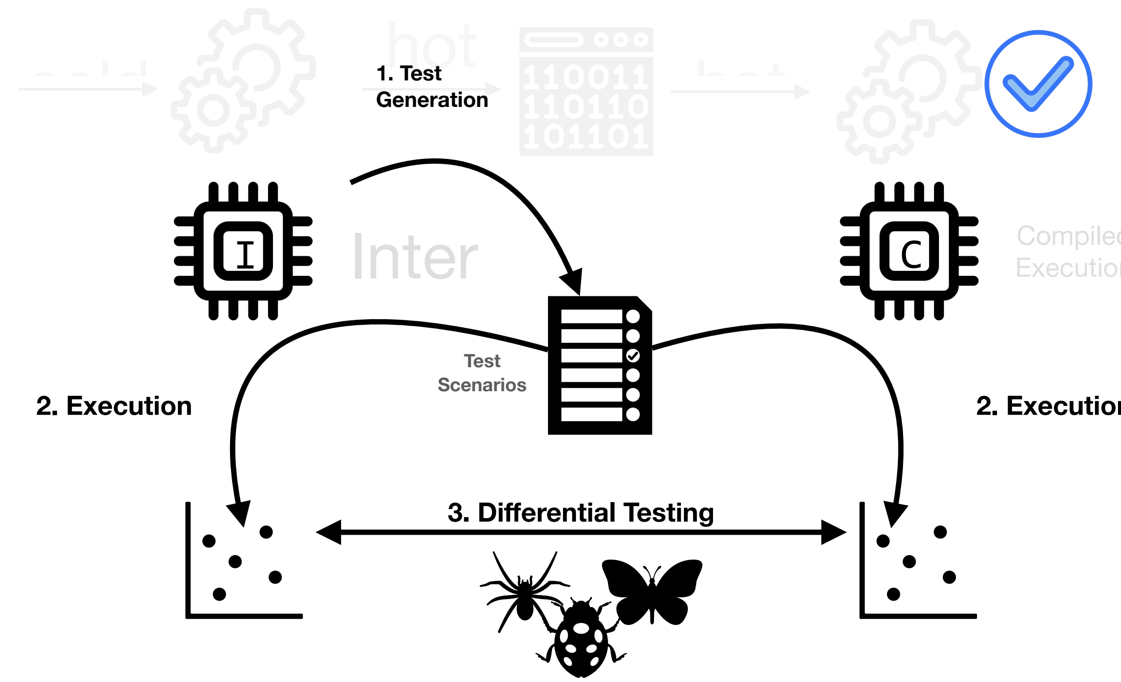
1 Introduction

Modern Virtual Machines support code generation fo
compilation and dynamic code patching for techniques
as inline caching. They are often structured around a b
code interpreter, a baseline JIT compiler, and a specula
inliner. This complexity is aggravated when the VM bu
and runs on multiple target architectures [1]. Validating
execution of interpreted code and its compiled counter
is challenging.

Several solutions have been proposed to aid in VM tes
tasks. Traditionally, VM simulation environments have

Conclusion

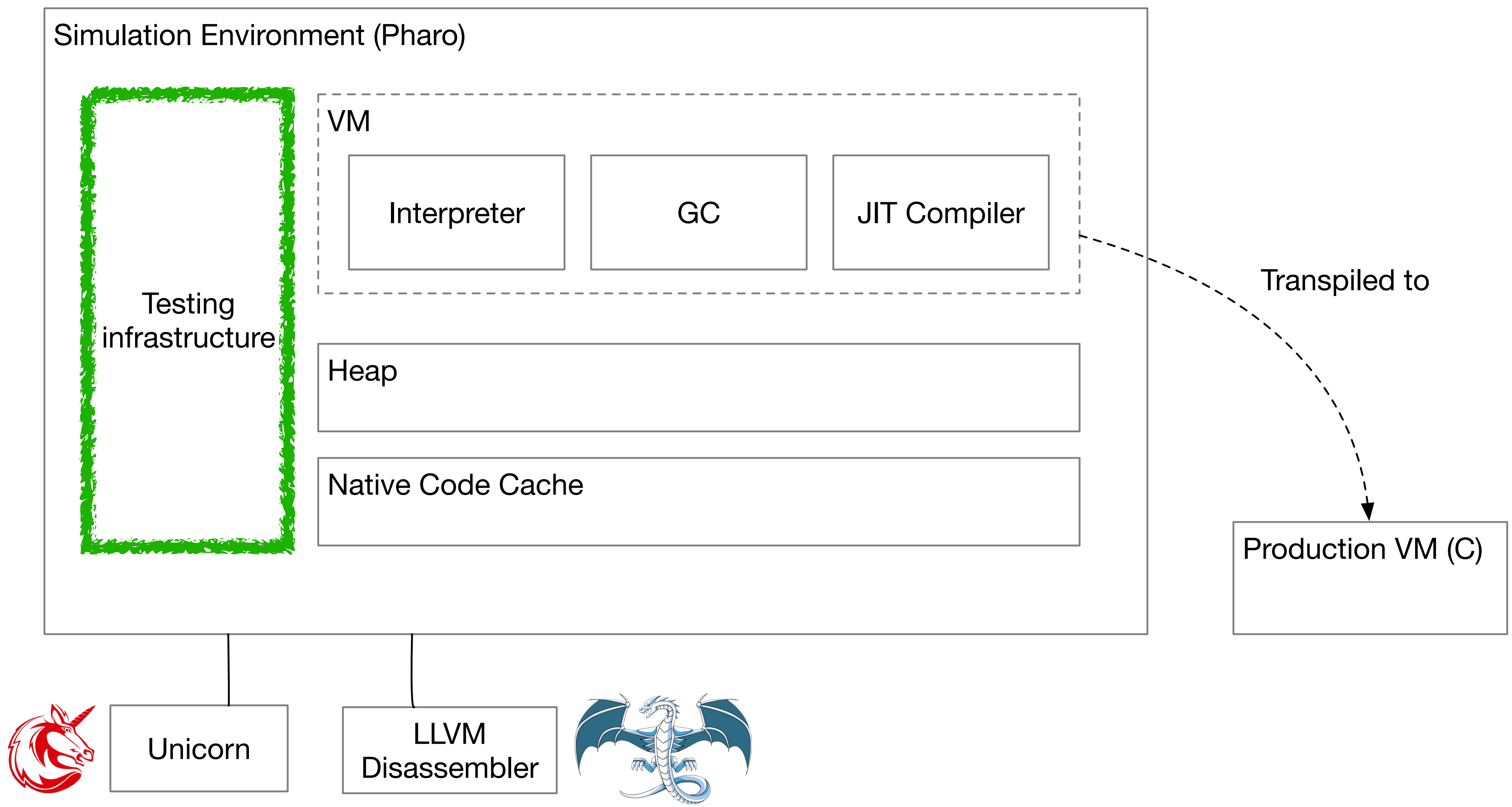
- 478 differences found, 91 causes, 6 categories
- Practical:
    - 4928 tests generated in ~8 minutes
    - 4928 tests run in ~40 seconds



Guille Polito - Pablo Tesone - Stéphane Ducasse
guillermo.polito@univ-lille.fr
@guillep

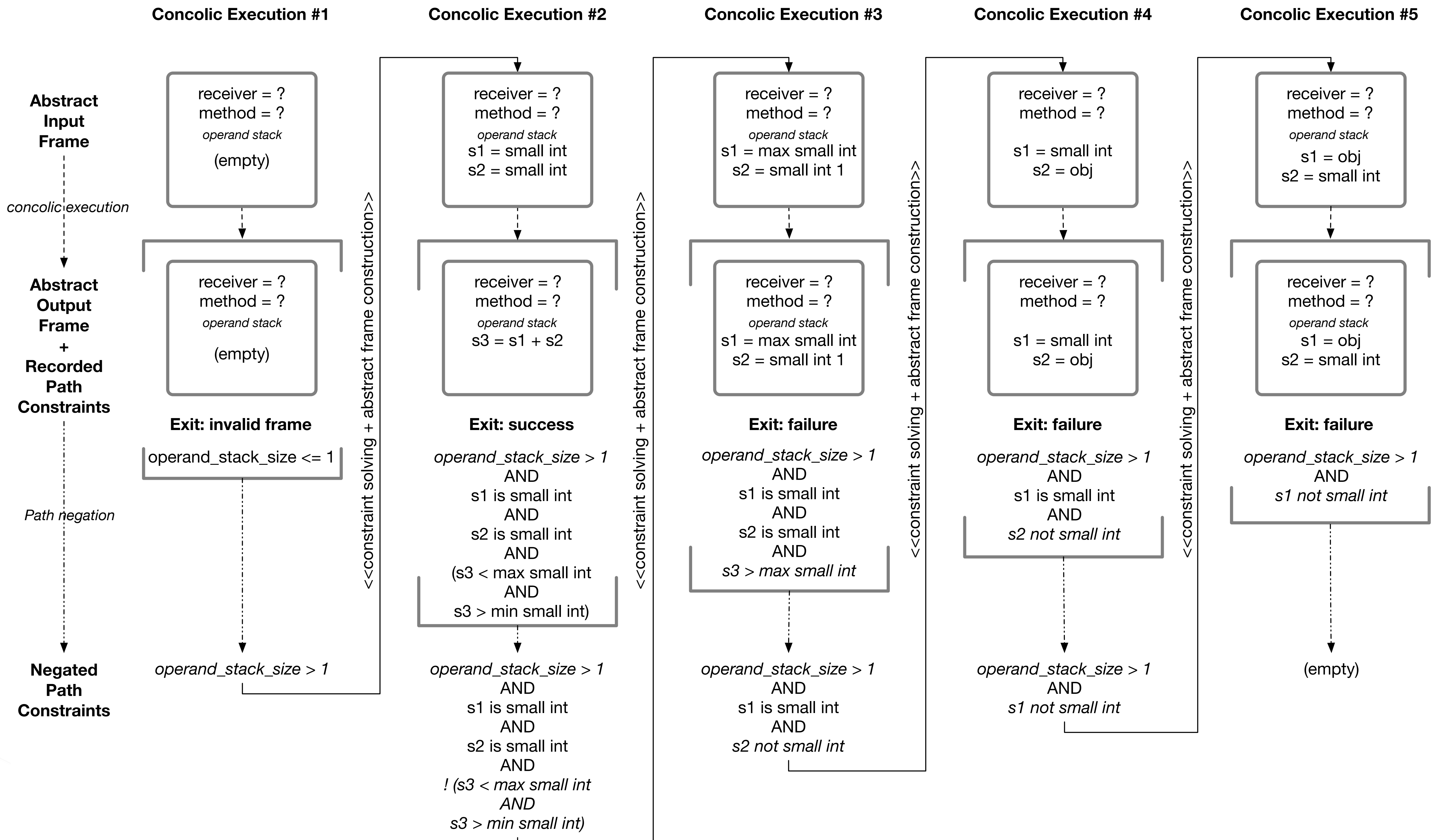
Extras

Simulation + Testing Environment



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


Concolic Execution #1
Concolic Execution #2
Concolic Execution #3
Concolic Execution #4
Concolic Execution #5
Abstract Input Frame
concolic execution
Abstract Output Frame + Recorded Path Constraints
Path negation
Negated Path Constraints
receiver = ?
method = ?
operand stack
(empty)
receiver = ?
method = ?
operand stack
(empty)
Exit: invalid frame
operand_stack_size <= 1
operand_stack_size > 1
<<constraint solving + abstract frame construction>>
receiver = ?
method = ?
operand stack
s1 = small int
s2 = small int
receiver = ?
method = ?
operand stack
s3 = s1 + s2
Exit: success
operand_stack_size > 1 AND s1 is small int AND s2 is small int AND (s3 < max small int AND s3 > min small int)
operand_stack_size > 1 AND s1 is small int AND s2 is small int AND ! (s3 < max small int AND s3 > min small int)
<<constraint solving + abstract frame construction>>
receiver = ?
method = ?
operand stack
s1 = max small int
s2 = small int 1
receiver = ?
method = ?
operand stack
s1 = max small int
s2 = small int 1
Exit: failure
operand_stack_size > 1 AND s1 is small int AND s2 is small int AND s3 > max small int
operand_stack_size > 1 AND s1 is small int AND s2 not small int
<<constraint solving + abstract frame construction>>
receiver = ?
method = ?
s1 = small int
s2 = obj
receiver = ?
method = ?
s1 = small int
s2 = obj
Exit: failure
operand_stack_size > 1 AND s1 is small int AND s2 not small int
operand_stack_size > 1 AND s1 not small int
<<constraint solving + abstract frame construction>>
receiver = ?
method = ?
operand stack
s1 = obj
s2 = small int
receiver = ?
method = ?
operand stack
s1 = obj
s2 = small int
Exit: failure
operand_stack_size > 1 AND s1 not small int
(empty)