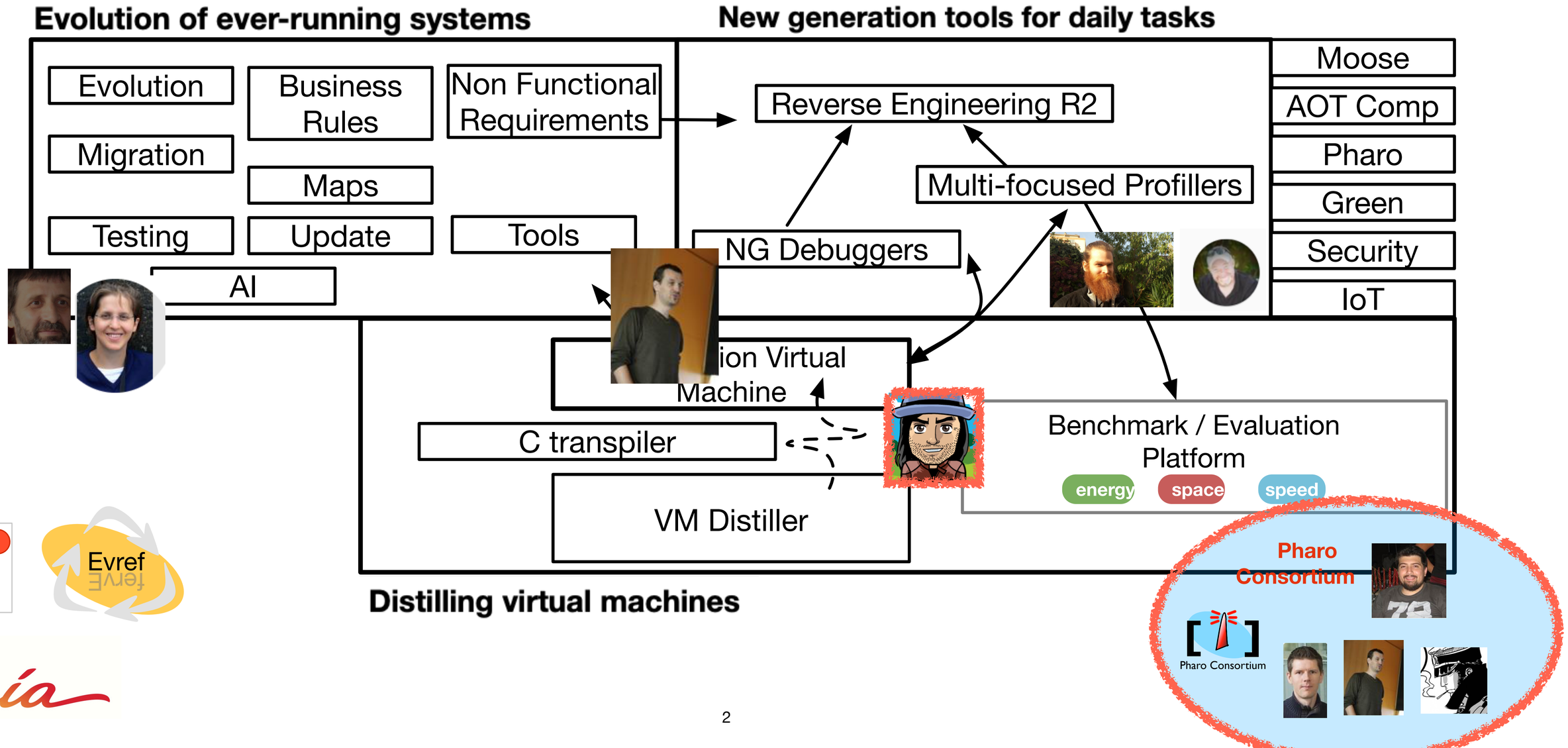
High-Performance and Robust Virtual Machines

Pharo as research engine

G. Polito - 08/11/2022 - LAFHIS

guillermo.polito@inria.fr
@guillep

RMoD Team (Soon Evref)

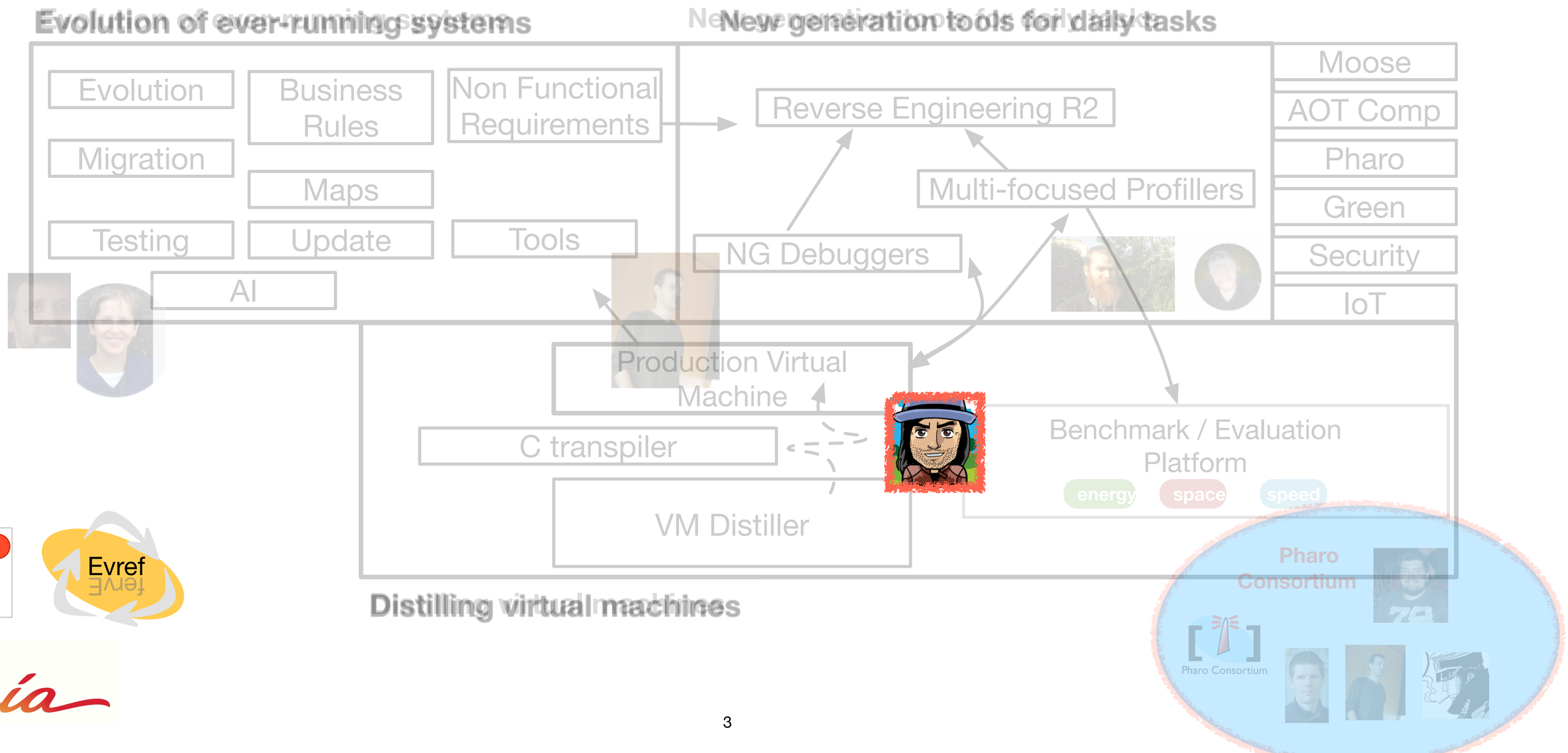


RMod

Evref

Inria

RMoD Team (Soon Evref)



RMod

Evref

Inria

First: About Me

guillermo.polito@inria.fr
@guillep

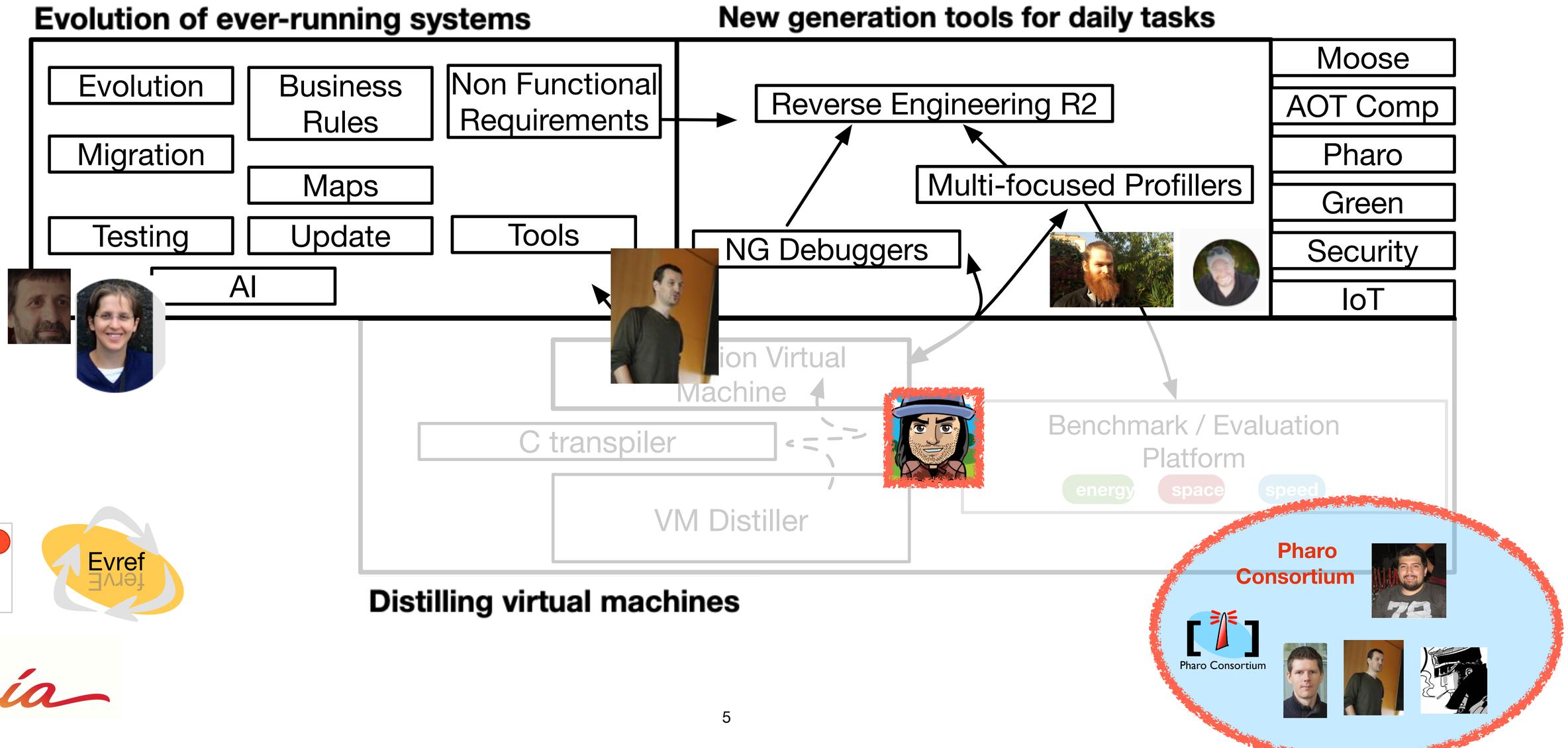
- **Now:** Permanent researcher (CRCN) at Inria - Lille since *2022*
- **Ph.D.:** Reflection, debloating, dynamic updates

- **Keywords:** compilers, testing, test generation
- **Interests:** tooling, benchmarking, 日本語, board games, batman, concurrency

I'll be around the whole week in **Smalltalks**, come talk to me!

Or: guillermo.polito@inria.fr

RMoD Team (Soon Evref)



RMod

Evref

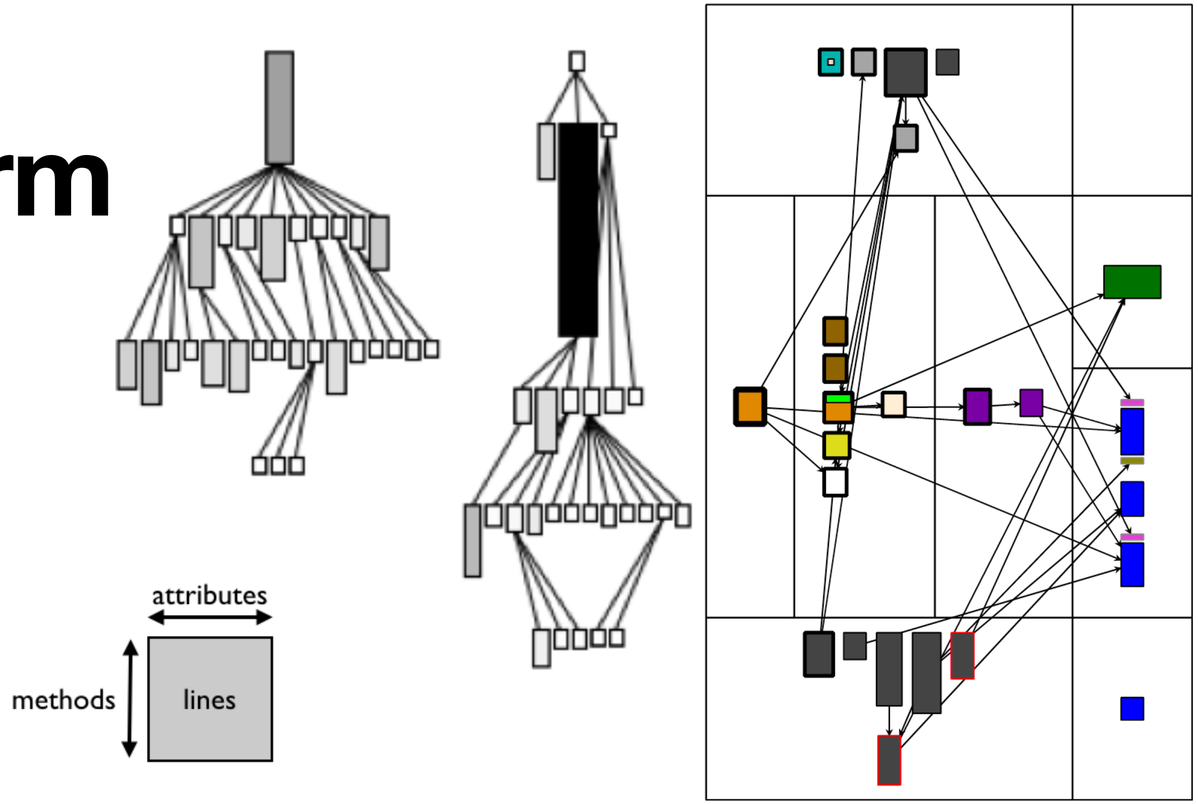
Inria

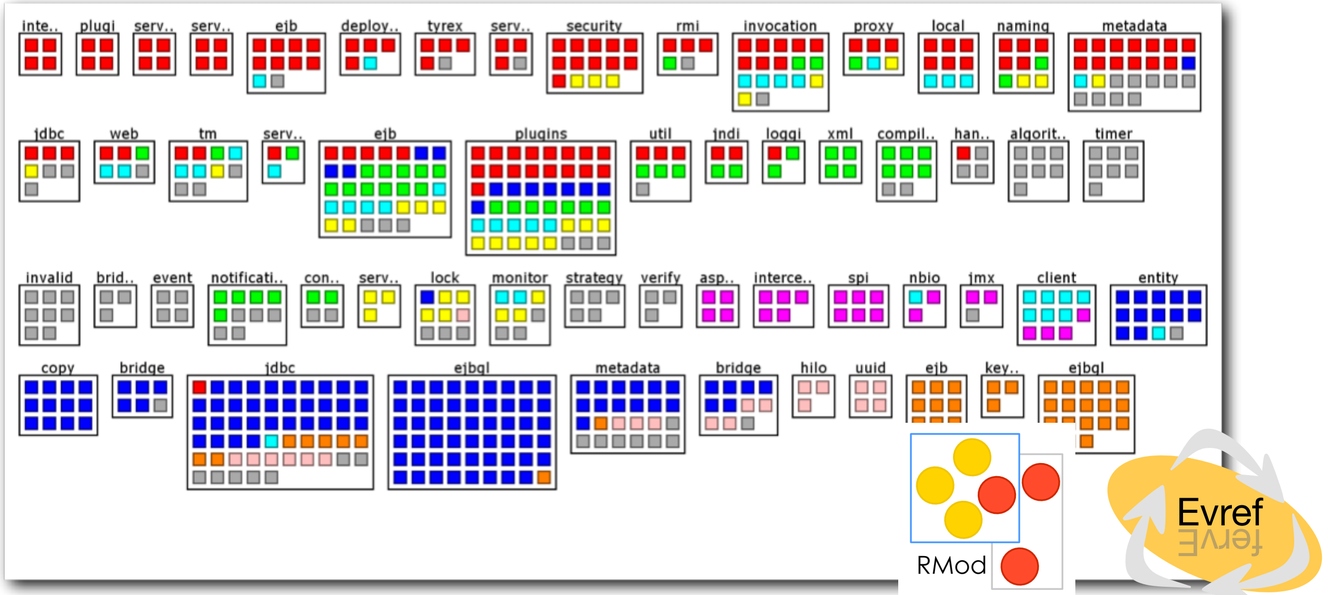
Moose Analysis Platform

Imagine:

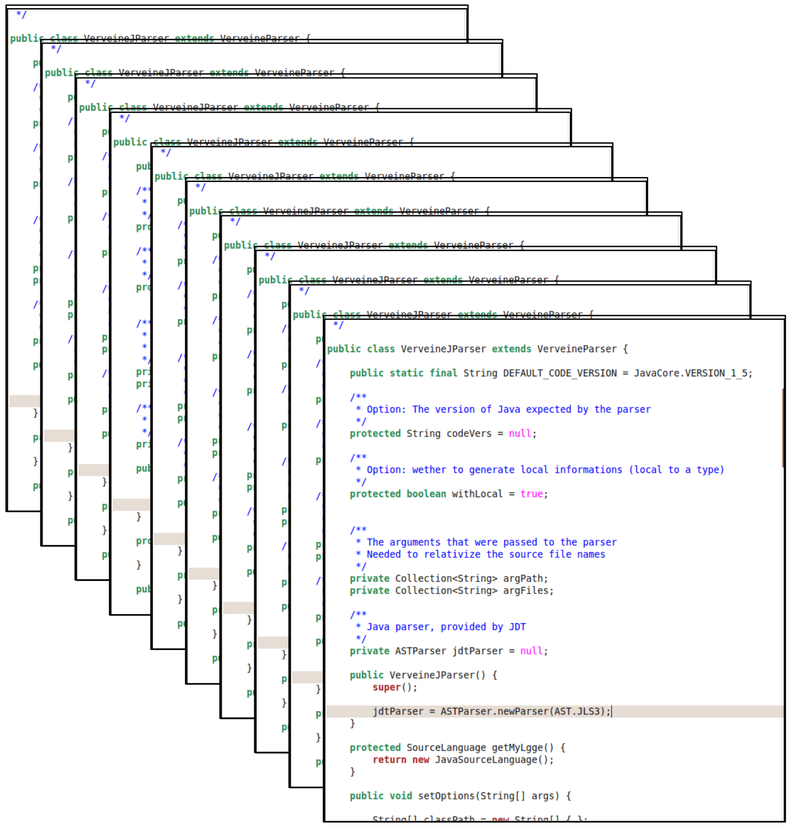
- A program with 1 000 000 LOC
- 2 000 000 seconds to **read** them
- 555 hours → 70 days (with 8h per days) → ~ 3 months (with 22d / month)
- To only read the code
- We need something else to understand program!!

Moose a meta-platform to query, visualize, analyze software systems or other





Business Rules Management



```
public class VerveineJParser extends VerveineParser {

    public static final String DEFAULT_CODE_VERSION = JavaCore.VERSION_1_5;

    /**
     * Option: The version of Java expected by the parser
     */
    protected String codeVers = null;

    /**
     * Option: wether to generate local informations (local to a type)
     */
    protected boolean withLocal = true;


    /**
     * The arguments that were passed to the parser
     * Needed to relativize the source file names
     */
    private Collection<String> argPath;
    private Collection<String> argFiles;

    /**
     * Java parser, provided by JDT
     */
    private ASTParser jdtParser = null;

    public VerveineJParser() {
        super();

        jdtParser = ASTParser.newParser(AST.JLS3);
    }

    protected SourceLanguage getMyLgge() {
        return new JavaSourceLanguage();
    }

    public void setOptions(String[] args) {
```

An Example: A Human Resource Application
If remote working or paternity leave conditions change, where should we make the code evolve?

More generally:

- How to extract Business rules from code?
- How to represent the spread of the Business rules into the code?
- How to make code evolve to follow the business rules?

Keywords:

- Business rules extraction
- Slicing
- Feature location
- Moose

RMod

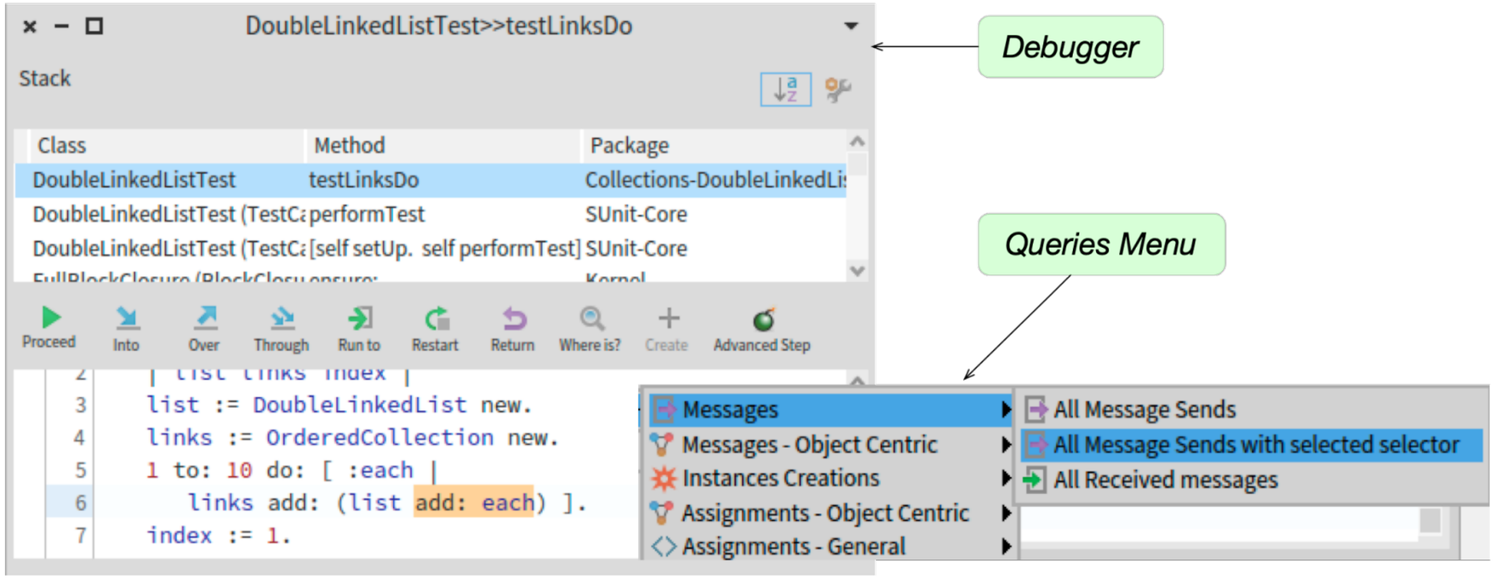
Evref

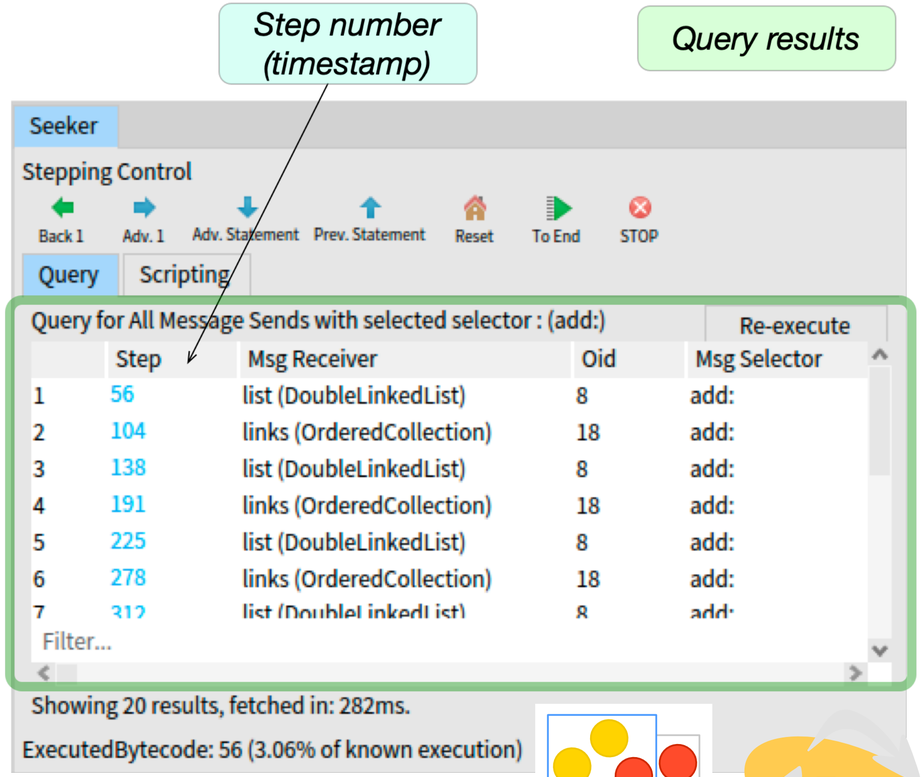
Time Travelling Queries

Do you have a debugging question?

Select a Time-Traveling Query from the Queries Menu!

Find execution data and explore your execution conveniently from the Query Results





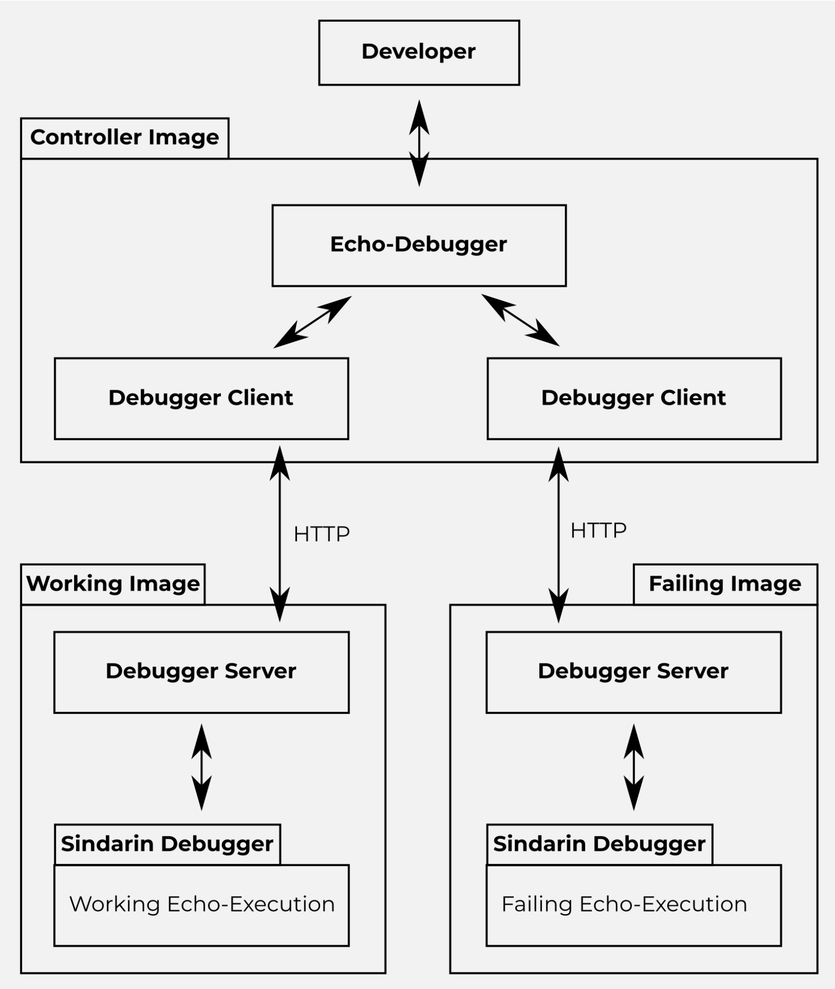
(*) M. Willembrinck, S. Costiou, A. Etien, S. Ducasse. Time-Traveling Debugging Queries: Faster Program Exploration. *International Conference on Software Quality, Reliability, and Security*, Dec 2021, Hainan Island, China.

RMod

Evref

Echo Debugging

Explore two executions in parallel to find and understand differences (or bugs):
- 1 working execution
- 1 failing execution



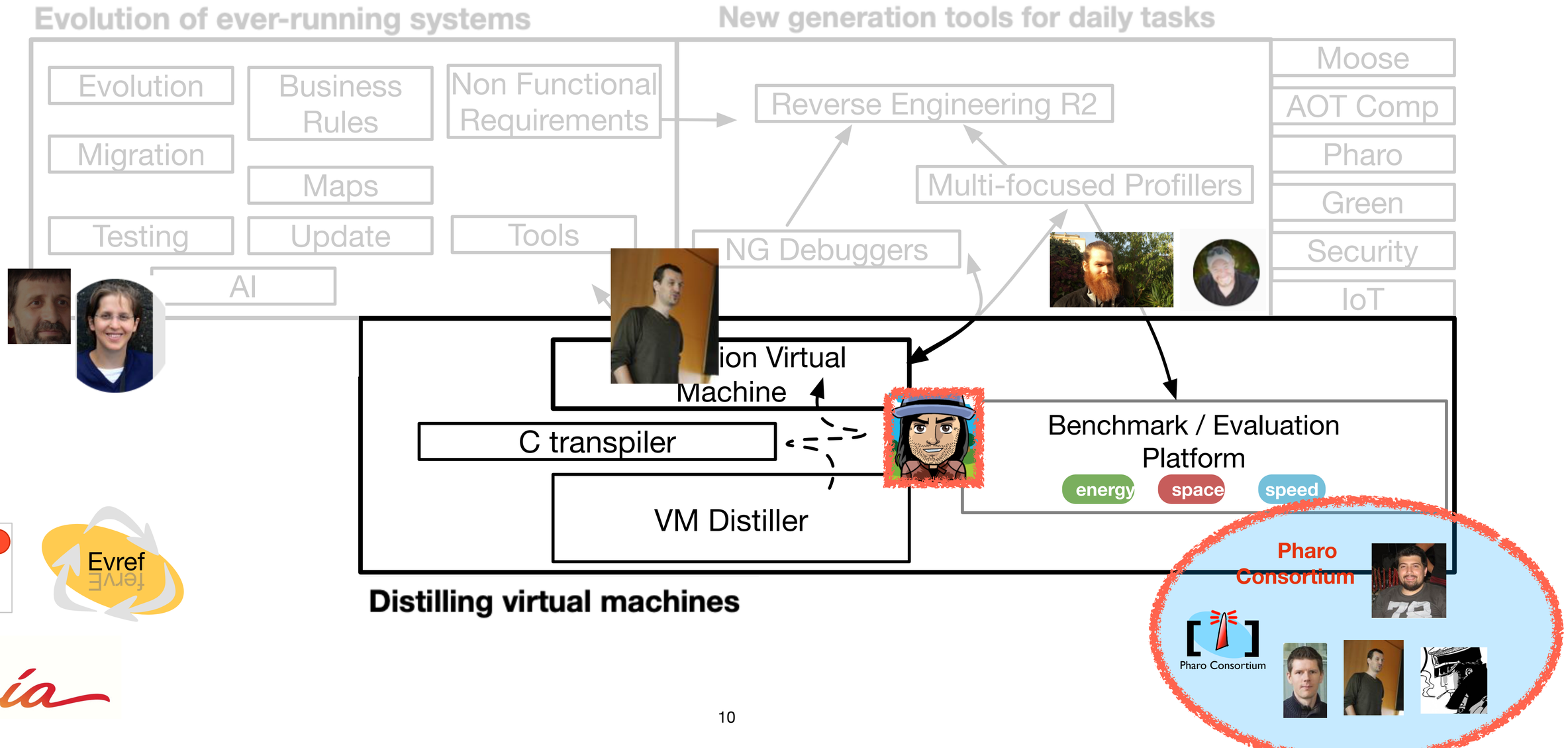
Research questions:
- what infrastructure?
- how to compare two (or more) executions? (*i.e.*, how to represent an execution and compare it to another one?)
- how to understand the differences? => the set of changes between two program executions can be gigantic, *e.g.*, the same program running on different Pharo versions
- how to build echo-debugging tools integrated into an IDE?

PhD positions to foresee!

RMod

Evref

Virtual Machines



The Plan

And let’s hope we get to fit it all

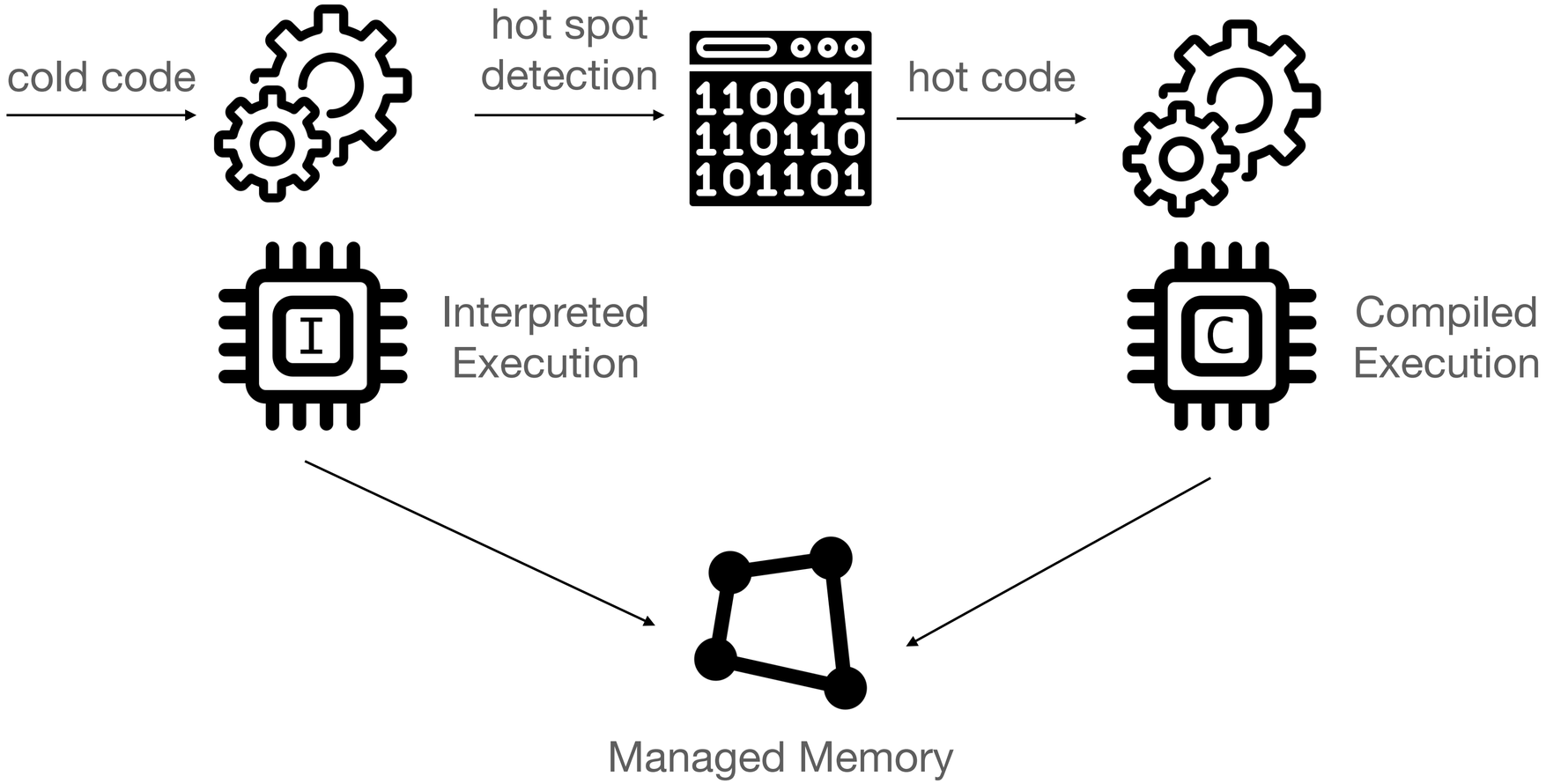
- Basic Virtual Machines notions
- A brief overview of the Pharo VM architecture

- A review of the techniques that have influenced the last ~30-40 years
  - Instruction Dispatch, OOP lookup optimisations

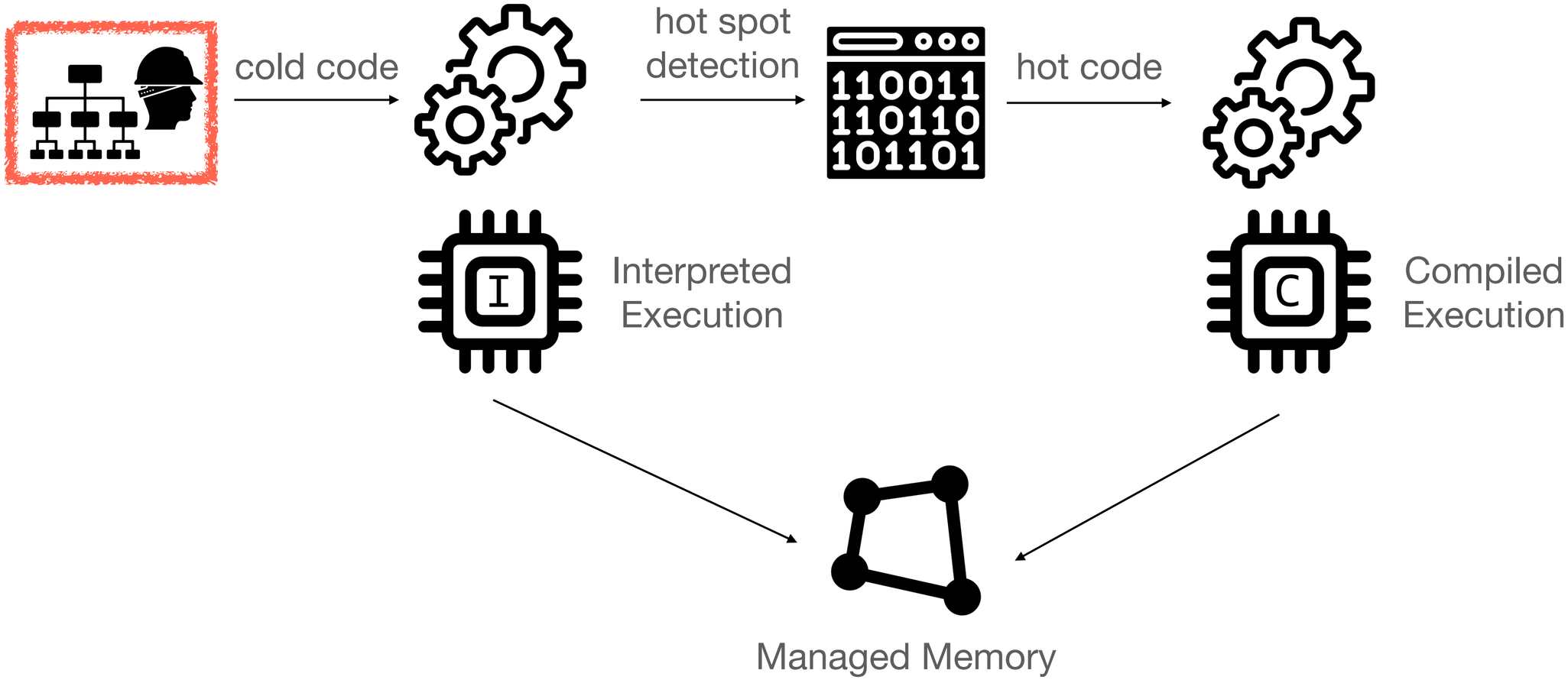
- Remarkable challenges in High-Performance VMs

Part 0: Virtual Machines 101

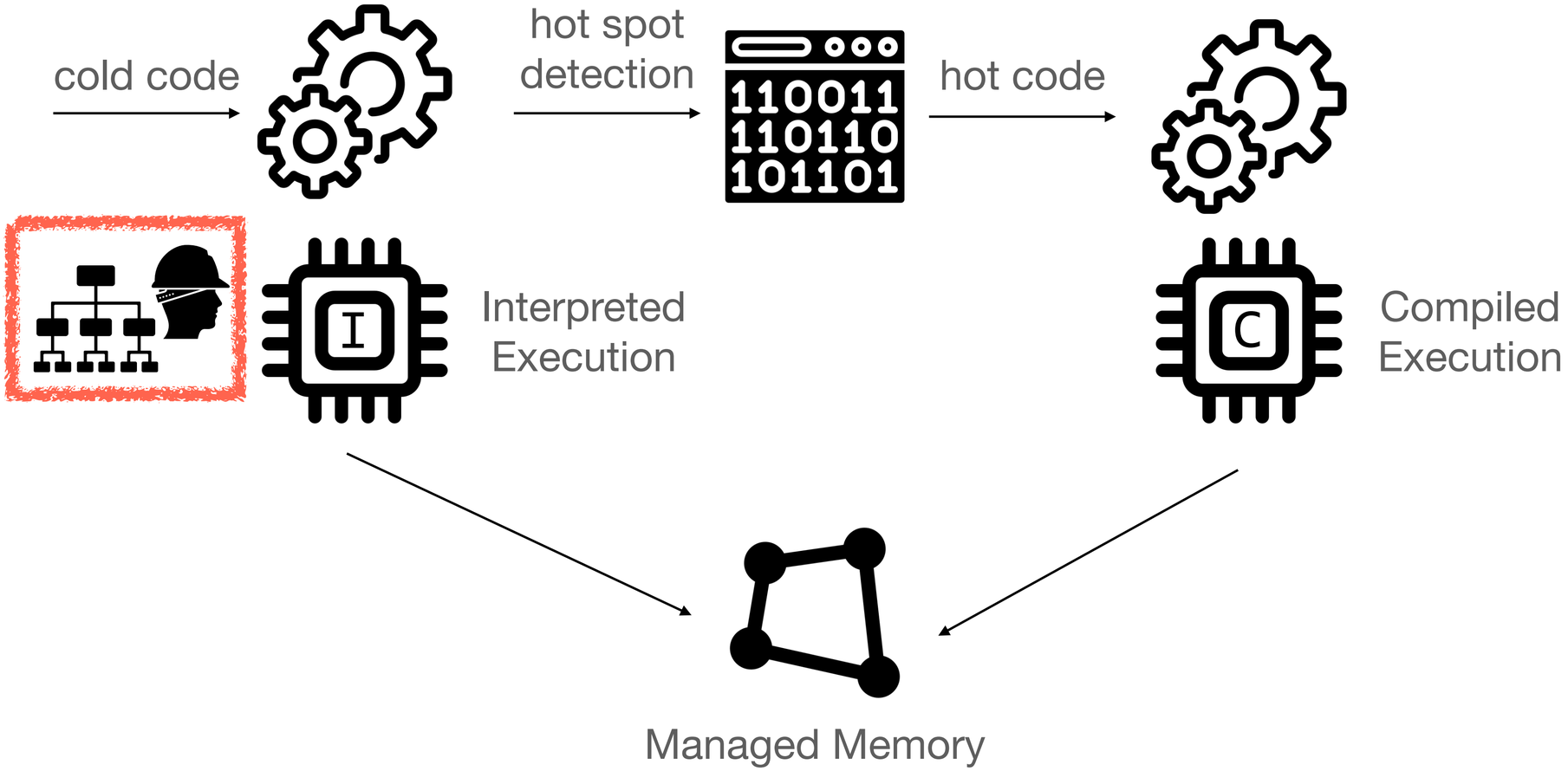
Virtual Machine Execution Engine



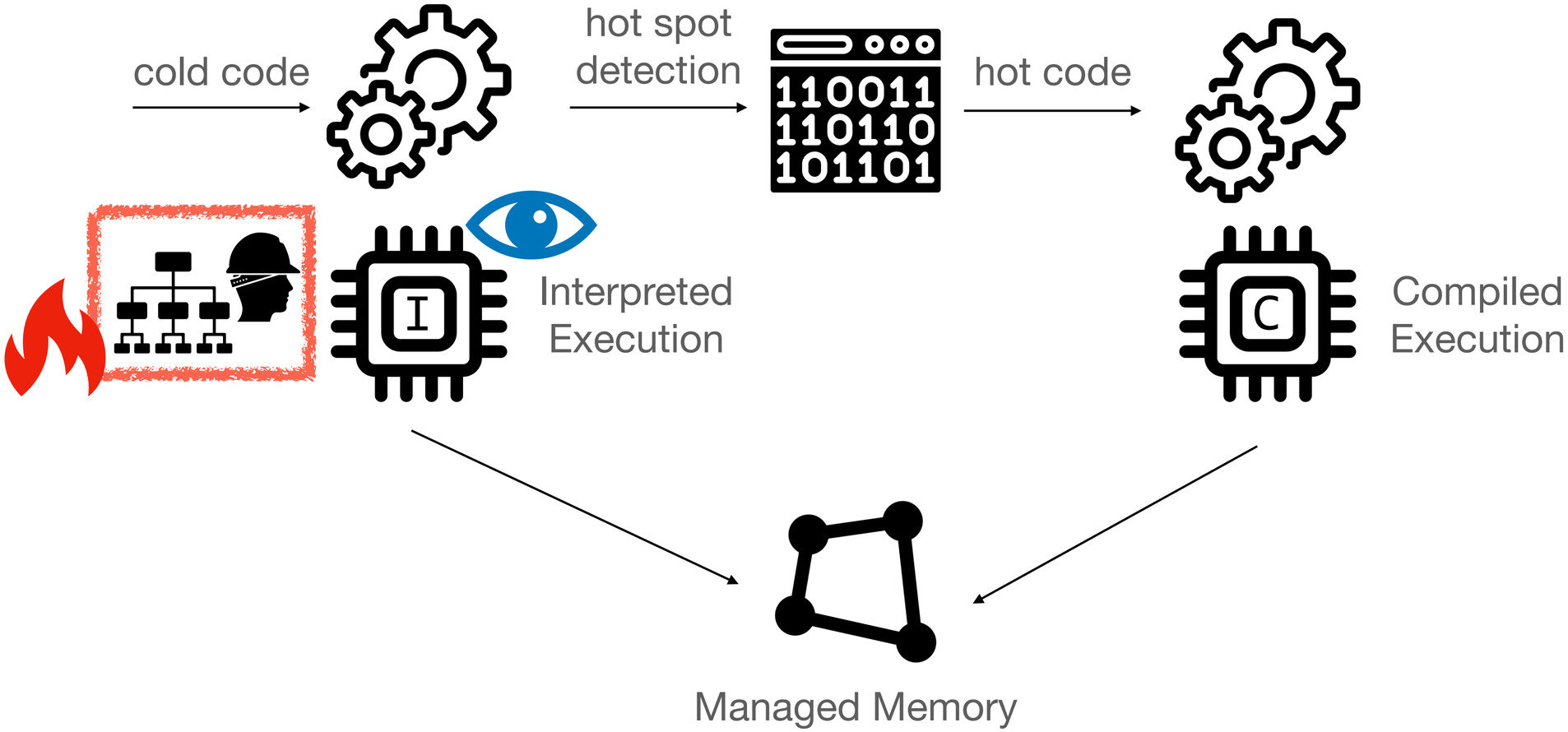
Virtual Machine Execution Engine



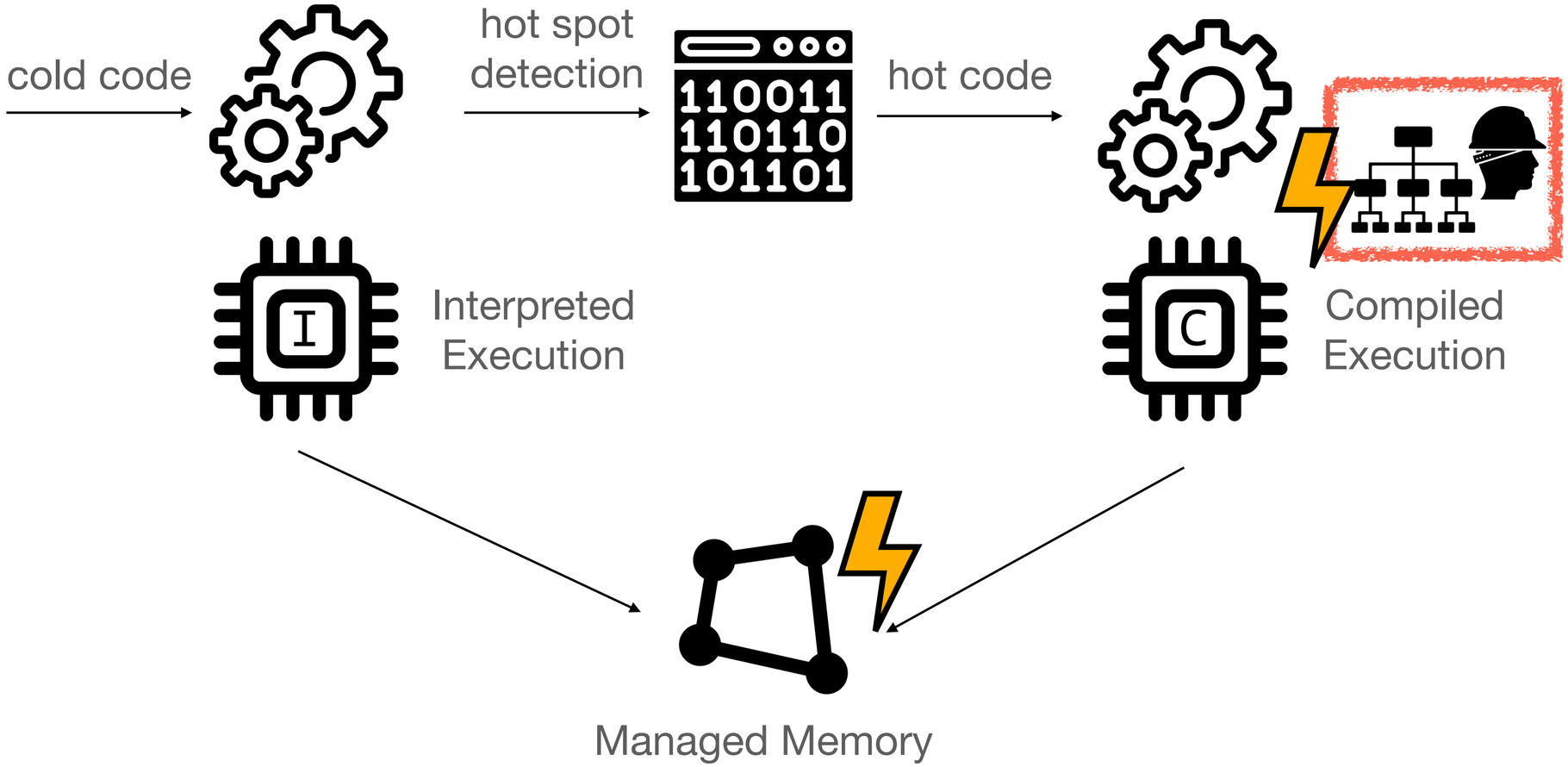
Virtual Machine Execution Engine



Virtual Machine Execution Engine

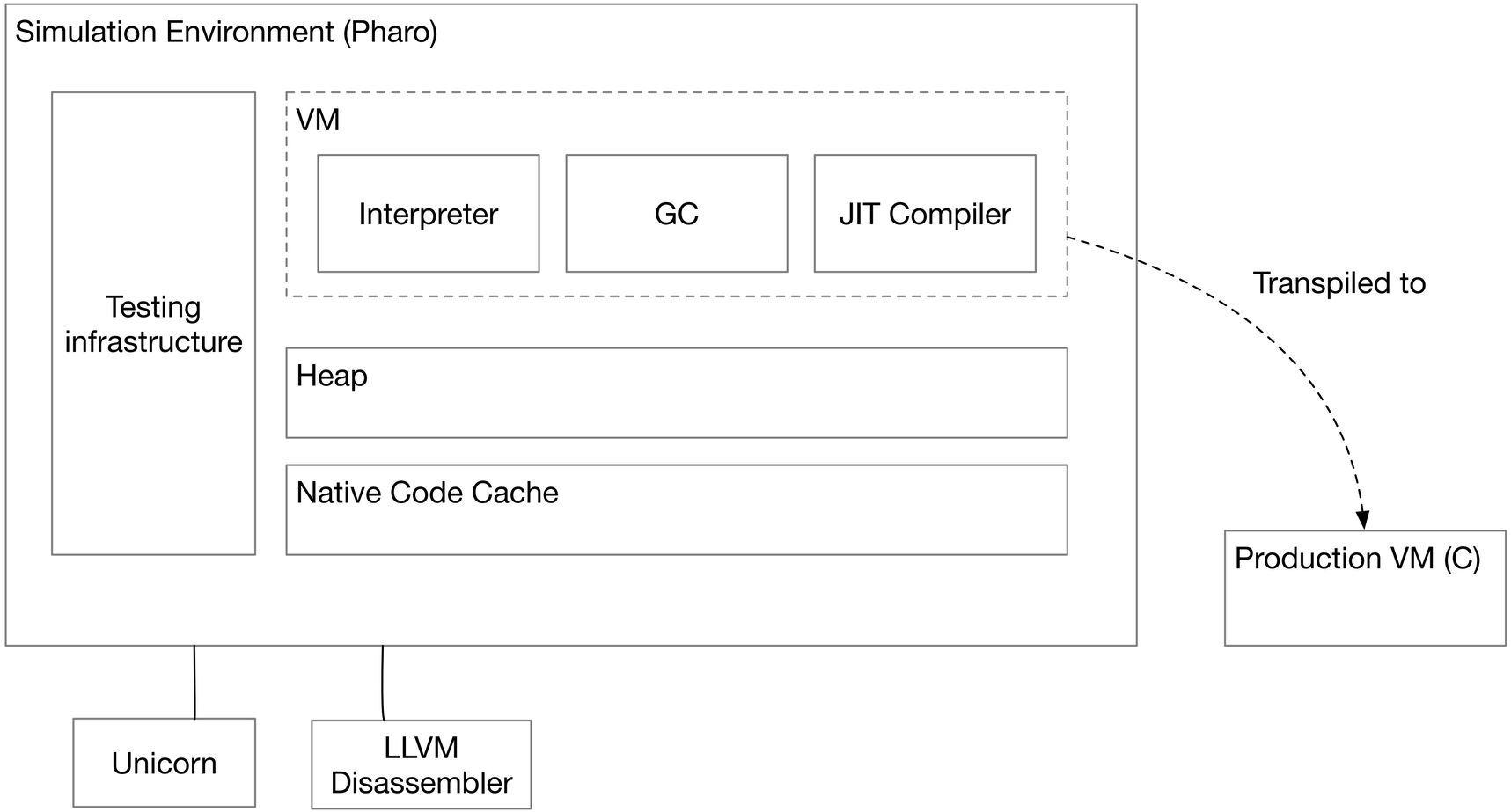


Virtual Machine Execution Engine



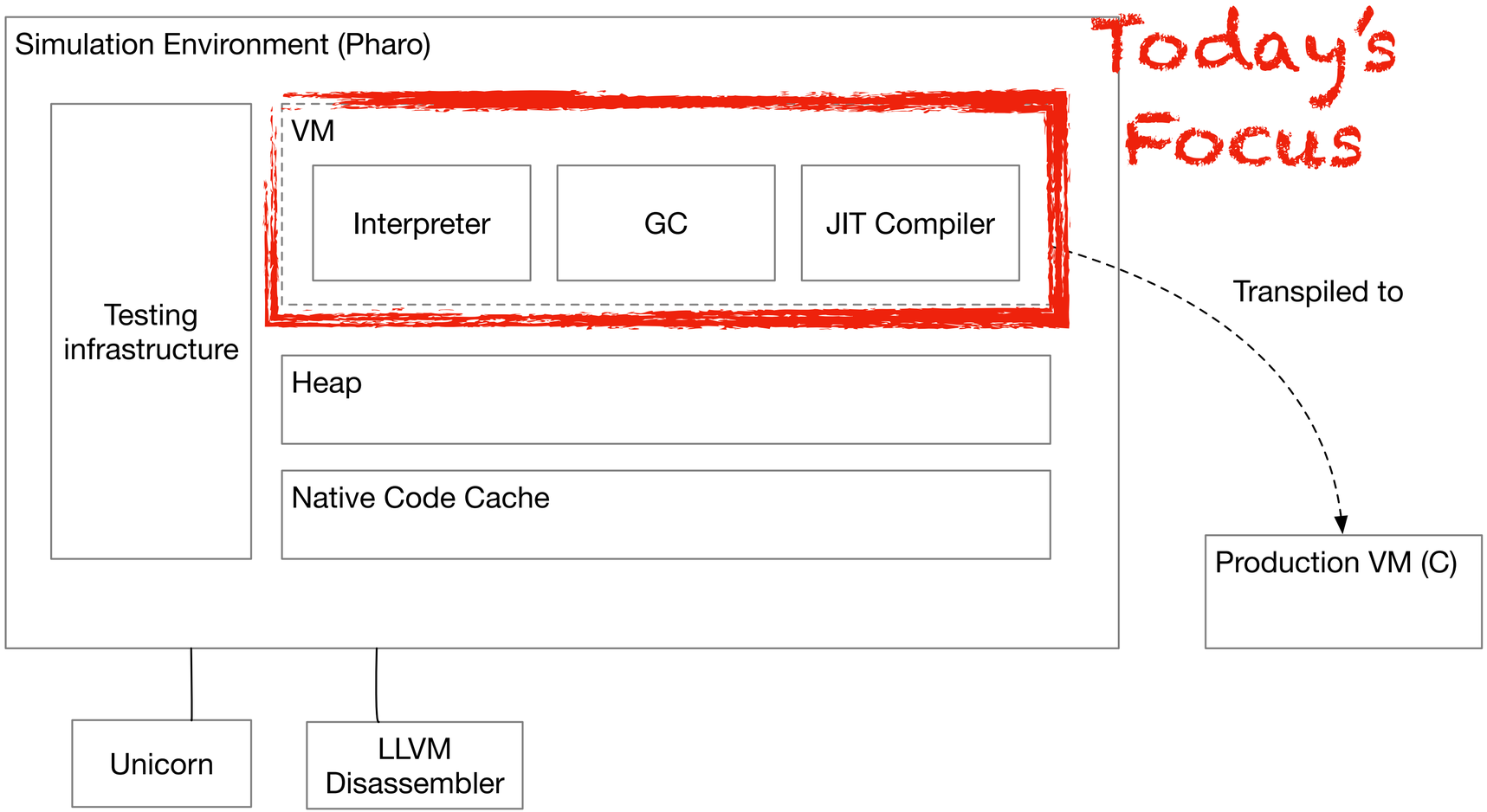
The Pharo VM Architecture

- Written in Pharo
- Transpiled to C
- Simulable
  - Interpreter
  - GC
  - Jitted code



The Pharo VM Architecture

- Written in Pharo
- Transpiled to C
- Simulable
  - Interpreter
  - GC
  - Jitted code



Part I: On Instruction Dispatch

Interpreters

```
interpret
 [ true ] whileTrue: [
    currentBytecode := self fetchNextBytecode.
    self dispatch: currentBytecode ]
```

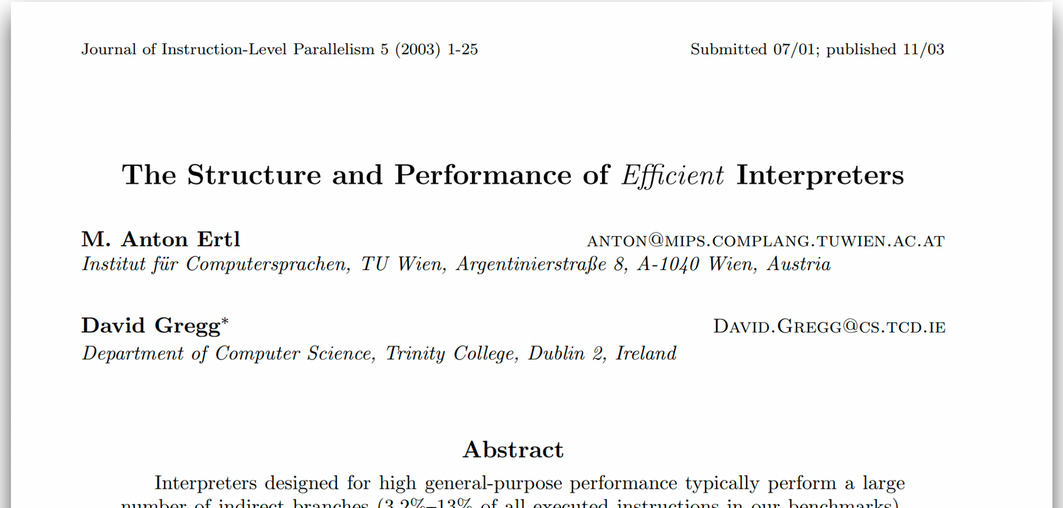
```
void interpret(){
  while (1){
    switch(nextInstruction){
      case push: …
      case pop: …
      case send: …
      …
    }
  }
}
```

Interpreter Perfomance Profile

- Dispatch overhead
  - Affects languages with simple instructions
  - More time in dispatch than actual work
  - Caused by CPU branch mispredictions

```
void interpret(){
  while (1){
    switch(nextInstruction){
      case push: …
      case pop: …
      case send: …
      …
    }
  }
}
```

```
interpret
 [ true ] whileTrue: [
    currentBytecode := self fetchNextBytecode.
    self dispatch: currentBytecode ]
```



Journal of Instruction-Level Parallelism 5 (2003) 1-25 Submitted 07/01; published 11/03

The Structure and Performance of *Efficient* Interpreters

M. Anton Ertl ANTON@MIPS.COMPLANG.TUWIEN.AC.AT
Institut für Computersprachen, TU Wien, Argentinierstraße 8, A-1040 Wien, Austria

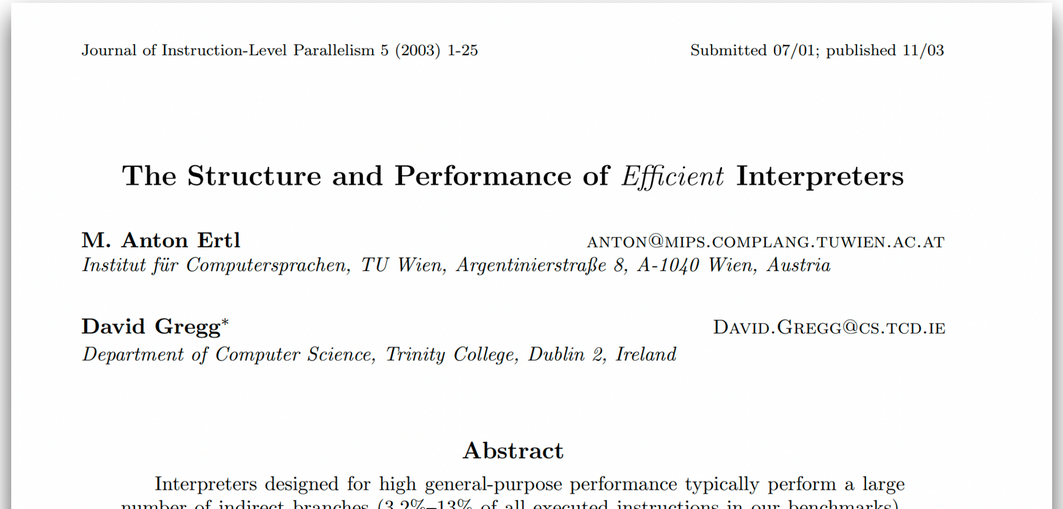
David Gregg* DAVID.GREGG@CS.TCD.IE
Department of Computer Science, Trinity College, Dublin 2, Ireland

Abstract

Interpreters designed for high general-purpose performance typically perform a large number of indirect branches (3.2%–13% of all executed instructions in our benchmarks)

Threaded Code Models

- Idea: restructure the code to *help branch predictors*
  - How? Multiply branch sites, use less indirect branches…
  - Lots of flavours: indirect, token, direct, context threading…



Journal of Instruction-Level Parallelism 5 (2003) 1-25 Submitted 07/01; published 11/03

The Structure and Performance of *Efficient* Interpreters

M. Anton Ertl ANTON@MIPS.COMPLANG.TUWIEN.AC.AT
Institut für Computersprachen, TU Wien, Argentinierstraße 8, A-1040 Wien, Austria

David Gregg* DAVID.GREGG@CS.TCD.IE
Department of Computer Science, Trinity College, Dublin 2, Ireland

Abstract

Interpreters designed for high general-purpose performance typically perform a large number of indirect branches (3.2%–13% of all executed instructions in our benchmarks)

```
void interpret(){
  while (1){
    switch(nextInstruction){
      case push: …
      case pop: …
      case send: …
      …
    }
  }
}
```

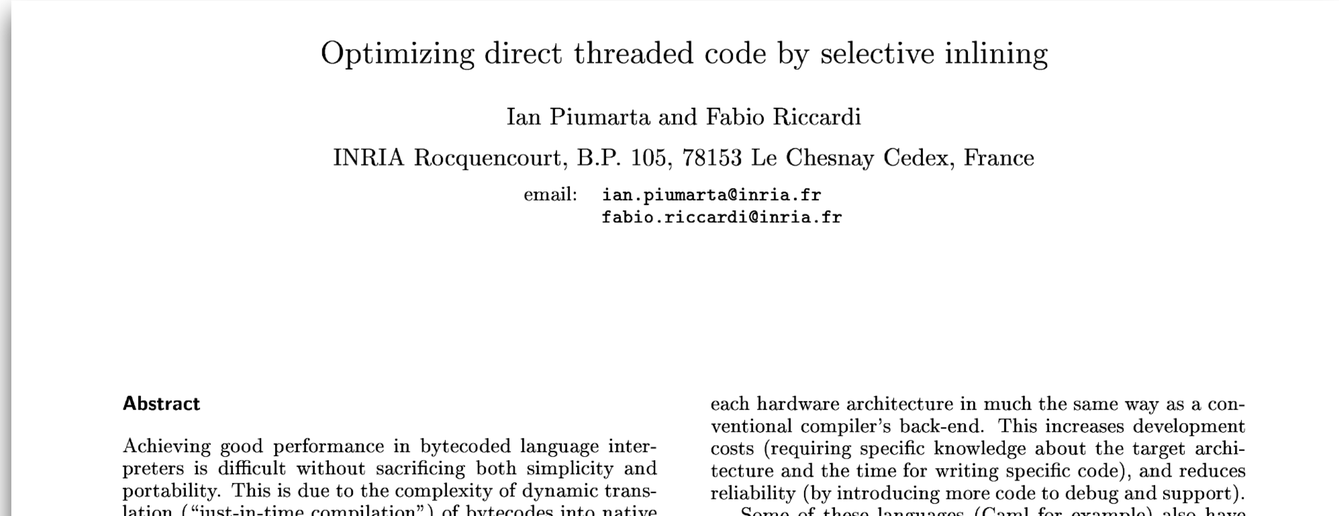
```
void interpret(){

  label push: … goto next;
  label pop: …  goto next;
  label send: … goto next;
  …

}
```

Super Instructions

- Super instructions
  - less but bigger instructions
  - => less dispatch

```
void interpret(){
  while (1){
    switch(nextInstruction){
      case push: …
      case pop: …
      case send: …
      …
    }
  }
}
```

```
void interpret(){
  while (1){
    switch(nextInstruction){
      case push: …
      case pop: …
      case send: …
      case push_send: …
      case push_send_pop: …
      …
    }
  }
}
```



Optimizing direct threaded code by selective inlining

Ian Piumarta and Fabio Riccardi

INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France

email: ian.piumarta@inria.fr
fabio.riccardi@inria.fr

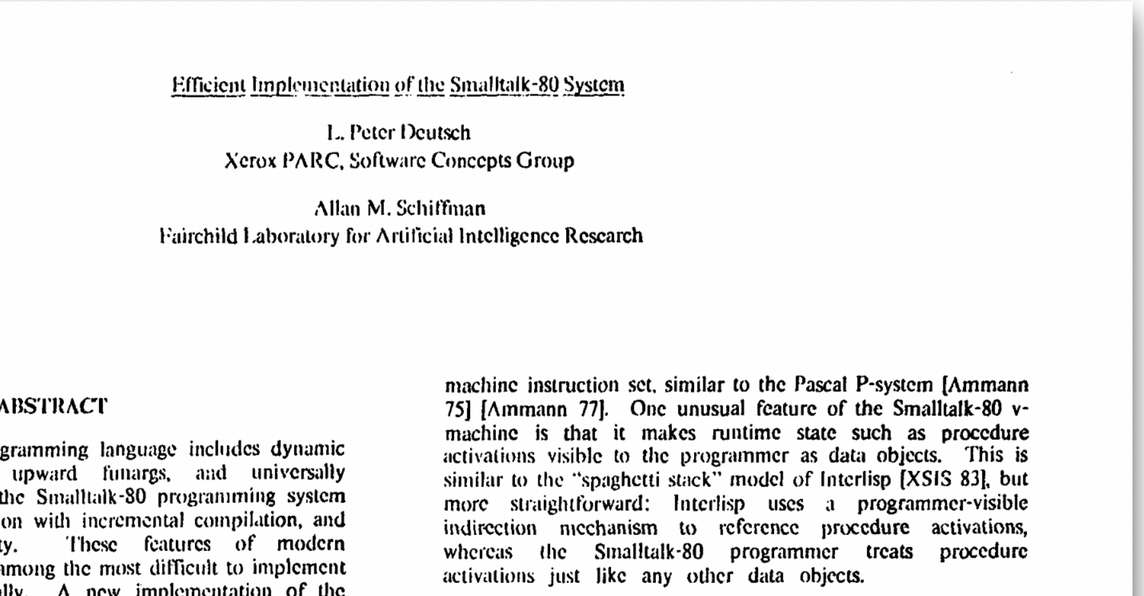
Abstract

Achieving good performance in bytecoded language interpreters is difficult without sacrificing both simplicity and portability. This is due to the complexity of dynamic translation ("just-in-time compilation") of bytecodes into native

each hardware architecture in much the same way as a conventional compiler's back-end. This increases development costs (requiring specific knowledge about the target architecture and the time for writing specific code), and reduces reliability (by introducing more code to debug and support).

Dynamic Translation - JIT Compilation

- Do not improve instruction dispatch: **remove** it !
- Translate functions/methods to machine code on the fly
- Techniques: dynamic compilation, code copying…



Efficient Implementation of the Smalltalk-80 System

L. Peter Deutsch
Xerox PARC, Software Concepts Group

Allan M. Schiffman
Fairchild Laboratory for Artificial Intelligence Research

ABSTRACT

gramming language includes dynamic
upward funargs, and universally
the Smalltalk-80 programming system
on with incremental compilation, and
ty. These features of modern
among the most difficult to implement

machine instruction set, similar to the Pascal P-system [Ammann 75] [Ammann 77]. One unusual feature of the Smalltalk-80 v-machine is that it makes runtime state such as procedure activations visible to the programmer as data objects. This is similar to the "spaghetti stack" model of Interlisp [XSIS 83], but more straightforward: Interlisp uses a programmer-visible indirection mechanism to reference procedure activations, whereas the Smalltalk-80 programmer treats procedure activations just like any other data objects.

```
void interpret(){
  while (1){
    switch(nextInstruction){
      case push: …
      case pop: …
      case send: …
      …
    }
  }
}
```

```
void m1(){
  push;
  send;
  push;
  send;
  pop;
}
```

```
void m2(){
  push;
  send;
  send;
  pop;
}
```

```
void m3(){
  push;
  send;
  pop;
}
```

Summary of Dispatch Optimisations

```
void interpret(){
  while (1){
    switch(nextInstruction){
      case push: …
      case pop: …
      case send: …
      …
    }
  }
}
```

Threading

```
void interpret(){
      label push: … goto next;
      label pop: …  goto next;
      label send: … goto next;
      …
}
```

Super Instructions

```
void interpret(){
  while (1){
    switch(nextInstruction){
      case push: …
      case pop: …
      case send: …
      case push_send: …
      case push_send_pop: …
      …
    }
  }
}
```

Dynamic Translation

```
void m2(){
  push;
  send;
  send;
  pop;
}
```

```
void m3(){
  push;
  send;
  pop;
}
```

```
void m1(){
  push;
  send;
  push;
  send;
  pop;
}
```

Part II: The Good and Bad of OOP

Object-Oriented Languages

- Dynamic binding
- Polymorphism
- Abstraction

```
object message.
```

```
object.message();
```

OOP Languages are HARD to Optimise

- Why?
  - Dynamic binding
  - Polymorphism
  - Abstraction

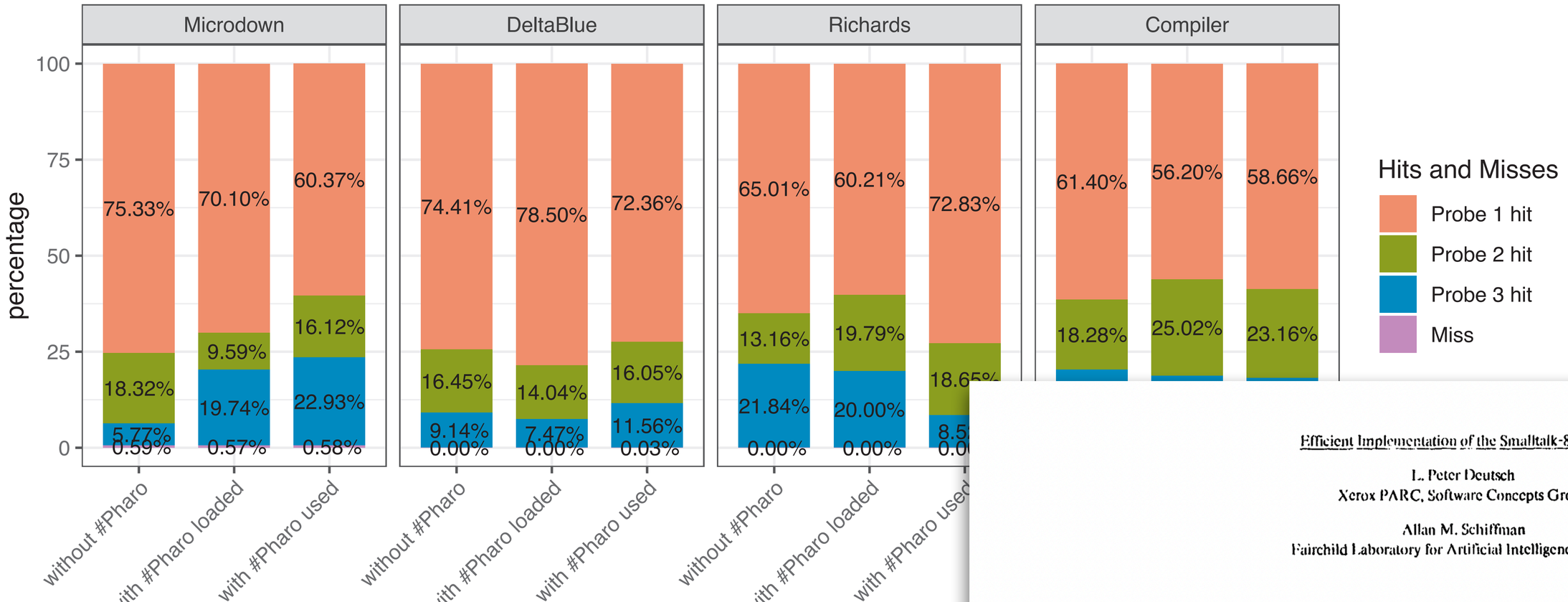
OOP Languages are HARD to Optimise

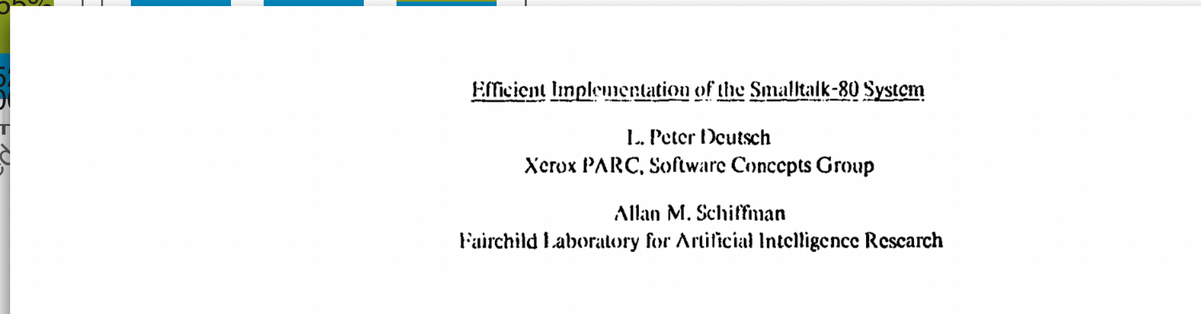
- Why?
  - **Dynamic binding:** method lookup is not **cache conscious**
  - **Polymorphism:** cannot easily **predict types** to optimise
  - **Abstraction:** small methods with lots of calls => lot of **call overhead**

Global Lookup Caches

- Hash table, LRU eviction

```
Interpreter >> lookupSelector: selector inClass: class
  method := self lookupInCacheSelector: selector inClass: class.
  method ifNotNil: [ ^ method ]

  “SLOW LOOKUP”
  …
```





Efficient Implementation of the Smalltalk-80 System

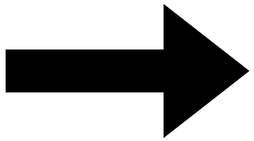
L. Peter Deutsch
Xerox PARC, Software Concepts Group

Allan M. Schiffman
Fairchild Laboratory for Artificial Intelligence Research

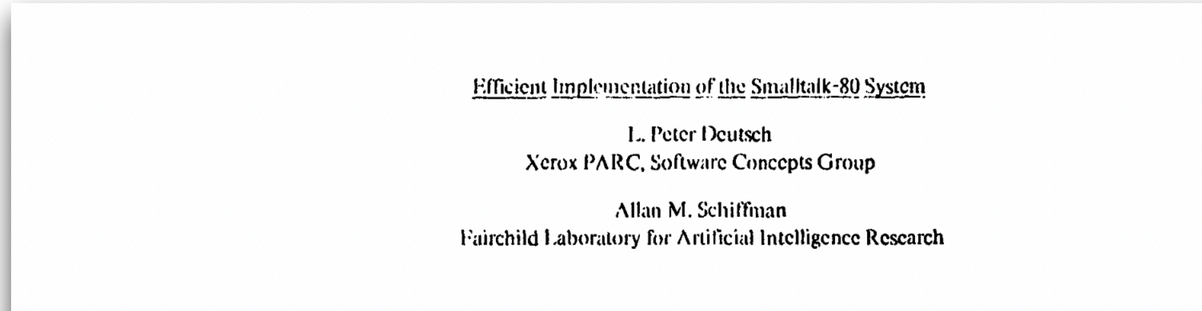
Inline Caches

- **Idea:** put the *cache in the call-site*
- Replace *each call-site* by a call to the target method
- Make it a direct call to help the branch prediction
- Rewrite the call if the decision is wrong

```
void m1(){
  push;
  send(foo);
  push;
  send(bar);
  pop;
}
```



```
void m1(){
  push;
  call TheClass>>#foo;
  push;
  call TheClass>>#bar;
  pop;
}
```

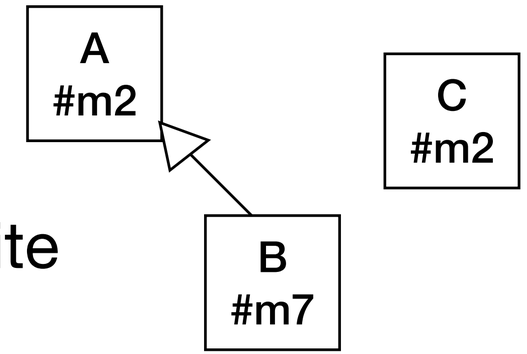


Efficient Implementation of the Smalltalk-80 System

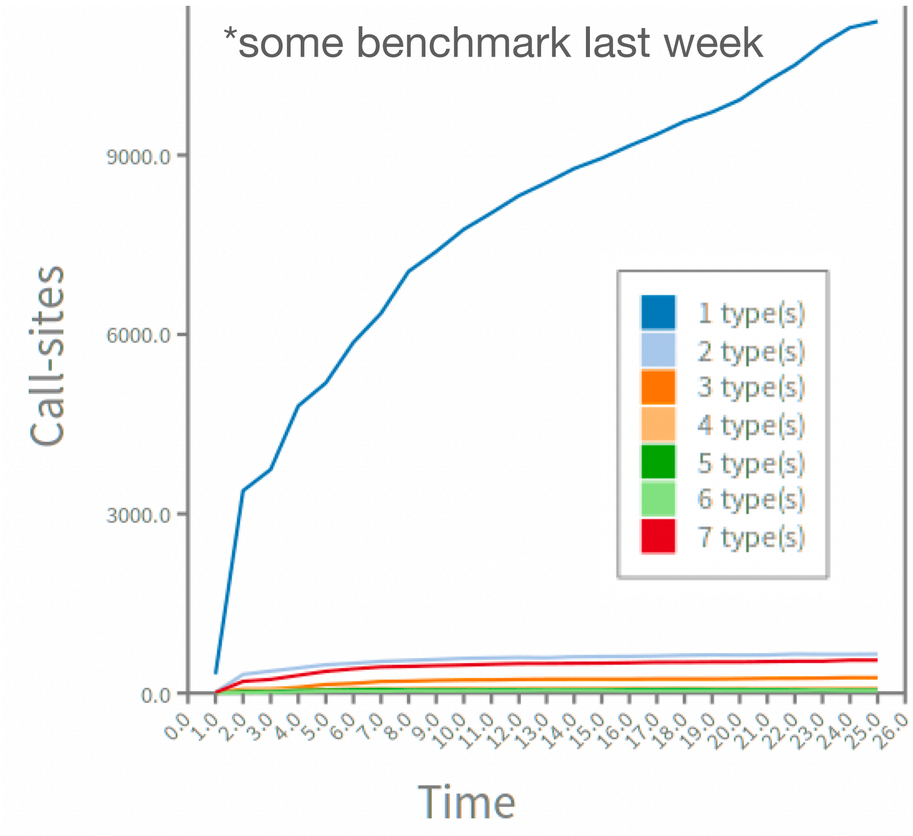
L. Peter Deutsch
Xerox PARC, Software Concepts Group

Allan M. Schiffman
Fairchild Laboratory for Artificial Intelligence Research

Polymorphic Inline Caches

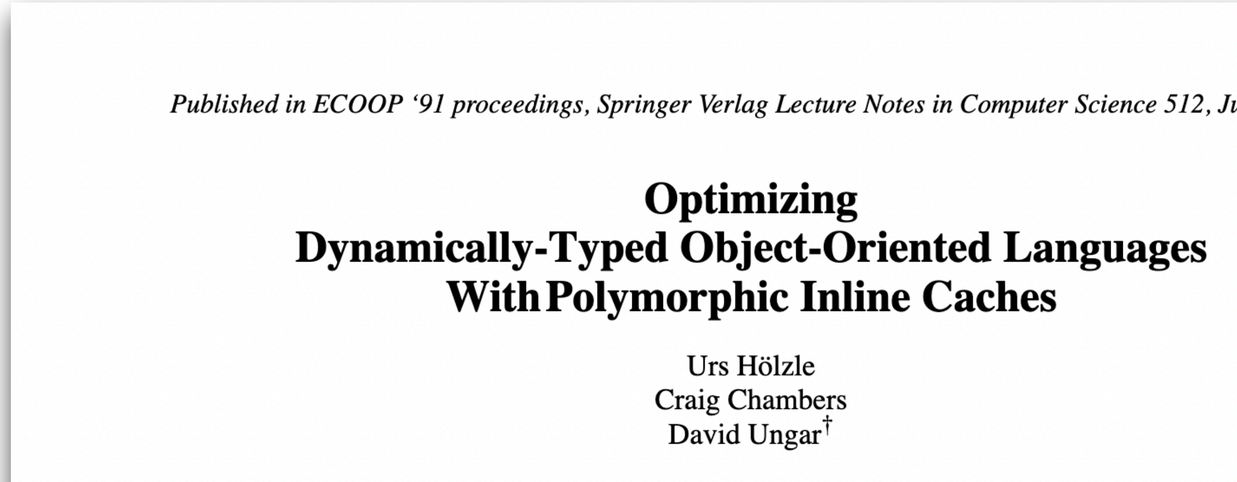


- **Problem:** polymorphic call-sites will often fail+lookup+rewrite
- **Idea:** cache the *last N results* (*e.g.,* as a little switch)




```
void m1(){
  push;
  call m2_PIC;
  pop;
}
```

Type	Method
A	A>>m2
B	A>>m2
C	C>>m2



Published in ECOOP '91 proceedings, Springer Verlag Lecture Notes in Computer Science 512, Ju

Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches

Urs Hölzle
Craig Chambers
David Ungar†

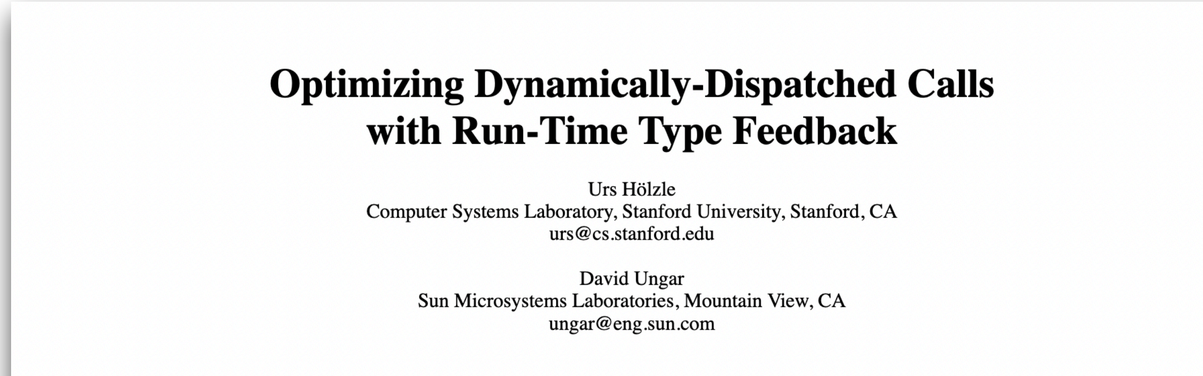
Adaptive Compilation

- PICs give type information for free!
- Optimise for the observed types: speculative inlinings (!!)
- Inlinings generate optimisation opportunities

```
void m1(){
  push;
  call m2_PIC;
  pop;
}
```

Type	Method
A	A>>m2
B	A>>m2
C	C>>m2

```
void m1(){
  push;
  if (type == A)
    // the code of A>>m2
  else call m2_PIC
  pop;
}
```

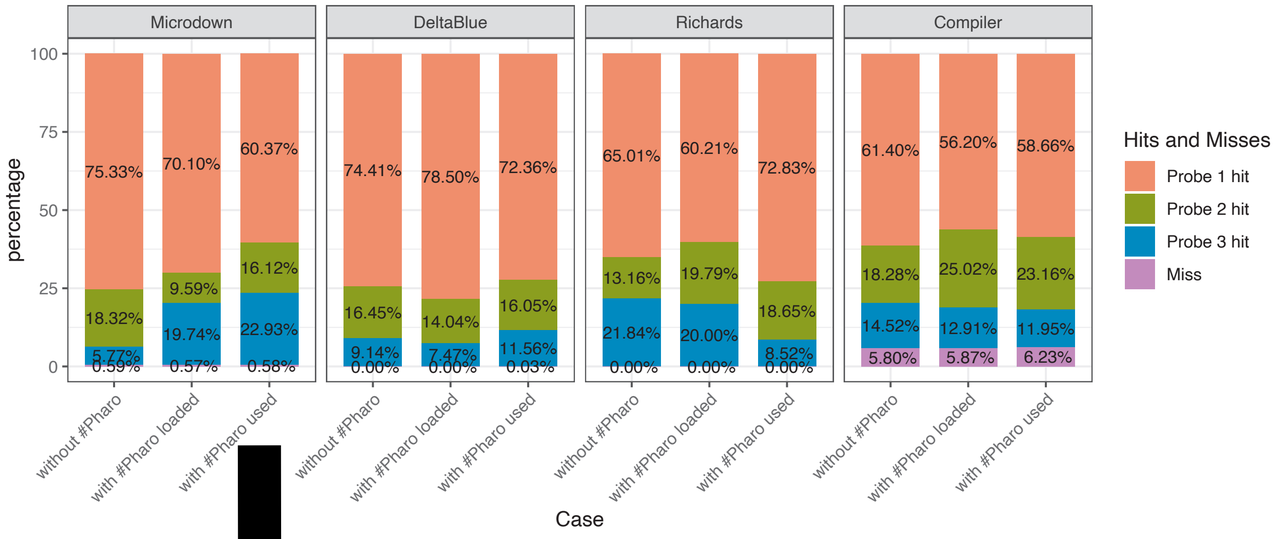


Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback

Urs Hölzle
Computer Systems Laboratory, Stanford University, Stanford, CA
urs@cs.stanford.edu

David Ungar
Sun Microsystems Laboratories, Mountain View, CA
ungar@eng.sun.com

Part 2: Summary




```
void m1(){
  push;
  call TheClass>>#foo;
  push;
  call TheClass>>#bar;
  pop;
}
```

```
void m1(){
  push;
  call m2_PIC;
  pop;
}
```

Type	Method
A	A>>m2
B	A>>m2
C	C>>m2

```
void m1(){
  push;
  if (type == A)
    // the code of A>>m2
  else call m2_PIC
  pop;
}
```

Part 3 - Software Engineering for Language Implementation

Remarkable Challenges in High-Performance VMs

- Managed Execution
- Performance Evaluation
- Memory Management
- Software Engineering
- Security

Tech Report'22

Remarkable Challenges of High-Performance Language Virtual Machines

G. Polito[1], S. Ducasse[1], P. Tesone[1], L. Fabresse[2], G. Thomas[3], M. Bacou[3], P. Cotret[4], and L. Lagadec[4]

[1] *Inria Lille – Nord Europe*
[2] *Telecom Nord Europe*
[3] *Telecom Paris*
[4] *ENSTA Bretagne*

1 Language Virtual Machines : Society Assets

Language Virtual Machines (VMs) are pervasive in every laptop, server, and smartphone, as is the case with Java or Javascript. They allow application portability between different platforms and better usage of resources. They are used in critical applications such as stock exchange, banking, insurance, and health [25]. Virtual machines are an important asset in companies because they allow the efficient execution of high-level programming languages. Nowadays, they even attract investments from large non-system companies, *e.g.,* Netflix [1], Meta [2], Shopify [3] and Amazon [4].

VMs achieve high-performance thanks to aggressive optimization techniques that observe and adapt the execution dynamically, either by doing just-in-time compilation [5] or by adapting the memory management strategies at runtime [90, 91]. For all these reasons Virtual Machines are highly-complex engineering pieces,

Making VMs is HARD and Expensive

- We do not **write a JIT compiler** over a weekend
- Difficult to **maintain and test:** performance vs modularity vs generality
- **Security and JIT** compilation are not close friends
- **New hardware**, new challenges
  - RISC-V processors
  - processor in memory
  - non-volatile and disaggregated memory…

Meta-Compilation

- Developing JIT compilers is HARD and expensive
- Meta-compilation approaches propose to auto-generate them

Tracing the Meta-Level: PyPy's Tracing JIT Compiler

Carl Friedrich Bolz
University of Düsseldorf
STUPS Group
Germany
cfbolz@gmx.de

Antonio Cuni
University of Genova
DISI
Italy
cuni@disi.unige.it

Maciej Fijalkowski
merlinux GmbH
fijal@merlinux.eu

Armin Rigo
arigo@tunes.org

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation, incremental compilers, interpreters, run-time environments*

ABSTRACT

We attempt to apply the technique of Tracing JIT Compilers in the context of the PyPy project, i.e., to programs that are interpreters for some dynamic languages, including Python,

tions use completely straightforward bytecode-interpreters without any advanced implementation techniques like just-in-time compilation. There are a number of reasons for this. Most of them boil down to the inherent complexities of using compilation. Interpreters are simple to implement, understand, extend and port whereas writing a just-in-time compiler is an error-prone task that is made even harder by the dynamic features of a language.

A recent approach to getting better performance for dynamic languages is that of tracing JIT compilers [18, 8].

Practical Partial Evaluation for High-Performance Dynamic Language Runtimes

Thomas Würthinger* Christian Wimmer* Christian Humer* Andreas Wöß*
Lukas Stadler* Chris Seaton* Gilles Duboscq* Doug Simon* Matthias Grimmer†

*Oracle Labs †Institute for System Software, Johannes Kepler University Linz, Austria
{thomas.wuerthinger, christian.wimmer, christian.humer, andreas.woess, lukas.stadler, chris.seaton, gilles.m.duboscq, doug.simon}@oracle.com matthias.grimmer@jku.at

Abstract

Most high-performance dynamic language virtual machines duplicate language semantics in the interpreter, compiler, and runtime system. This violates the principle to not repeat yourself. In contrast, we define languages solely by writing

1. Introduction

High-performance virtual machines (VMs) such as the Java HotSpot VM or the V8 JavaScript VM follow the design that was first implemented for the SELF language [25]: a multi-tier optimization system with adaptive optimization and de-

Duplicated semantics

```
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
```

Code Generation: From Interpreter to Compiler

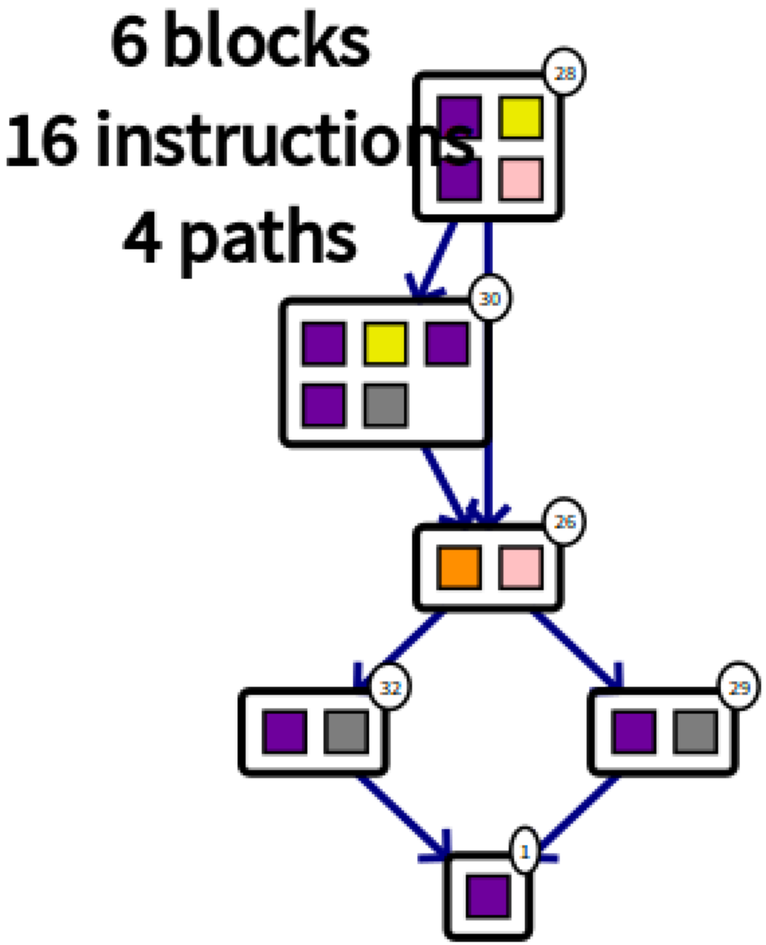
```
Interpreter >> bytecodePrimAdd
  | rcvr arg result |
  rcvr := self internalStackValue: 1.
  arg := self internalStackValue: 0.
  (objectMemory areIntegers: rcvr and: arg) ifTrue: [
    result := (objectMemory integerValueOf: rcvr) + (
      objectMemory integerValueOf: arg).
    "Check for overflow"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

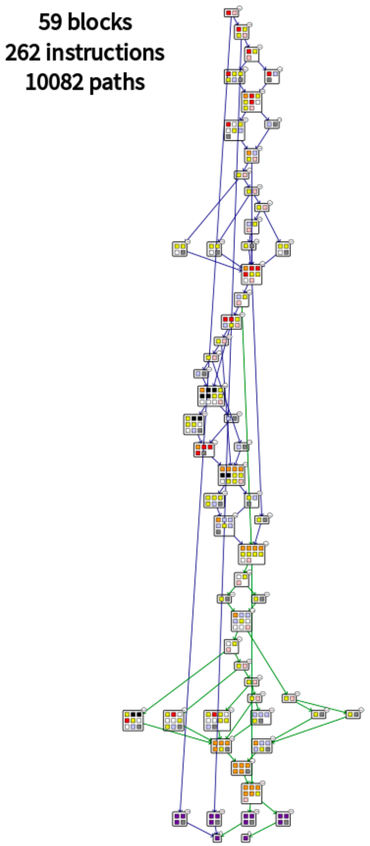
```
  ... # previous bytecode IR
        checkSmallInteger t0
        jumpzero notsmi
        checkSmallInteger t1
        jumpzero notsmi
        t2 := t0 + t1
        jumpIfNotOverflow continue
notsmi: #slow case first send
        t2 := send #+ t0 t1
continue:
  ... # following bytecode IR
```

Druid: a (Meta) Compiler Infrastructure

For Pharo, in Pharo

- Optimizing Compiler
- SSA register-based IR
- Multiple Frontends
  - Pharo code
  - Pharo Meta-interpretation
- Backends
  - JIT compiler code
  - Pharo VM bytecode





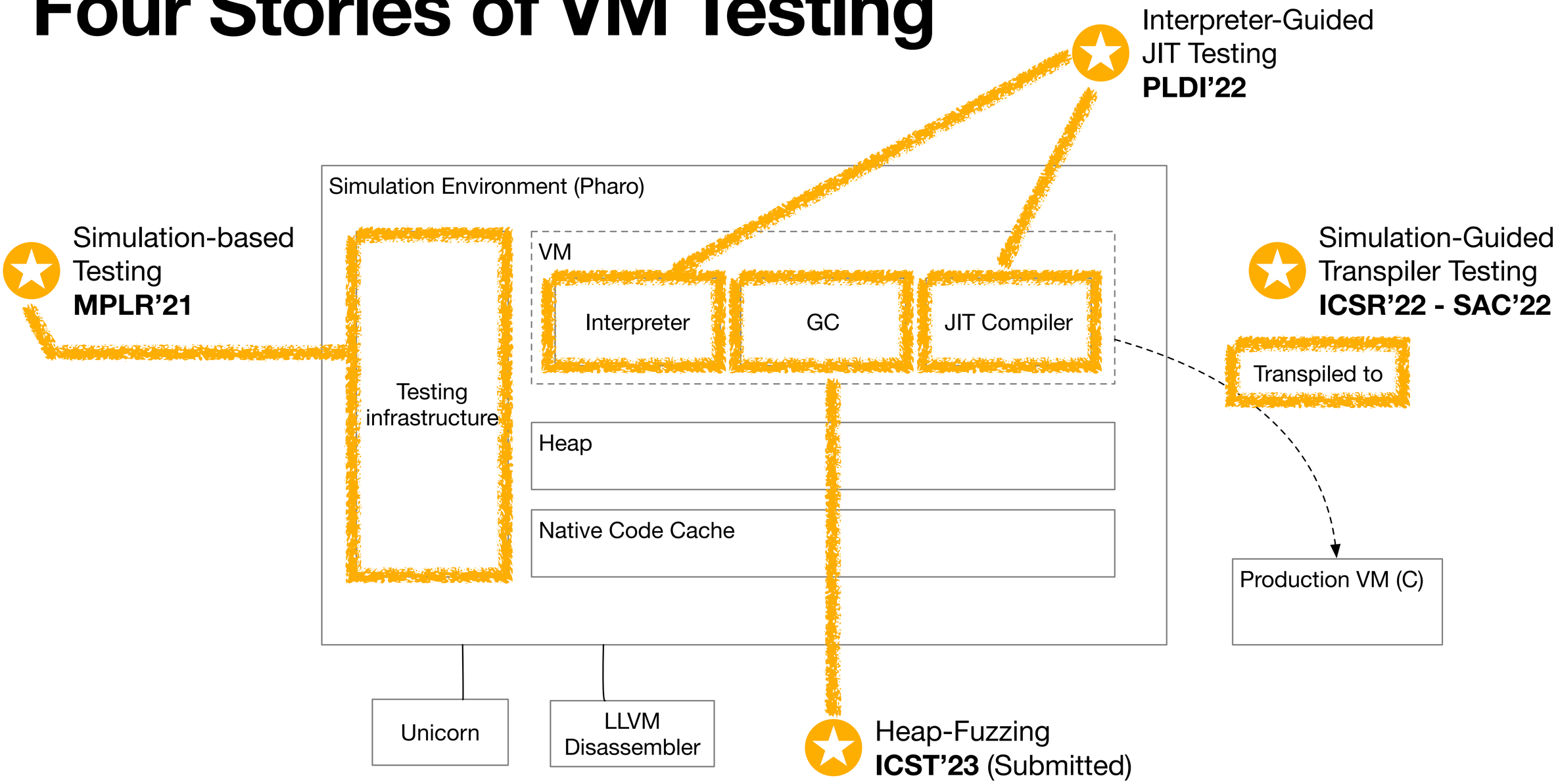
RMod

Evref

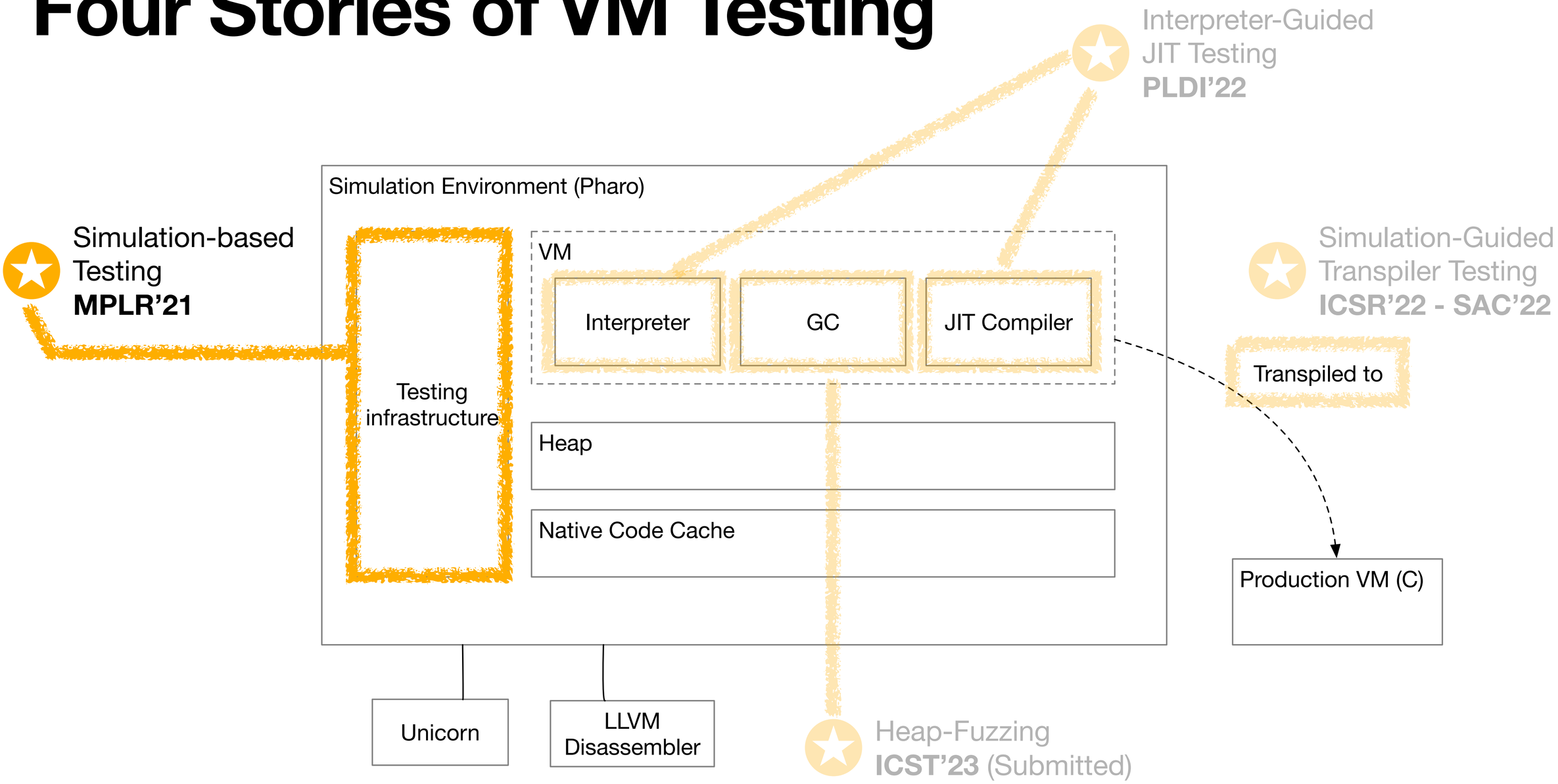
Virtual Machine Testing is HARD

- Hardware **unavailability**
- **Slow** Change-Compile-Test **cycle**
- **Bug reproduction** is a demanding task
- VMs are **complex**, **non-deterministic** beasts

Four Stories of VM Testing

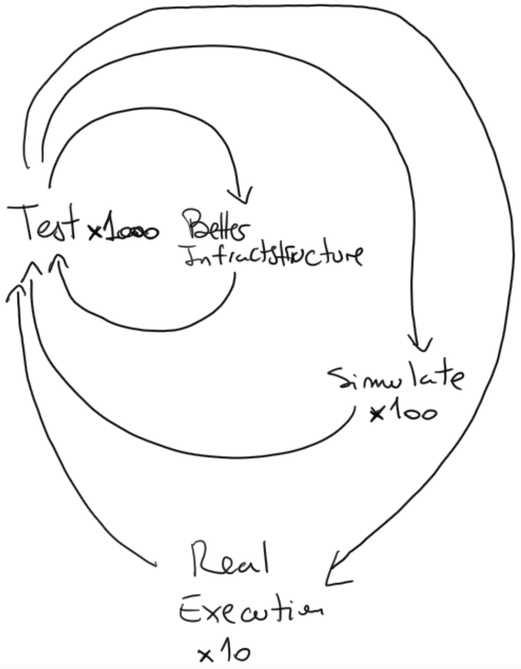


Four Stories of VM Testing



Simulation-Based Unit Testing

- Simulate Hardware
- **Fast** Change-Compile-Test **cycle**
- **Unit testing** for bug reproduction



- Case Studies
  - ARM64 and RISC-V ports
  - Testing GC memory corruptions

MPLR'21

Cross-ISA Testing of the Pharo VM: Lessons Learned While Porting to ARMv8

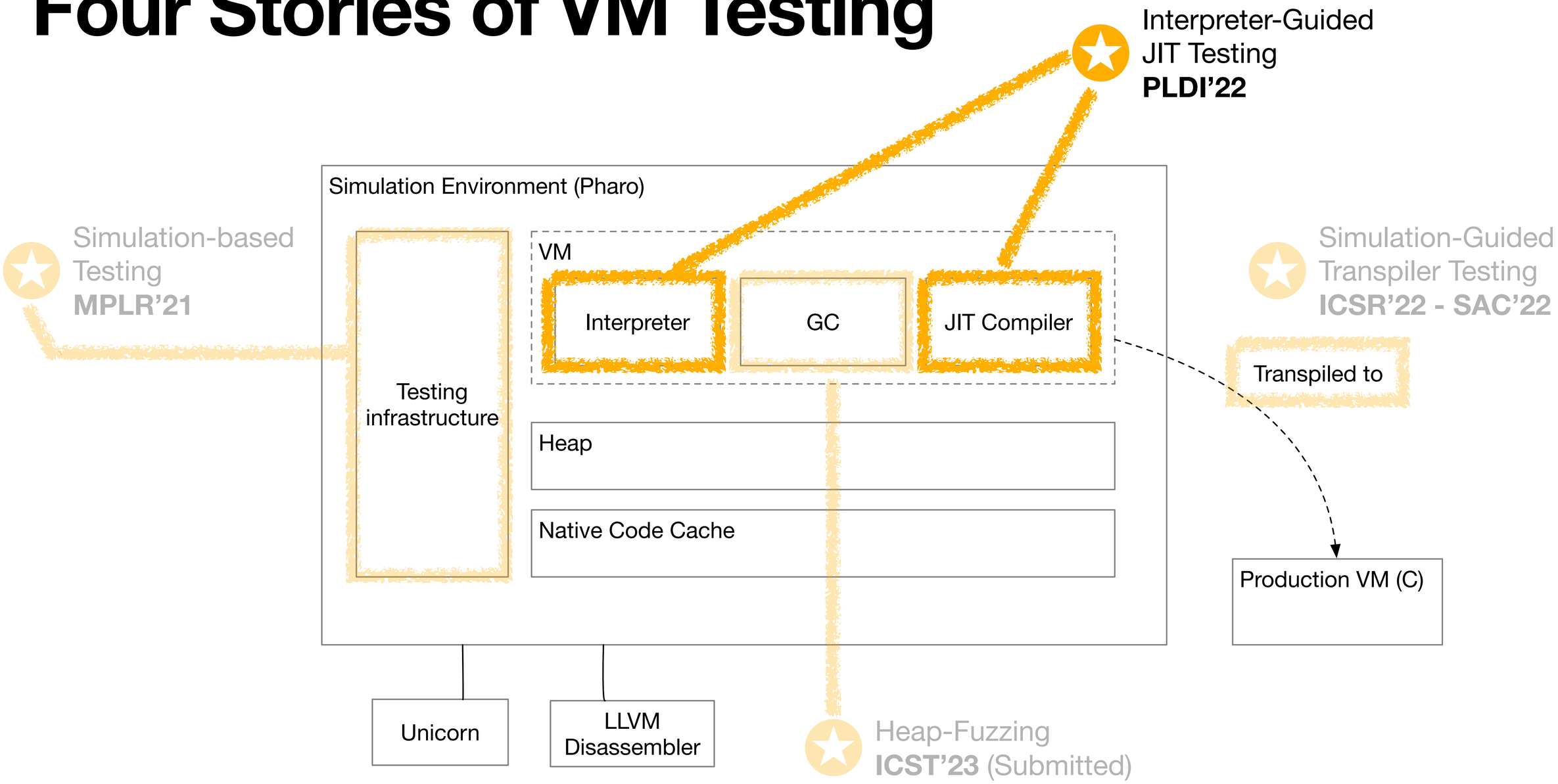
Guillermo Polito
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000
Lille, France
guillermo.polito@univ-lille.fr

Pablo Tesone
Stéphane Ducasse
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
Lille, France
{name}.{surname}@inria.fr

Luc Fabresse
IMT Lille Douai, Institut Mines-Télécom, Univ. Lille, Centre for Digital Systems, F-59000
Lille, France
luc.fabresse@imt-lille-douai.fr

Théo Rogliano
Pierre Misse-Chanabier
Carolina Hernandez Phillips
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL

Four Stories of VM Testing



Duplicated semantics (again)

```
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-Guided Automatic JIT Compiler Unit Testing

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 Lille, 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

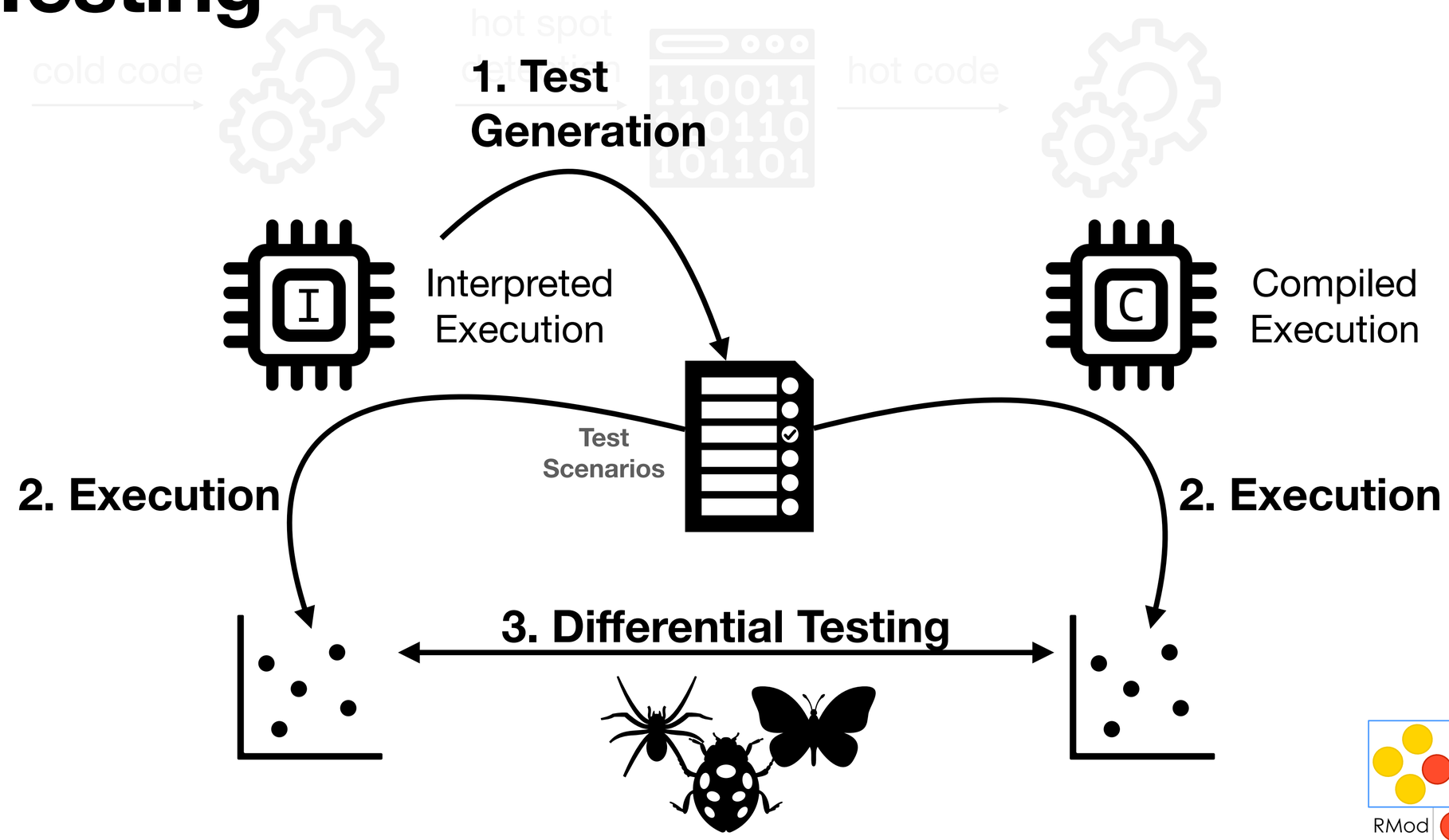
San Diego, CA, USA. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3519939.3523457

1 Introduction

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

Several solutions have been proposed to aid in VM testing tasks. Traditionally, VM simulation environments have ap-

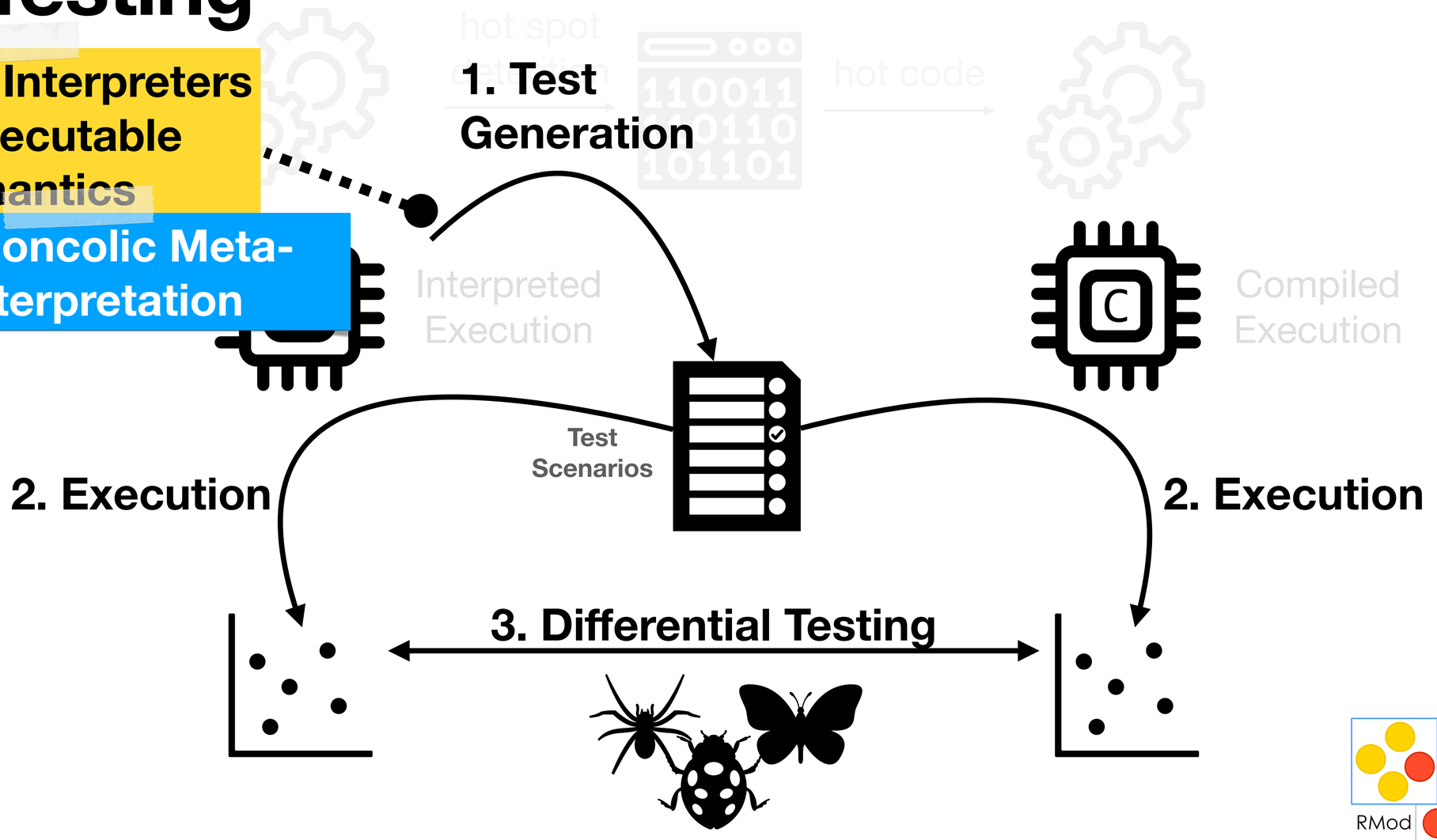
Interpreter-Guided Automatic JIT Compiler Unit Testing



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
```

If both operands are integers

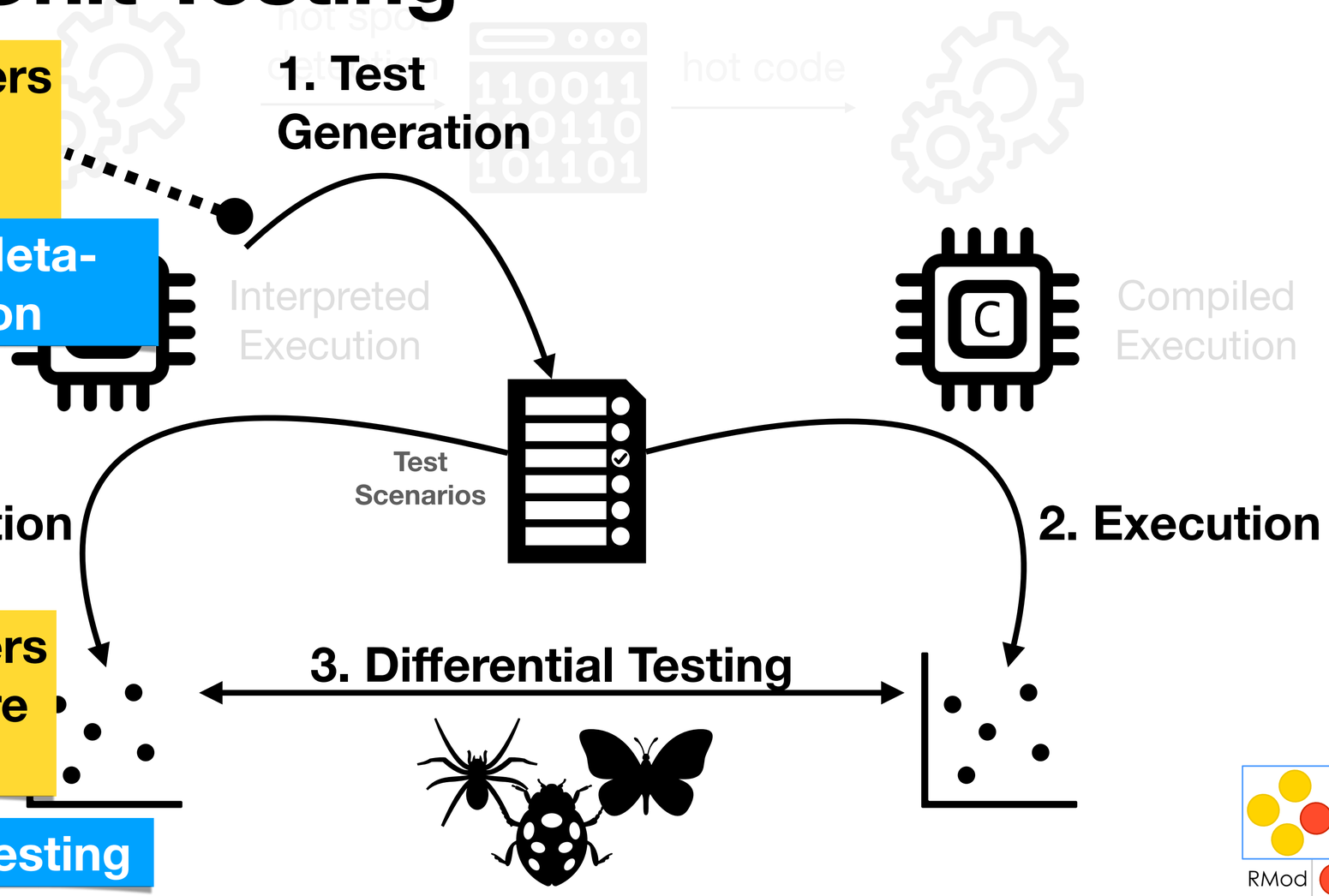
If their sum does not overflow

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
```

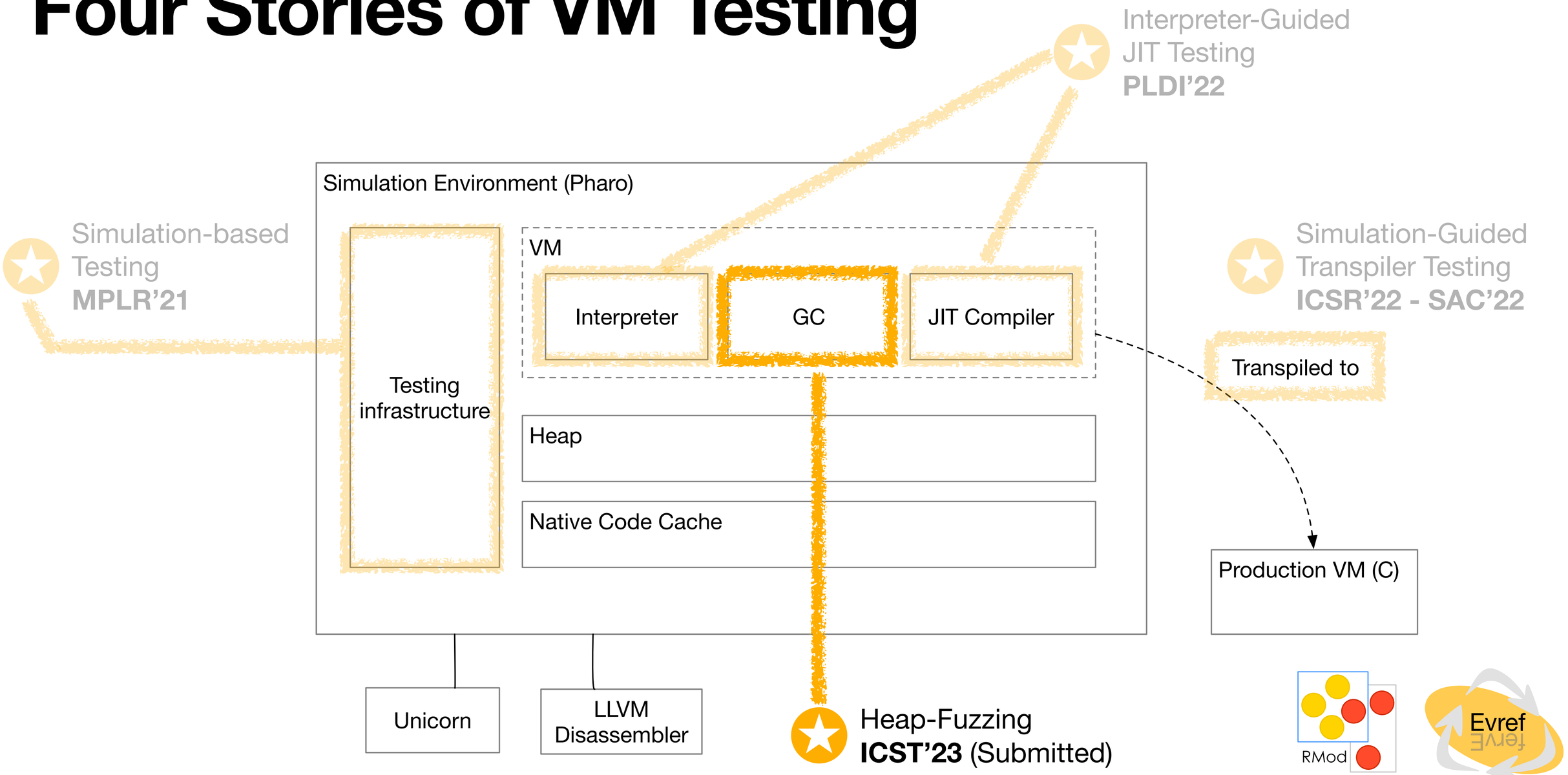
```
... # previous bytecode IR
    checkSmallInteger t0
    jumpzero notsmi
    checkSmallInteger t1
    jumpzero notsmi
    t2 := t0 + t1
    jumpIfNotOverflow continue
  notsmi: #slow case first send
    t2 := send #+ t0 t1
  continue:
    ... # following bytecode IR
```

JIT + Interpreter Bugs!

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

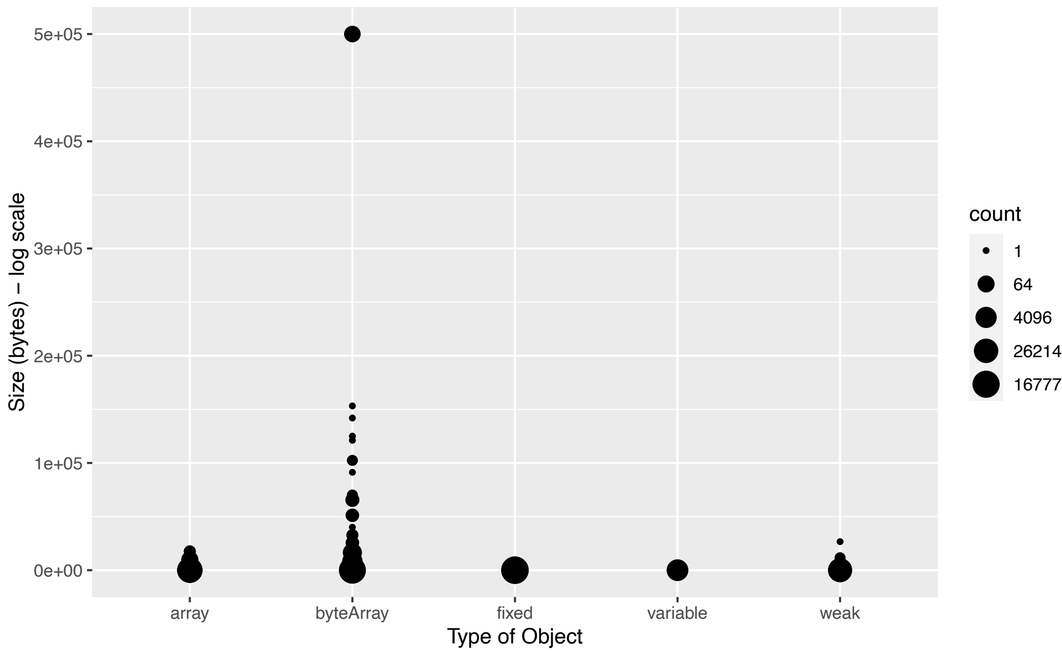
Compiler	# Tested Instructions	# Interpreter Paths	# Curated Paths	# Differences (%)
Native Methods (primitives)	112	2024	1520	440 (28,95%)
Simple Stack BC Compiler	175	1308	1136	18 (1,59%)
Stack-to-Register BC Compiler	175	1308	1136	10 (0,88%)
Linear-Scan Allocator BC Compiler	175	1308	1136	10 (0,88%)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

Four Stories of VM Testing



Garbage Collection Testing

- Applications have stable allocation patterns
  - partial exploration of the *search space*
  - low probability to *hit interesting* scenarios



- GC memory corruptions
  - far away from the *cause*
  - masked by side-effects

ICST'23 (Submitted), **Smalltalks'22**

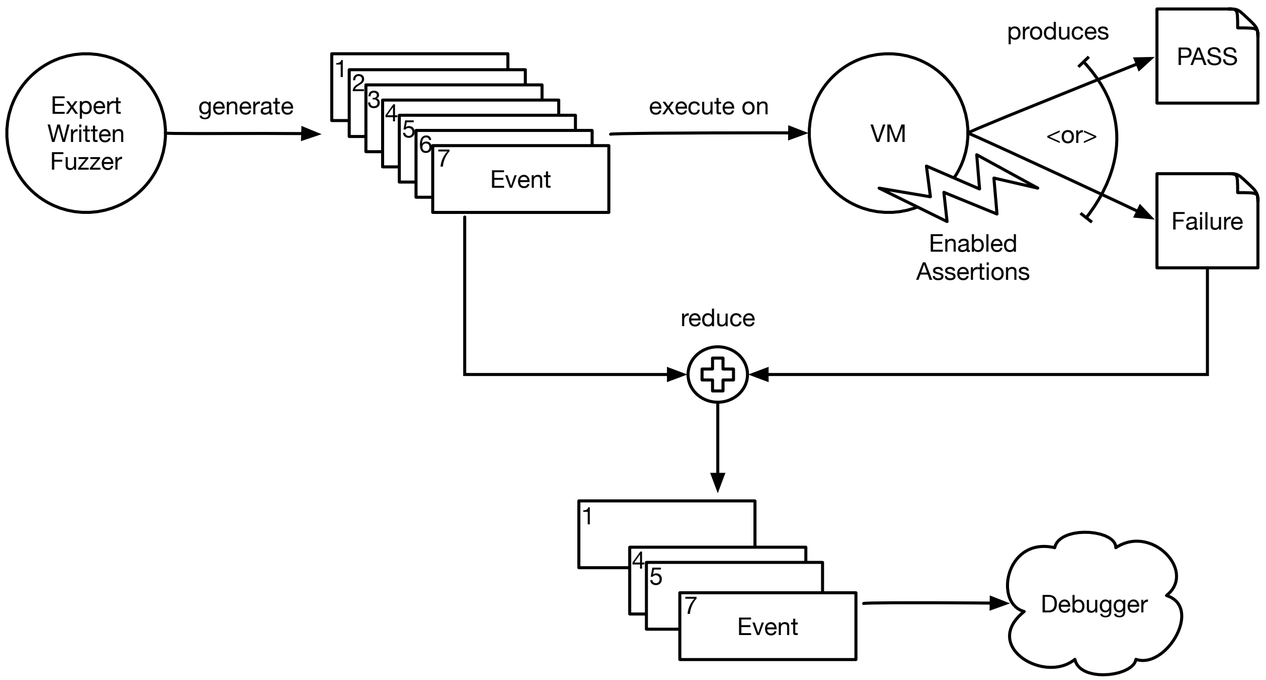
Heap Fuzzing: Automatic Garbage Collection Testing with Expert-Guided Random Events

***Abstract*—Producing robust memory manager implementa-**

such approaches, although shown suitable to find compilation

Heap Fuzzing

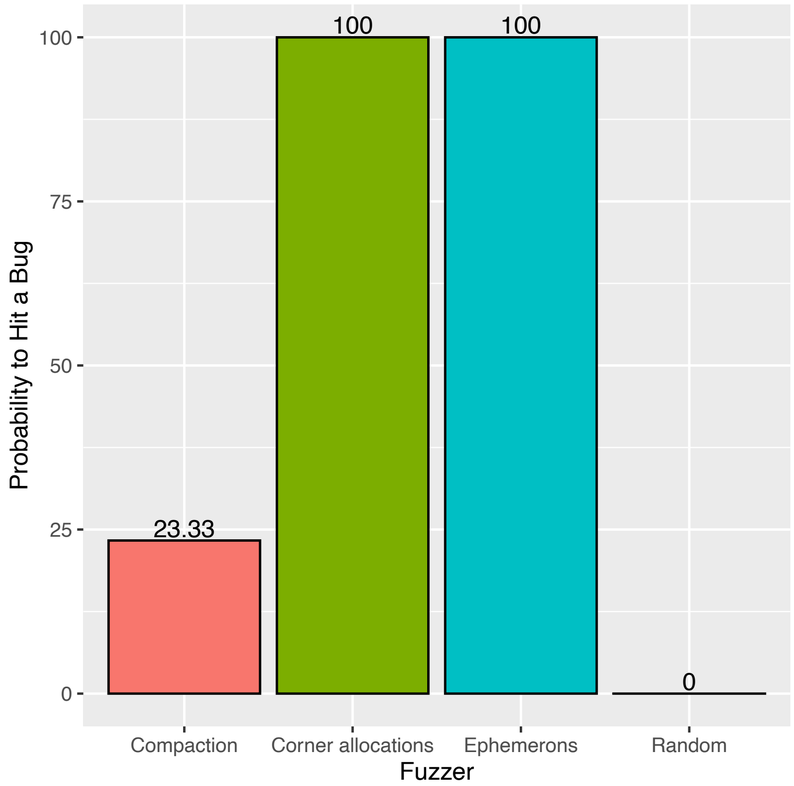
- Fuzz the GC and allocator, not the App (!!)
  - Control + determinism
- Experts guide the fuzzing
  - change event probability
  - add new kind of events

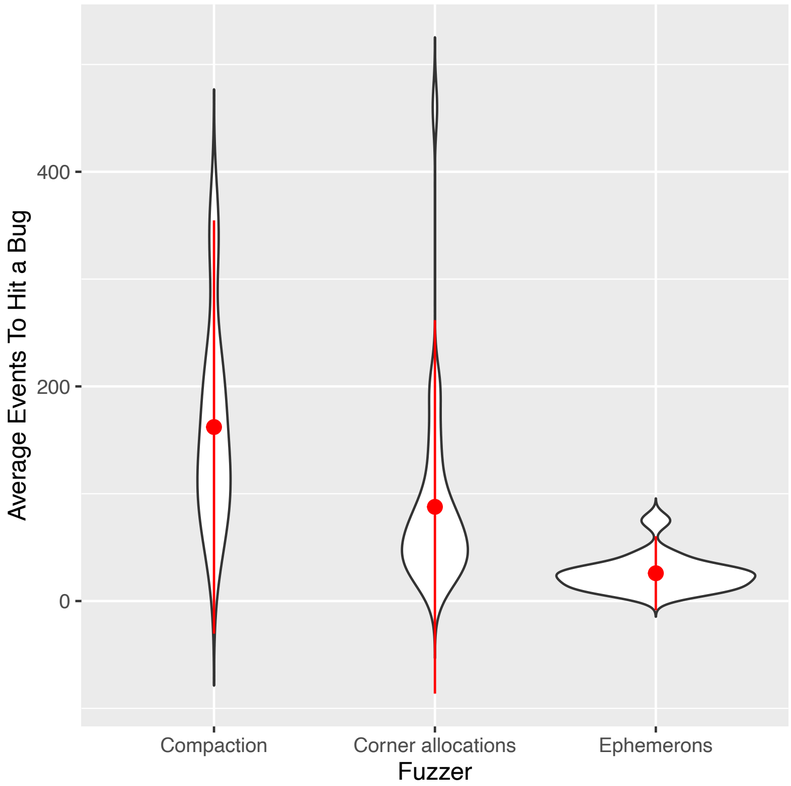


Heap Fuzzing Effectiveness

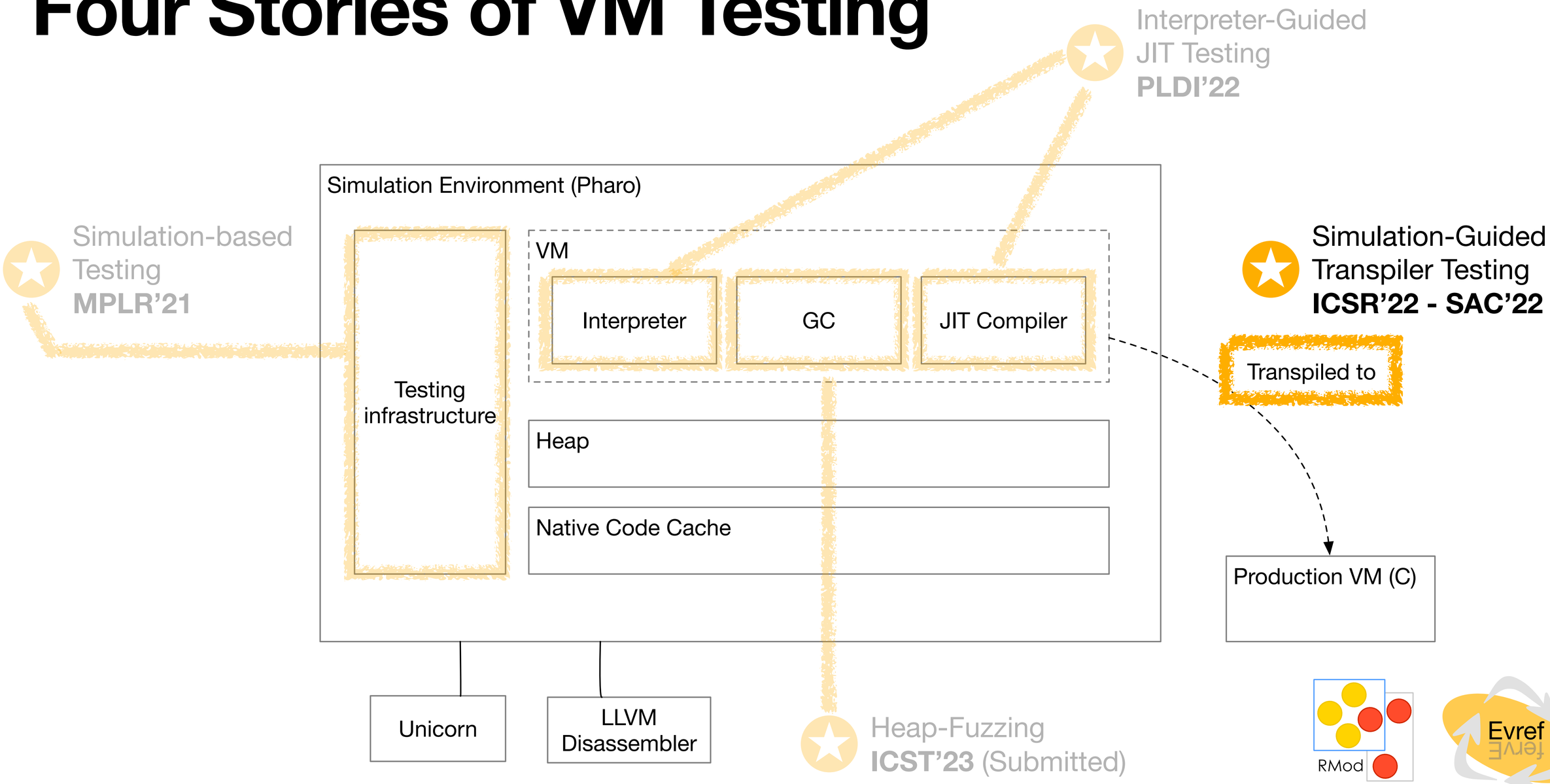
- Expert-guided fuzzers are better than random fuzzing
  - may need refinement
  - practical

- > 6 bugs found **(!)**

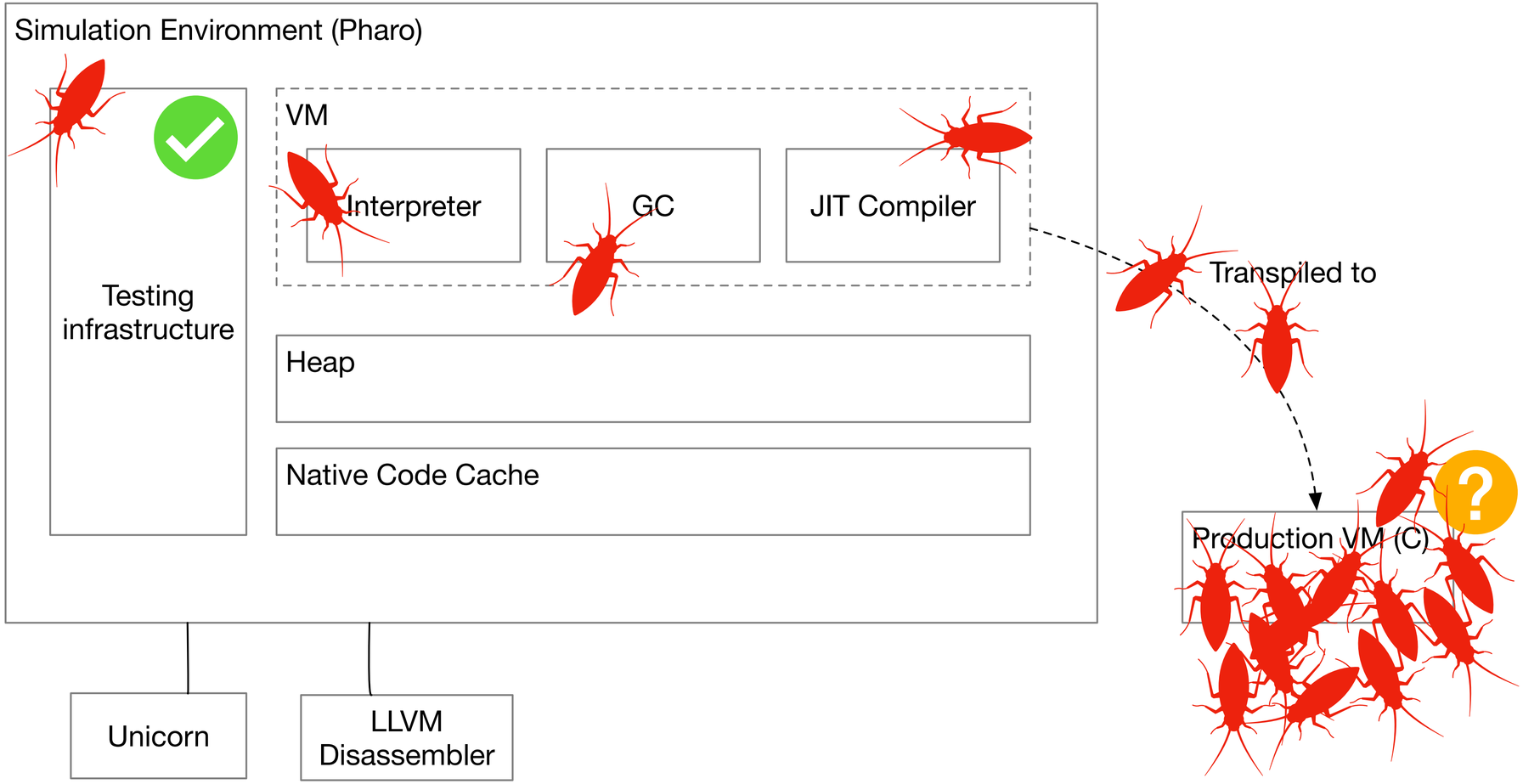




Four Stories of VM Testing



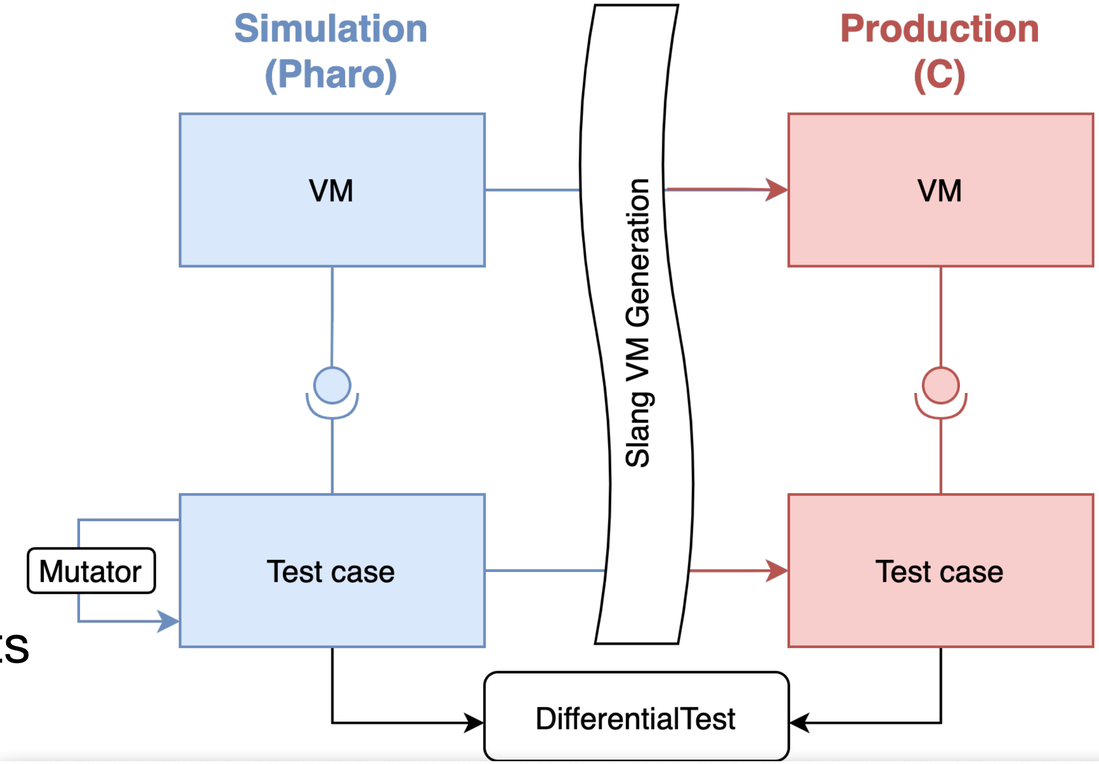
Simulation Semantic Gaps



VM Test Transmutation

- Existing simulation tests as input programs
- Simulation vs Production differential testing
- Non-semantic-preserving mutations add variability

- Experiments: 256 tests, ~500 mutants, ~2k tests
  - Stack allocation + inlining bugs
  - Division translation bugs
  - Assertion behavior differences



ICSR'22

Differential Testing of Simulation-Based Virtual Machine Generators

Automatic Detection of VM Generator Semantic Gaps Between Simulation and Generated VMs

Pierre Misse-chanabier[1], Guillermo Polito[2], Noury Bouraqadi[3], Stéphane Ducasse[1],

And more, already here or coming

- VM Debugging
- Code Cache Profiling
- Benchmark Generation
- Deep JIT fuzzing
- Speculative Compilation
- Vectorisation
- …

MPLR'22

Porting a JIT compiler t
Oppor

Quentin Ducasse [1], Guille
Pascal Cotret

[1] Laboratoire Lab-STICC - ENS
firstname.lastname
[2] CNRS, INRIA - (

The RISC-V Ins
sible ISA. The abilit
to accelerate VM co
on top. However, t
and therefore softwa
codes and instructio
present the challeng
ISA, on RISC-V, W

VMIL'22

Interpreter Register Autolocalisation: Improvin
performance of efficient interpreters

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

Nahuel Palumbo
Soufyane Labsari
Stéphane Ducasse
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
name.surname@inria.fr

Pablo T
Pharo Con
Univ. Lille, Inria, CNRS
9189 CR
pablo.teson

KEYWORDS

These two requirements are opposing

MPLR'21

Profiling Code Cache Behaviour via Events

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

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

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

Abstract

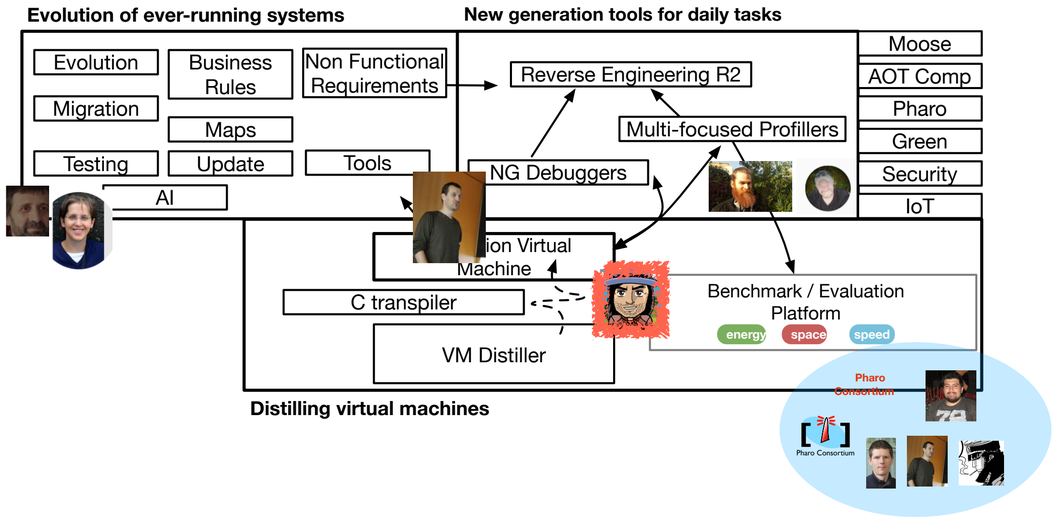
Virtual machine performance tuning for a given application is an arduous and challenging task. For example, parametrizing the behaviour of the JIT compiler machine code caches affects the overall performance of applications while being rather obscure for final users not knowledgeable about VM internals. Moreover, VM components are often heavily coupled and changes in some parameters may affect several seemingly unrelated components and may have unclear performance impacts. Therefore, choosing the best parametrization requires to have precise information

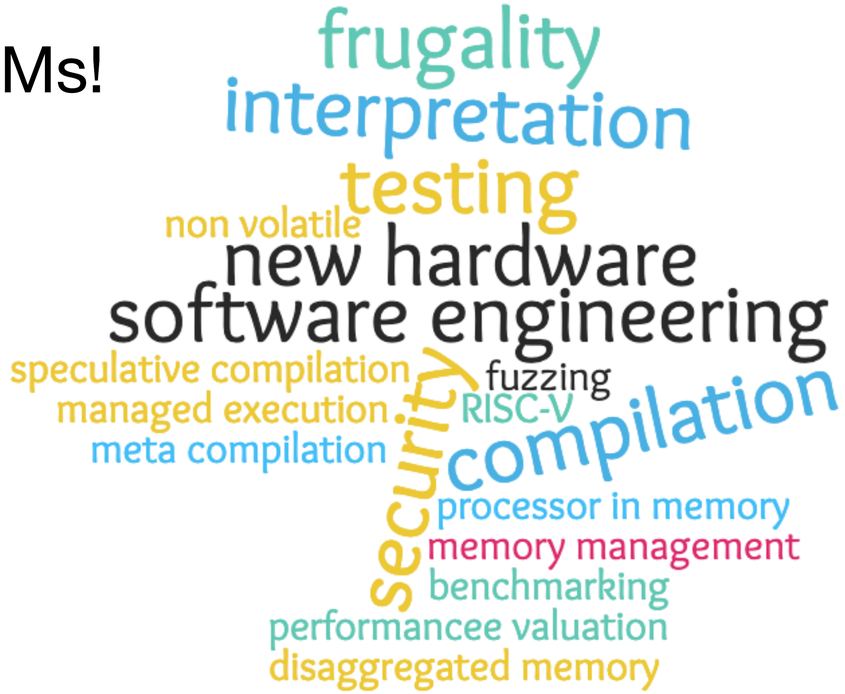
any more space for new compiled methods, some existing methods should be discarded, and the code cache should be compacted to allow new methods to be compiled [10, 19].

All the behaviour of the code cache is governed by a set of parameters: *e.g.,* cache size, initial set of VM routines and primitives, minimum size to recover after compaction, method retention policy, and maximum length of the methods to compile in the cache. In the scope of the paper, we name *specific configuration* a given set of parameter values. Due to their close interdependencies, we refer to them as a single entity.

Takeovers

- **RMoD**: Software Evolution, Language Design & Implementation
- Techniques in the 80s, 90s are the basis of today's VMs!
- Unique and exciting challenges!





guillermo.polito@inria.fr
@guillep

Extra - Concolic Testing

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

- **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

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

Example

Argument 0 (type)	Argument 1(type)	Path
0 (integer)	0 (integer)	isInteger(arg0), isInteger(arg1), isInteger(arg0+arg1)
0xFFFFFFFF (integer)	1 (integer)	isInteger(arg0), isInteger(arg1), isNotInteger(arg0+arg1)
0 (integer)	object1 (object)	isInteger(arg0), isNotInteger(arg1)
object1 (object)	0 (integer)	isNotInteger(arg0), isInteger(arg1)
object1 (object)	object2 (object)	isNotInteger(arg0), isNotInteger(arg1)

```
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
```

Listing 1. Excerpt of the byte-code interpretation implementing addition in the Pharo Virtual Machine.

```
    ... # 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
```

Listing 2. Illustration of the Intermediate Representation instructions created when compiling the byte-code instruction in Listing 1.

Analysis of Differences through Manual Inspection

Interpreter-Guided JIT Compiler Unit Testing

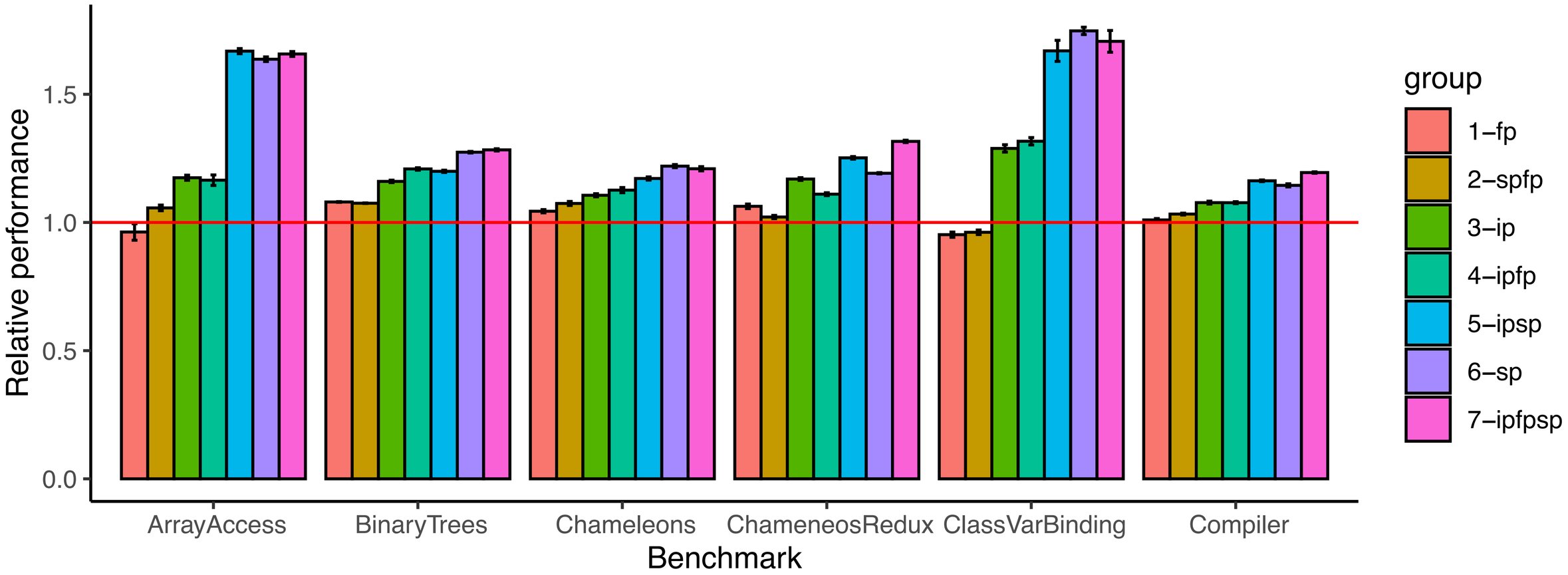
- 91 causes, *6 different categories*
- Errors both in the interpreter AND the compilers
- 14 causes of ***segmentation faults***!

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

Extra - Interpreter Register Autolocalisation

Some Benchmarks

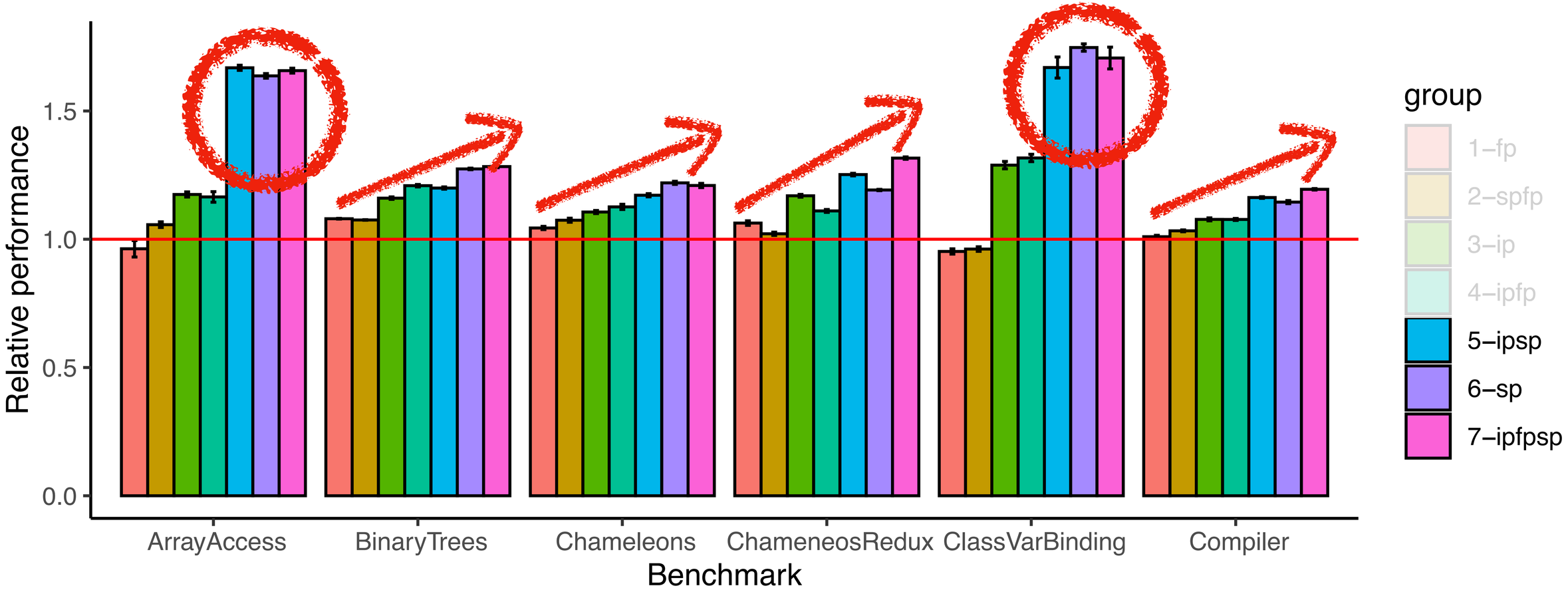
Intel x86-64



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

Some Benchmarks

Intel x86-64

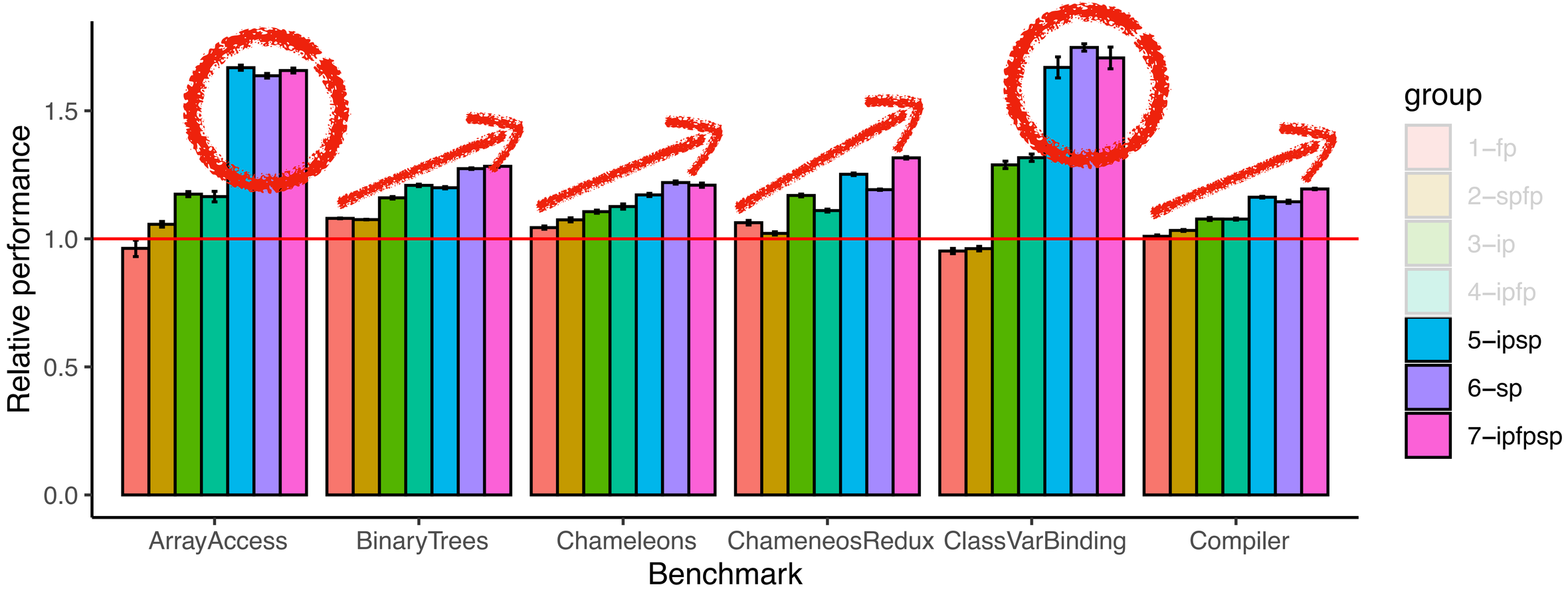


Averages of 100 iterations + stdev. Relative to baseline (no optimisation). Higher is better.

Some Benchmarks

Intel x86-64

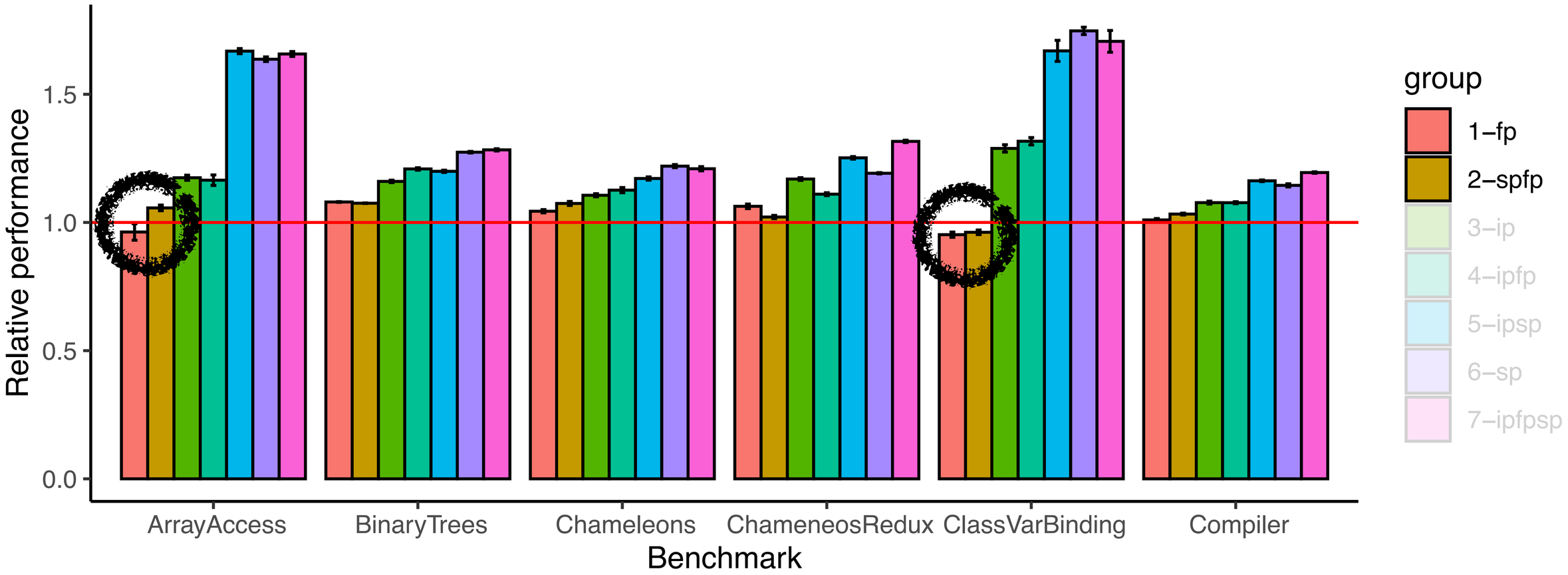
Is there an optional combination for different setups? 🤔



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

Some Benchmarks

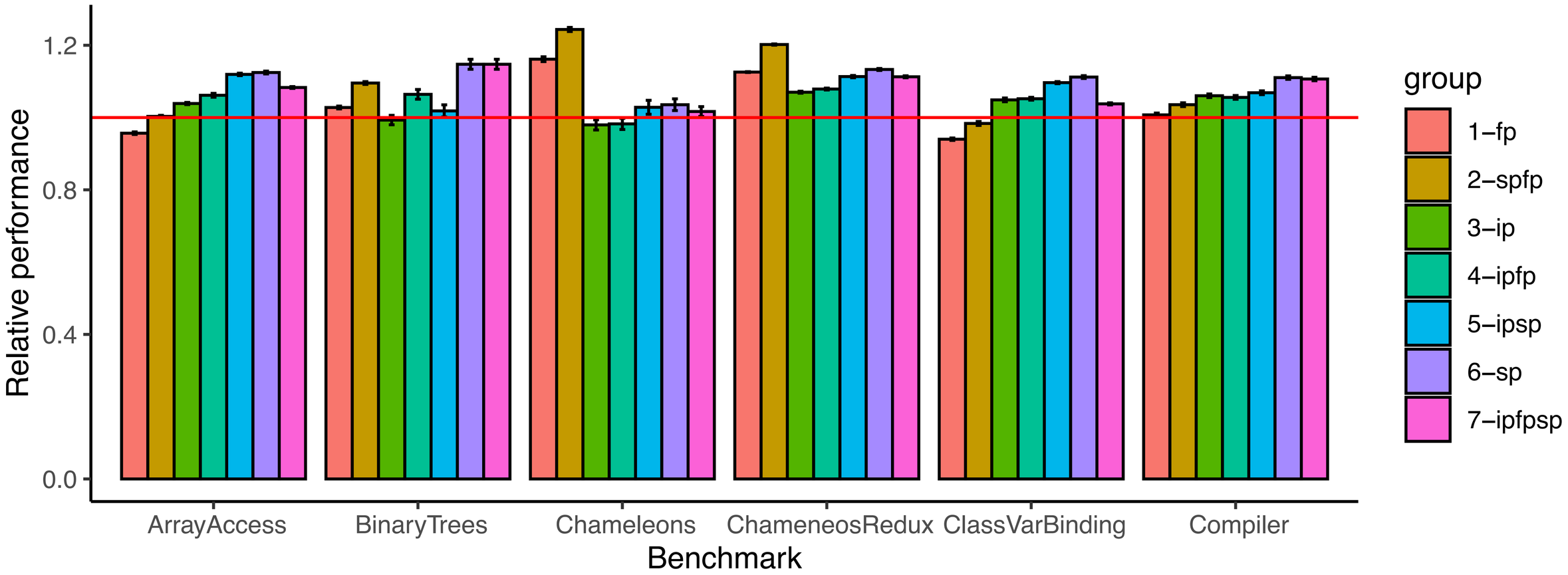
Intel x86-64



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

Some Benchmarks

ARM64 - Raspberry Pi

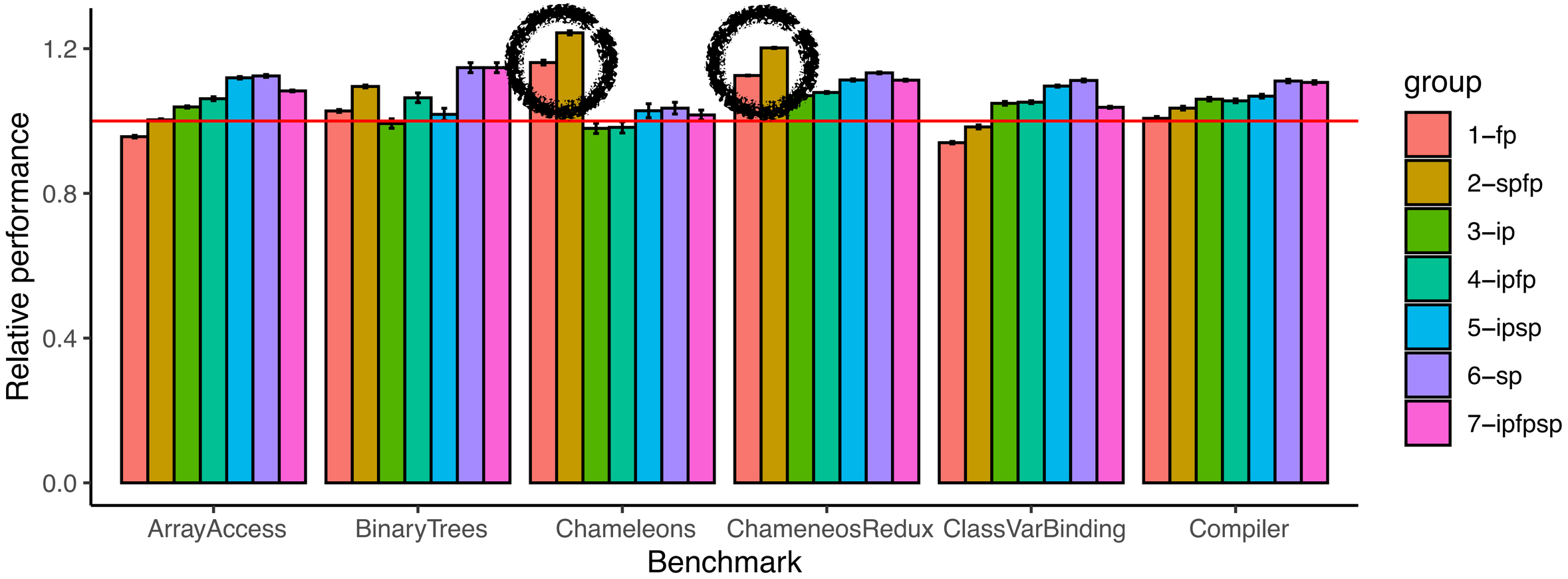


Averages of 100 iterations + stdev. Relative to baseline (no optimisation). Higher is better.

Some Benchmarks

ARM64 - Raspberry Pi

- Study the impact in different architectures
- Study the CPU and cache impact of these optimizations



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

Extra - RISC-V Port

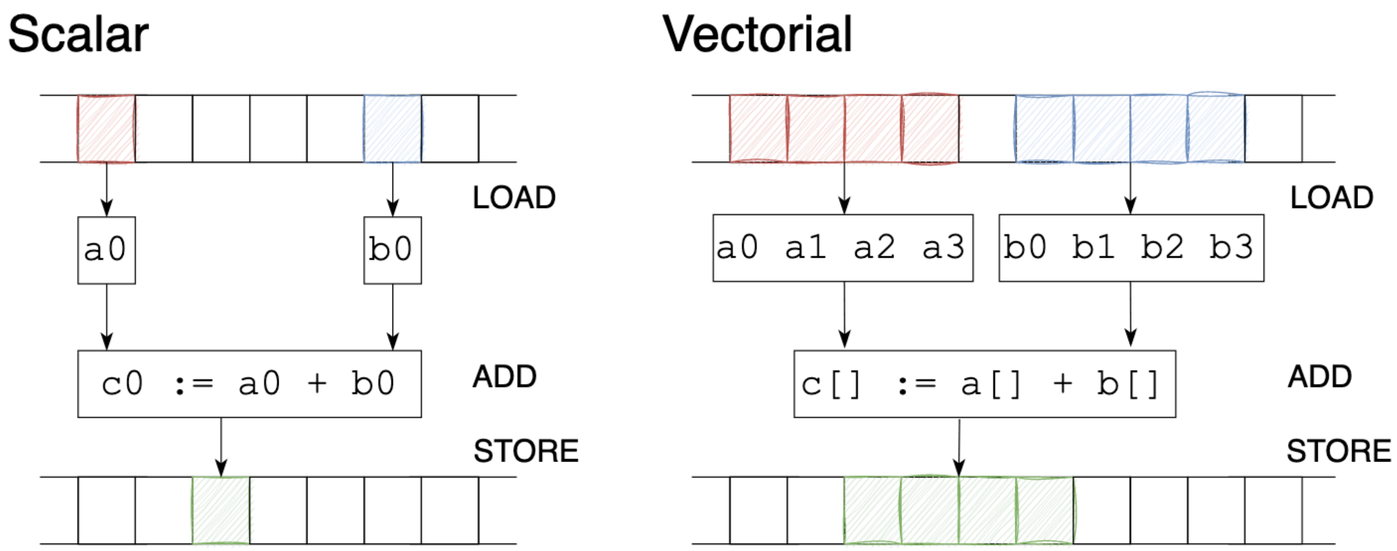
Ongoing RISCV64 Port

- 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*

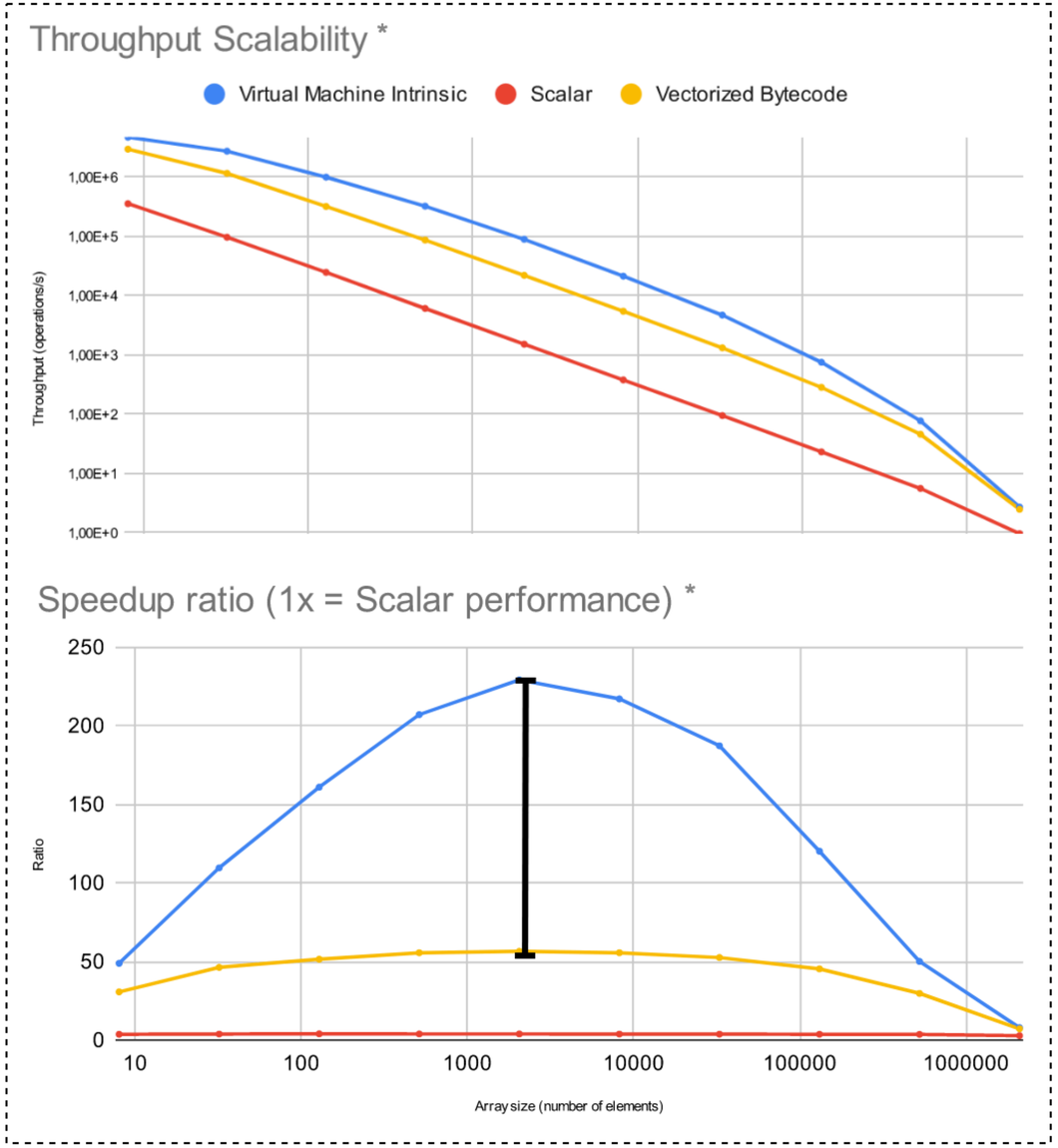
Extra - SIMD + Vectorisation

Single Instruction Multiple Data Extensions



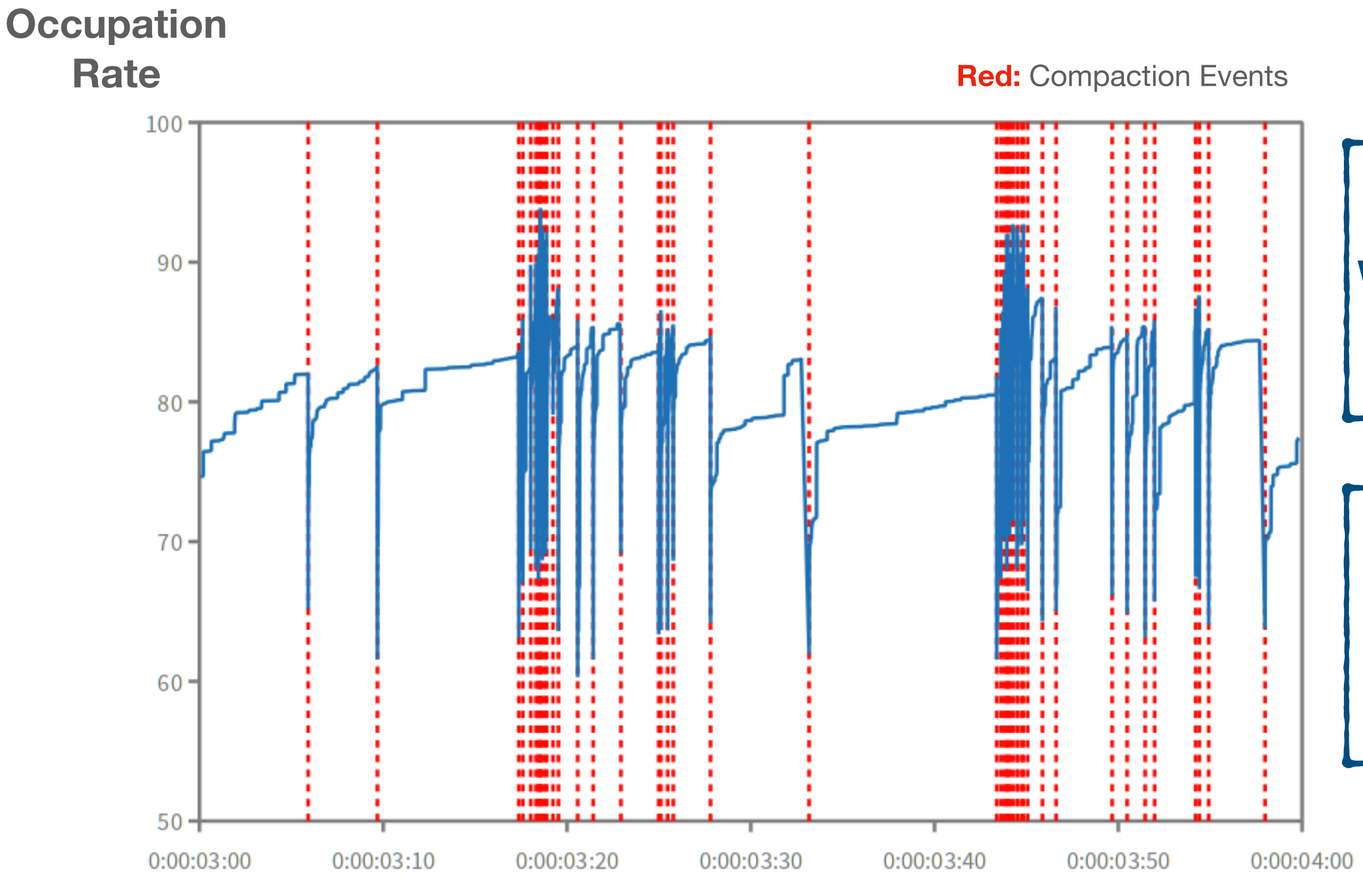
SIMD Design Space

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



Extra - Code Cache Profiling Story

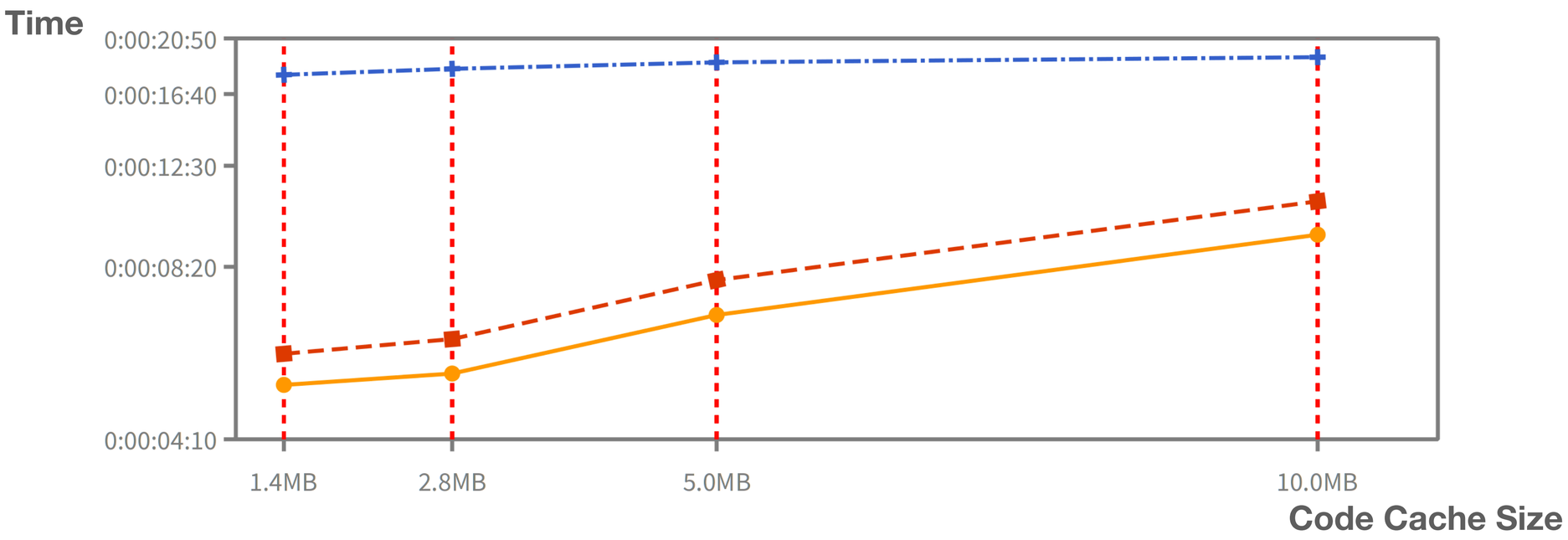
Analysing Code Cache Behavior



Analysing Events
We see trashing in the code cache

We need to increase the size of the code cache

Code Cache Unexpected results



Young Space 1MB 10 MB 100 MB

Loading Moose

Extra - JITt’ed Code Debugging

JIT Code Debugger

Address	Name
16r1000000	ceCaptureCStackPointers
16r1000070	ceEnterCogCodePopReceiverReg
16r10000A0	ceCallCogCodePopReceiverReg
16r10000D8	ceCallCogCodePopReceiverAndClassRegs
16r1000118	cePrimReturnEnterCogCode
16r10001B8	cePrimReturnEnterCogCodeProfiling
16r10002A8	send0argsTrampoline
16r10002B0	send1argsTrampoline
16r10002B8	send2argsTrampoline
16r10002C0	send3argsTrampoline
16r10002C8	ceCPICMissTrampoline
16r10002D0	cePICAbortTrampoline
16r10002D8	ceMethodAbortTrampoline
16r1000530	methodZoneBase

Name	Op1	Op2	Op3
MoveCqR	0	ReceiverResul	
PushR	LinkReg		
Call	16r10002D8/1		
AlignmentNop	8		
Label	1		
AndCqRR	7	ReceiverResul	TempReg
JumpNonZerc	(Label 2)		
MoveMwrR	0	ReceiverResul	TempReg
AndCqR	16r3FFFFF/41	TempReg	
Nop			
Nop			
Label	2		
CmpRR	ClassReg	TempReg	
JumpNonZerc	(PushR 1 FF81		
Label	3		
RetN	0		
AlignmentNop	8		
Literal	16r3FFFFF/41		

Address		ASM	Bytes
16r10008C8		mv s8, zero	#[19 12 0 0]
16r10008CC		addi sp, sp, -8	#[19 1 129 255
16r10008D0		sd ra, 0(sp)	#[35 48 17 0]
16r10008D4		auipc t0, 0	#[151 2 0 0]
16r10008D8	➡	jalr -1532(t0)	#[231 128 66 1
16r10008DC		nop	#[19 0 0 0]
16r10008E0		andi s6, s8, 7	#[19 123 124 0
16r10008E4		mv t5, zero	#[19 15 0 0]
16r10008E8		mv t6, zero	#[147 15 0 0]
16r10008EC		slti t4, s6, 0	#[147 46 11 0]
16r10008F0		seqz t3, s6	#[19 62 27 0]
16r10008F4		beqz t3, 44	#[99 6 14 2]
16r10008F8		ld s6, 0(s8)	#[3 59 12 0]
16r10008FC		auipc t0, 0	#[151 2 0 0]
16r1000900		ld t0, 76(t0)	#[131 178 194
16r1000904		and s6, s6, t0	#[51 123 91 0]
16r1000908		mv t5, zero	#[19 15 0 0]
16r100090C		mv t6, zero	#[147 15 0 0]
16r1000910		slti t4, s6, 0	#[147 46 11 0]
16r1000914		seqz t3, s6	#[19 62 27 0]
16r1000918		nop	#[19 0 0 0]
16r100091C		nop	#[19 0 0 0]
16r1000920		sub t3, s6, s7	#[51 14 123 65
16r1000924		slti t1, s7, 1	#[19 163 27 0]
16r1000928		slt t2, t3, s6	#[179 35 110 1
16r100092C		xor t5, t1, t2	#[51 79 115 0]
16r1000930		seqz t5, t5	#[19 63 31 0]
16r1000934		sltu t6, s6, t3	#[179 63 203 1

Name	Machine Alias	Smalltalk Alias	Value
fp			'16r1003000
x0	zero		'16r0'
x1	ra		'16r1001000
x2	sp	sp	'16r1002FE8
x3	gp		'16r0'
x4	tp		'16r0'
x5	t0	ip1	'16r10008D4
x6	t1	ip2	'16r0'
x7	t2	ip3	'16r0'
x8	fp	fp	'16r1003000
x9	s1		'16r0'
x10	a0	carg0	'16r0'
x11	a1	carg1	'16r0'
x12	a2	carg2	'16r0'
x13	a3	carg3/arg0	'16r0'
x14	a4	arg1	'16r0'
x15	a5		'16r0'
x16	a6		'16r0'
x17	a7		'16r0'
x18	s2	extra0	'16r0'
x19	s3	extra1	'16r0'
x20	s4	extra2	'16r0'
x21	s5		'16r0'
x22	s6	temp	'16r1012F70
x23	s7	class	'16r0'
x24	s8	receiver	'16r0'
x25	s9	argnum	'16r0'
x26	s10	varbase	'16r7FFFFFF

Pointer	Address	Value
	16r1002FC8	16r0
	16r1002FD0	16r0
	16r1002FD8	16r0
	16r1002FE0	16r0
SP	16r1002FE8	16r1001000
	16r1002FF0	16r0
	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

Disassemble Trampoline | Step | 16r10008C8 Jump to | Disassemble at PC | Set SP to | Refresh Stack