M1 - Compiler Testing 101 & Best Engineering Practices

Guillermo Polito

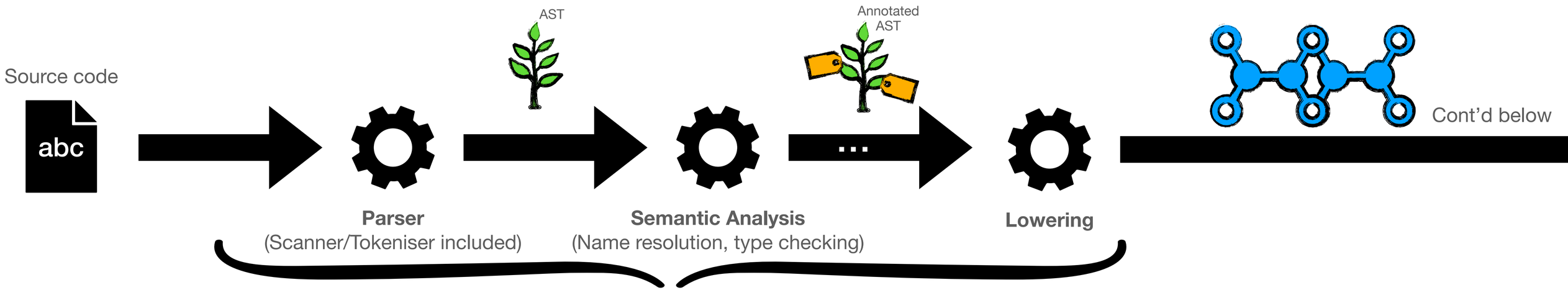
guillermo.polito@inria.fr
@guillep

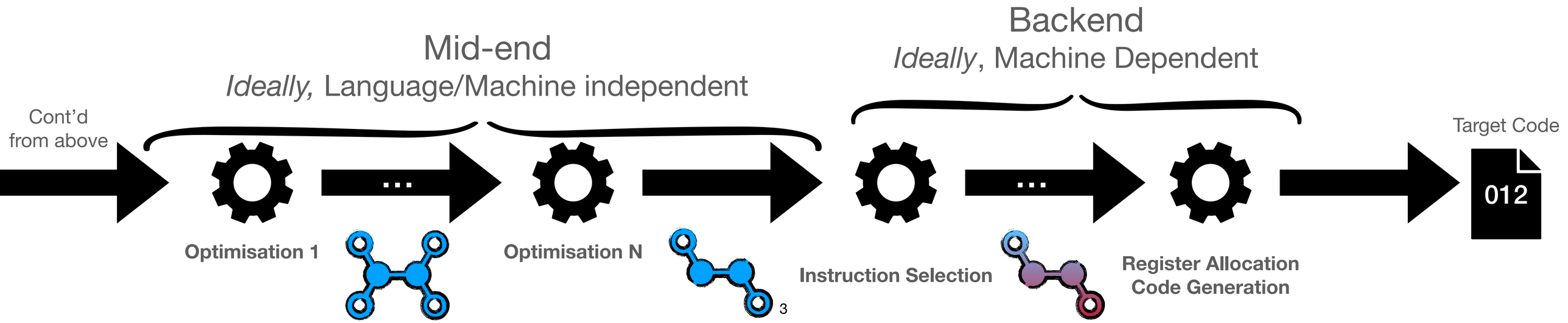
Goals

- Understand how compiler architecture influences the way we test it
- Trade-offs of compiler unit tests
- End-to-end compiler regression tests

- How we can use CPU emulators and simulations in our favor

Compiler Pipelines



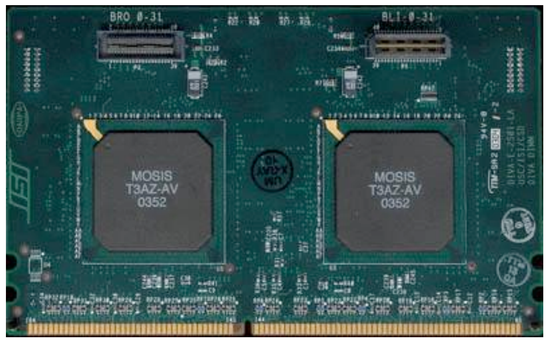


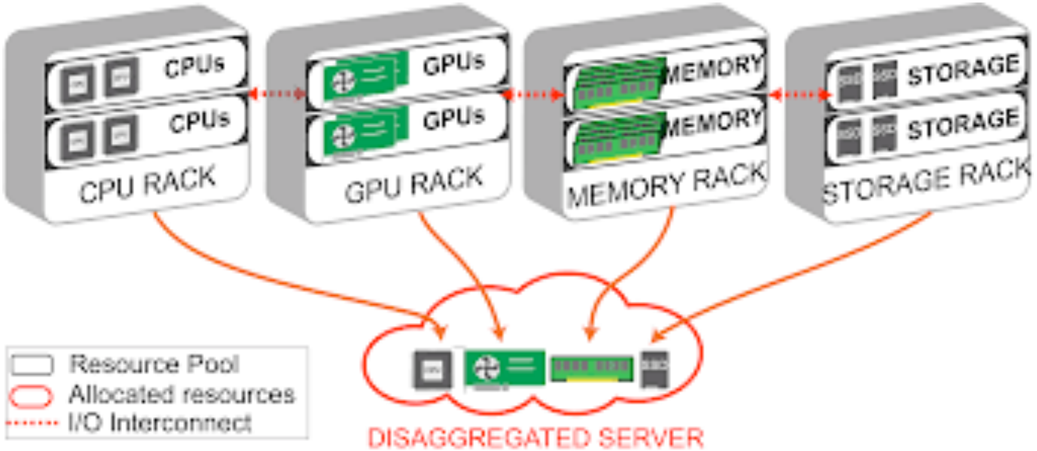
Why do we need Compiler Testing?

- Languages **evolve**
  - C++ ’11, ’14, ’17, ’20, ‘23
  - C ’11, ’17, ’23
  - Javascript has yearly releases
- Compilers too
  - Rust ~1 release per month!
  - Python, Ruby, Java, JS, Lua have many implementations for the spec

Hardware Evolves

- GPUs
- FPGAs
- Enclaves
- Non-volatile RAM
- Disaggregated Memory
- Processor in Memory
- Extensible ISAs





Importance of Compiler Testing - Examples

- 2023 - 122 loop bugs (LLVM, GCC & others)
- 2022 - 6 JVM bugs/CVEs
- 2022 - 91 bugs Pharo VM

ASE'22

Compiler Testing using Template Java Programs

Zhiqiang Zang
The University of Texas at Austin
Austin, Texas, USA
zhiqiang.zang@utexas.edu

Nathan Wiatrek
The University of Texas at Austi
Austin, Texas, USA
nwiatrek@utexas.edu

PLDI'22

Interpreter-guided Differential JIT Compiler U Testing

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

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

Stéphane Ducass
Univ. Lille, Inria, CNRS, Cent
UMR 9189 CRIStAL
stephane.ducasse@inri

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

switch views between the program-level and th
tation (VM)-level. These solutions are indeed
identify and track problems once an issue has
and reproduced. However, reproducing bugs sti
expensive and time-consuming task because n
structions may need to be executed before hitti

PLDI'23

Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages

VSEVOLOD LIVINSKII, University of Utah, USA
DMITRY BABOKIN, Intel Corporation, USA
JOHN REGEHR, University of Utah, USA

Compilers are part of the foundation upon which software systems are built; they need to be as co
as possible. This paper is about stress-testing loop optimizers; it presents a major reimplementation o
Another Random Program Generator (YARPGen), an open-source generative compiler fuzzer. This new ve
has found 122 bugs, both in compilers for data-parallel languages, such as the Intel® Implicit SPMD Pro
Compiler and the Intel® oneAPI DPC++ compiler, and in C++ compilers such as GCC and Clang/LLVM. Th
main contribution of our work is a novel method for statically avoiding undefined behavior when gener
loops; the resulting programs conform to the relevant language standard, enabling automated testing
second main contribution is a collection of mechanisms for increasing the diversity of generated loop co

Challenges of Compiler Testing

Challenges of Compiler Testing

- Phase **interactions**

  E.g., constant folding after/before value numbering?

- **Complex** test scenarios

- Different **configurations**

  E.g., multiple supported platforms, cross-compilation

Challenges of Compiler Testing II

- Optimization **heuristics**

  E.g., inlining decisions depend on complex conditions to setup

- **Large** set of features

  E.g., hundreds of different instructions, lots of different phases

- Language semantics can be **difficult to translate**

  E.g., closures, continuations, co-routines…

Challenges of Compiler Testing III

- JIT compilers? **Runtime** interactions

  E.g., trampolines, code patching

- Optimization vs **Debugging**

  E.g., deoptimisation meta-data, limits to optimizations

- **Hardware** specific expertise
- **Profile-Guided** Optimisations

What strategies could we take?

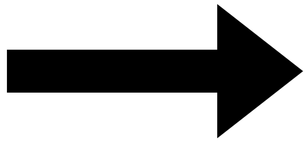
Compiler Unit Testing

- Test each component in isolation. E.g.,
  - parser
  - lowerer
  - optimisations
  - regalloc…
- Sensitive to implementation changes
- Very verbose and hard to maintain

Unit Testing Scenario: Register Allocator

- Scenario:
  - If a value is not used anymore (e.g., `R0` below after the assignment)
  - Its register can be used in subsequent instructions

```
R0 := 2.
R1 := R0 + 1.
```



```
RAX := 2.
RAX := RAX + 1.
```

Unit Testing Example: Register Allocator

RegisterAllocationTest >> testNonInterferingIntervals

```
| cfg basicBlock r |
cfg := DRControlFlowGraph new.
basicBlock := cfg newBasicBlockWith: [ :block |
  | r0 r1 |
  r0 := block copy: 2.
  r1 := block add: r0 to: 1 ].
cfg initialBasicBlock jumpTo: basicBlock.

r := DRPhysicalGeneralPurposeRegister name: 'RAX'.
DRLinearScanRegisterAllocator new
  integerRegisters: { r };
  allocateRegistersIn: cfg.

self assert: basicBlock first result equals: r.
self assert: basicBlock second result equals: r.
```

Unit Testing Example: Register Allocator

RegisterAllocationTest >> testNonInterferingIntervals

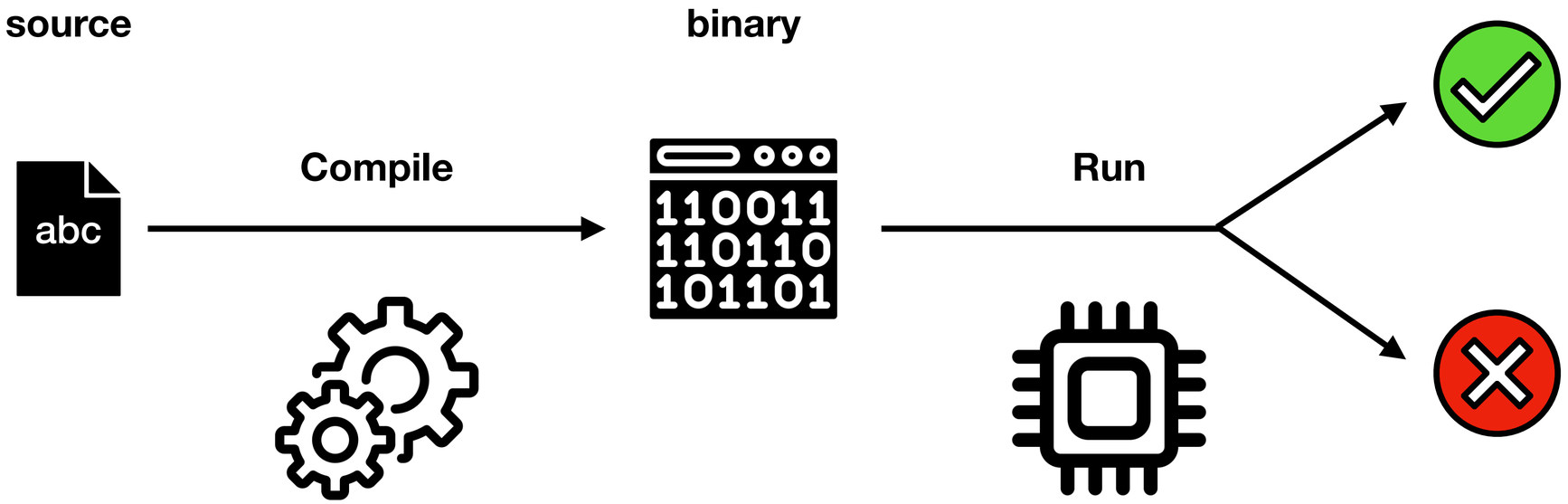
```
| cfg basicBlock r |
cfg := DRControlFlowGraph new.
basicBlock := cfg newBasicBlockWith: [ :block |
  | r0 r1 |
  r0 := block copy: 2.
  r1 := block add: r0 to: 1 ].
cfg initialBasicBlock jumpTo: basicBlock.

r := DRPhysicalGeneralPurposeRegister name: 'RAX'.
DRLinearScanRegisterAllocator new
  integerRegisters: { r };
  allocateRegistersIn: cfg.

self assert: basicBlock first result equals: r.
self assert: basicBlock second result equals: r.
```

Set up the input CFG and instructions

Set up the register allocator with specific registers

Assert renames

Main Strategy: Regression Testing



Compiler Regression Tests

- All benefits of normal tests PLUS:
  - **Resistant** to implementation changes
  - **Reusable** across configurations
  - **Simulable/emulable**
- But, difficult for fault localisation

Resistant to Implementation Changes

- Program to interface not to implementation

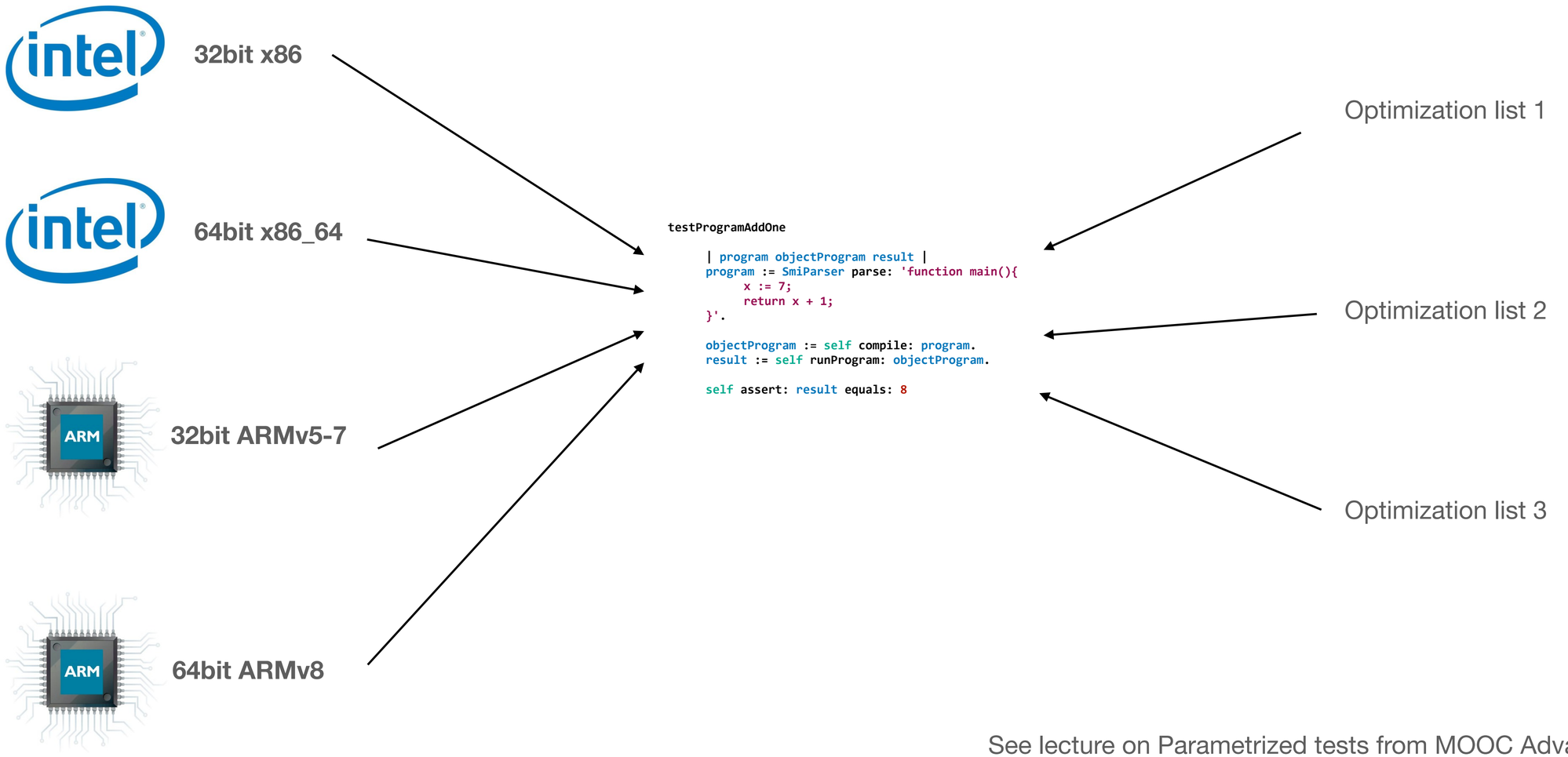
  => Depend on observable behavior

- Same test for multiple configurations. E.g.,
  - different hardware
  - different optimisation levels
  - changing compiler implementation

E2E Compiler Test Example

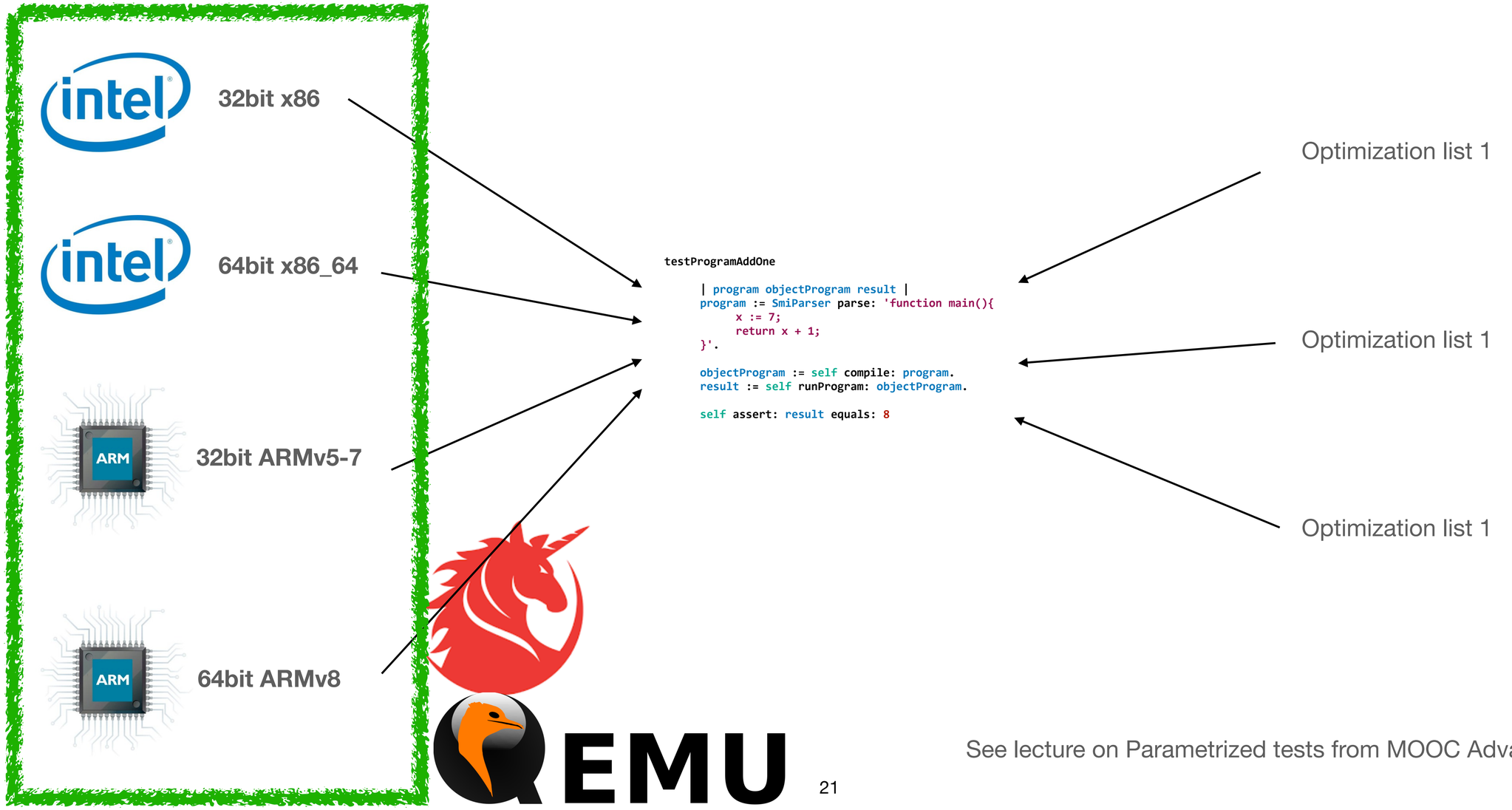
```
testProgramAddOne

  | program objectProgram result |
  program := SmiParser parse: 'function main(){
    x := 7;
    return x + 1;
  }'.

  objectProgram := self compile: program.
  result := self runProgram: objectProgram.

  self assert: result equals: 8
```

Parametrised tests



See lecture on Parametrized tests from MOOC Advanced Design - M2-5

Parametrised tests + Simulation



See lecture on Parametrized tests from MOOC Advanced Design - M2-5

No Silver Bullet

	Real Hardware Execution	E2E + Simulation/Emulation	Unit-Testing + Simulation/Emulation
Feedback-cycle speed	Very Low	Low	High
Availability	Low	High	High
Reproducibility	Low	High	High
Precision	High	Low	Low if executes simulated
Debuggability	Low	Mid	High
Resistance to Change	High	High	Low in transformation phases

No Silver Bullet Explained

- Simulators are cheap, but not 100% trustworthy

  E.g., QEMU does not perform stack alignment checks on ARM64

- Real HW execution requires access to the HW

- Unit tests of transformations phases are difficult to maintain in the long run

Takeaways

- Compilers are complex beasts to tame and test
- Unit tests are good for small components
- Regression tests are more maintainable and reusable in the long term
  - Test what the compiled code does, not what it looks like

Material

- Cross-ISA testing of the Pharo VM: lessons learned while porting to ARMv8. MPLR'21. Polito, G., Tesone, P., Ducasse, S., Fabresse, L., Rogliano, T., Misse-Chanabier, P., & Hernandez Phillips, C.
- How is Pypy tested? https://www.pypy.org/posts/2022/04/how-is-pypy-tested.html
- Testing language implementations: PLISS'17 https://www.youtube.com/watch?v=ZJUk8_k1HbY
- How is LLVM Tested? https://systemundertest.org/llvm/
- Testing LLVM https://blog.regehr.org/archives/1450