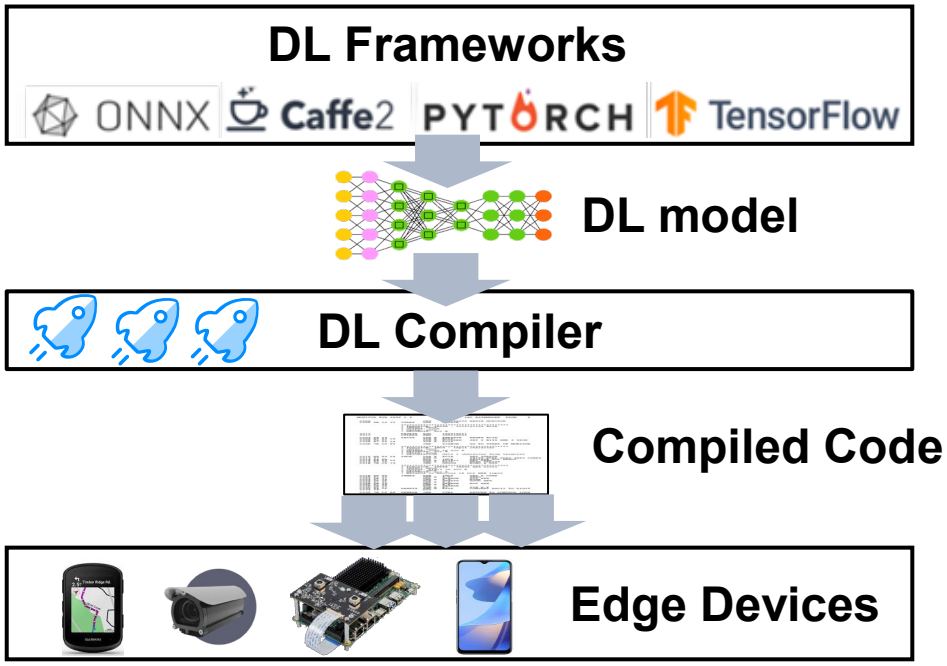

# PolyJuice: Detecting Mis-compilation Bugs in Tensor Compilers with Equality Saturation Based Rewriting

**Chijin Zhou**†, Bingzhou Qian‡, Gwihwan Go†,
Quan Zhang†, Shanshan Li‡, and Yu Jiang†

†Tsinghua University
‡National University of Defense Technology

# DL models are being ***compiled***




### Optimized Inference Engines

NVIDIA TensorRT Cloud is a developer service for compiling and creating optimized inference engines for ONNX. Developers can use their own model and choose the target RTX GPU. Then TensorRT Cloud builds the optimized inference engine, which can be downloaded and integrated into an application. TensorRT Cloud also provides prebuilt, optimized engines for popular LLMs on RTX GPUs.

### Inductor CPU backend debugging and profiling

Overview

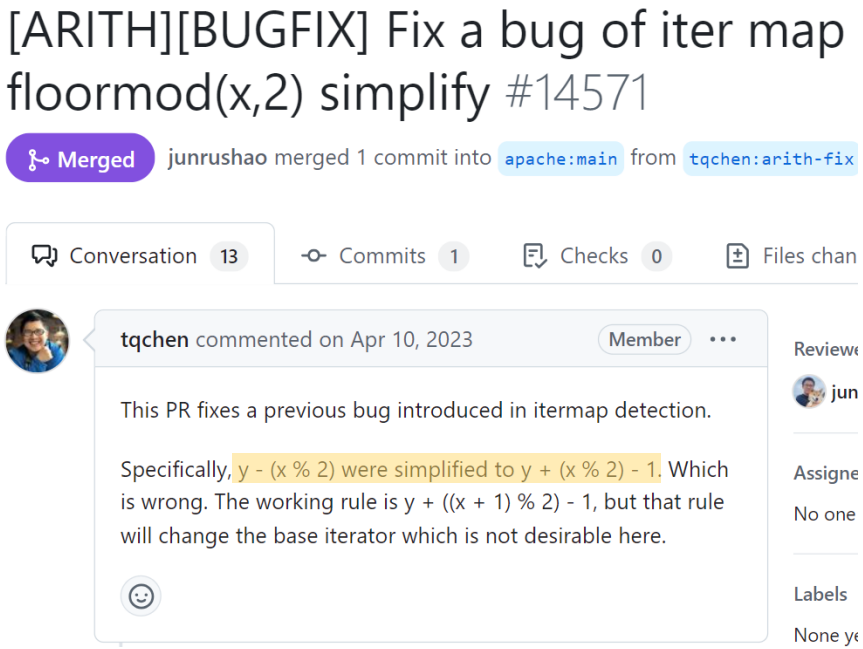
PyTorch 2.0 introduced the compilation API called `torch.compile`. This new feature offers a significant speedup over eager mode execution through graph-level optimization powered by the default Inductor backend.

This tutorial is intended to provide an in-depth introduction on the debugging and performance profiling on Inductor CPU backend by delving into the intricacies of `torch.compile`.

Meanwhile, you may also find related tutorials about `torch.compile` around basic usage, comprehensive troubleshooting and GPU-specific knowledge like GPU performance profiling.

XLA (Accelerated Linear Algebra) is an open-source compiler for machine learning. The XLA compiler takes models from popular frameworks such as PyTorch, TensorFlow, and JAX, and optimizes the models for high-performance execution across different hardware platforms including GPUs, CPUs, and ML accelerators. For example, in a BERT MLPerf submission, using XLA with 8 Volta V100 GPUs achieved a ~7x performance improvement and ~5x batch-size improvement compared to the same GPUs without XLA.

# Mis-compilation ***bugs*** in tensor compilers




https://github.com/apache/tvm/pull/14571

| | $x\%2 == 0$ | $x\%2 == 1$ |
|---|---|---|
| $y - (x\%2)$ | $y$ | $y - 1$ |
| $y + (x\%2) - 1$ | $y - 1$ | $y$ |

**Incorrect Rewrite**

# Mis-compilation ***bugs*** in tensor compilers

[ARITH][BUGFIX] Fix a bug of iter map floormod(x

Merged junrush

Conversation 13

tqchen comm

This PR fixes

Specifically, y
is wrong. The
will change t

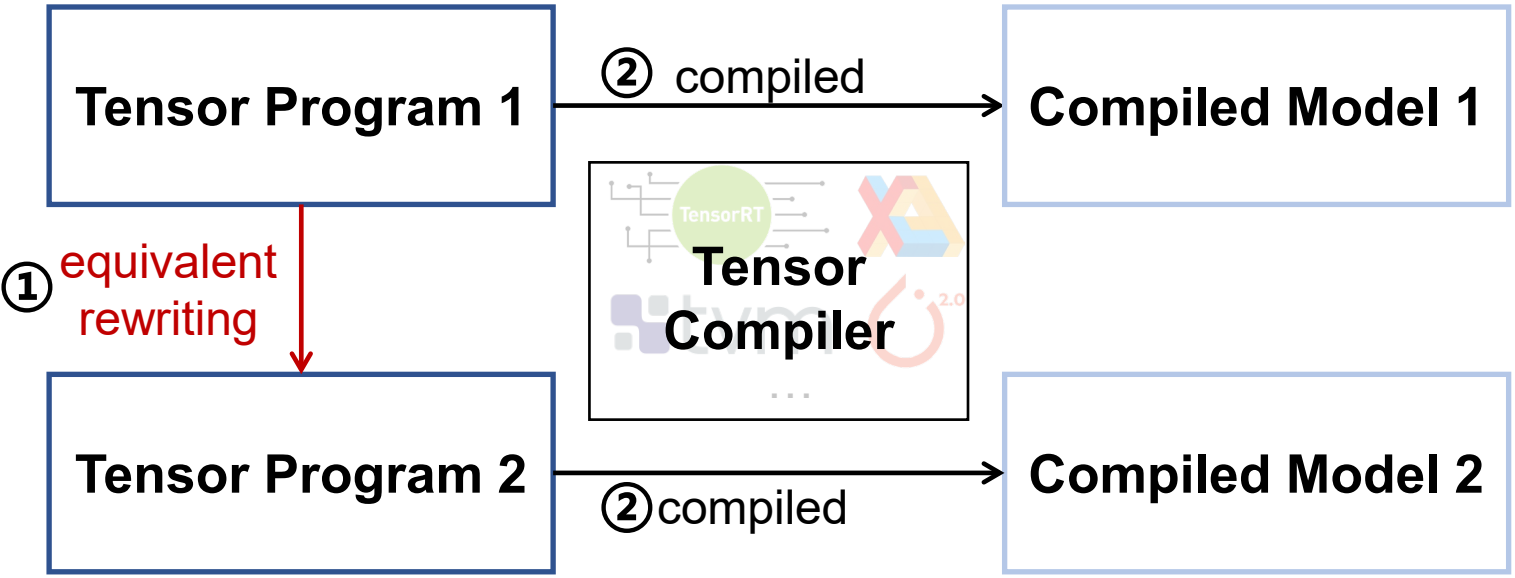
**Lunderberg** commented on Apr 11, 2023 Contributor

Thank you for finding the error there. The rewrite were initially introduced to allow simplification of cases of `(x+const)%2` introduced by the `InjectSoftwarePipeline` pass, with equivalent changes made for `DetectIterMap` so that it could handle the simplified expressions.

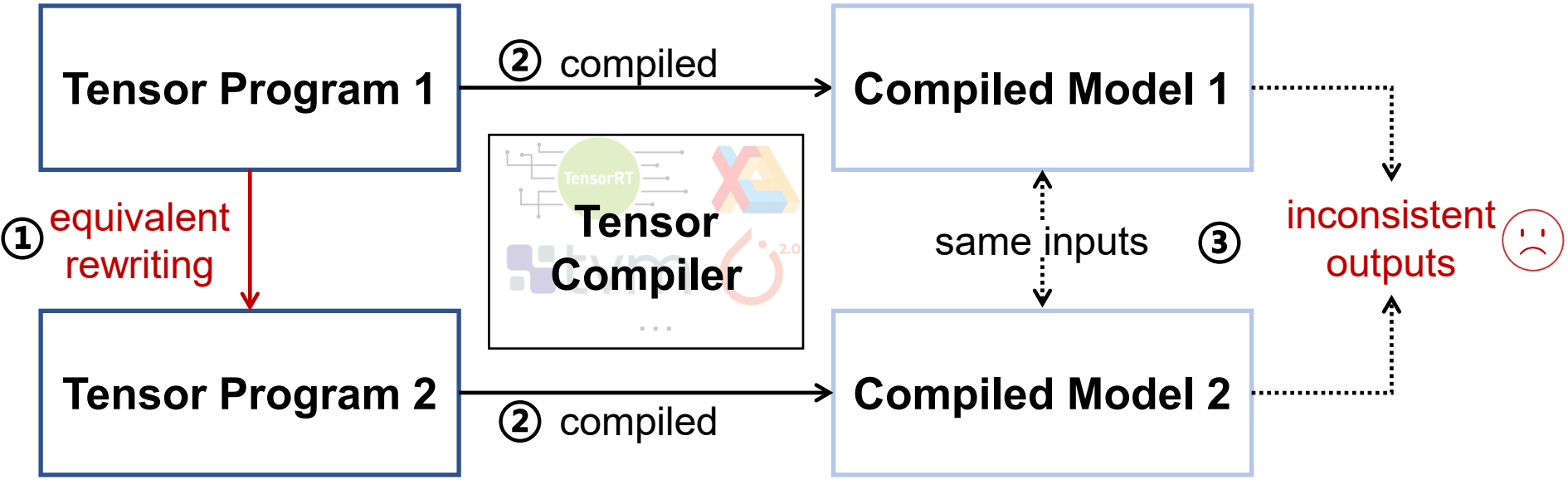
I agree that the `DetectIterMap` changes are definitely incorrect, and am going through the rewrite rules introduced in that PR to check whether they had a similar error, or whether the error was solely in the `IterMapSimplify` changes.

*How do we automatically detect such non-crash bugs?*

# The basic idea of ***PolyJuice***



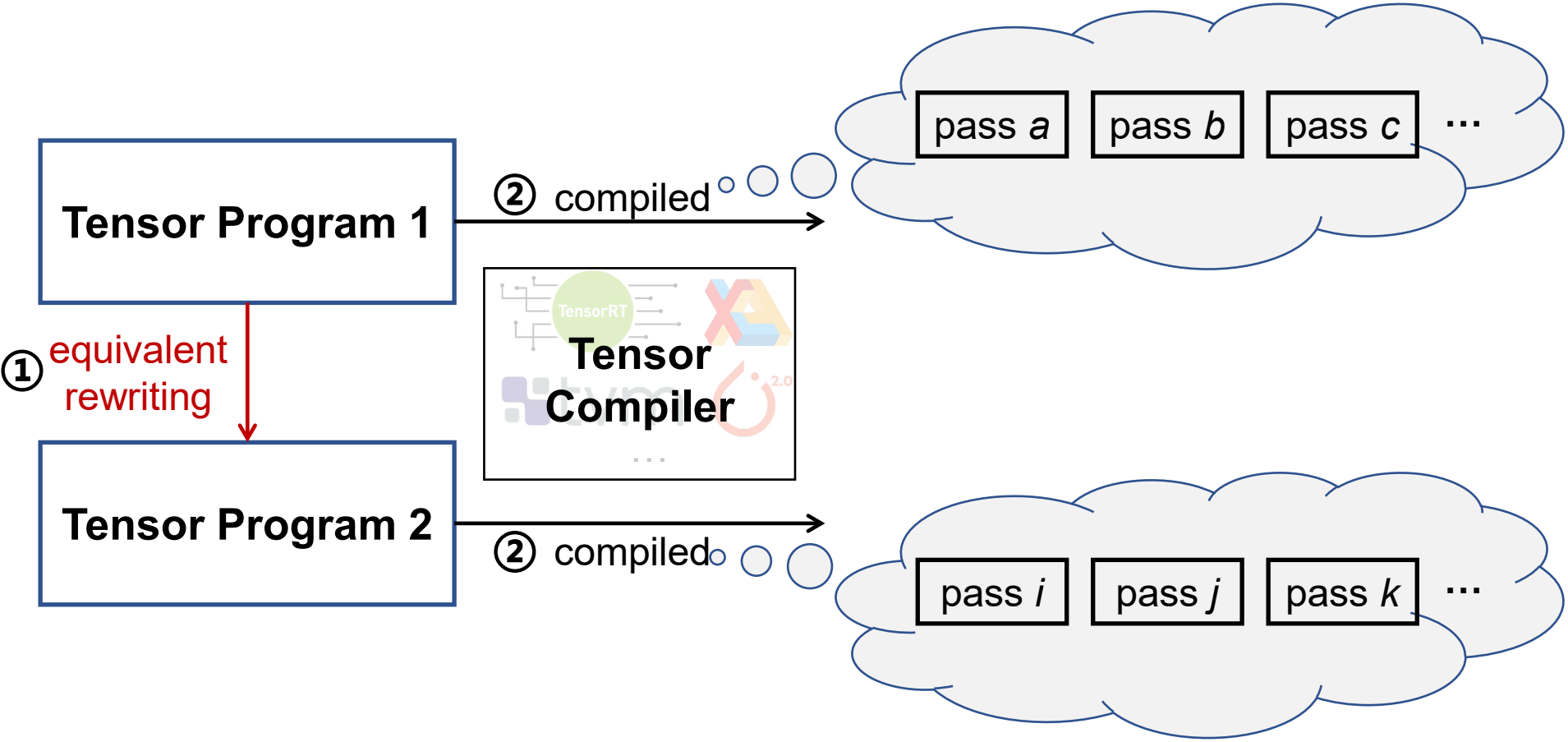
# The basic idea of ***PolyJuice***




The same as Equivalence Modulo Inputs [PLDI'14]

*How do we rewrite the tensor program?*

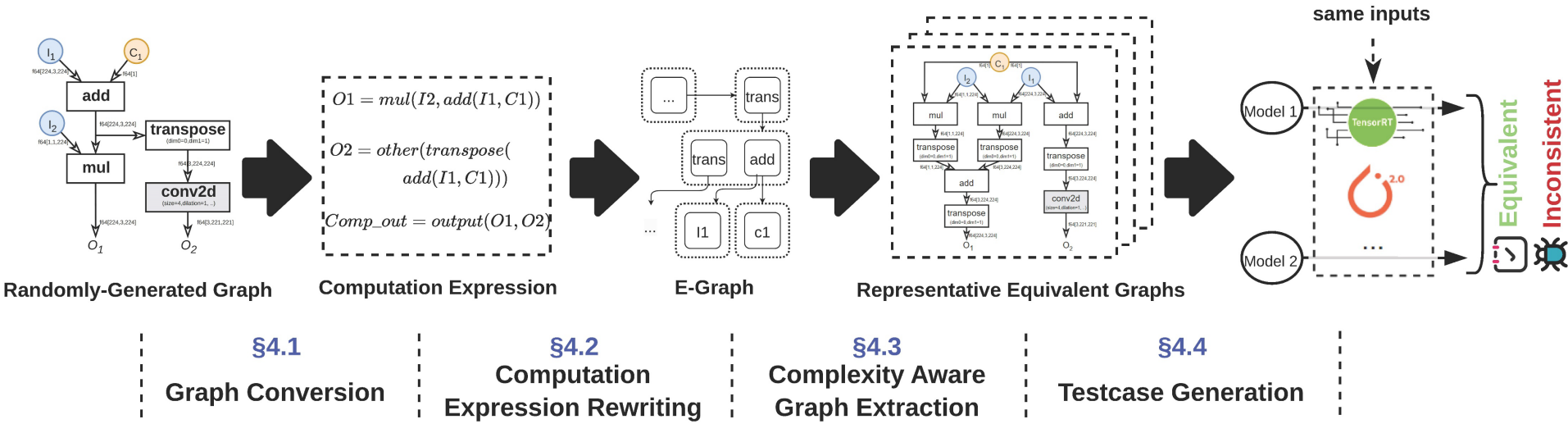
# Goal of Rewriting



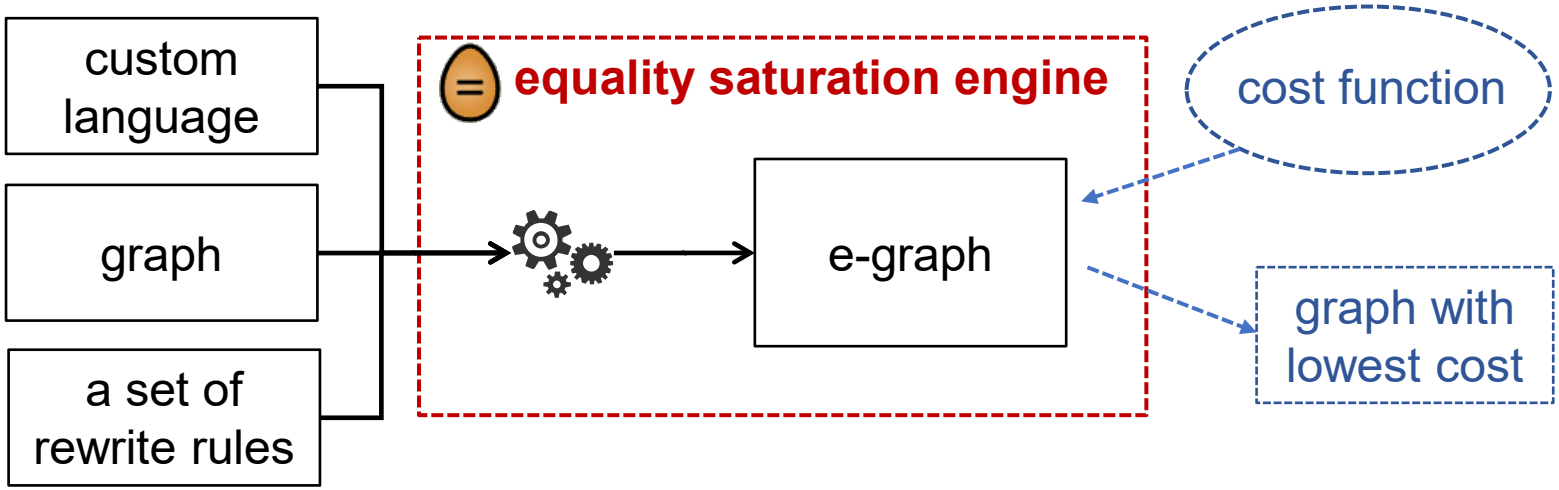

The two equivalent programs should undergo passes that are as ***different*** as possible

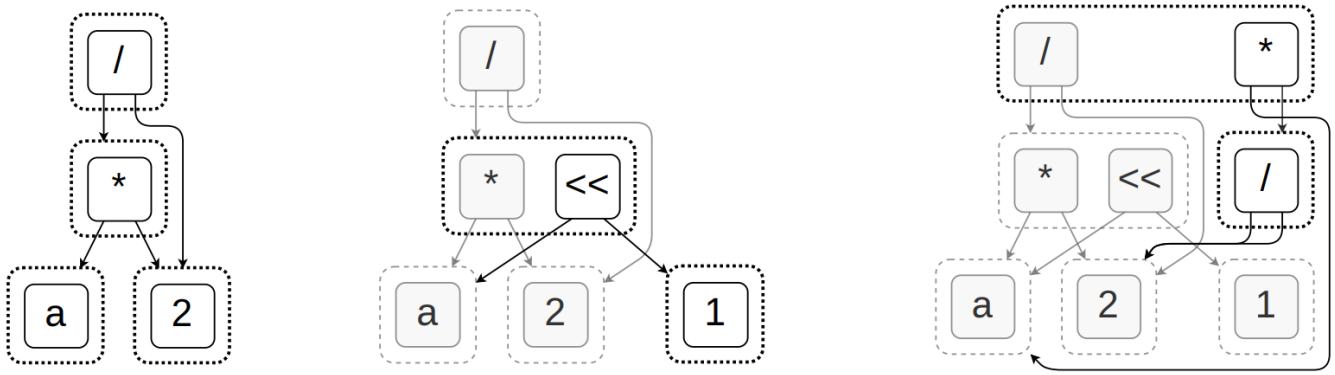
# Design

- Goal: construct two equivalent programs with highly different dataflow
- Approach:
  - leverage *equality saturation* to rewrite a computation graph
  - find the simplest graph and the most complex graph for testing



# Equality Saturation



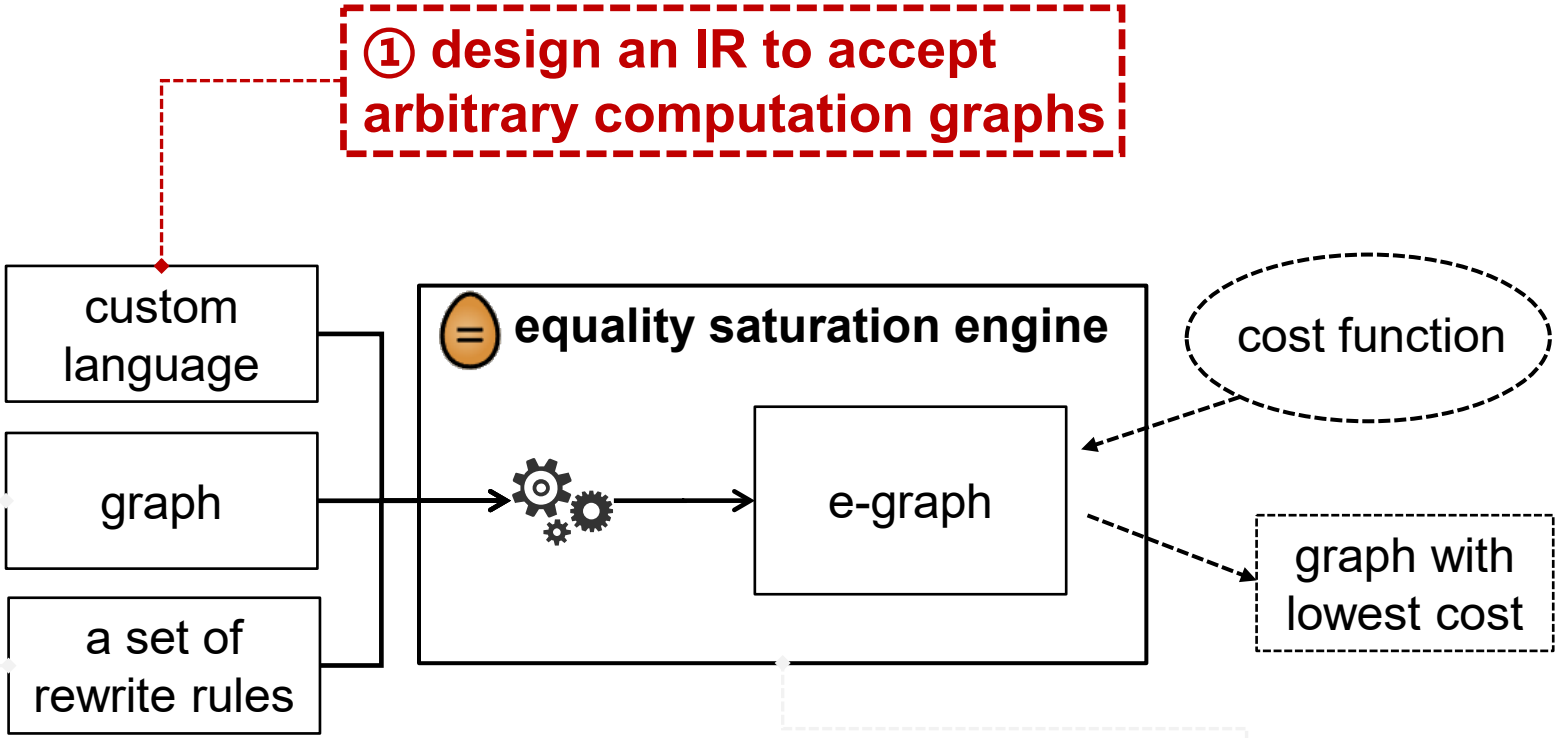




(a) The e-graph of $(a * 2)/2$.

(b) After applying rewrite rule $x * 2 \rightarrow x \ll 1$.

(c) After applying rewrite rule $(x * y)/z \rightarrow x * (y/z)$.

# Implementation



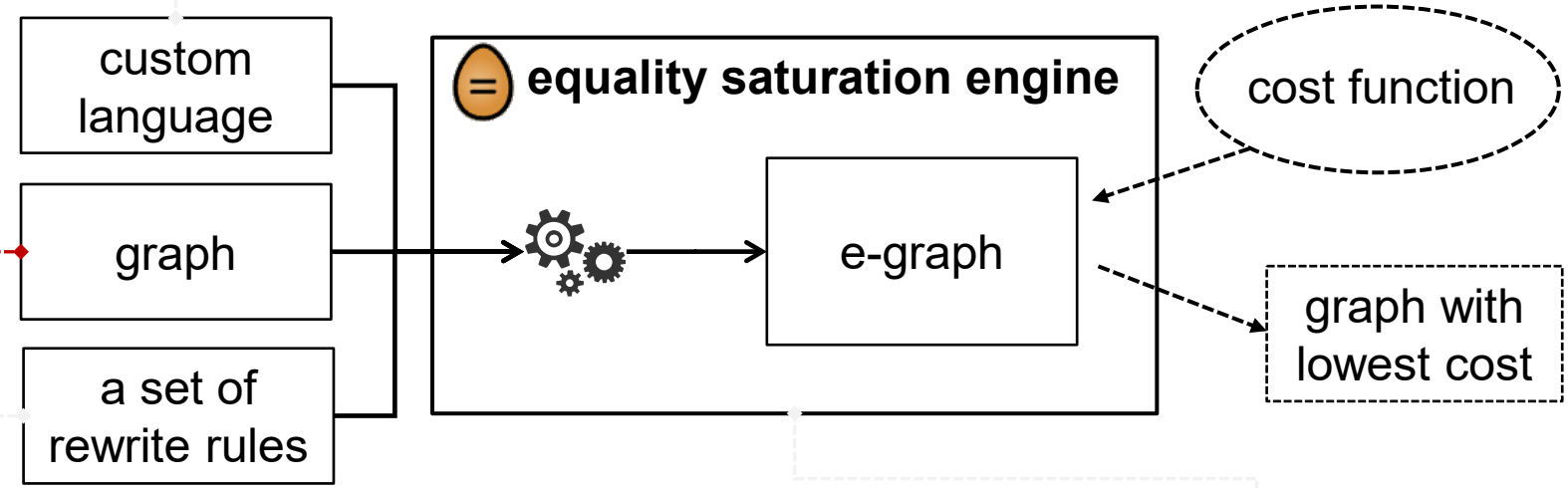

② leverage NNSmith [ASPLOS'23] to generate original graphs

④ use Egg [POPL'21] for equality saturation

③ propose several arithmetic and structural rewrite rules

# Implementation

① design an IR to accept arbitrary computation graphs



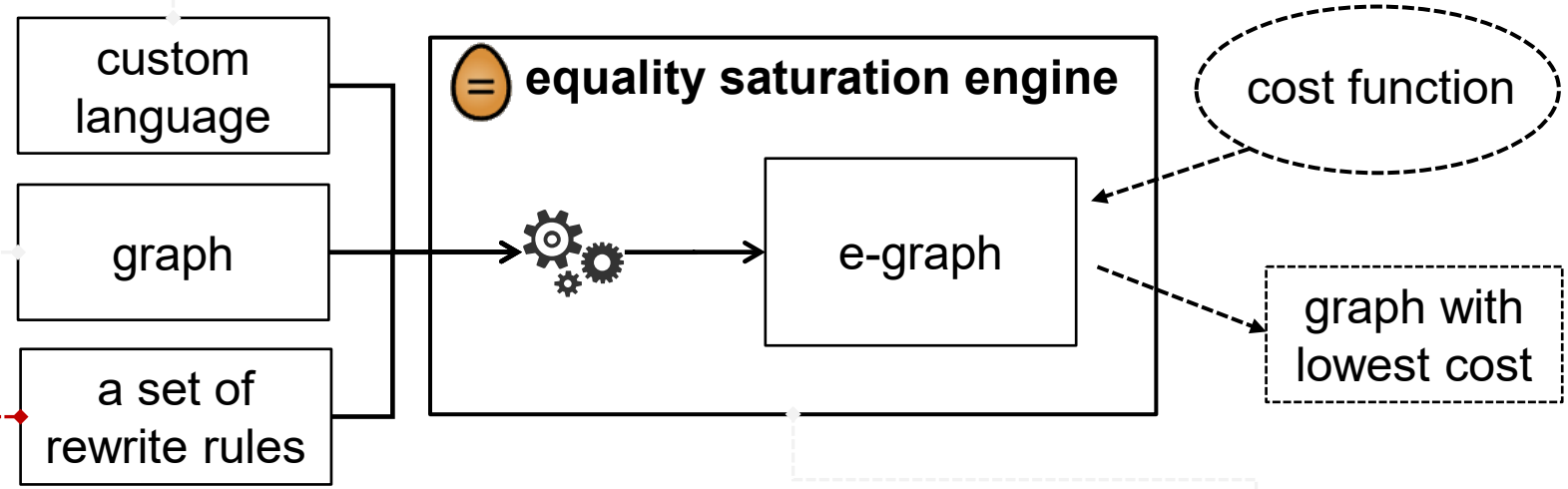

② leverage NNSmith [ASPLOS'23] to generate the original graph

④ use Egg [POPL'21] for equality saturation

③ propose several arithmetic and structural rewrite rules

# Implementation

① design an IR to accept arbitrary computation graphs



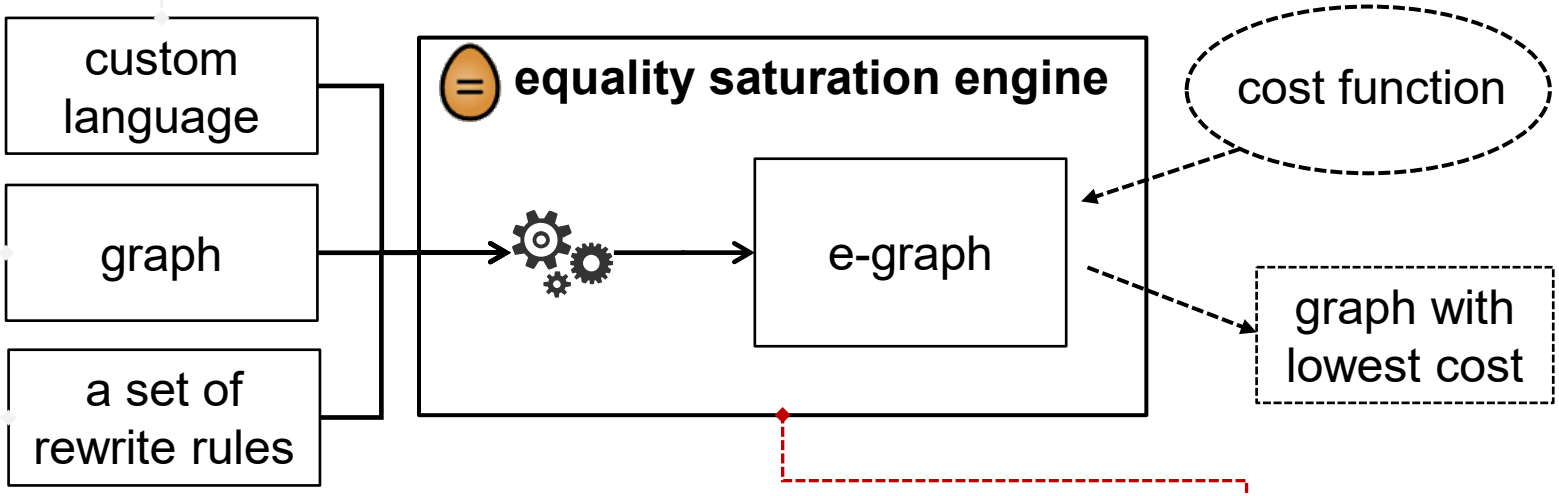

② leverage NNSmith [ASPLOS'23] to generate original graphs

④ use Egg [POPL'21] for equality saturation

**③ propose several arithmetic and structural rewrite rules**

# Implementation

① design an IR to accept arbitrary computation graphs




② leverage NNSmith [ASPLOS'23] to generate original graphs

③ propose several arithmetic and structural rewrite rules

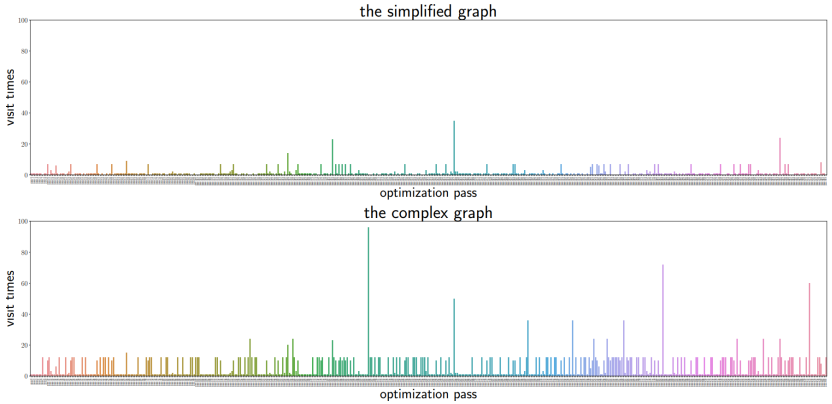
④ use Egg [POPL'21] for equality saturation

# Evaluation

## highlight

- PolyJuice found **84** non-crash bugs in 7 tensor compilers, with **49** confirmed.
- Compared with naïve rewriting, our rewriting can improve the difference of compilation paths by **112%-150%**.
- PolyJuice only introduces **0.11%-1.53%** runtime overhead compared to NNSmith

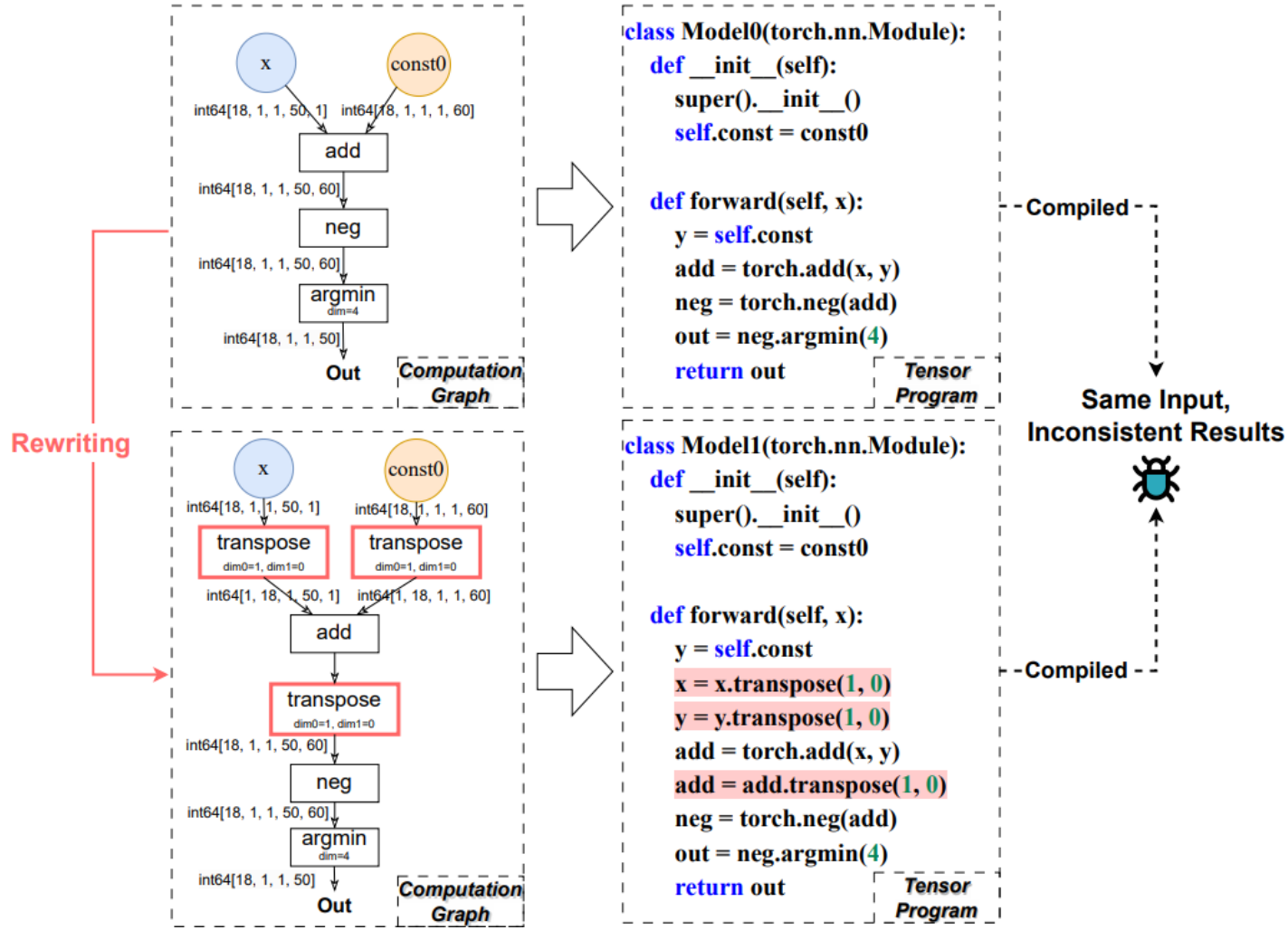
| | Reported | Confirmed | Fixed |
|---|---|---|---|
| Torch Inductor | 12 | 9 | 9 |
| TensorRT | 10 | 8 | 7 |
| ONNXRuntime | 11 | 0 | 0 |
| TVM | 7 | 0 | 0 |
| XLA | 25 | 13 | 0 |
| Hidet | 4 | 4 | 4 |
| EinNet | 15 | 15 | 0 |
| Total | 84 | 49 | 20 |




| Compiler | # graph node | original graph generation | expression rewriting | graph extraction | compiler execution |
|---|---|---|---|---|---|
| TVM | 5 | 26.76ms | 0.33ms | 1.16ms | 1367.85ms |
| | 10 | 41.72ms | 0.33ms | 1.63ms | 1865.97ms |
| | 15 | 53.04ms | 0.35ms | 1.64ms | 2059.14ms |
| XLA | 5 | 22.35ms | 0.27ms | 1.23ms | 74.09ms |
| | 10 | 35.92ms | 0.26ms | 1.23ms | 97.24ms |
| | 15 | 87.00ms | 0.26ms | 1.25ms | 121.11ms |

# Interesting Cases

**TVM 0.12.0**



# Interesting Cases

**TVM 0.12.0**

**Tensor Information**

x: int64[3,1,1,1,1]
const0: int64[3,1,1,1,60]

```
class Model0(torch.nn.Module):
  def __init__(self):
    super().__init__()

  def forward(self, x):
    add = torch.add(x, const0)
    neg = torch.neg(add)
    argmin = neg.argmin(4)
    return argmin
```

Fusion

**Tensor Information**

x: int64[3,1,1,1,1]
const0: int64[3,1,1,1,60]

```
class Model1(torch.nn.Module):
  def __init__(self):
    super().__init__()

  def forward(self, x):
    trans_0 = x.transpose(1, 0)
    trans_1 = const0.transpose(1, 0)
    add = torch.add(trans_0, trans_1)
    add_trans = add.transpose(1, 0)
    neg = torch.neg(add_trans)
    argmin = neg.argmin(4)
    return argmin
```

Fusion

**Converted to LLVM IR by TVM**

```
# fused_add_negative_argmin
for .. in T.parallel(3):
  min_arg = -1
  min_val = T.int64().MAX
  for index in range(60):
    min_arg = if(min_val < T.int64(0) -
              input0[index] - input1[index]) {
                min_arg
              } else { index }
  min_val = if(...) { ... } else { ... }
```

```
# fused_negative_argmin
for ... in T.parallel(3):
  min_arg = -1
  min_val = T.int64().MAX
  for index in range(60):
    min_arg = if(min_val + input0[index] < T.int64(0)) {
                min_arg
              } else { index }
  min_val = if(...) { ... } else { ... }
```

Integer Overflow

# Interesting Cases

**PyTorch 2.1.0**

**Tensor Information**
x: uint8[1]
const0: uint8[40,30]
const1: uint8[30]

```
class Model0(torch.nn.Module):
  def __init__(self):
    super().__init__()

  def forward(self, x):
    reshape = const0.reshape(30)
    neg = torch.neg(const1)
    add = torch.add(reshape, x)
    mul = torch.mul(add, neg)
    sum = mul.sum(0)
    return sum
```

**Tensor Information**
x: uint8[1]
const0: uint8[40,30]
const1: uint8[30]

```
class Model1(torch.nn.Module):
  def __init__(self):
    super().__init__()

  def forward(self, x):
    reshape = const0.reshape(30)
    neg = torch.neg(const1)
    add = torch.add(reshape, x)
    mul = torch.mul(neg, add)
    sum = mul.sum(0)
    return sum
```

**Converted to C++ code by Torch Inductor**

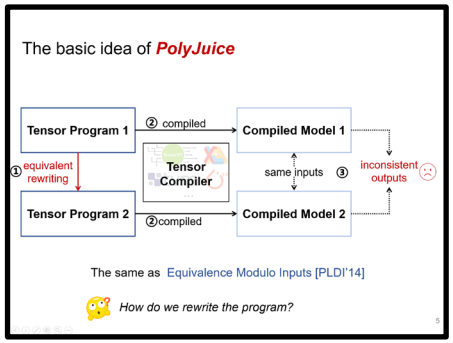
```
const unsigned char* in_ptr0,in_ptr1,in_ptr2 = ...;
// long i0: range(0,30); long i1: range(0,40);
auto tmp0 = in_ptr0[i0 + (30L*i1)];
auto tmp2 = in_ptr1[i0];
auto tmp3 = in_ptr2[0L];
auto tmp1 = decltype(tmp0)(-tmp0); // tmp1 is "char"
auto tmp4 = tmp2 + tmp3; // tmp4 is "int"
auto tmp5 = decltype(tmp1)(tmp1 * tmp4);
auto tmp6 = static_cast<long>(tmp5);
```

```
const unsigned char* in_ptr0,in_ptr1,in_ptr2 = ...;
// long i0: range(0,30); long i1: range(0,40);
auto tmp0 = in_ptr0[i0 + (30L*i1)];
auto tmp2 = in_ptr1[i0];
auto tmp3 = in_ptr2[0L];
auto tmp1 = decltype(tmp0)(-tmp0); // tmp1 is "char"
auto tmp4 = tmp2 + tmp3; // tmp4 is "int"
auto tmp5 = decltype(tmp4)(tmp4 * tmp1);
auto tmp6 = static_cast<long>(tmp5);
```

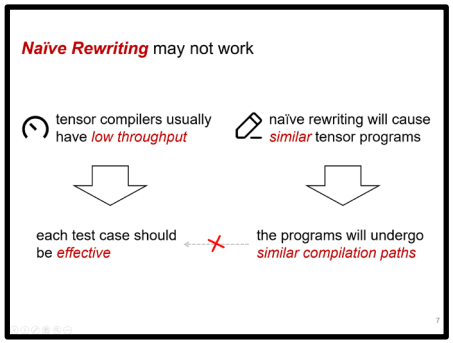
**Inconsistent because of type casting for tmp5**

# Summary

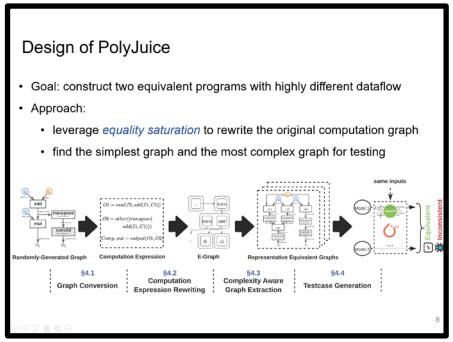
Goal: detecting mis-compilation bugs



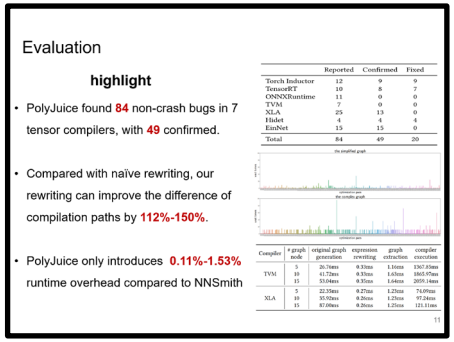

Problem: how to effectively rewrite programs




Method: equality saturation for rewriting




Evaluation: able to find real-world bugs




| | Reported | Confirmed | Fixed |
|---|---|---|---|
| Torch Inductor | 12 | 9 | 9 |
| TensorRT | 10 | 8 | 7 |
| ONNXRuntime | 11 | 0 | 0 |
| TVM | 7 | 0 | 0 |
| XLA | 25 | 13 | 0 |
| Hidet | 4 | 4 | 4 |
| EinNet | 15 | 15 | 0 |
| Total | 84 | 49 | 20 |

| Compiler | # graph node | original graph generation | expression rewriting | graph extraction | compiler execution |
|---|---|---|---|---|---|
| TVM | 5 | 26.76ms | 0.33ms | 1.16ms | 1367.35ms |
| | 10 | 41.72ms | 0.33ms | 1.63ms | 1865.97ms |
| | 15 | 53.04ms | 0.35ms | 1.64ms | 2059.14ms |
| XLA | 5 | 22.35ms | 0.27ms | 1.23ms | 74.09ms |
| | 10 | 35.92ms | 0.26ms | 1.25ms | 97.24ms |
| | 15 | 87.08ms | 0.26ms | 1.25ms | 121.11ms |

**Prototype:** https://github.com/ChijinZ/PolyJuice-Fuzzer

**Special thanks:**

- NNSmith's authors, for the well-structured and reusable code
- Egg community, for the well-developed tool and responsive community