HEC: Equivalence Verification Checking for Code Transformation via Equality Saturation

Jiaqi Yin, Zhan Song, Nicolas Bohm Agostini, Antonino Tumeo, Cunxi Yu

University of Maryland, College Park
Pacific Northwest National Laboratory

A. JAMES CLARK
SCHOOL OF ENGINEERING

Pacific Northwest
NATIONAL LABORATORY

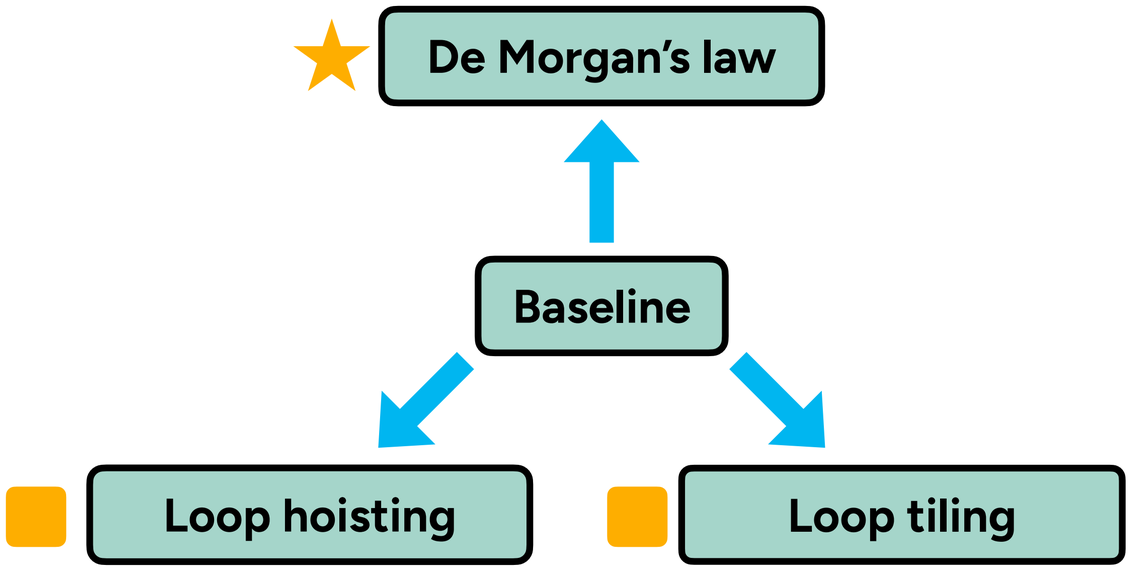
Motivation

- Source-to-source code transformations are common in compilers
- **Equivalence checking** for **code transformation** is critical but neglected

$$P_{src} \xrightarrow[=?]{\text{Pass}_1} P_1 \xrightarrow[=?]{\text{Pass}_2} P_2 \xrightarrow[=?]{\text{Pass}_3} \cdots \xrightarrow[=?]{\text{Pass}_n} P_{dst}$$

Equivalence checking: Are P_{src} and P_{dst} functionally equivalent?

Motivation



- **Datapath Transformation**
  - **Arithmetic**: Associative law, Commutative law, Distributive law
  - **Boolean**: De Morgan's laws, Commutative law, Double Negation law
- **Control Flow Transformation**
  - Loop unrolling, Loop tiling, Loop hoisting, Loop fusion
- ***Unified Verification Approach?***

Motivation

Baseline

```
0  %av, %bv: memref<101xi1>
1
2  %true = arith.constant true
3  affine.for %arg1=0 to 101 {
4    %1=affine.load %av[%arg1]:memref<101xi1>
5    %2=affine.load %bv[%arg1]:memref<101xi1>
6    %3=arith.andi %1, %2:i1
7    %4=arith.xori %3, %true:i1
8  }
```

Loop hoisting

```
0  %av, %bv: memref<101xi1>
1
2  affine.for %arg1=0 to 101 {
3   %true = arith.constant true
4   %1=affine.load %av[%arg1]:memref<101xi1>
5   %2=affine.load %bv[%arg1]:memref<101xi1>
6   %3=arith.andi %1, %2:i1
7   %4=arith.xori %3, %true:i1
8 }
```

De Morgan's law

```
0  %av, %bv: memref<101xi1>
1
2  %true = arith.constant true
3  affine.for %arg1=0 to 101 {
4    %1=affine.load %av[%arg1]:memref<101xi1>
5    %2=affine.load %bv[%arg1]:memref<101xi1>
6    %3=arith.xori %1, %true:i1
7    %4=arith.xori %2, %true:i1
8    %5=arith.ori %3, %4:i1
9  }
```

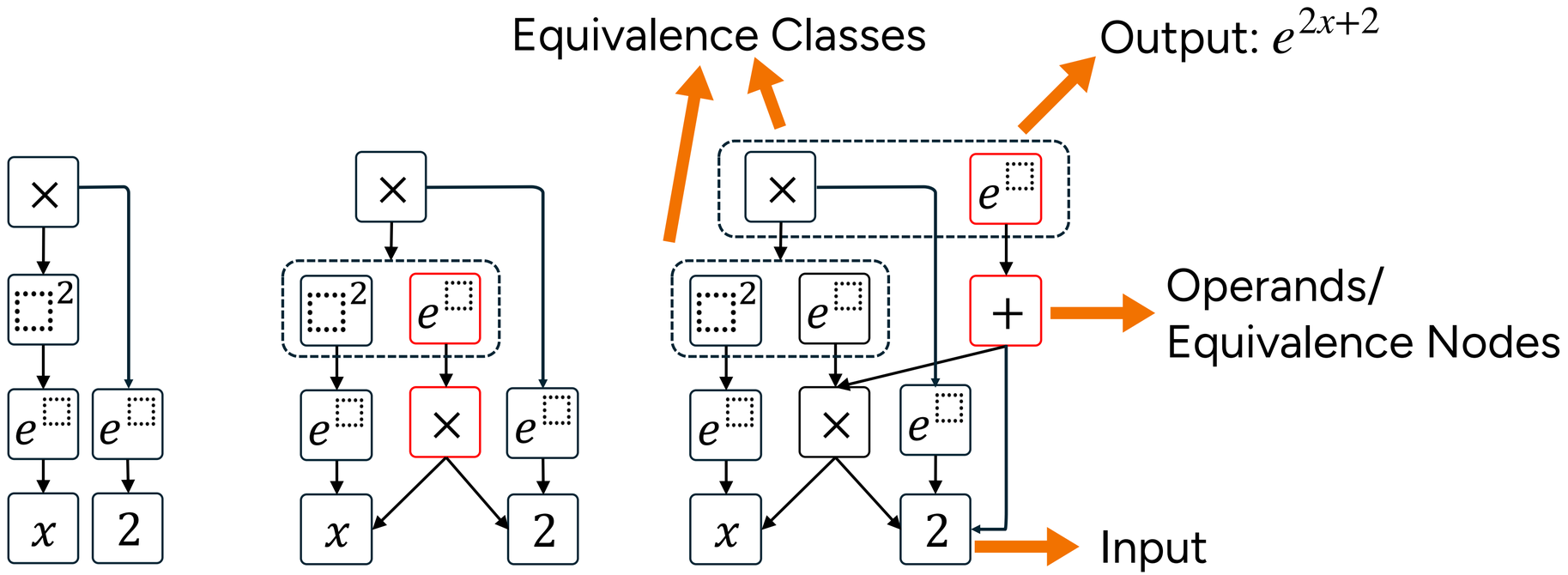
Loop tiling

```
0  %av, %bv: memref<101xi1>
1
2  %true = arith.constant true
3  affine.for %arg1=0 to 101 step 3 {
4   affine.for %arg2= %arg1 to min (%arg1+3, 101) {
5    %1=affine.load %av[%arg2]:memref<101xi1>
6    %2=affine.load %bv[%arg2]:memref<101xi1>
7    %3=arith.andi %1, %2:i1
8    %4=arith.xori %3, %true:i1
9   }
10  }
```

Code Transformation Equivalence Checking

- Existing tools are limited to specific transformation domains
  - Control flow transformation
    - [POPL 2016]: PolyCheck: dynamic verification of iteration space transformations on affine programs
    - [TOPLAS 2012]: Equivalence Checking of Static Affine Programs Using Widening to Handle Recurrences
  - Datapath transformation
    - [CAV 2022]: SMT-based Translation Validation for Machine Learning Compiler
    - [FMCAD 2023]: Datapath Verification via Word-Level E-Graph Rewriting

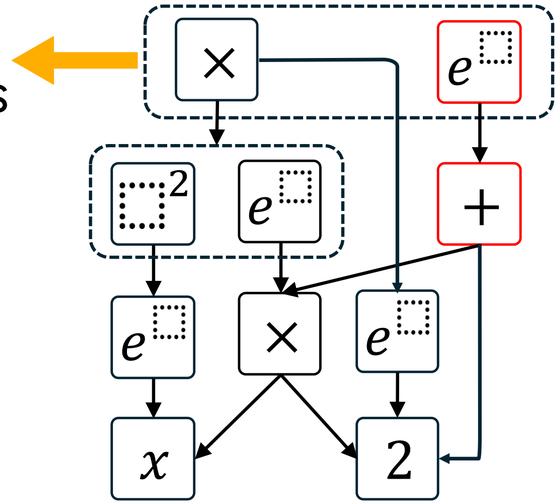
E-Graphs

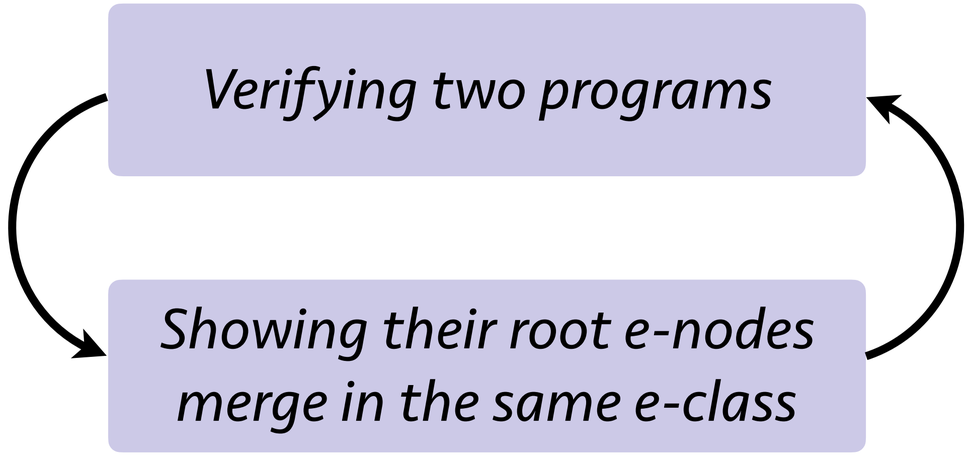


- E-graph rewriting: **Left**: $(e^x)^2 \times e^2$; **Middle**: $e^{2x} \times e^2$; **Right**: e^{2x+2}
  - Compact representation of **equivalent terms**
  - Data structure - **explore all possible rewrites of a program**
  - Capture semantic equivalence by merging equivalent forms into a single class.
  - **Constructive** rewrite application - phase ordering

E-Graphs

Groups multiple equivalent terms

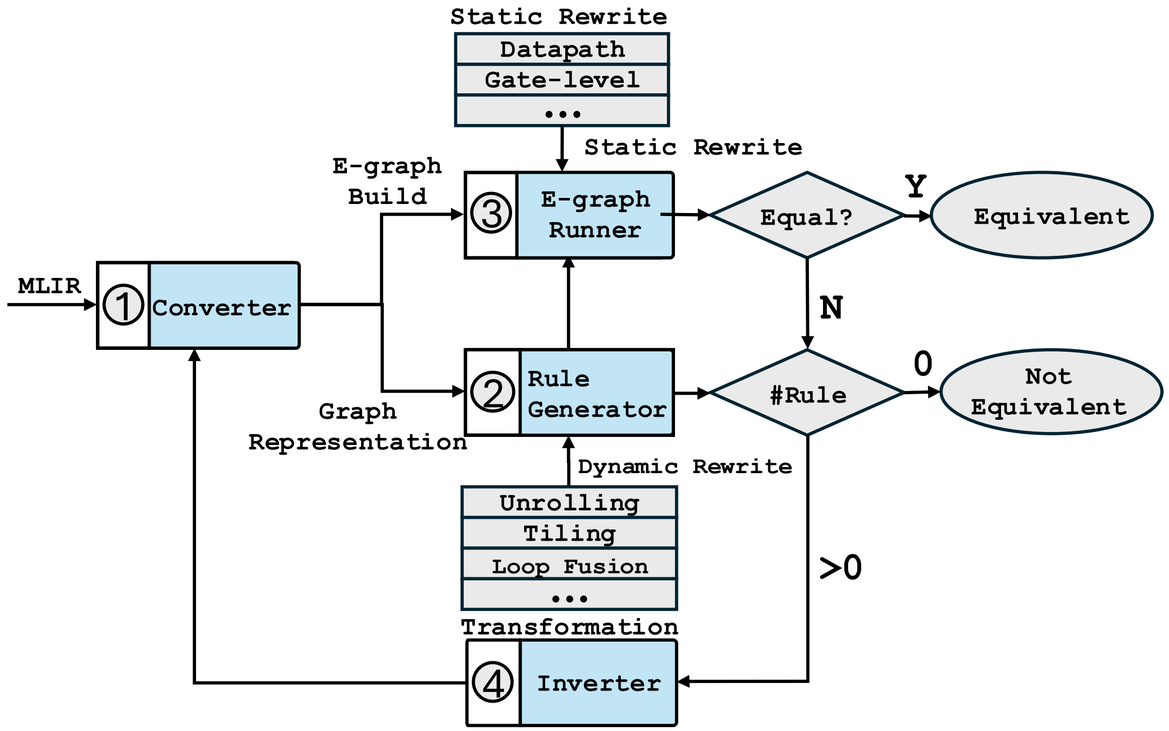




- Equivalence Checking ↔ Rewrite-Based Proof
  - E-graph rewriting and equivalence checking both seek a sequence of valid passes that map one program to another
  - Can we show that the original and transformed versions of a program collapse into the same equivalence class?

HEC: Hybrid Equivalence Checking

- Goals: Verification of both control flow and datapath transformation
- Target: MLIR code
- Overview
  - Rewriting Engine (Static + Dynamic)
  - MLIR → Graph Representation
  - E-graph Runner



Graph Representation

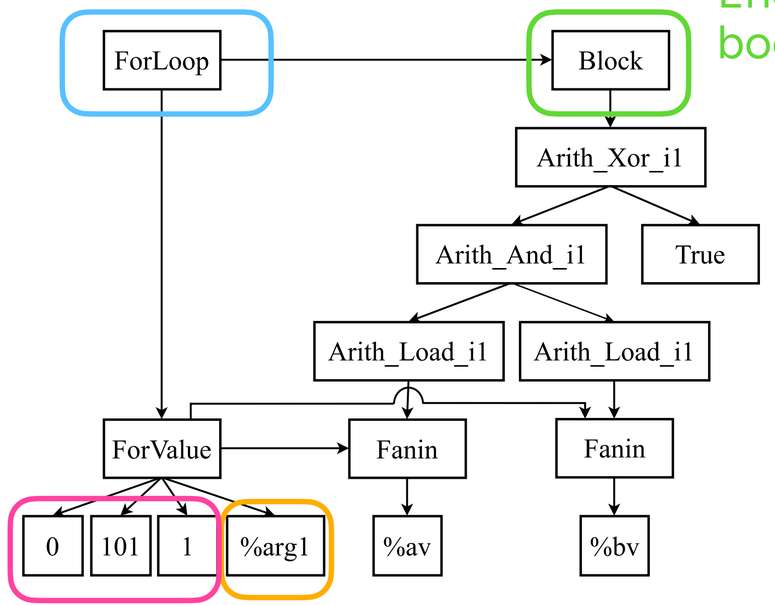
- AST-Inspired Model
  - MLIR operation → Graph Vertex
  - Variable → Graph Edges
- Variable Renaming
- Expose parameters for static/ dynamic rewriting

```
0  %av, %bv: memref<101xi1>
1
2  %true = arith.constant true
3  affine.for %arg1=0 to 101 {
4    %1=affine.load %av[%arg1]:memref<101xi1>
5    %2=affine.load %bv[%arg1]:memref<101xi1>
6    %3=arith.andi %1, %2:i1
7    %4=arith.xori %3, %true:i1
8  }
```

Baseline

Loop decomposition

Encapsulating body operations



Loop bounds, Loop steps

Variable identity

Loop Hoisting - Quick Recap

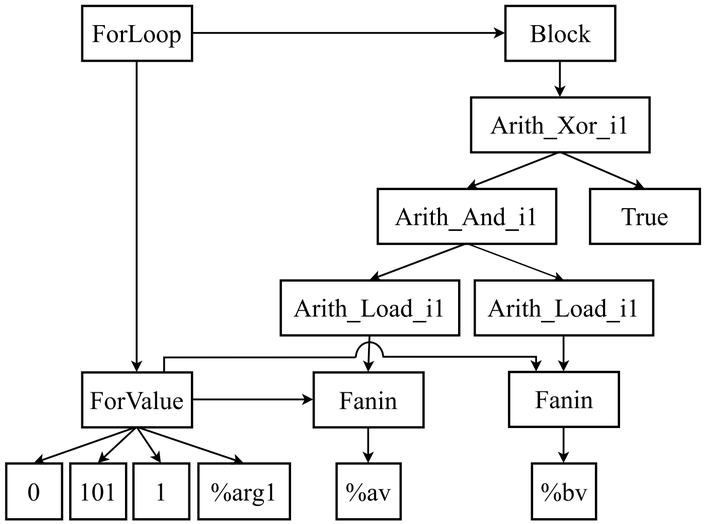
Baseline

```
0  %av, %bv: memref<101xi1>
1
2  %true = arith.constant true
3  affine.for %arg1=0 to 101 {
4    %1=affine.load %av[%arg1]:memref<101xi1>
5    %2=affine.load %bv[%arg1]:memref<101xi1>
6    %3=arith.andi %1, %2:i1
7    %4=arith.xori %3, %true:i1
8  }
```

Loop hoisting

```
0  %av, %bv: memref<101xi1>
1
2  affine.for %arg1=0 to 101 {
3   %true = arith.constant true
4   %1=affine.load %av[%arg1]:memref<101xi1>
5   %2=affine.load %bv[%arg1]:memref<101xi1>
6   %3=arith.andi %1, %2:i1
7   %4=arith.xori %3, %true:i1
8  }
```

Automatically verifies transformations like loop hoisting before rewriting



Hybrid E-graph Rewriting

- Static Rewriting Rules
  - Bitwidth-aware rewriting rules: arithmetic laws, boolean identities
  - Selection of static rewriting rules

Class	Left-hand Side	Right-hand Side
Datapath	$a \ll b$	$a \times 2^b$
	$(a \times b) \ll c$	$(a \ll c) \times b$
	$a \times b \times c$	$a \times (b \times c)$
	$(a \ll b) \ll c$	$a \ll (b + c)$
Gate-level	$\neg(a \& b)$	$\neg a \parallel \neg b$
	$a \oplus b$	$(a \wedge \neg b) \vee (\neg a \wedge b)$
	$a \oplus 0$	a
	$\neg a$	$a \oplus True$

De Morgan's law - Quick Recap

Baseline

```
0  %av, %bv: memref<101xi1>
1
2  %true = arith.constant true
3  affine.for %arg1=0 to 101 {
4    %1=affine.load %av[%arg1]:memref<101xi1>
5    %2=affine.load %bv[%arg1]:memref<101xi1>
6    %3=arith.andi %1, %2:i1
7    %4=arith.xori %3, %true:i1
8  }
```

De Morgan's law

```
0  %av, %bv: memref<101xi1>
1
2  %true = arith.constant true
3  affine.for %arg1=0 to 101 {
4    %1=affine.load %av[%arg1]:memref<101xi1>
5    %2=affine.load %bv[%arg1]:memref<101xi1>
6    %3=arith.xori %1, %true:i1
7    %4=arith.xori %2, %true:i1
8    %5=arith.ori %3, %4:i1
9  }
```

- **De Morgan's Law:** map `NAND(a, b)` and `OR(¬a, ¬b)` to the two sides of $\overline{a \,\&\, b} \Leftrightarrow \bar{a} \| \bar{b}$
- **NOT via XOR:** represent `¬a` as `XOR(a, True)` using the identity $\neg a \Leftrightarrow a \oplus \text{True}$
- Merge both variants into the same e-class, proving semantic equivalence.

Hybrid E-graph Rewriting

- Dynamic Rewriting
  - Control flow Transformation (e.g., unrolling, tiling)
  - Use Z3 solver to verify transformation conditions

	Left-hand Side	Right-hand Side	Condition

Unrolling	for **m1** to **n2** step **k2**:
	`Loop-body-2`
		for **m1** to **n1** step **k1**:
		`Loop-body-1`
		for **m2** to **n2** step **k2**:
		`Loop-body-2`
			1. $\lceil(\mathbf{n2}-\mathbf{m1})/\mathbf{k2}\rceil == \lceil(\mathbf{n2}-\mathbf{m2})/\mathbf{k2}\rceil + \lceil(\mathbf{n1}-\mathbf{m1})/\mathbf{k1}\rceil \times (\mathbf{k1}/\mathbf{k2})$
			2. `Loop-body-1` is **k1/k2** times replication of `Loop-body-2`

Tiling	for **%1** = **m1** to **n1** step **k2**: `Loop-body`	for **%1** = **m1** to **n1** step **k1**: for **%2** = **%1** to **n2** step **k2**: `Loop-body`	1. $\mathbf{k1} == f * \mathbf{k2}$ (f is the tiling factor) 2. $\mathbf{n2} = \min(\mathbf{\%1} + \mathbf{k1}, \mathbf{\%2})$

Fusion	for **m** to **n** step **k1**:
	`Loop-body-1`
	for **m** to **n** step **k2**:
	`Loop-body-2`
		for **m** to **n** step **k1** × **k2**:
		`Loop-body-3`
			1. `Loop-body-3` is **k2** times replication of `loop-body-1` plus **k1** times replication of `loop-body-2`
			2. No memory RAW violation across `Loop-body-1` and `Loop-body-2`

Coalesing	for **%1** = **m1** to **n1** step **k1**: for **%2** = **%1** to **n2** step **k2**: `Loop-body`	for **%3** = 0 to **n1** × **n2**: **%1**=(floordiv **%3 n2**) **%2**=(mod **%3 n2**) `Loop-body`	1. The reference of **%1**, **%2** is replaced by (floordiv **%3 n2**) and (mod **%3 n2**), respectively.

Loop tiling - Quick Recap

Baseline

```
0  %av, %bv: memref<101xi1>
1
2  %true = arith.constant true
3  affine.for %arg1=0 to 101 {
4    %1=affine.load %av[%arg1]:memref<101xi1>
5    %2=affine.load %bv[%arg1]:memref<101xi1>
6    %3=arith.andi %1, %2:i1
7    %4=arith.xori %3, %true:i1
8  }
```

Loop tiling

```
0  %av, %bv: memref<101xi1>
1
2  %true = arith.constant true
3  affine.for %arg1=0 to 101 step 3 {
4   affine.for %arg2= %arg1 to min (%arg1+3, 101) {
5    %1=affine.load %av[%arg2]:memref<101xi1>
6    %2=affine.load %bv[%arg2]:memref<101xi1>
7    %3=arith.andi %1, %2:i1
8    %4=arith.xori %3, %true:i1
9   }
10  }
```

```
1  affine.for %arg1 = m1 to n1 step k2:
2    // Loop-body
```

Transformation Pattern Condition:

$1.k_1 = f * k_2\ (f\ is\ the\ tiling\ factor) \quad 2.n_2 = \min(\%arg1 + k1, n1)$

```
1  affine.for %arg1 = m1 to n1 step k1:
2    affine.for %arg2 = %arg1 to n2 step k2:
3      // Loop-body
```

E-Graph Runner

- Input:
  - Graph representation of MLIR code
  - Hybrid rule set: all static rewriting rules + any dynamic rules
- Rewrite-Based Verification
  - Apply Static Rules
  - Attempt Equivalence Check
  - If the root nodes of the two programs have merged
    - Yes: Finished
    - No: scan the current e-graph for control-flow candidates and inject dynamic rules
  - Repeat until no new merges occur
- Result:
  - **Equivalent** if root nodes collapse into the same e-class
  - **Not Equivalent** if saturation exhausts with distinct classes

Formal Guarantees – Soundness and Completeness

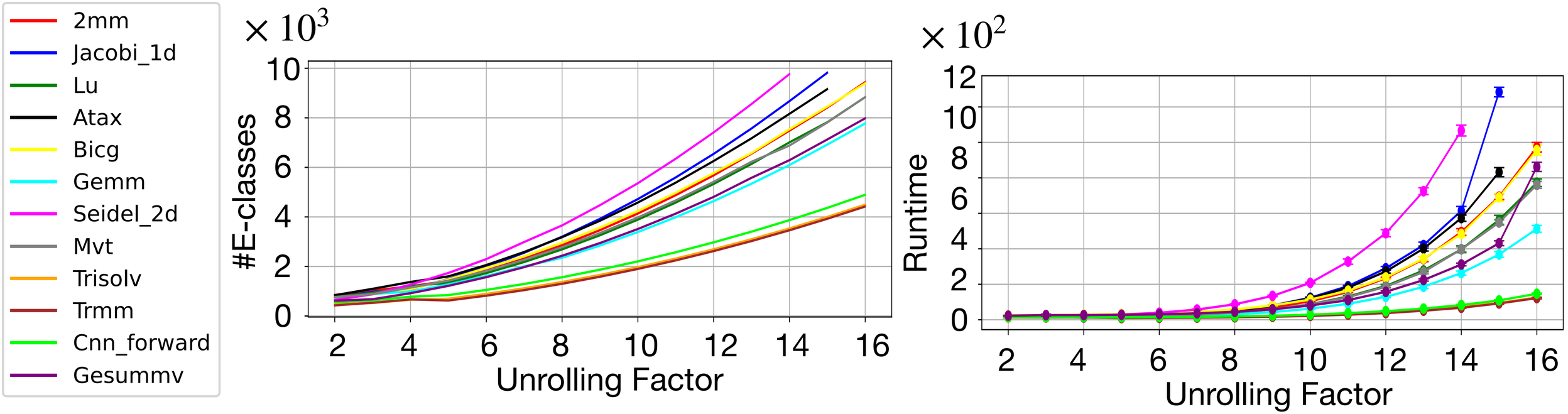
- Sound but not Complete
  - No false positives
    - If we declare two programs equivalent, they truly are
  - False negatives
    - Some equivalence may escape our rule set

Results

- Benchmarks: GEMM [1], LU [1], 2MM [1], ATAX [1], BiCG [1], GESUMMV [1], MVT [1], GEMM [1], TRISOLV [1], TRMM [1], CNN_Forward [2], Jacobi_1D [1], Seidel_2D [1]
- Control flow transformation:
  - Loop unrolling (up to nested U16×U16)
  - Tiling, Fusion
- Datapath transformation
- Mining Bugs in PolyBenchC MLIR

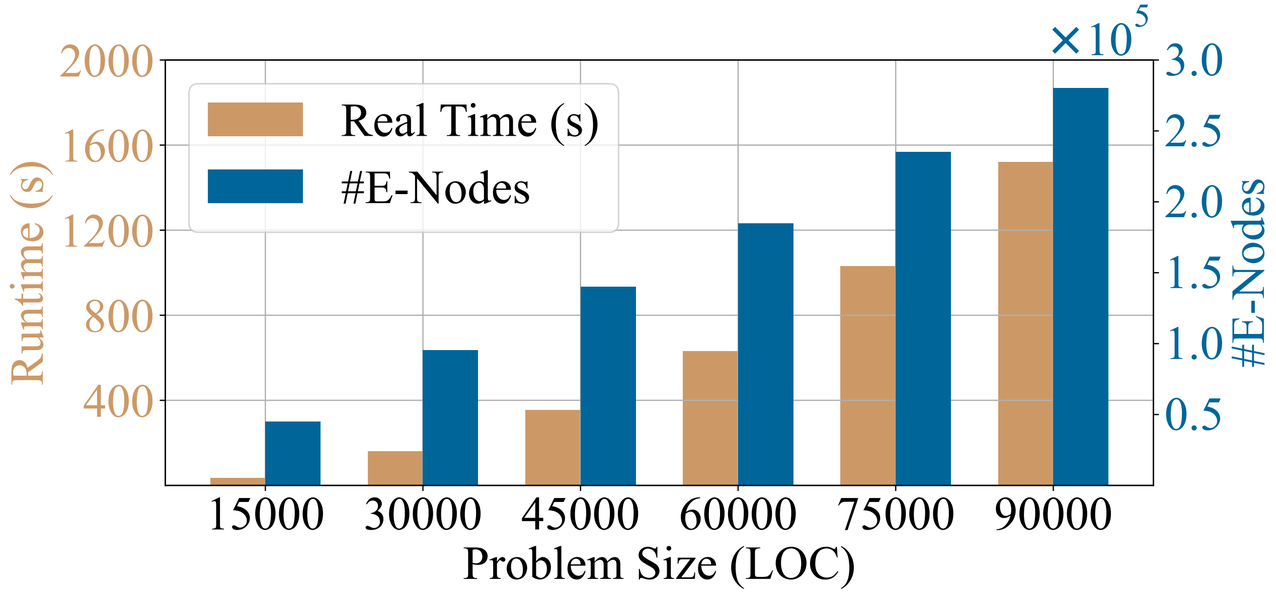
[1]. https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1.git
[2] https://github.com/IITH-Compilers/PolyBench-NN.git

Control FLow Transformation



- Both runtime and e-class count **grow quadratically** as the unrolling factor increases.
- Nested unrolling **quadratically** inflates code size → more e-nodes → more equivalence classes

Datapath Transformation



- All benchmarks (up to ~108 K LOC) complete within ~40 min
- Static (datapath) rewriting dominates runtime growth

Mining Bugs in PolyBenchC MLIR

```
0  func.func @testing2(%arg0: memref<10xi32>, %arg1: memref<10xi32>) {
1    %cst = arith.constant 1 : i32
2    // Replace elements indexed from 1 to 10 with %arg0[0]
3    affine.for %arg2 = 1 to 10 {
4      %1 = affine.load %arg0[%arg2 - 1] : memref<10xi32>
5     affine.store %1, %arg0[%arg2] : memref<10xi32>
6   }
7    // Elements indexed from 1 to 10 are incremented by 1
8   affine.for %arg2 = 1 to 10 {
9      %1 = affine.load %arg0[%arg2] : memref<10xi32>
10     %2 = arith.addi %1, %cst : i32
11     affine.store %2, %arg0[%arg2] : memref<10xi32>
12   }
13   // Finally, memory in %arg0 will be replaced to
14   // %arg0[0], %arg0[0]+1, %arg0[0]+1, ...
15   return
16 }
```

```
0  func.func @testing2(%arg0: memref<10xi32>, %arg1: memref<10xi32>) {
1    %cst = arith.constant 1 : i32
2    affine.for %arg2 = 1 to 10 {
3      // Load data from %arg0[%arg2 - 1]
4      %0 = affine.load %arg0[%arg2 - 1] : memref<10xi32>
5      affine.store %0, %arg0[%arg2] : memref<10xi32>
6      %1 = affine.load %arg0[%arg2] : memref<10xi32>
7      %2 = arith.addi %1, %cst : i32
8      // Elements are incremented by 1
9      affine.store %2, %arg0[%arg2] : memref<10xi32>
10   }
11   // Finally, memory in %arg0 will be replaced to
12   // %arg0[0], %arg0[0]+1, %arg0[0]+2, ...
13   return
14 }
```

- What happened?
  - The MLIR optimizer fused two sequential loops that share an array dependency
  - Origin %arg0[0], %arg0[0]+1, %arg0[0]+1, ...
  - After loop fusion: %arg0[0], %arg0[0]+1, %arg0[0]+2, ...
- Alters execution order of writes and reads → **incorrect semantic** behavior.
- Such RAW hazards can silently corrupt results.

Thank You!