

# Enabling AI-based Automated Code Generation with Guaranteed Correctness

E. Vatai, A. Drozd, I. R. Ivanov, Y. Ren, M. Wahib

R-CCS, High Performance Artificial Intelligence Systems Research Team



THIS IS YOUR MACHINE LEARNING SYSTEM?

YUP! YOU POUR THE DATA INTO THIS BIG  
PILE OF LINEAR ALGEBRA, THEN COLLECT  
THE ANSWERS ON THE OTHER SIDE.

WHAT IF THE ANSWERS ARE WRONG?

JUST STIR THE PILE UNTIL  
THEY START LOOKING RIGHT.



# ML approaches to correctness in code generation

- ▶ Extensive unit testing
  - ▶ Coverage
  - ▶ Writing tests is not trivial
- ▶ Surrogate ML model
- ▶ Round-trip
  - ▶ Trust issues
- ▶ Limit ML to short sequences of instructions
- ▶ Limit ML to a set of determined transformations
  - ▶ Not general
- ▶ Symbolic execution
  - ▶ Not well established

Edsger W. Dijkstra

*“Program testing can be used to show the presence of bugs, but never to show their absence!”*

# ML approaches to correctness in code generation

- ▶ Extensive unit testing
  - ▶ Coverage
  - ▶ Writing tests is not trivial
- ▶ Surrogate ML model
- ▶ Round-trip
  - ▶ Trust issues
- ▶ Limit ML to short sequences of instructions
- ▶ Limit ML to a set of determined transformations
  - ▶ Not general
- ▶ Symbolic execution
  - ▶ Not well established

Edsger W. Dijkstra

*“Program testing can be used to show the presence of bugs, but never to show their absence!”*

# ML approaches to correctness in code generation

- ▶ Extensive unit testing
  - ▶ Coverage
  - ▶ Writing tests is not trivial
- ▶ Surrogate ML model
- ▶ Round-trip
  - ▶ Trust issues
- ▶ Limit ML to short sequences of instructions
- ▶ Limit ML to a set of determined transformations
  - ▶ Not general
- ▶ Symbolic execution
  - ▶ Not well established

Edsger W. Dijkstra

*“Program testing can be used to show the presence of bugs, but never to show their absence!”*

# ML approaches to correctness in code generation

- ▶ Extensive unit testing
  - ▶ Coverage
  - ▶ Writing tests is not trivial
- ▶ Surrogate ML model
- ▶ Round-trip
  - ▶ Trust issues
- ▶ Limit ML to short sequences of instructions
- ▶ Limit ML to a set of determined transformations
  - ▶ Not general
- ▶ Symbolic execution
  - ▶ Not well established

Edsger W. Dijkstra

*“Program testing can be used to show the presence of bugs, but never to show their absence!”*



# ML approaches to correctness in code generation

- ▶ Extensive unit testing
  - ▶ Coverage
  - ▶ Writing tests is not trivial
- ▶ Surrogate ML model
- ▶ Round-trip
  - ▶ Trust issues
- ▶ Limit ML to short sequences of instructions
- ▶ Limit ML to a set of determined transformations
  - ▶ Not general
- ▶ Symbolic execution
  - ▶ Not well established

Edsger W. Dijkstra

*“Program testing can be used to show the presence of bugs, but never to show their absence!”*



All you need is to sample the set of correct transformations

- ▶ ML codegen is **transformations** on a reference implementation
- ▶ ML can **explore the space of transformations** to find a faster version
- ▶ But we also need **correctness**

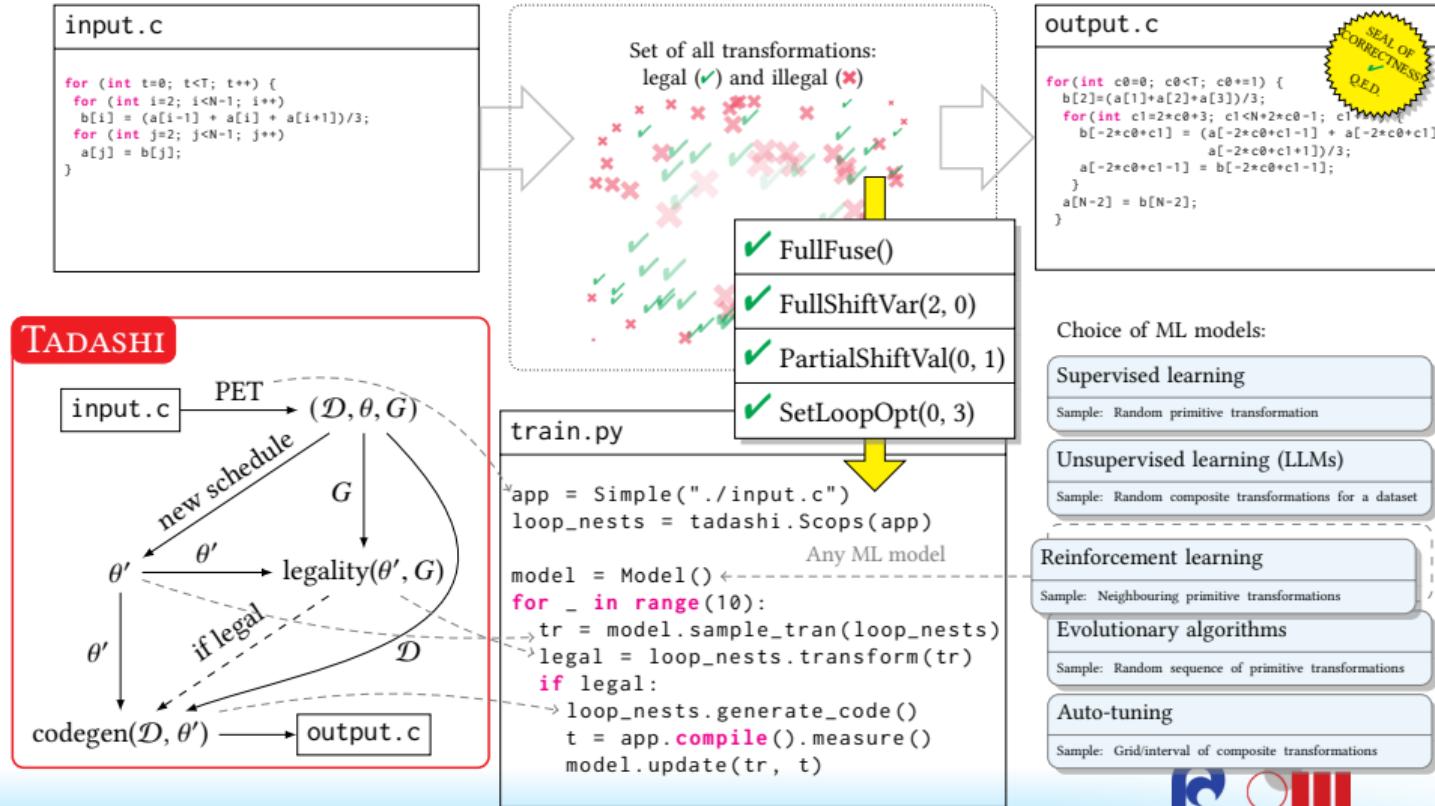
All you need is to sample the set of correct transformations

- ▶ ML codegen is transformations on a reference implementation
- ▶ ML can explore the space of transformations to find a faster version
- ▶ But we also need correctness

All you need is to sample the set of correct transformations

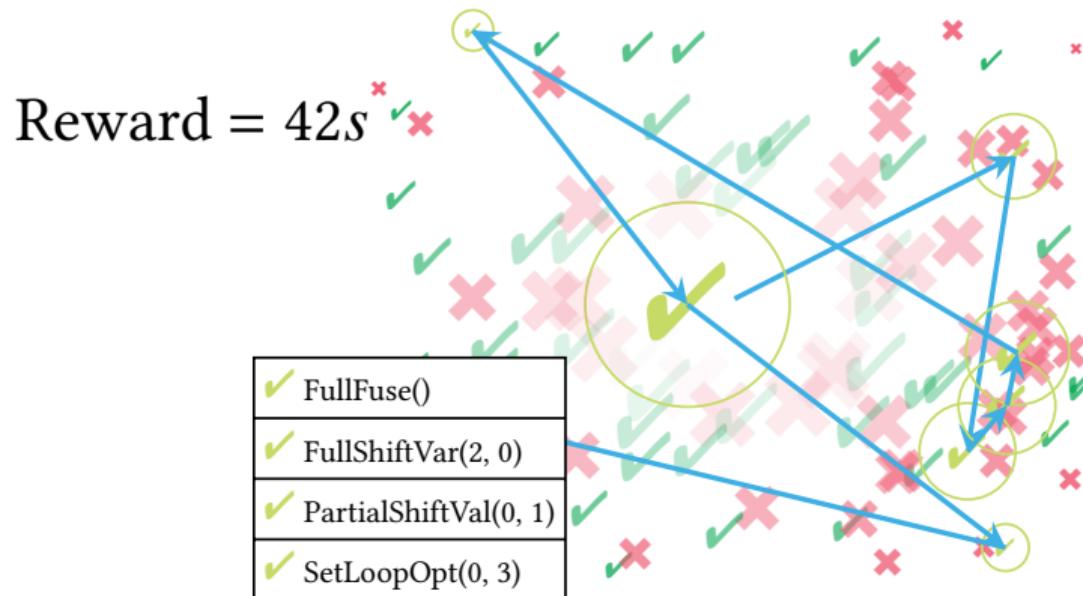
- ▶ ML codegen is transformations on a reference implementation
- ▶ ML can explore the space of transformations to find a faster version
- ▶ But we also need correctness

# TADASHI: loop transformations with correctness check



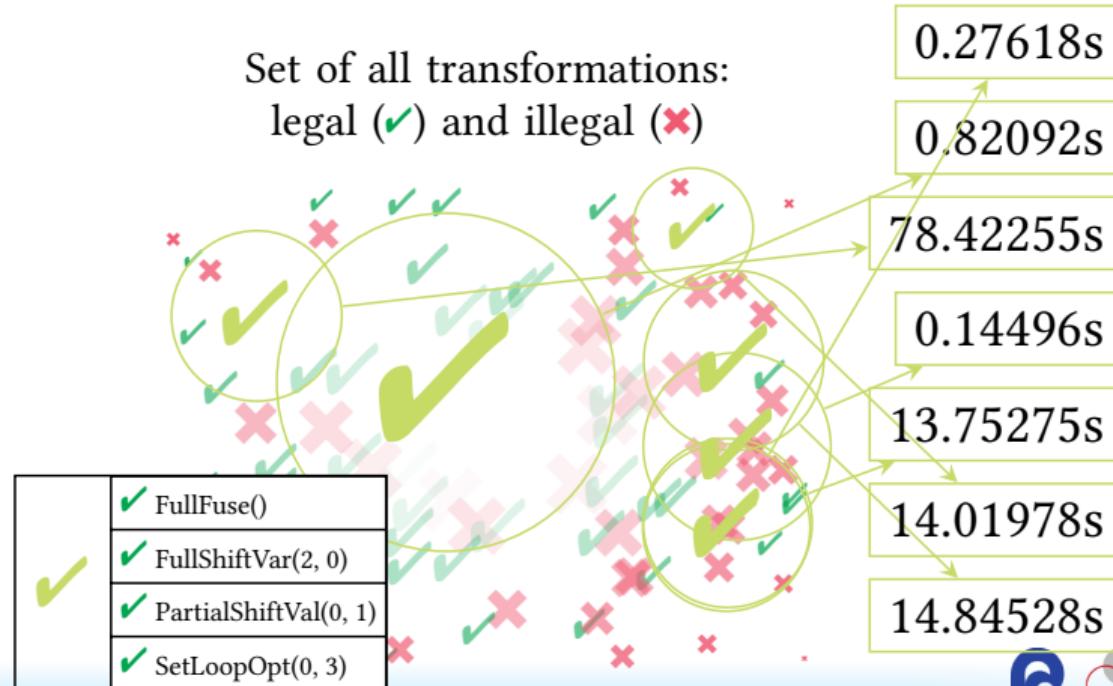
# Reinforcement learning

- Actions = primitive transformations; Reward = walltime; Environment = correctness



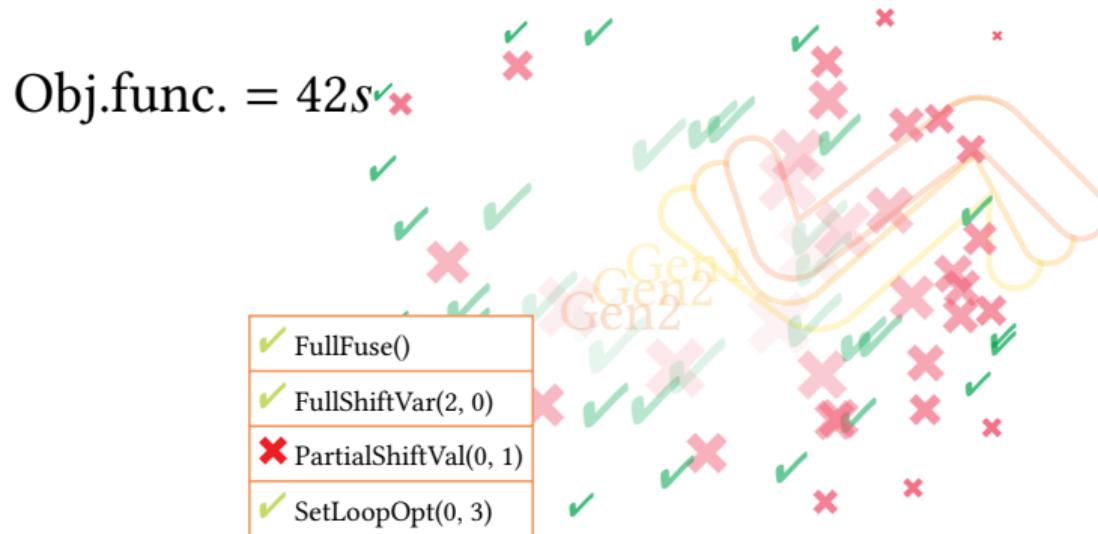
# Supervised learning

- Dataset generation (legal transformations); Sample label: runtime



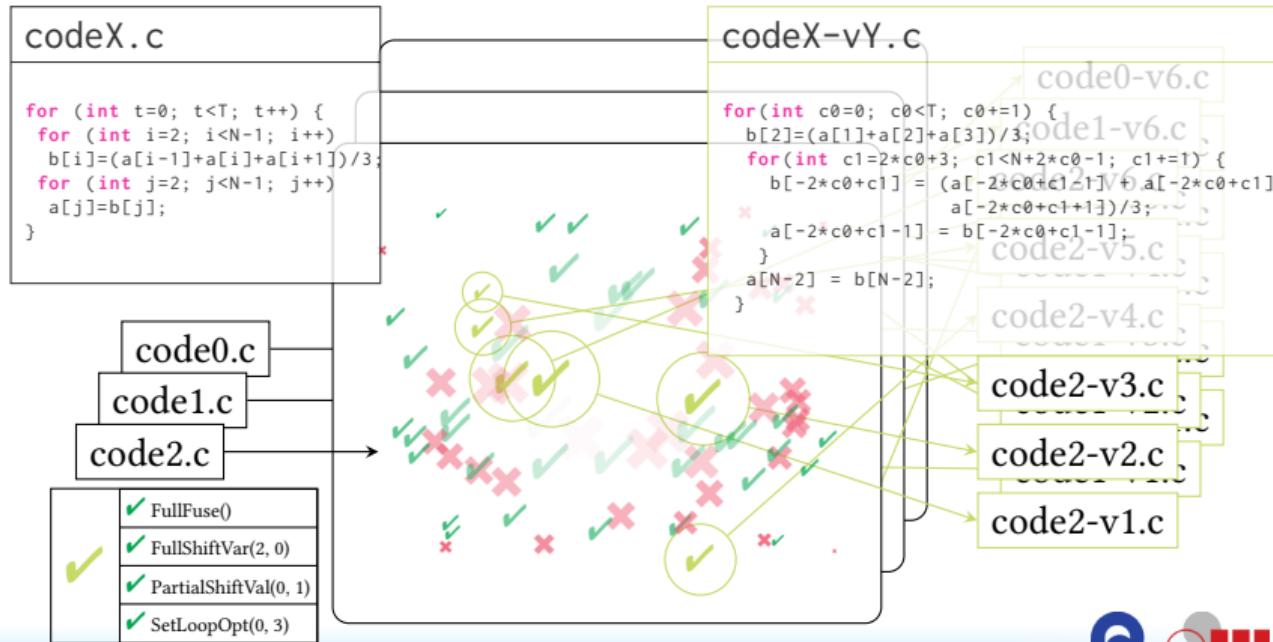
# Evolutionary algorithms

- ▶ Exploration and explorations; Candidates = transformations; Objective function = runtime



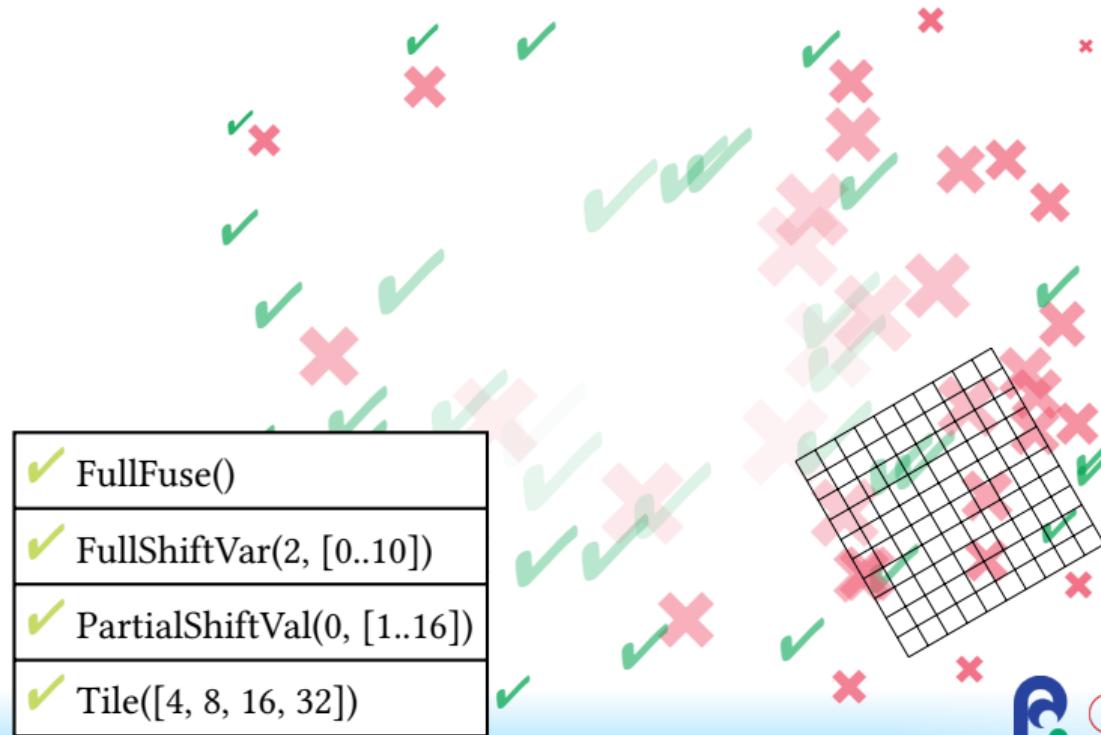
# LLM agents

- ▶ Dataset generation; High quality correct transformations; Caveat!  
Hallucinations are possible!



# Auto-Tuning

- Brute forcing on a bounded region of the space (grid search)

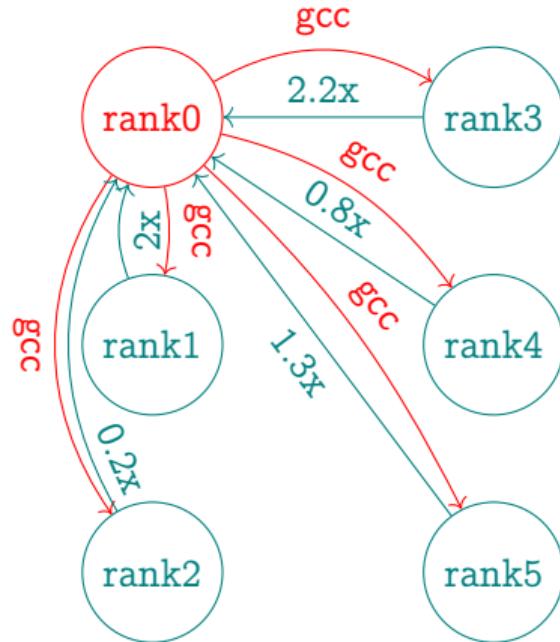


# Benchmarking harness

The **compiler** spends a few minutes optimising the code on a single node.

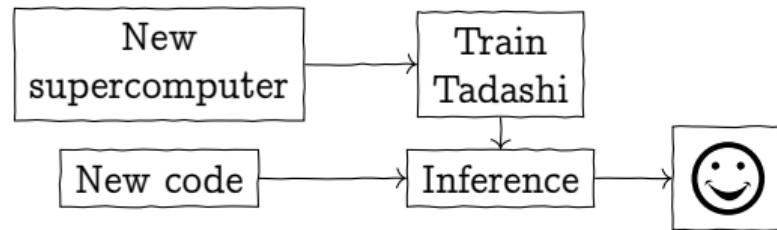
Why not use the entire supercomputer?

- ▶ Benchmarking harness
  - ▶ Distribute compiling and running of benchmarks
  - ▶ Collect the speedups to remote nodes
  - ▶ Uses **MPI4py**, and exposed to the user as **concurrent.futures** (from the Python standard)



# Grand vision

## Large scale optimisation framework



## Wise words

*Spending 6h on the whole machine to make the code 5% faster will pay off. – N. D.*

## Now: Support C, Harness, and Transformations

TrEnum	Args
TILE	tile size
INTERCHANGE	-
FULL_FUSE	-
FUSE	2 loop indices
FULL_SHIFT_VAL	const shift value
PARTIAL_SHIFT_VAL	stmt idx, const shift value
FULL_SHIFT_VAR	coeff, shift iter idx
PARTIAL_SHIFT_VAR	stmt idx, coeff, shift iter idx
FULL_SHIFT_PARAM	coeff, shift param idx
PARTIAL_SHIFT_PARAM	stmt idx, coeff, shift param idx
SCALE	coeff
SET_PARALLEL	num\threads
SET_LOOP_OPT	AstLoopType enum

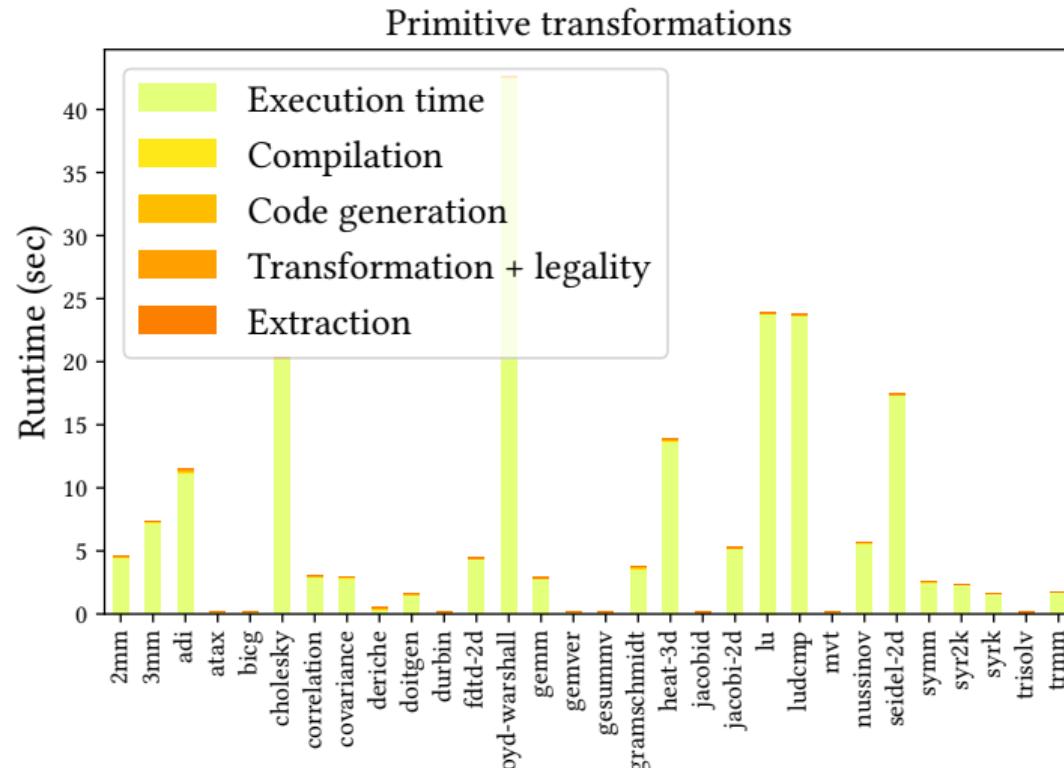
# Future:

- ▶ New language
  - ▶ Fortran (MLIR, Polygeist)
  - ▶ CUDA (PPCG?)
- ▶ Better use of the polyhedral model
  - ▶ New transformations: Loop fission, unrolling, etc.
  - ▶ Expansion and extension nodes
  - ▶ More efficient legality check
- ▶ ML (target list)
  - ▶ Reinforcement Learning
  - ▶ LLM Agents
  - ▶ Auto-Tuning
  - ▶ Supervised Learning
  - ▶ Evolutionary Algorithms

# Random RL

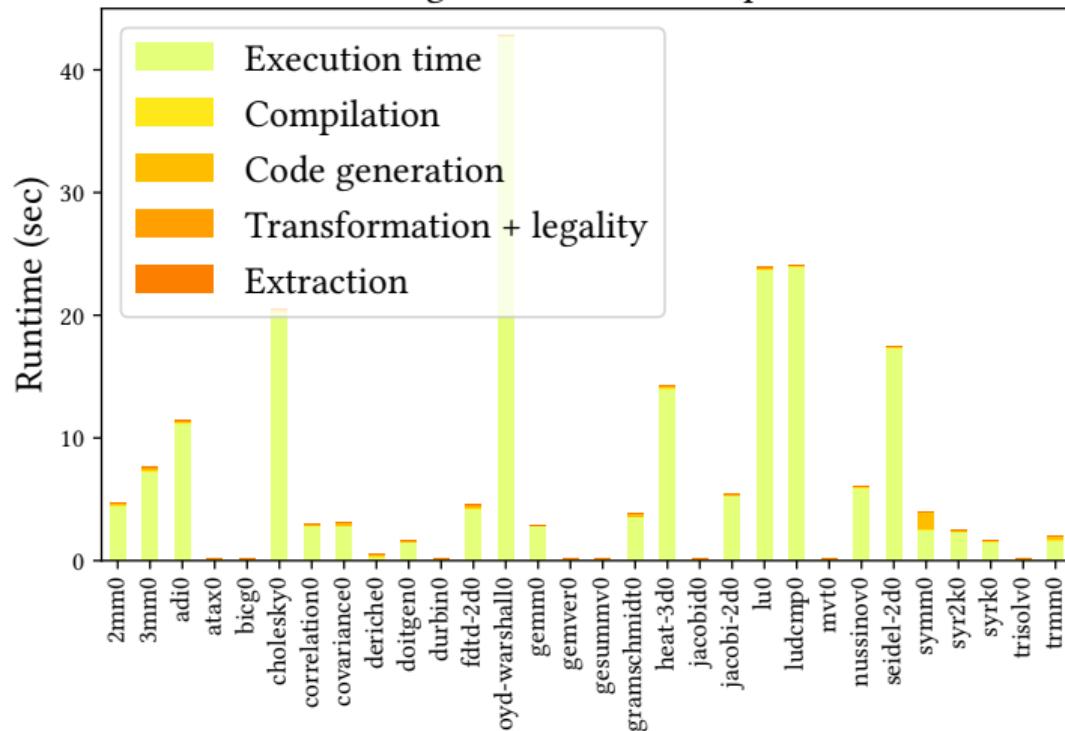
```
def train_model(app, num_iter=3, net=Model()):  
    loop_nests = LoopNests(app)  
    ln = loop_nests[0]  
    for i in range(num_iter):  
        loop_idx, tr, args = net.transform(ln)  
        loop = ln.schedule_tree[loop_idx]  
        legal = loop.transform(tr, *args)  
        if not legal:  
            node.rollback()  
            continue  
        loop_nests.generate_code("output.c")  
        app.compile()  
        t = app.measure()  
        # net.update(loop_idx, tr, args, legal, t)
```

# Overhead breakdown (primitive)

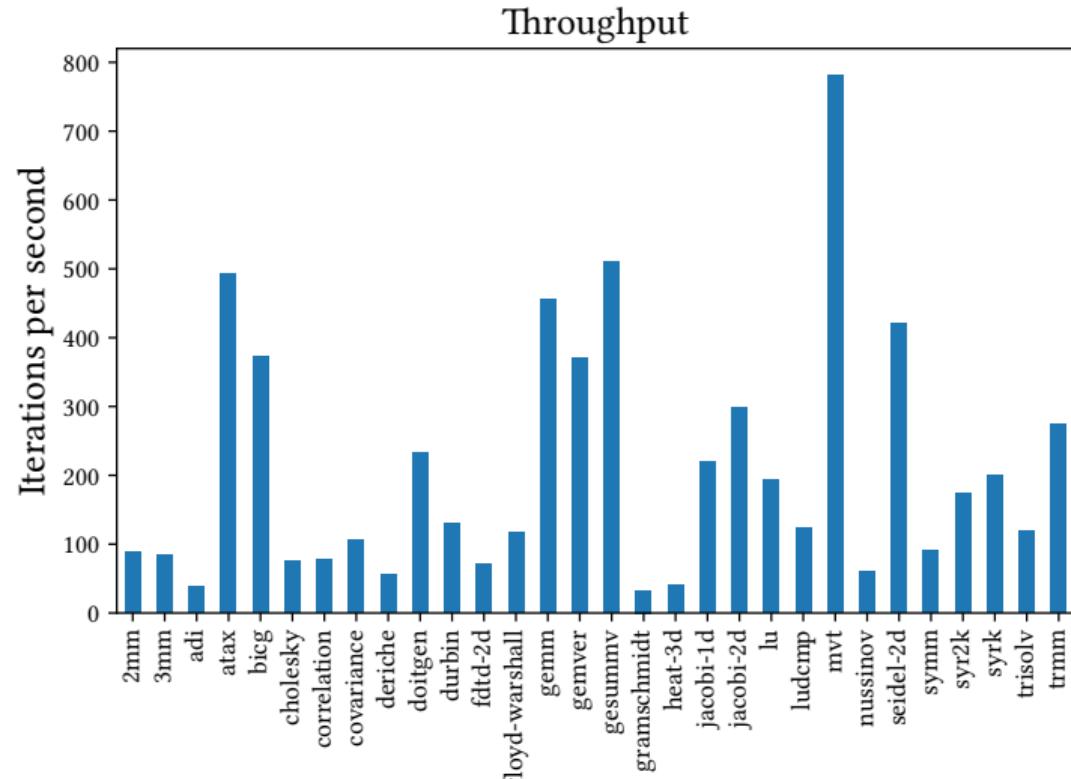


# Overhead breakdown (10 seq)

10-long transformation sequence



# Throughput (of transformations)



# Representations

Different levels of abstraction

- ▶ source code [Stein: Paraphrasing etc.]
- ▶ abstract syntax tree (AST) [Shido: Tree-LSTM etc.]
- ▶ polyhedral [Baghdadi: Tiramisu]
- ▶ dependency graphs [Cummins: ProGraML]
- ▶ intermediate Representations (IR) [Ben-nun: inst2vec]
- ▶ assembly instructions [Deepmind: Faster sorting]

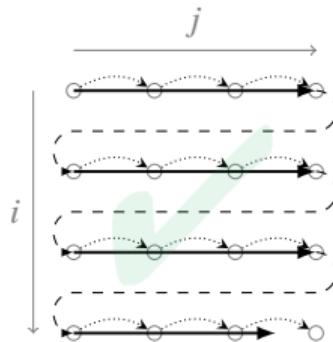
# Polyhedral model

## Components

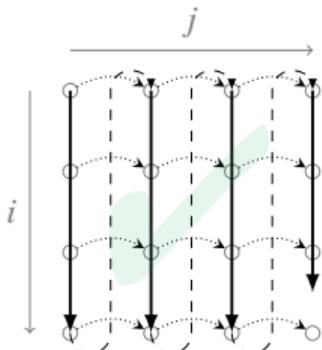
1.  $\mathcal{D}_S = \{S[\vec{i}] \in \mathbb{Z}^n : \mathbf{A}\vec{i} + \mathbf{b} \leq 0\}$
2.  $\theta(S[i, j]) = t = (i, j)$
3.  $G = (V, E)$ :
  - ▶  $V = \{S_0, S_1, \dots\}$ ,
  - ▶  $E = \{S_i[\vec{d}] \mapsto S_j[\vec{r}], \dots\}$

## Mini example

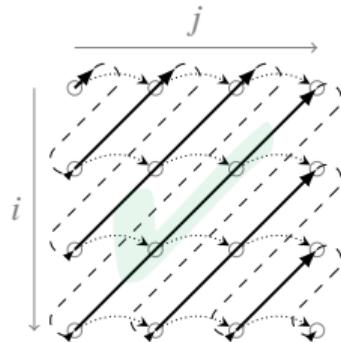
```
for(int i = 0; i < N; i++)
    for(int j = 1; j < M; j++)
        A[i, j] += A[i, j-1]; // S[i][j]
```



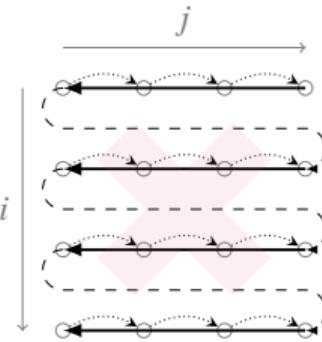
(a)  $\theta(S[i, j]) = (i, j)$



(b)  $\theta(S[i, j]) = (j, i)$



(c)  $\theta(S[i, j]) = (i + j, j)$



(d)  $\theta(S[i, j]) = (i, -j)$

## Legality check

$G:$



$$[S[i, j - 1] \mapsto S[i, j]]$$

$$\theta(S[i, j]) = (i + j, j)$$

$$(i + j - 1, j - 1) \mapsto (i + j, j)$$

$$\begin{aligned} \delta &= (i + j, j) - (i + j - 1, j - 1) \\ &= (1, 1) >_{\text{LEX}} \vec{0} \end{aligned}$$

$G:$



$$[S[i, j - 1] \mapsto S[i, j]]$$

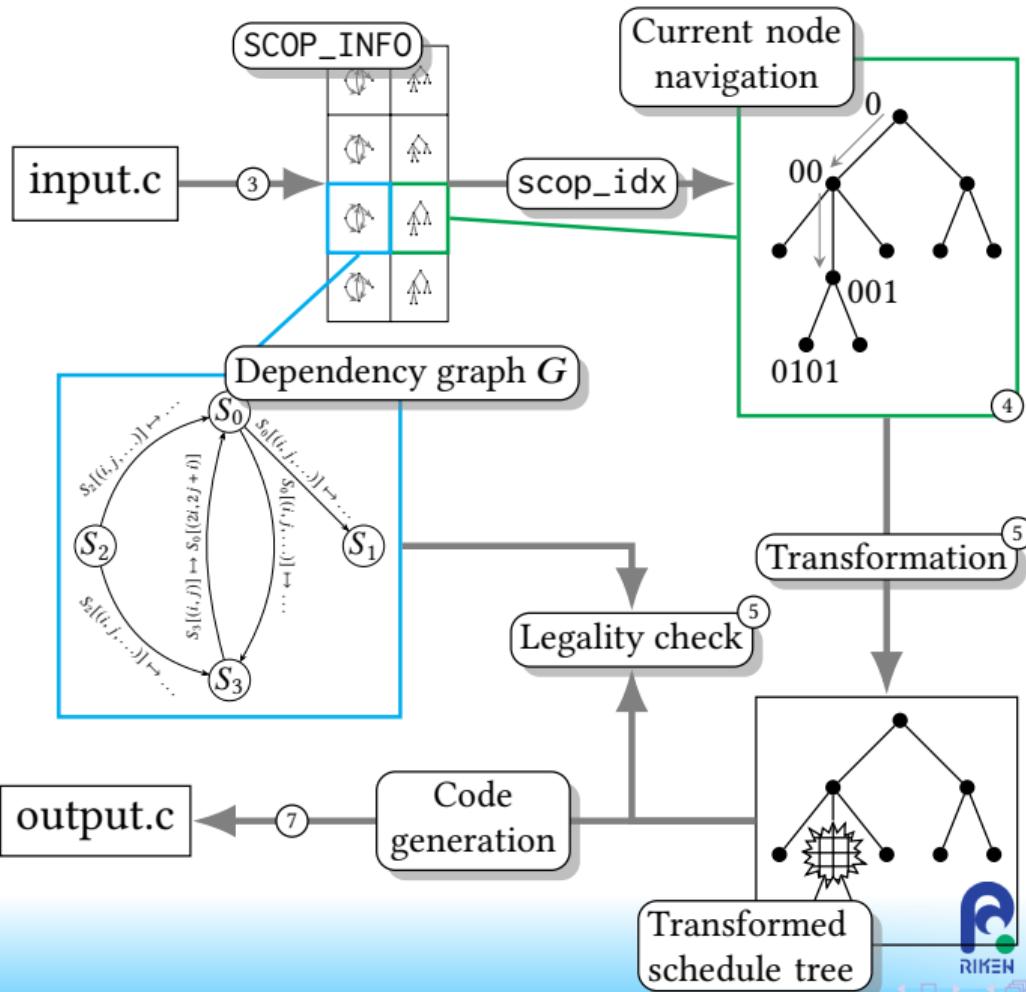
$$\theta(S[i, j]) = (i, -j)$$

$$(i, -j + 1) \mapsto (i, -j)$$

$$\begin{aligned} \delta &= (i, -j) - (i, -j + 1) \\ &= (0, -1) \not>_{\text{LEX}} \vec{0} \end{aligned}$$

## End-to-end example

```
import tadashi
app = tadahis.Simple("input.c")
scops = tadahi.Scops(app)
node = scops[0].schedule_tree[1]
tr, args = tadashi.TrEnum.TILE, 16
legal = node.transform(tr, args)
if legal:
    scops.generate_code(output_path="output.c")
    app.compile()
    t = app.measure()
    printf(f"Walltime: {t}")
```



That's all folks!

- ▶ i. <https://vatai.github.io/>
- ▶ ii. <https://github.com/vatai/tadashi/>
- ▶ iii. <https://arxiv.org/abs/2410.03210>

