Verified Validation for Affine Scheduling in Polyhedral Compilation

Xuyang Li, Hongjin Liang, Xinyu Feng

Nanjing University

Background

- Performance of modern software relies on compiler optimizations
- **Nested loops** are optimization targets due to its heavy numerical computation, like in scientific computing and machine learning
- Optimization Goals
  - Enhance locality of memory access
  - Increase program parallelism

```
for i in [1, N]:
S:  A[i] = B[i-1] + C[i+1]
```

→

```
parallel for i in [1, N]:
S:  A[i] = B[i-1] + C[i+1]
```

(Suppose arrays A, B, C are non-aliasing)

Background

- For nested loops with complex memory access pattern, loop transformations are needed to achieve better optimization
  - Like loop fusion, interchange, skewing, etc...
- We reason the dependences of instruction(s)'s iterations for such transformation's correctness, according to the memory access expression

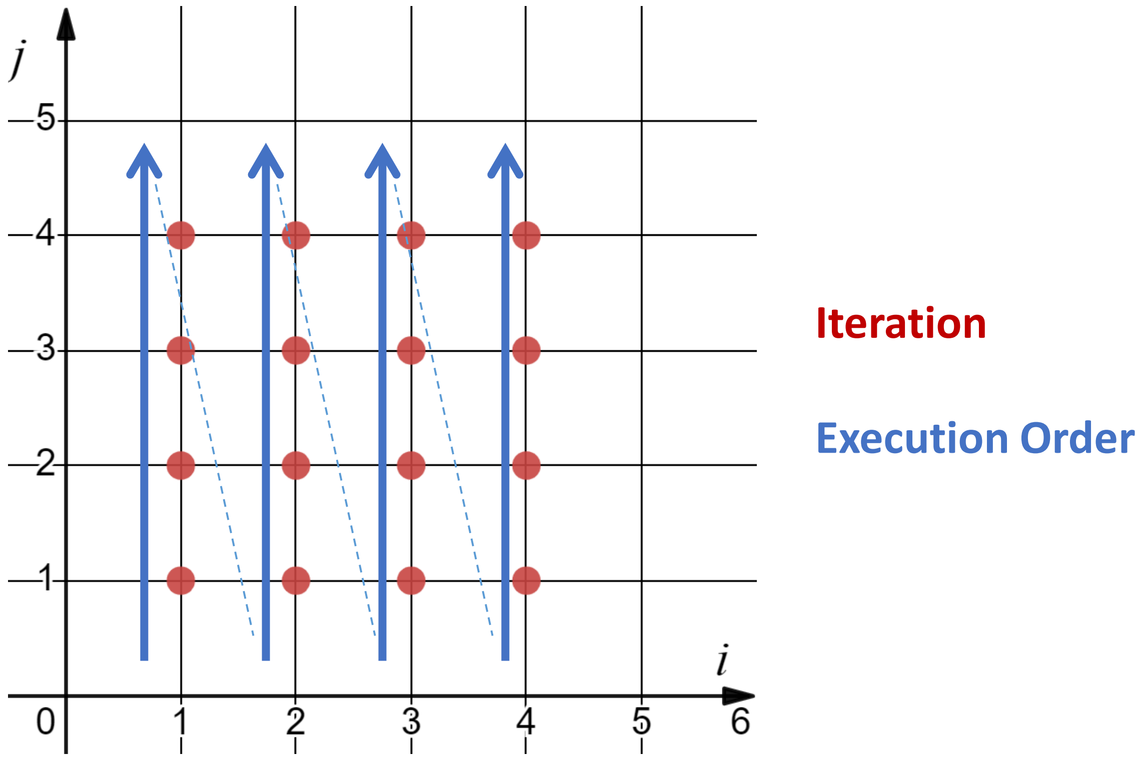
```
for i in [1, N]:
  for j in [1, N]:
S:  A[i][j] = A[i+1][j] + A[i-1][j-1]
```

Is the loop parallelizable?

Example

Unroll each iteration of the instruction(s) into a coordinate system with the loop variables as the axis.

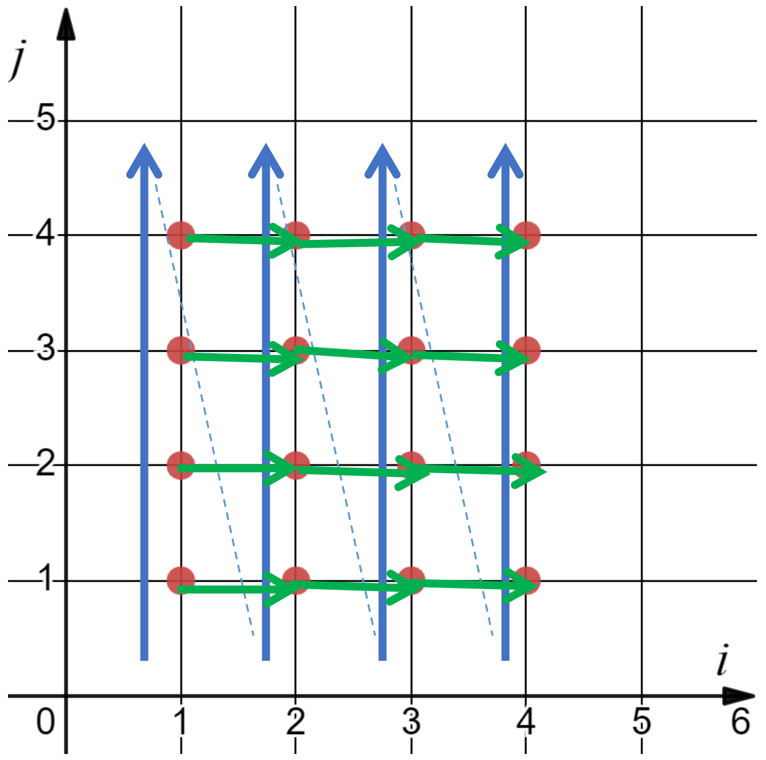
```
for i in [1, N]:
  for j in [1, N]:
S:  A[i][j] = A[i+1][j] + A[i-1][j-1]
```



Example

Write after Read (WAR) **dependence**: iteration (i, j) reads A[i+1][j], iteration (i+1, j) writes A[i+1][j]. Not permutable!

```
for i in [1, N]:
  for j in [1, N]:
S:  A[i][j] = A[i+1][j] + A[i-1][j-1]
```



Iteration

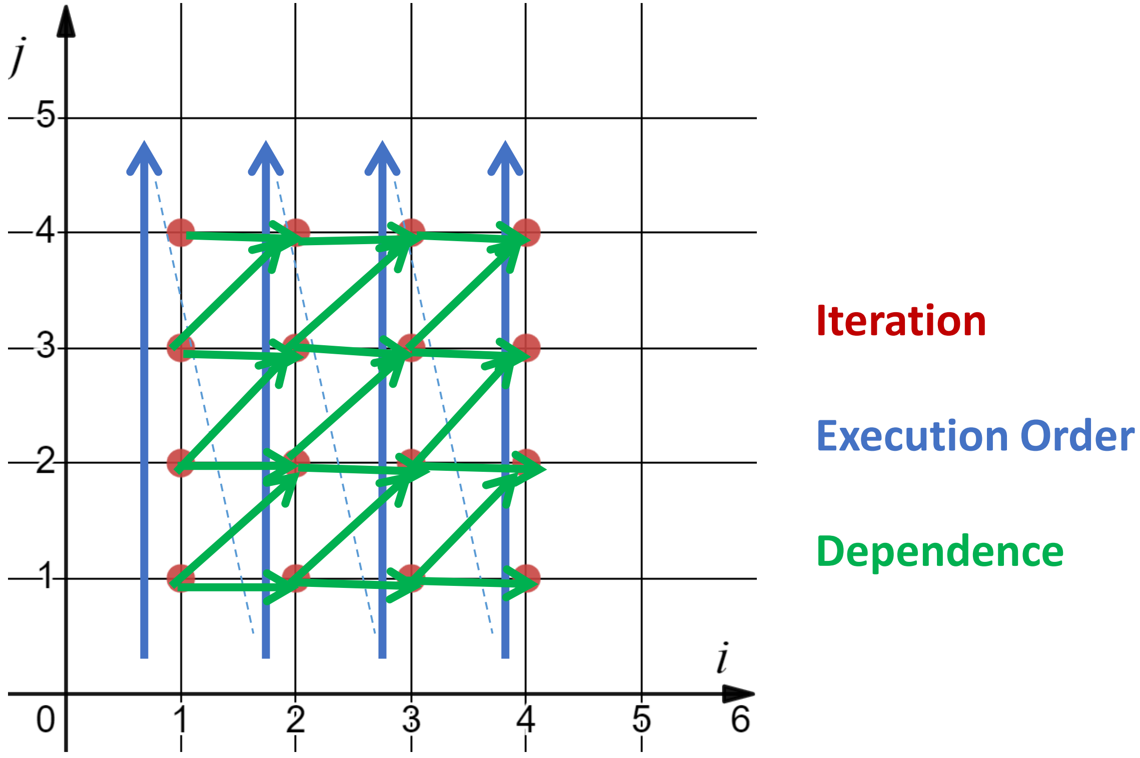
Execution Order

Dependence

Example

Read after Write (RAW) **dependence** :
iteration (i,j) writes A[i][j], iteration
(i+1,j+1) reads A[i][j]. Not permutable!

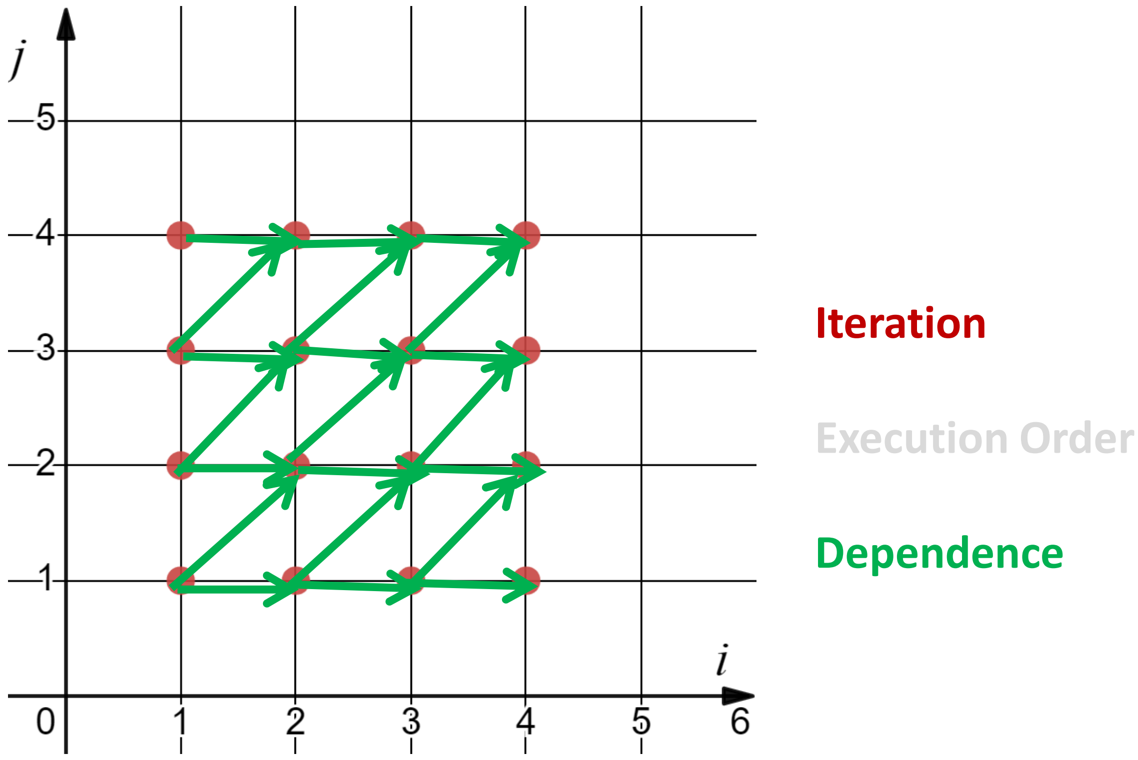
```
for i in [1, N]:
  for j in [1, N]:
S:   A[i][j] = A[i+1][j] + A[i-1][j-1]
```



Example

Old **execution order** is now useless.
Only **dependences** matter.

```
for i in [1, N]:
  for j in [1, N]:
S:  A[i][j] = A[i+1][j] + A[i-1][j-1]
```

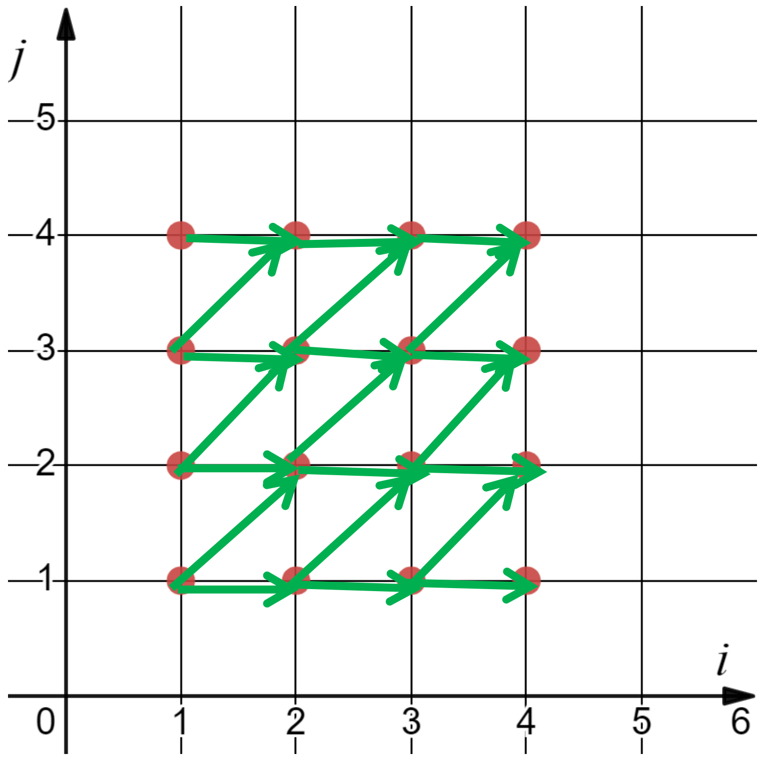


Example

Is the loop parallelizable?

Without breaking dependences?

```
for i in [1, N]:
  for j in [1, N]:
S:  A[i][j] = A[i+1][j] + A[i-1][j-1]
```



Iteration

Execution Order

Dependence

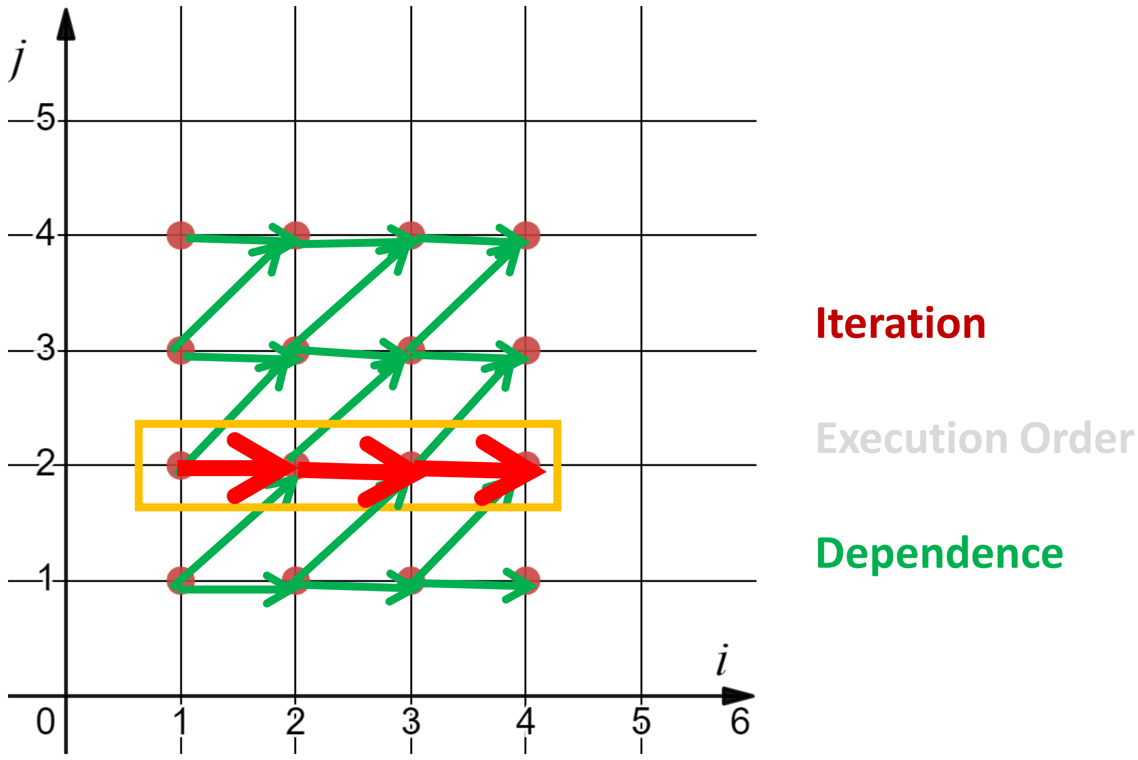
Example

Is the loop parallelizable?

Without breaking dependences?

- for iterations with same j
    - no due to inner dependences
- other possibilities?

```
for i in [1, N]:
  for j in [1, N]:
S:  A[i][j] = A[i+1][j] + A[i-1][j-1]
```



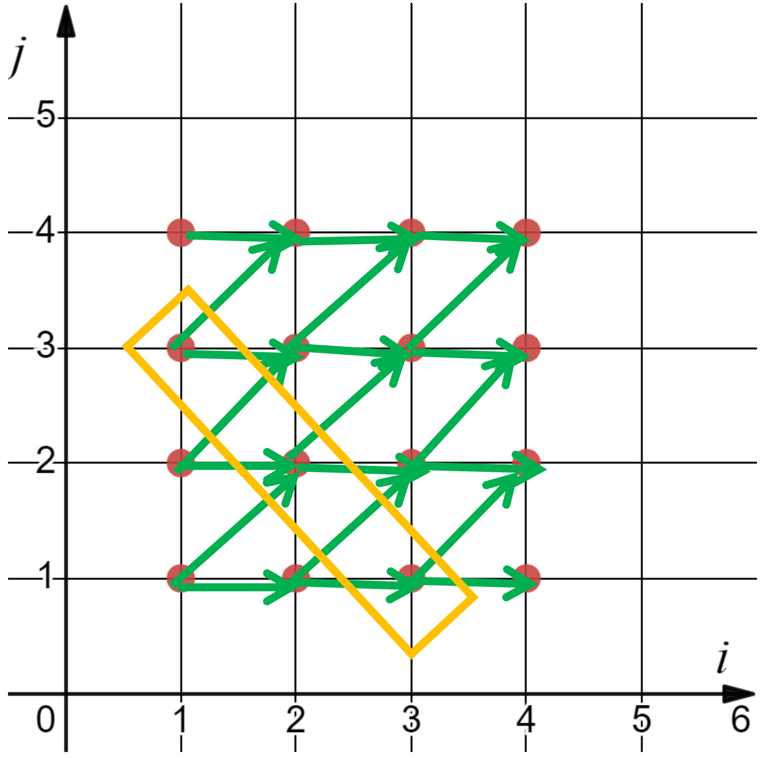
Example

Is the loop parallelizable?

Without breaking dependences?

- for iterations with same j
  - no due to inner dependences
- other possibilities?
  - iterations with same i+j

```
for i in [1, N]:
  for j in [1, N]:
S:  A[i][j] = A[i+1][j] + A[i-1][j-1]
```



Iteration

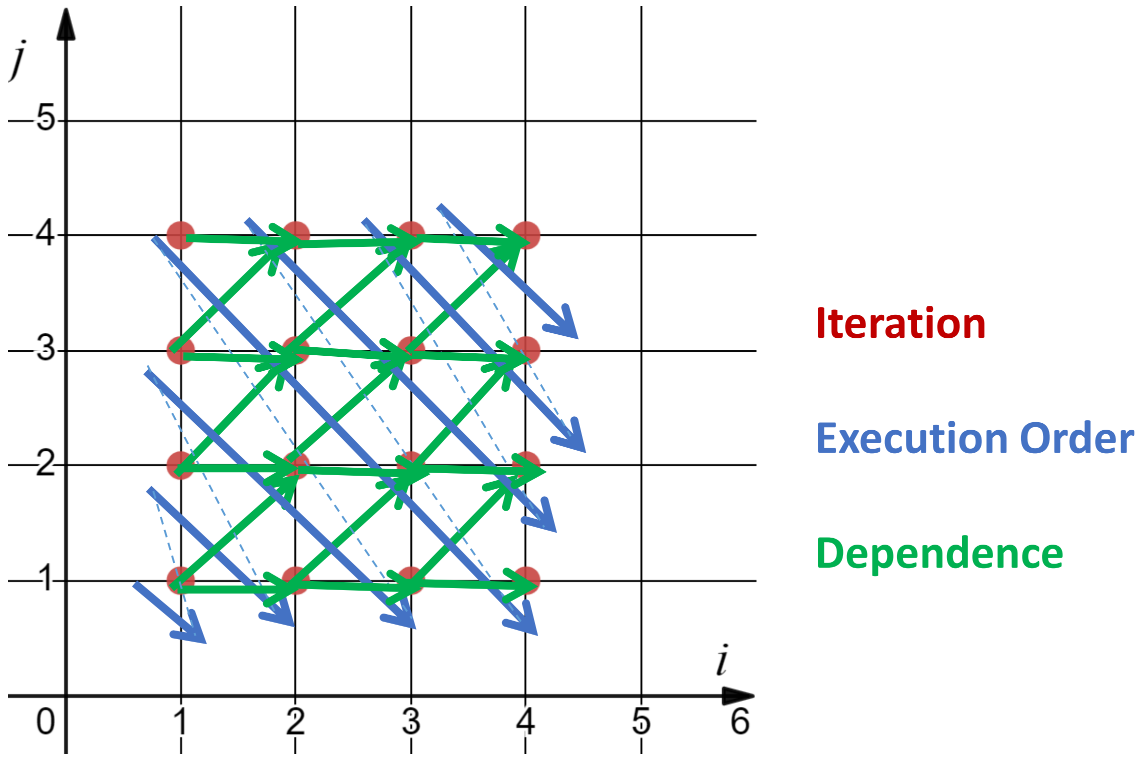
Execution Order

Dependence

Example

Now we can determine a new execution order, not breaking dependences

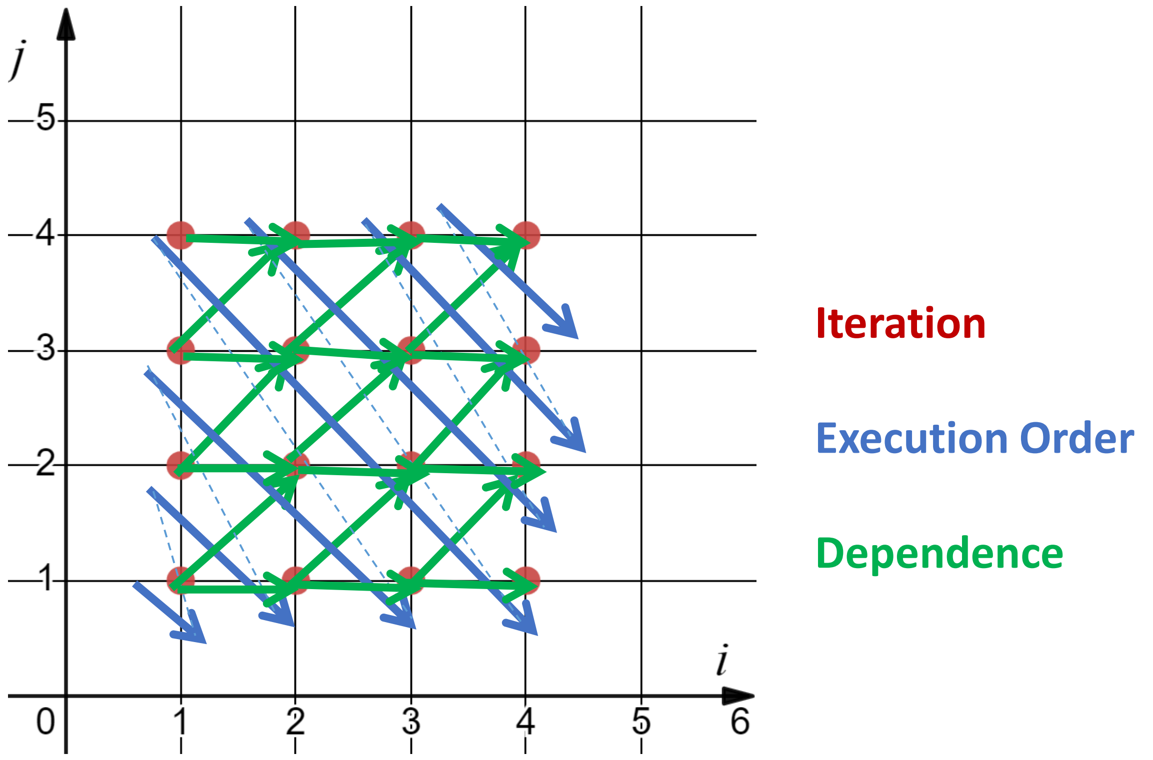
```
for i in [1, N]:
  for j in [1, N]:
S:  A[i][j] = A[i+1][j] + A[i-1][j-1]
```



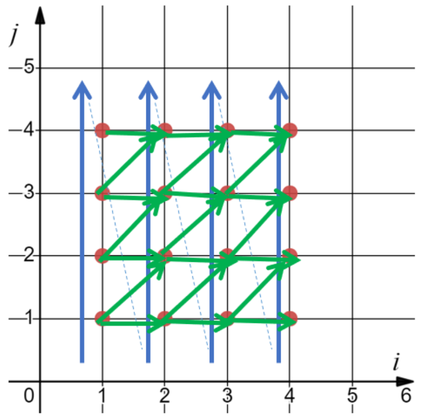
Example

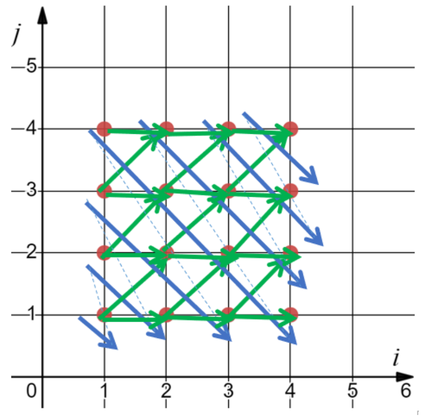
And we can regenerate a new nested loop respecting new execution order, whose inner loop is parallelizable.

```
for j' in [1, 2*N-1]:
  for i' in [max(1, j'-N+1),
             min(N-1, j'-1)]:
S:  A[i'][(j'-i')] = A[i'+1][(j'-i')]
                   + A[i'-1][(j'-i')-1]
```



Example



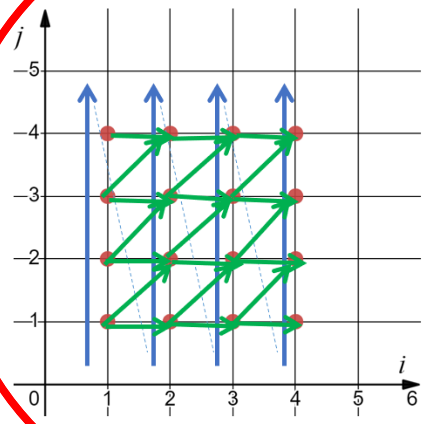


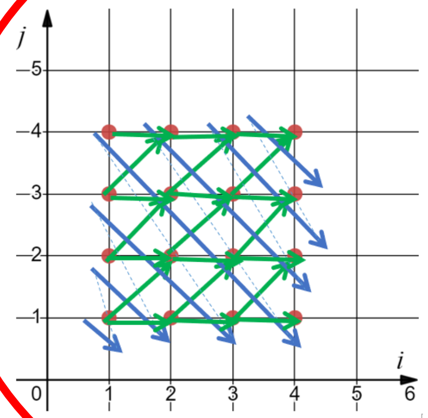

```
for i in [1, N]:
  for j in [1, N]:
S:  A[i][j] = A[i+1][j]
              + A[i-1][j-1]
```

```
for j' in [1, 2*N-1]:
  for i' in [max(1, j'-N+1),
             min(N-1, j'-1)]:
S:  A[i'][(j'-i')] = A[i'+1][(j'-i')]
                   + A[i'-1][(j'-i')-1]
```

loop skewing + loop interchange

Polyhedral Model





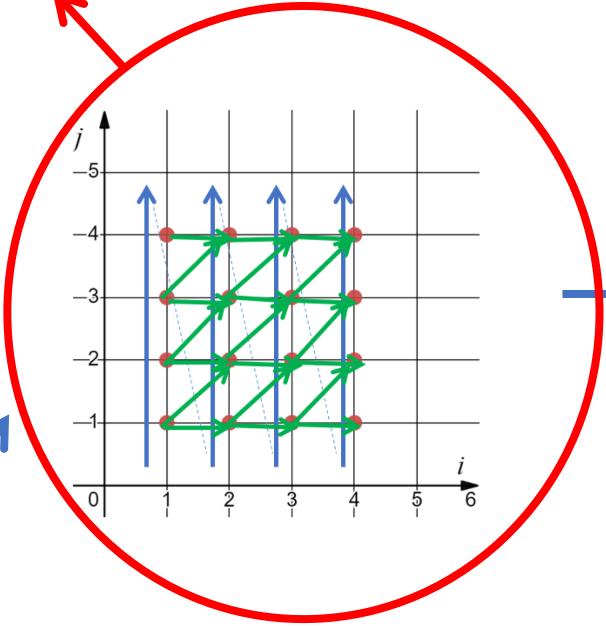

```
for i in [1, N]:
  for j in [1, N]:
S:  A[i][j] = A[i+1][j]
              + A[i-1][j-1]
```

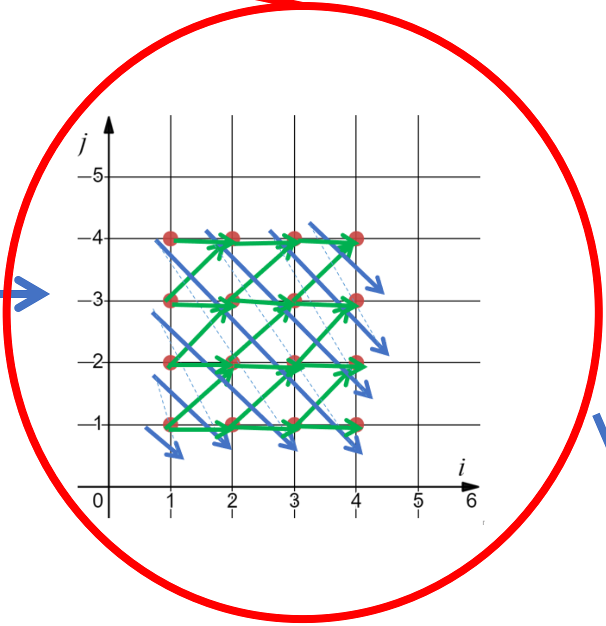
```
for j’ in [1, 2*N-1]:
  for i’ in [max(1, j’-N+1),
             min(N-1, j’-1)]:
S:  A[i’][(j’-i’)] = A[i’+1][(j’-i’)]
                     + A[i’-1][(j’-i’)-1]
```

loop skewing + loop interchange

Polyhedral Model

Iteration
– Domain – "Polyhedron"
Execution order
– Schedule
Dependence



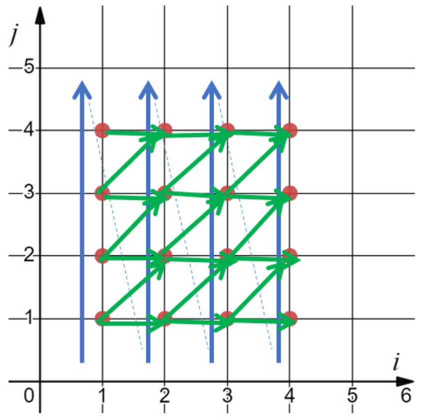


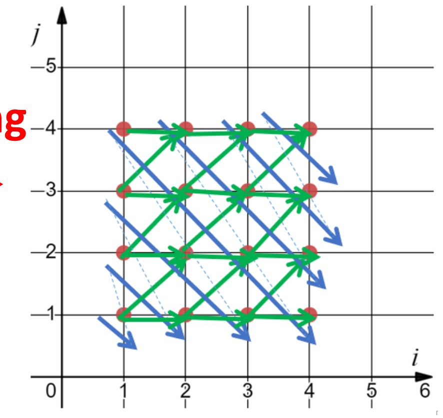

```
for i in [1, N]:
  for j in [1, N]:
S:  A[i][j] = A[i+1][j]
            + A[i-1][j-1]
```

```
for j' in [1, 2*N-1]:
  for i' in [max(1, j'-N+1),
            min(N-1, j'-1)]:
S:  A[i'][(j'-i')] = A[i'+1][(j'-i')]
                   + A[i'-1][(j'-i')-1]
```

loop skewing + loop interchange

Polyhedral Compilation





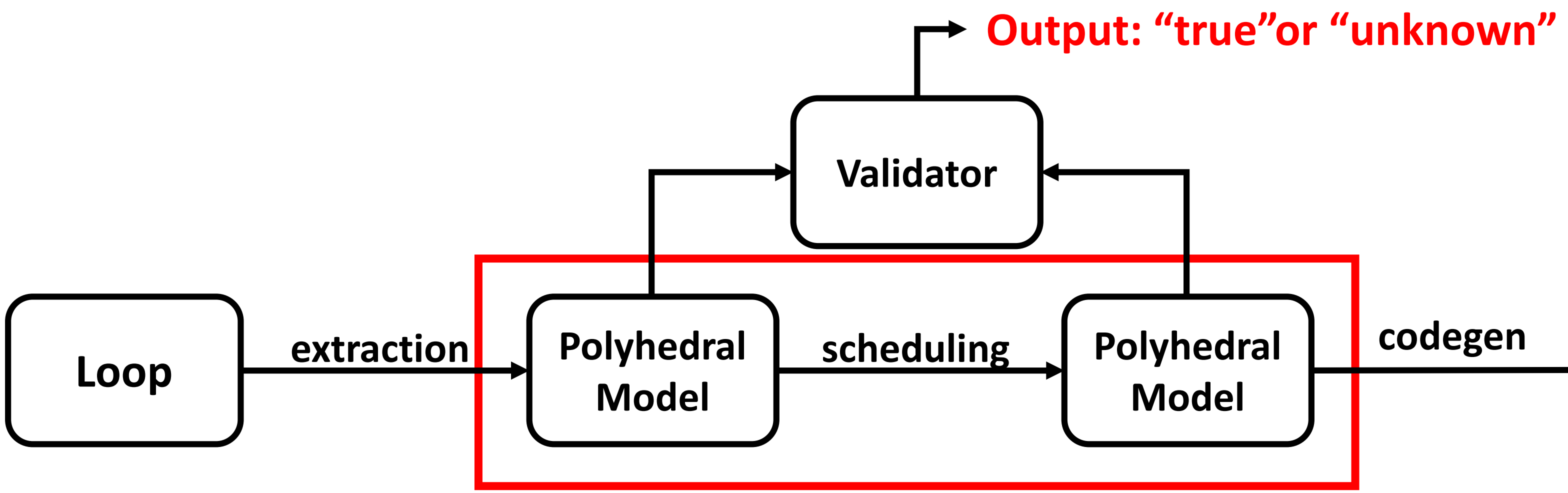

```
for i in [1, N]:
  for j in [1, N]:
S:  A[i][j] = A[i+1][j]
            + A[i-1][j-1]
```

```
for j' in [1, 2*N-1]:
  for i' in [max(1, j'-N+1),
            min(N-1, j'-1)]:
S:  A[i'][(j'-i')] = A[i'+1][(j'-i')]
                   + A[i'-1][(j'-i')-1]
```

Polyhedral Compilation

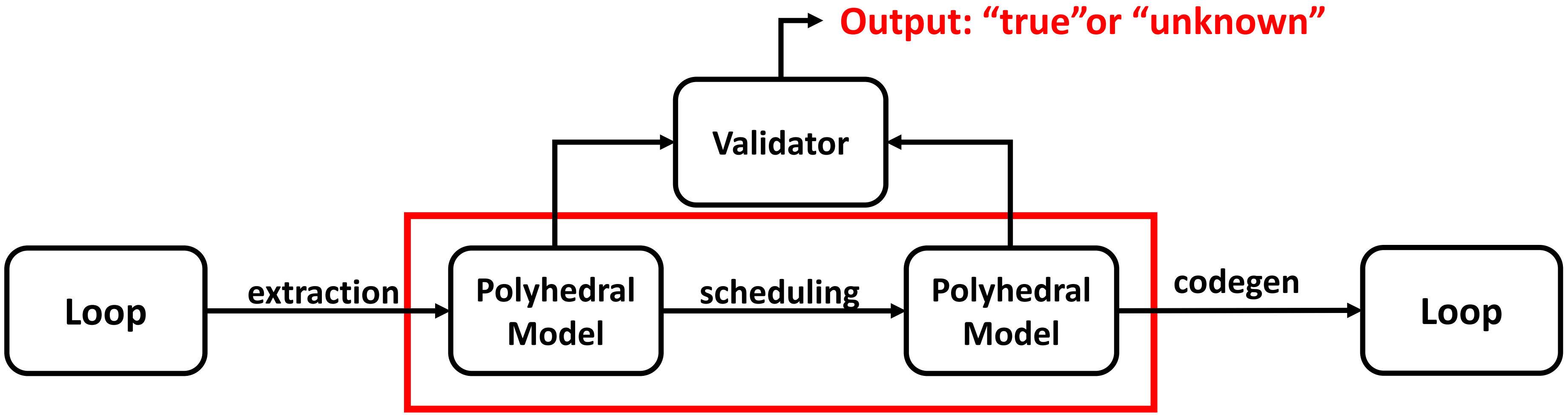
Our work:
Verified Validation for Polyhedral Scheduling

- **Implement and verify** a validator for (affine) scheduling in polyhedral compilation
- **Apply to** Xavier Leroy et al.'s verified compiler **CompCert [1],** showing its usability
- **Apply and evaluate** the validator with the (affine) scheduler of Uday Bondhugula et al's polyhedral compiler **Pluto [2]**, showing its practicality



Our work: Verified Validation for Polyhedral Scheduling

- **Implement and verify** a validator for (affine) scheduling in polyhedral compilation
- **Apply to** Xavier Leroy et al.'s verified compiler **CompCert [1],** showing its usability
- **Apply and evaluate** the validator with the (affine) scheduler of Uday Bondhugula et al's polyhedral compiler **Pluto [2]**, showing its practicality



Compilation correctness

- For compiler *Comp,* programs $\mathcal{P}_s$ and $\mathcal{P}_t$ that *Comp*($\mathcal{P}_s$) = Some $\mathcal{P}_t$.
- *If* $\mathcal{P}_t$ ***refines*** $\mathcal{P}_s$ (written as $\mathcal{P}_t \sqsubseteq \mathcal{P}_s$), we say this compilation is correct.
  - **It says, from the same beginning state, whenever $\mathcal{P}_t$ terminates at some state, then $\mathcal{P}_s$ is able to stop at the same final state.**

Compilation correctness

- For compiler *Comp,* programs $\mathcal{P}_s$ and $\mathcal{P}_t$ that *Comp*($\mathcal{P}_s$) = Some $\mathcal{P}_t$.
- *If* $\mathcal{P}_t$ ***refines*** $\mathcal{P}_s$ (written as $\mathcal{P}_t \sqsubseteq \mathcal{P}_s$), we say this compilation is correct.
  - **It says, from the same beginning state, whenever $\mathcal{P}_t$ terminates at some state, then $\mathcal{P}_s$ is able to stop at the same final state.**
- Two ways to guarantee correct compilation:
  - Compiler proof: reasoning on *Comp*'s concrete definition to prove

$$\forall \mathcal{P}_s, \mathcal{P}_t.\ \mathit{Comp}(\mathcal{P}_s) = \texttt{Some}\ \mathcal{P}_t \implies \mathcal{P}_t \sqsubseteq \mathcal{P}_s.$$

  - Verified validation: define a separate validator *Validate* and prove

$$\forall \mathcal{P}_s, \mathcal{P}_t.\ \mathit{Validate}(\mathcal{P}_s, \mathcal{P}_t) = \texttt{true} \implies \mathcal{P}_t \sqsubseteq \mathcal{P}_s.$$

    - And run *Validate* after ***each run*** of *Comp*.

Compilation correctness

Why not directly verify the scheduling algorithm?

- **On the one hand, it contains complex heuristic with heavy mathematics. Hard/Impractical/Uneconomical to verify.**
- **On the other hand, it has simple validation algorithm due to its simple correctness criterion: not breaking dependence.**

- Two ways to guarantee correct compilation:
  - Compiler proof: reasoning on *Comp*'s concrete definition to prove

$$\forall \mathcal{P}_s, \mathcal{P}_t.\ \mathit{Comp}(\mathcal{P}_s) = \mathrm{Some}\ \mathcal{P}_t \implies \mathcal{P}_t \sqsubseteq \mathcal{P}_s.$$

  - Verified validation: define a separate validator *Validate* and prove

$$\forall \mathcal{P}_s, \mathcal{P}_t.\ \mathit{Validate}(\mathcal{P}_s, \mathcal{P}_t) = \mathtt{true} \implies \mathcal{P}_t \sqsubseteq \mathcal{P}_s.$$

    - And run *Validate* after ***each run*** of *Comp*.

Implementation and Verification of the Validator

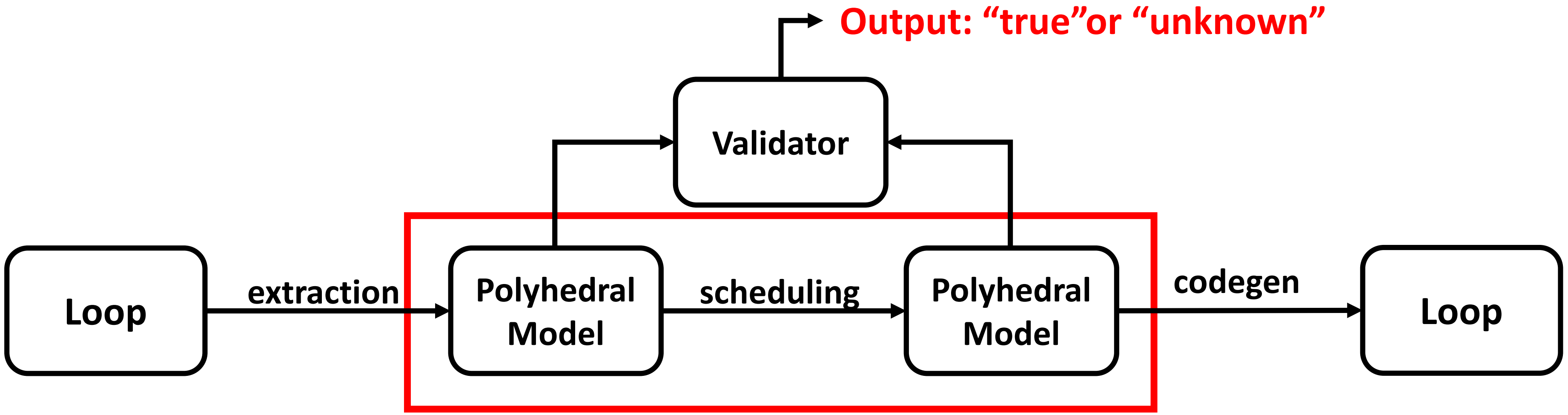
- We define a validation function *Validate* that checks the violation of dependences within the realm of polyhedral model, and mechanize its correctness. All in Coq proof assistant.
- It is parametrized by instruction language to be **reusable**.
- Proof Goal:

Definition (correctness of the validator)

$$
\begin{aligned}
\mathit{Correct}(\mathit{Validate}) &\triangleq \forall \mathcal{P}_s, \mathcal{P}_t. \mathit{Validate}(\mathcal{P}_s, \mathcal{P}_t) = \texttt{true} \\
&\quad \implies \mathcal{P}_t \sqsubseteq \mathcal{P}_s. \\
\mathcal{P}_t \sqsubseteq \mathcal{P}_s &\triangleq \forall \sigma, \sigma'. \\
&\quad\quad \models \mathcal{P}_t, \sigma \Rightarrow \sigma' \implies \models \mathcal{P}_s, \sigma \Rightarrow \sigma'.
\end{aligned}
$$

Our work:
Verified Validation for Polyhedral Scheduling

- **Implement and verify** a validator for (affine) scheduling in polyhedral compilation
- **Apply to** Xavier Leroy et al.'s verified compiler **CompCert [1],** showing its usability
- **Apply and evaluate** the validator with the (affine) scheduler of Uday Bondhugula et al's polyhedral compiler **Pluto [2]**, showing its practicality

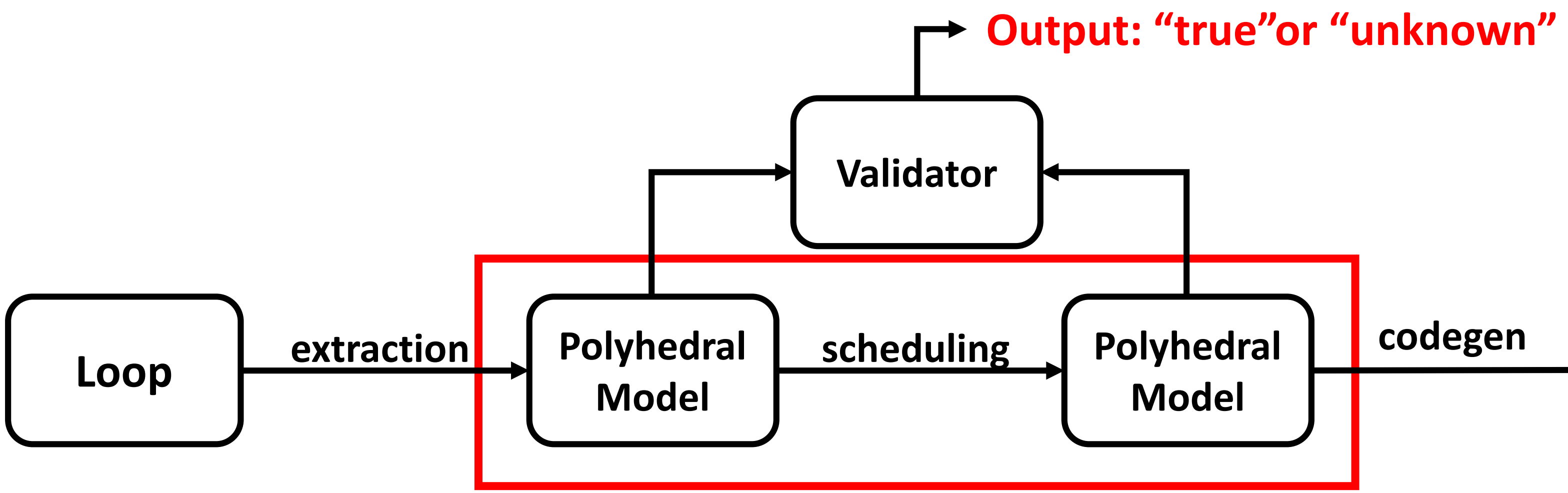


Case study: CompCert

- What is CompCert [1]?
  - A formally verified optimizing C compiler developed by Xavier Leroy et al.
  - Not optimizing enough than industial compilers like Clang and GCC [3].
    - Aggressive optimizations like polyhedral compilation could help!
- We successfully instantiate *Validate* (its implementation and proof) with CompCert's semantics model, showing the possibility towards a fully verified polyhedral extension to CompCert.

Our work:
Verified Validation for Polyhedral Scheduling

- **Implement and verify** a validator for (affine) scheduling in polyhedral compilation
- **Apply to** Xavier Leroy et al.'s verified compiler **CompCert [1],** showing its usability
- **Apply and evaluate** the validator with the (affine) scheduler of Uday Bondhugula et al's polyhedral compiler **Pluto [2]**, showing its practicality



Case study: Pluto

- Loop optimizers like polyhedral-based ones are error prone [4] ! **So formal methods do help.**
- We evaluate on Pluto [2], one of the famous polyhedral compiler.
  - Pluto: ACM SIGPLAN PLDI Most Influential Paper award in 2018

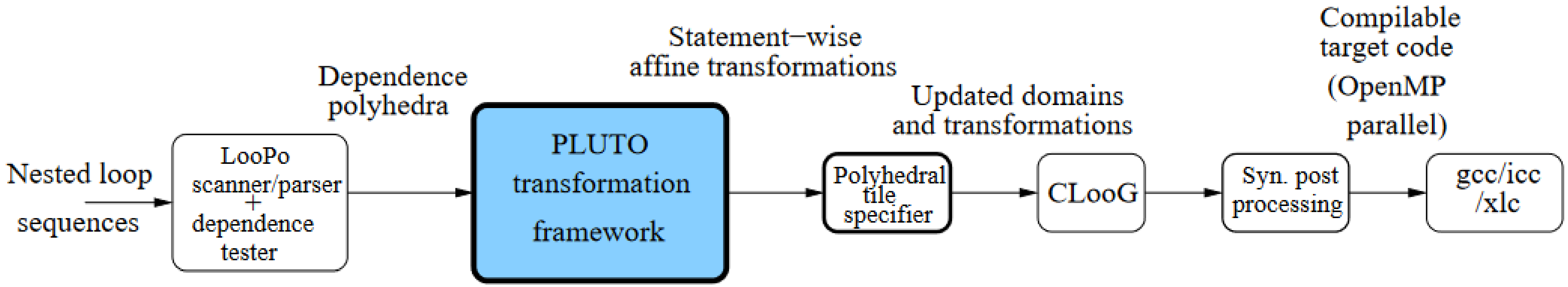
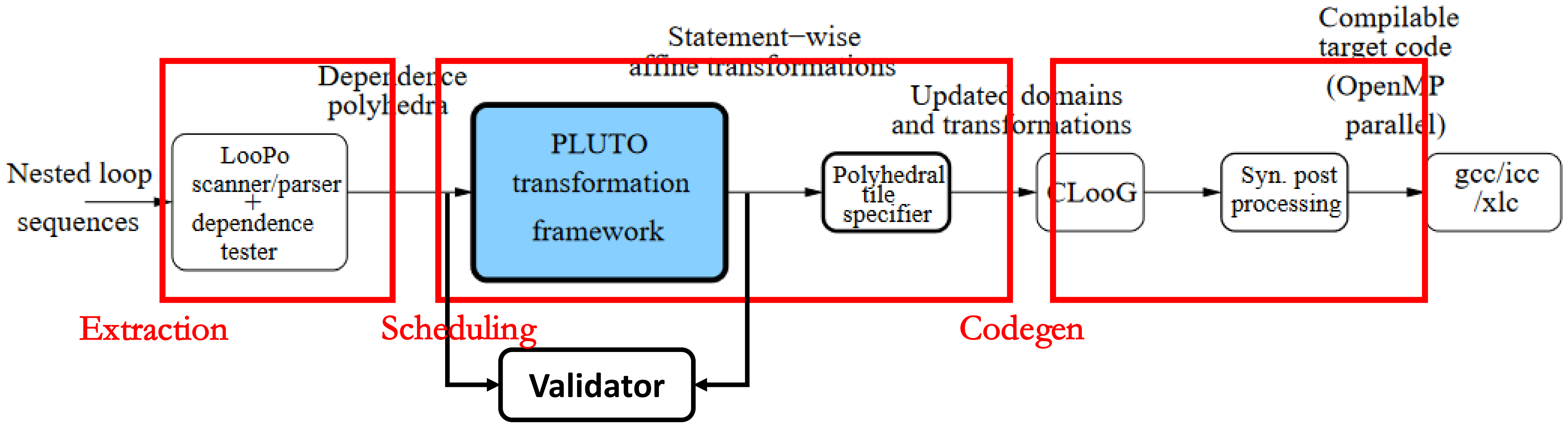


Figure from https://www.csa.iisc.ac.in/~udayb/publications/uday-thesis.pdf, Page 98

Case study: Pluto

- Loop optimizers like polyhedral-based ones are error prone [4] ! **So formal methods do help.**
- We evaluate on Pluto [2], one of the famous polyhedral compiler.
  - Pluto: ACM SIGPLAN PLDI Most Influential Paper award in 2018



Case study: Pluto

- Result shows the validator works well with Pluto, successfully verify the affine scheduling of 62 test cases from Pluto's repository [5]
  - Overhead is reasonable
  - “unknown” is not reported
- Not only an academic prototype

Case st

- Result sh ... ly verify the affin ... y [5]
  - Overh
  - “unkn
- Not only

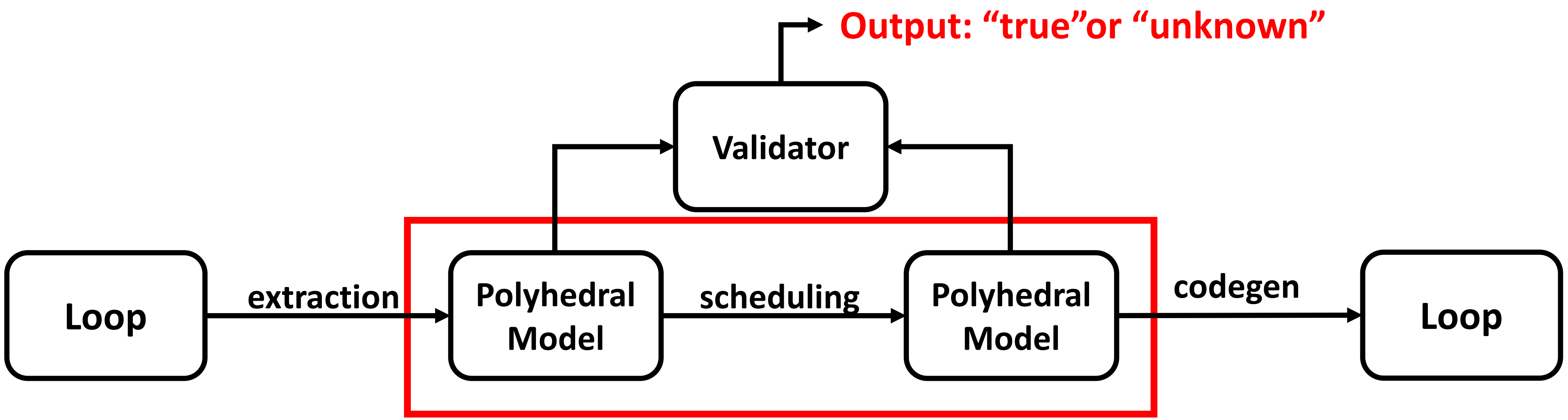
Test	Time of Pluto (ms)	Time of Validation (ms,ms)	Result
covcol	3.5	434.6, 320.7	EQ
dsyr2k	2.6	106.0, 83.4	EQ
fdtd-2d	46.4	1615.5, 1296.3	EQ
gemver	7.0	247.9, 240.4	EQ
lu	6.1	410.6, 331.2	EQ
mvt	2.2	70.2, 56.3	EQ
ssymm	40.7	726.0, 551.2	EQ
tce	568.6	4442.0, 4422.5	EQ
adi	77.5	2531.7, 2377.8	EQ
corcol	5.5	442.5, 362.1	EQ
dct	21.8	879.4, 739.4	EQ
dsyrk	1.8	96.8, 78.9	EQ
floyd	12.1	502.6, 421.7	EQ
jacobi-1d-imper	3.8	184.0, 167.8	EQ
matmul-init	2.9	257.8, 192.4	EQ
pca	202.5	2923.6, 2679.5	EQ
strmm	1.9	141.4, 110.8	EQ
tmm	1.6	109.7, 89.6	EQ
advect3d	1023.1	579.1, 498.1	EQ
corcol3	13.6	851.3, 733.4	EQ
doitgen	10.4	1069.2, 837.4	EQ
fdtd-1d	6.0	268.7, 229.9	EQ
jacobi-2d-imper	17.7	619.5, 543.5	EQ
matmul	3.2	157.1, 125.5	EQ
seidel	24.5	818.1, 725.5	EQ
strsm	6.4	209.3, 161.2	EQ
trisolv	5.1	338.9, 248.8	EQ
1dloop-invar	0.3	6.7, 6.0	EQ
costfunc	0.8	47.4, 35.0	EQ
fusion1	0.9	15.3, 13.9	EQ
...	...	...	...

Table 1: Evaluation results on Pluto's test suits

Thank you!

Our work:
Verified Validation for Polyhedral Scheduling

- **Implement and verify** a validator for (affine) scheduling in polyhedral compilation
- **Apply to** Xavier Leroy et al.'s verified compiler **CompCert [1],** showing its usability
- **Apply and evaluate** the validator with the (affine) scheduler of Uday Bondhugula et al's polyhedral compiler **Pluto [2]**, showing its practicality



Open source at https://github.com/verif-scop/PolCert/

Reference

[1]. Xavier Leroy. 2009. Formal verification of a realistic compiler. Commun. ACM 52, 7 (July 2009), 107–115.

[2]. Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08). Association for Computing Machinery, New York, NY, USA, 101–113.

[3]. Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux, and Alexandre Bérard. 2023. Formally Verifying Optimizations with Block Simulations. Proc. ACM Program. Lang. 7, OOPSLA2, Article 224 (October 2023), 30 pages.

[4]. Vsevolod Livinskii, Dmitry Babokin, and John Regehr. 2023. Fuzzing Loop Optimizations in Compilers for C++ and Data-Parallel Languages. Proc. ACM Program. Lang. 7, PLDI, Article 181 (June 2023), 22 pages.

[5]. Uday Bondhugula. https://github.com/bondhugula/pluto/.

Original code π_{cov} for covariance matrix calculation, 1.84s

```
for (j1 = 1; j1 <= M; j1++) {
    for (j2 = j1; j2 <= M; j2++) {
        for (i = 1; i <= N; i++) {
            I0: symmat[j1][j2] += data[i][j1] * data[i][j2];
        }
        I1: symmat[j2][j1] = symmat[j1][j2];
    }
}
```

Optimized code π'_{cov} for covariance matrix calculation, 0.43s, with loop distribution and loop interchange

```
for (i = 1; i <= N; i++) {
  for (j1 = 1; j1 <= M; j1++) {
    for (j2 = j1; j2 <= M; j2++) {
        symmat[j1][j2] += data[i][j1] * data[i][j2];
    }
  }
}
for (j1 = 1; j1 <= M; j1++) {
  for (j2 = j1; j2 <= M; j2++) {
    symmat[j2][j1] = symmat[j1][j2];
  }
}
```

M=N=1500

See at https://github.com/verif-scop/speed-up.

Polyhedral compilation does high-level structural tranformations and only impose a few properties of the underlying instruction language (called $\mathbb{I}$). The validation function given in this work is parameterized by $\mathbb{I}$.

$\mathbb{I}$ allows user define the syntax, types, state, semantics of the language, how it initializes, and its the non-alias proposition. It demands user to provide a verified *Checker* function to validate the consistency between the read and write access function and an instruction's semantics, and prove that any two instances that satisfy Bernstein's conditions are permutable.

We assume Pluto's instruction language satisfy this abstraction.

Module Type $\mathbb{I} \triangleq$

$$
\left\{
\begin{array}{l}
\mathsf{T}, \mathsf{I}, \mathsf{S} : \mathit{Type} \\
\quad \models \; : \mathsf{I} \to \mathit{List}(\mathbb{Z}) \quad \to \mathit{Memory\ Cells} \\
\qquad\qquad\qquad\qquad\quad \to \mathit{Memory\ Cells} \to \mathsf{S} \to \mathsf{S} \to \mathit{Prop} \\
\mathsf{Compat} : \mathit{List}(\mathit{Identifier}) \to \mathsf{S} \to \mathit{Prop} \\
\mathsf{Consistent} : \mathit{List}(\mathit{Identifier} \times \mathsf{T}) \to \mathit{List}(\mathbb{Z}) \to \mathsf{S} \to \mathit{Prop} \\
\mathsf{NonAlias} : \mathsf{S} \to \mathit{Prop} \\
\mathsf{NonAliasPsrv} : \\
\qquad \forall \mathrm{I}, \sigma, \sigma'. \ \mathsf{NonAlias}(\sigma) \wedge \boldsymbol{p} \models \mathrm{I}, \sigma \xrightarrow{\_,\_} \sigma' \implies \mathsf{NonAlias}(\sigma'). \\
\quad \mathit{Checker} : \mathsf{I} \quad \to \mathit{Access\ Functions} \\
\qquad\qquad\qquad\ \to \mathit{Access\ Functions} \to \mathit{Bool} \\
\mathit{Correct}(\mathit{Checker}) : \\
\qquad \forall \mathrm{I}, \mathcal{W}, \mathcal{R}. \mathit{Checker}(\mathrm{I}, \mathcal{W}, \mathcal{R}) = \mathsf{true} \\
\qquad\qquad \implies (\forall \sigma, \sigma', \boldsymbol{p}, \Delta_r, \Delta_w. \ \boldsymbol{p} \models \mathrm{I}, \sigma \xrightarrow{\Delta_r, \Delta_w} \sigma' \implies \Delta_r \subseteq \mathcal{R}(\boldsymbol{p}) \wedge \Delta_w \subseteq \mathcal{W}(\boldsymbol{p})). \\
\mathsf{BCPermut} : \\
\qquad \forall \mathrm{I}_1, \mathrm{I}_2, \boldsymbol{p}_1, \boldsymbol{p}_2, \sigma, \sigma', \sigma'', \Delta_r, \Delta_w, \Delta'_r, \Delta'_w. \\
\qquad (\boldsymbol{p}_1 \models \mathrm{I}_1, \sigma \xrightarrow{\Delta_r, \Delta_w} \sigma' \wedge \boldsymbol{p}_2 \models \mathrm{I}_2, \sigma' \xrightarrow{\Delta'_r, \Delta'_w} \sigma'' \\
\qquad\qquad \wedge \Delta_r \cap \Delta'_w = \emptyset \wedge \Delta_w \cap \Delta'_r = \emptyset \wedge \Delta_w \cap \Delta'_w = \emptyset) \\
\qquad\qquad \implies \exists \sigma^*. \ \boldsymbol{p}_2 \models \mathrm{I}_2, \sigma \xrightarrow{\Delta'_r, \Delta'_w} \sigma^* \wedge \boldsymbol{p}_1 \models \mathrm{I}_1, \sigma^* \xrightarrow{\Delta_r, \Delta_w} \sigma''.
\end{array}
\right.
$$

Figure 1: Definition of Instruction Language Module $\mathbb{I}$

Case study: Towards verified polyhedral compilation for CompCert

We instantiate the validation function with CompCert C's type, state and subset of its instruction language, and implement *Checker* with symbolic execution. All verified. Only differences are, affine expression is evaluated in $\mathbb{Z}$ (no overflow), and multi-dimensional array access is sugarized.

$$
\begin{array}{rccl}
\textit{(Base Type)} & \tau_\perp & ::= & \texttt{int32s} \\
\textit{(Type)} & \tau & \in & \textit{Base Type} \times \textit{List}(\mathbb{Z}) \\
\textit{(Value)} & v & ::= & \texttt{I32}(n) \mid \dots \\
\textit{(Iterator)} & i & \in & \mathbb{N} \\
\textit{(Unop)} & op_1 & ::= & \texttt{-} \mid \dots \\
\textit{(Binop)} & op_2 & ::= & \texttt{+} \mid \texttt{*} \mid \dots \\
\textit{(May Affine Expression)} & \varepsilon & ::= & z \mid i \mid op_1\ \varepsilon \mid \varepsilon_1\ op_2\ \varepsilon_2 \\
\textit{(Access Expression)} & \epsilon & \in & \begin{array}{l} \quad \textit{Identifier} \quad \times \textit{Base Type} \\ \qquad\qquad\qquad \times \textit{List(May Affine Expression)} \end{array} \\
\textit{(Expression)} & e & ::= & v \mid i \mid \epsilon \mid op_1\ e \mid e_1\ op_2\ e_2 \\
\textit{(Base Instruction)} & \mathtt{I} & ::= & \texttt{skip} \mid \epsilon\texttt{:=}e
\end{array}
$$

Future work

- Complete verified polyhedral compilation.
  - Verified extractor.
  - Engineering in CompCert's driver & frontend.
  - Apply optimistic approach[8] to deal with polyhedral model's heavy assumptions, like integer overflow[9].
- Support validation for other polyhedral transformations, like index set split (as a pre-phrase), tiling (as a post-phrase), layout transformation (as an orthogonal phrase).
- Support vectorization, parallelization, GPU compilation …

[8] https://dl.acm.org/doi/10.5555/3049832.3049864

[9] https://inria.hal.science/hal-00655485