Verified Validation for Polyhedral Scheduling

Xuyang Li, Hongjin Liang, Xinyu Feng

Nanjing University

Background

- Human needs highly optimized compilation techniques for modern software.
- **Nested loops** are optimization targets due to its heavy numerical computation.
- Common loop optimizations:
  - Apply combination of loop fusion, distribution, interchange, skewing, reverse, tiling, …, to improve memory locality and/or parallelizability.
    - May further apply vectorization (SIMD) and parallelization (openmp primitives).
  - Map it automatically to domain-specific hardware like GPU.

Background

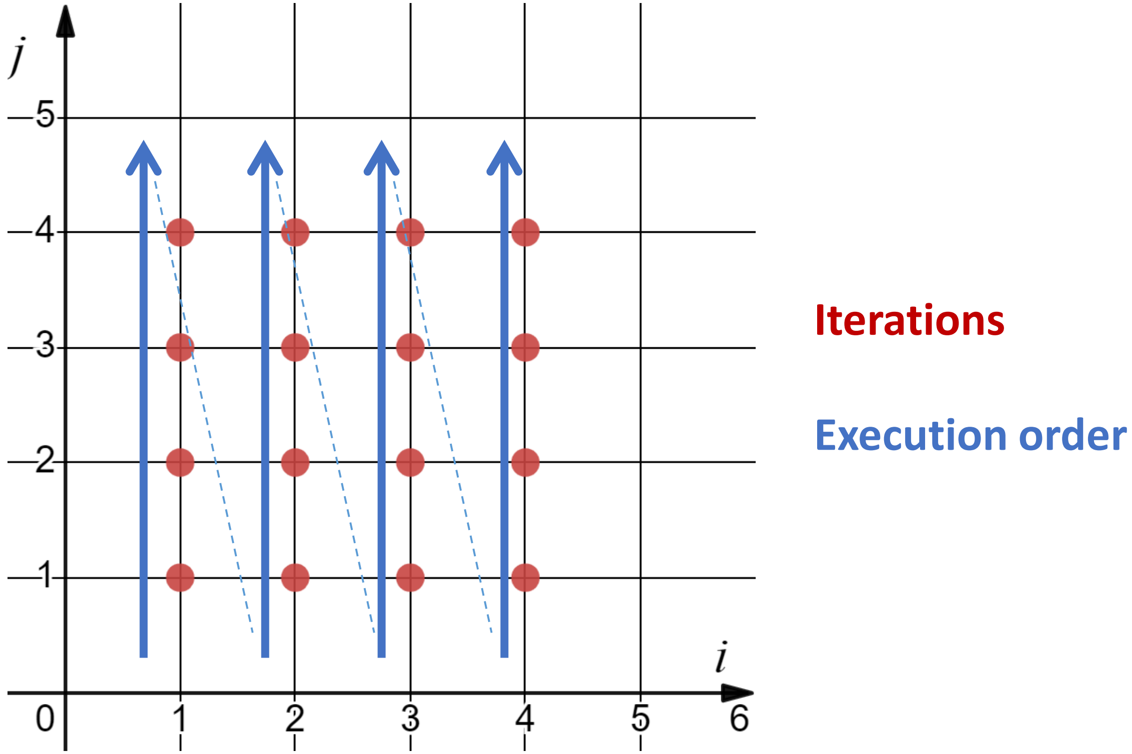
- To do such loop transformations, we need to analyze dependences between instructions of different iterations.

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

Can we parallelize it?

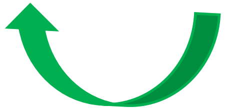
Example

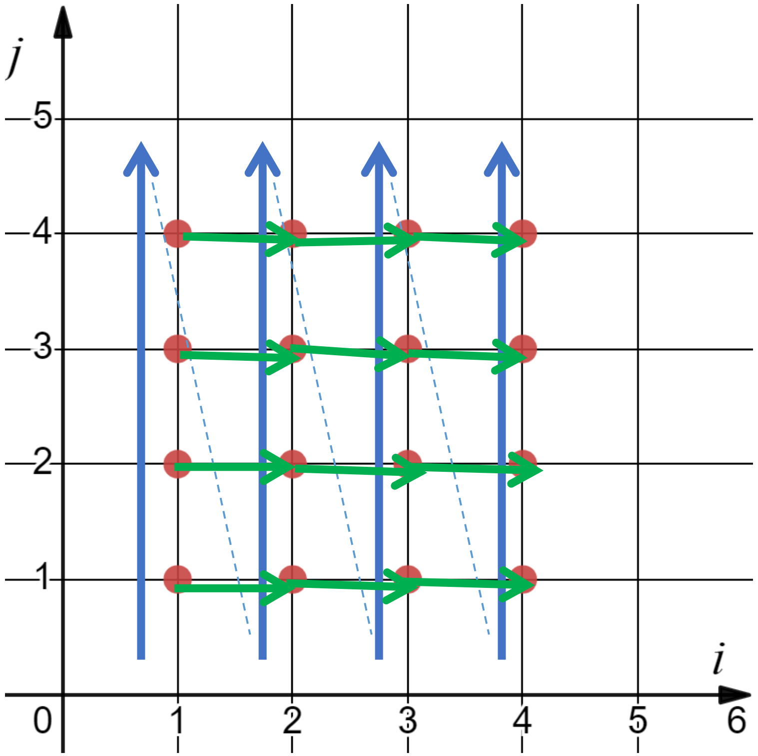
```
for i in [1, N]:
   for j in [1, N]:
S:   A[i][j] = A[i+1][j] + A[i-1][j-1]
```



Example

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





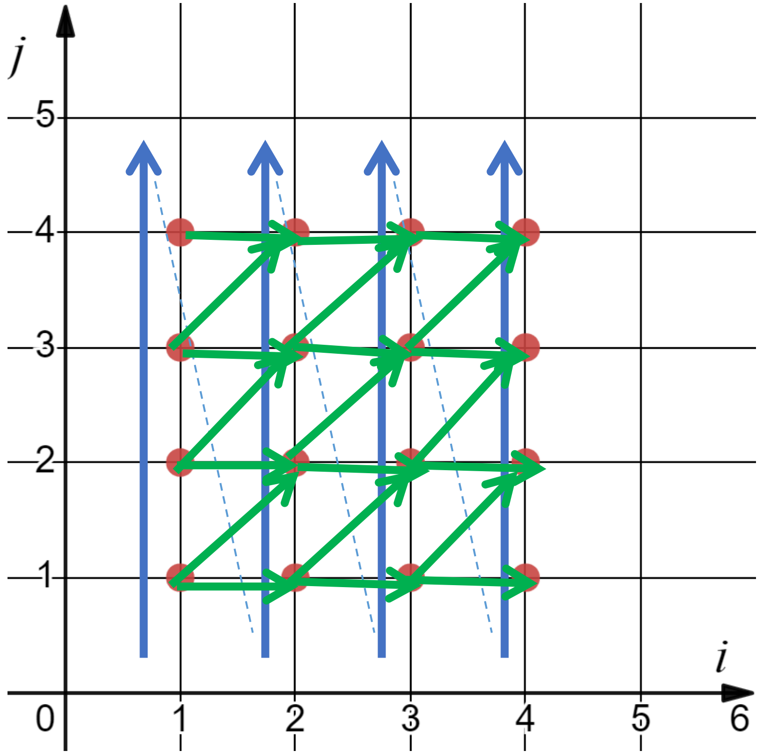
Iterations

Execution order

Dependence

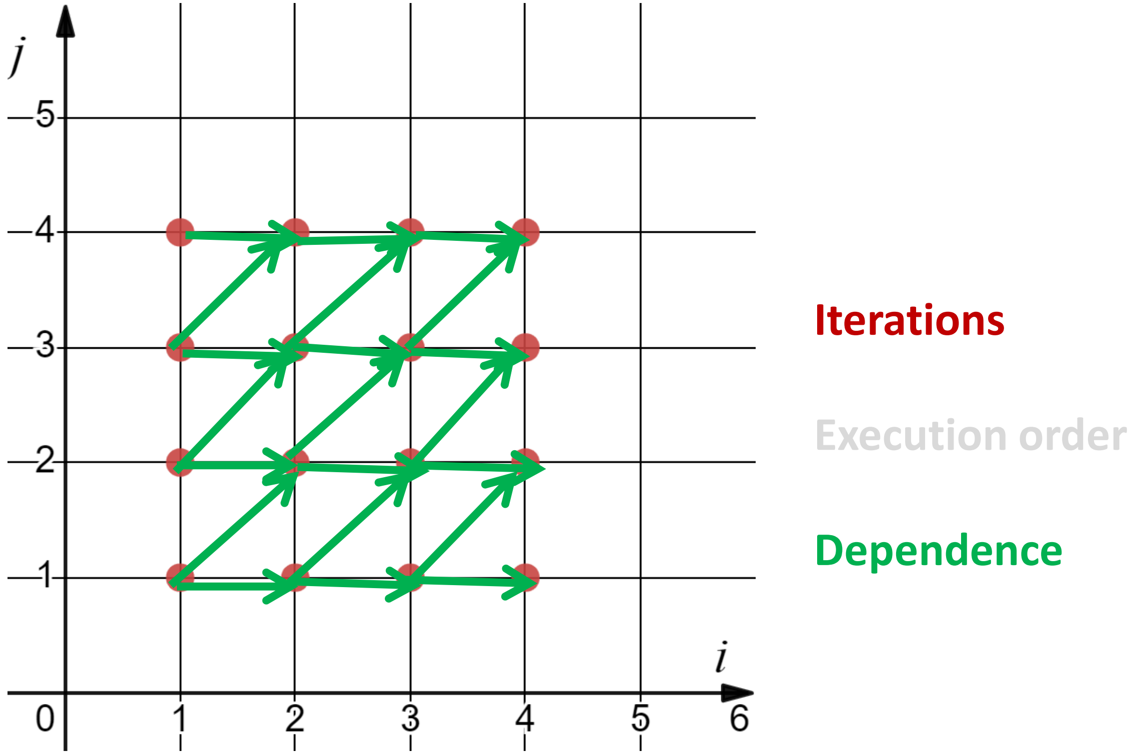
Example

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

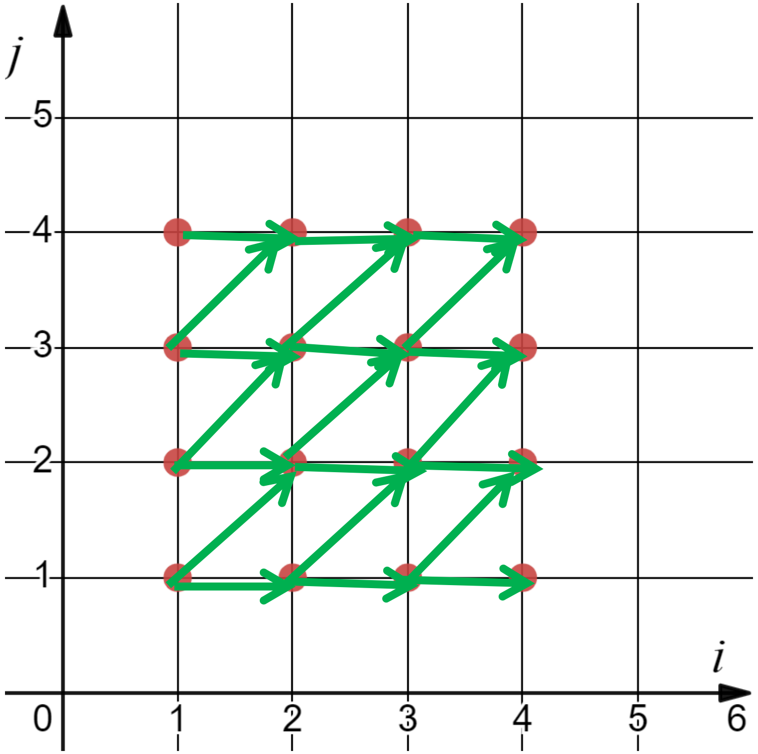


Example

Can we parallelize it?

Respecting dependences?

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



Iterations

Execution order

Dependence

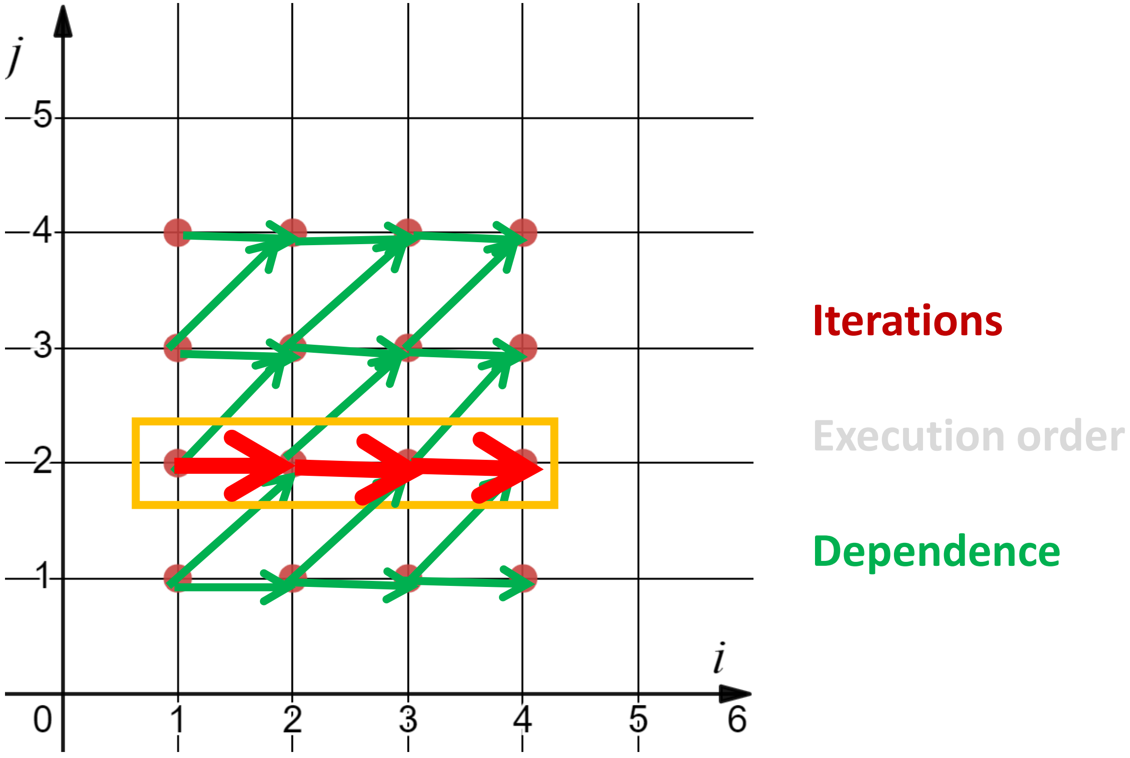
Example

Can we parallelize it?

Respecting dependences?

- Not along j. There is inner dependence.
- What about other axis?

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



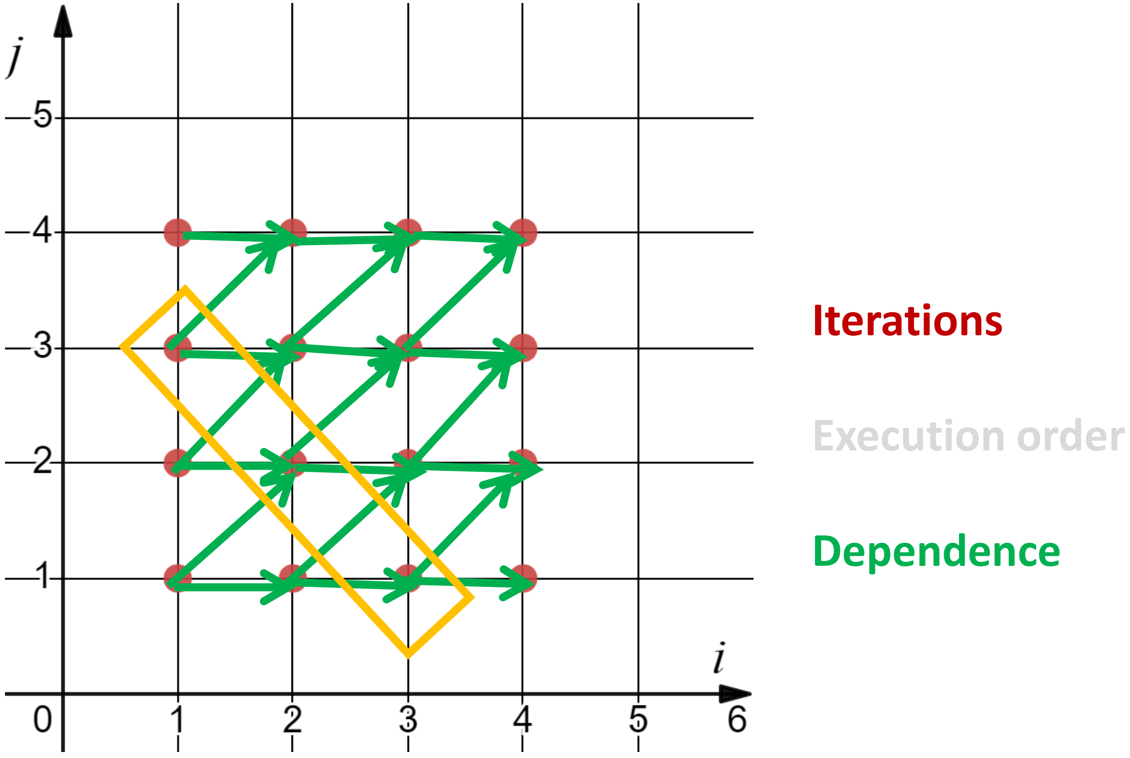
Example

Can we parallelize it?

Respecting dependences?

- Not along j. There is inner dependence.
- What about other axis?
  - Iteration with same i + j is parallelizable

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

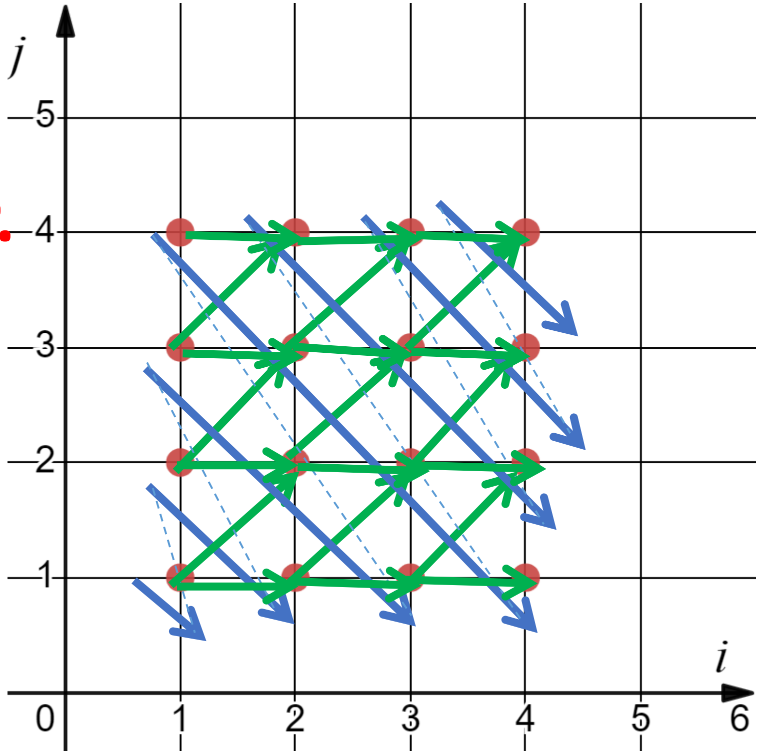


Example

Now we may find new execution order.

(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]
```



Iterations

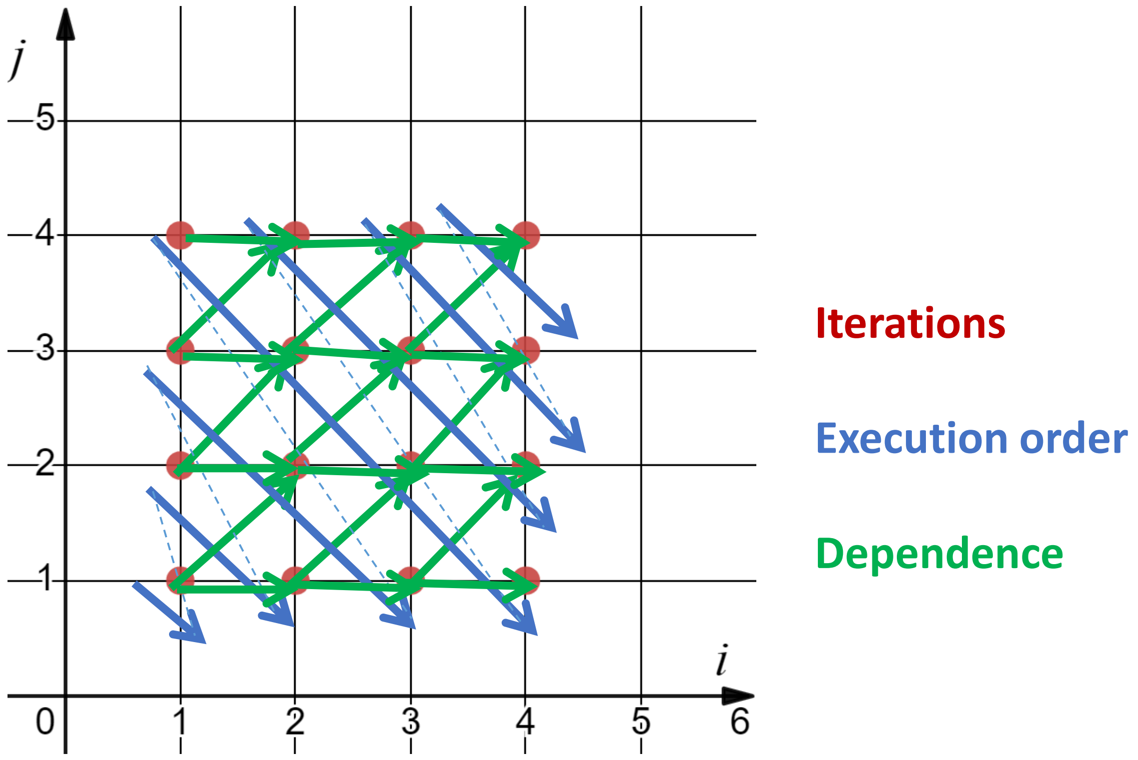
Execution order

Dependence

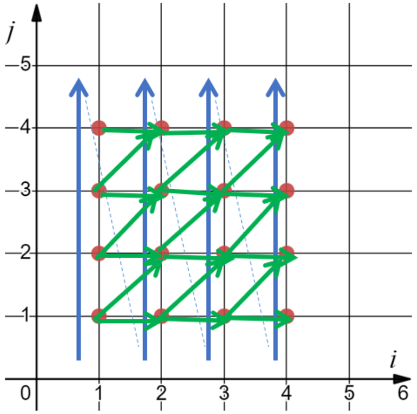
Example

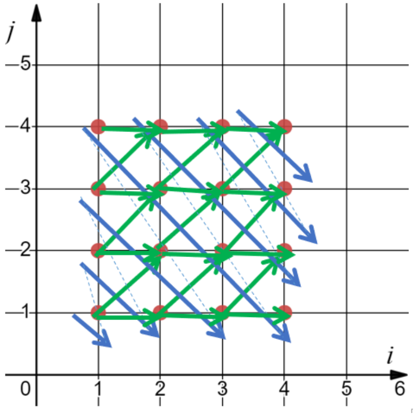
Now we generate a new nested loop respecting new execution order.

```
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



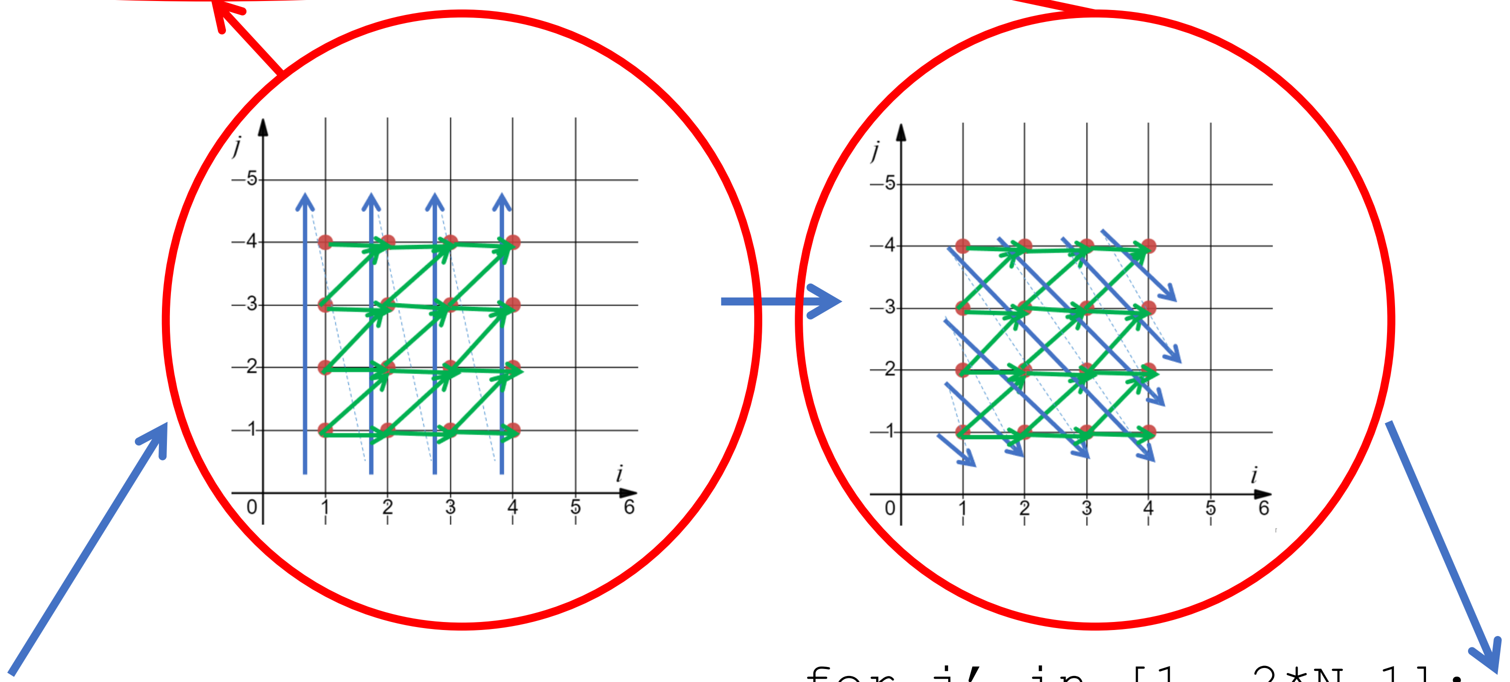


```
for i in [1, N]:
  for j in [1, N]:
S:  A[i][j] = A[i+1][j]
            + A[i-1][j-1]
```

```
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]
```

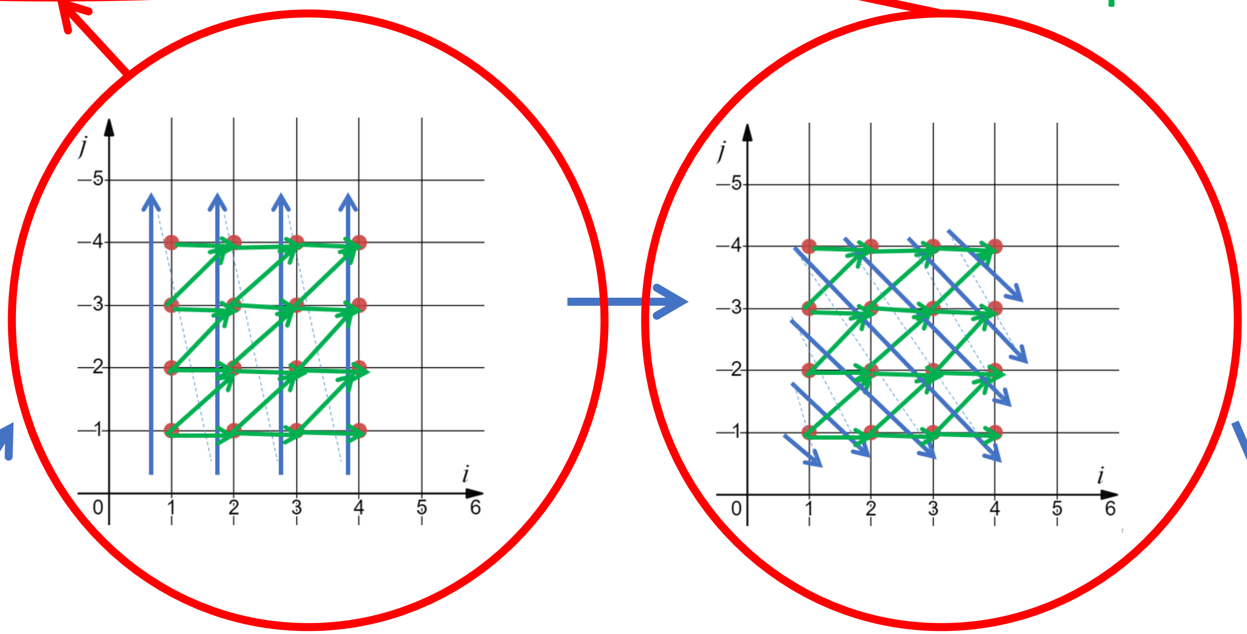
loop skewing + loop interchange

Polyhedral Model

Iterations – Domain – “The 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]
```

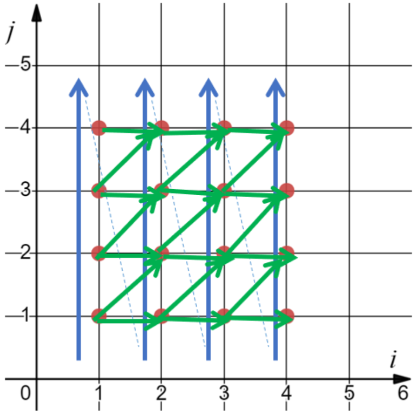
```
for j’ in [1, 2*N-1]:
  for i’ in [max(1, j’-N+1),
             min(N-1, j’-1)]:
S:  A[i’][(j’-i’)] = A[i’+1][(j’-i’)]
                   + A[i’-1][(j’-i’)-1]
```

loop skewing + loop interchange

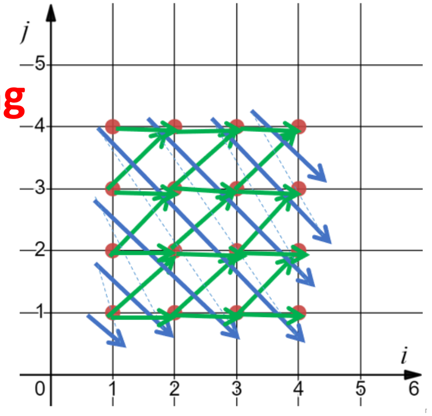
Polyhedral Compilation

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

extraction



scheduling



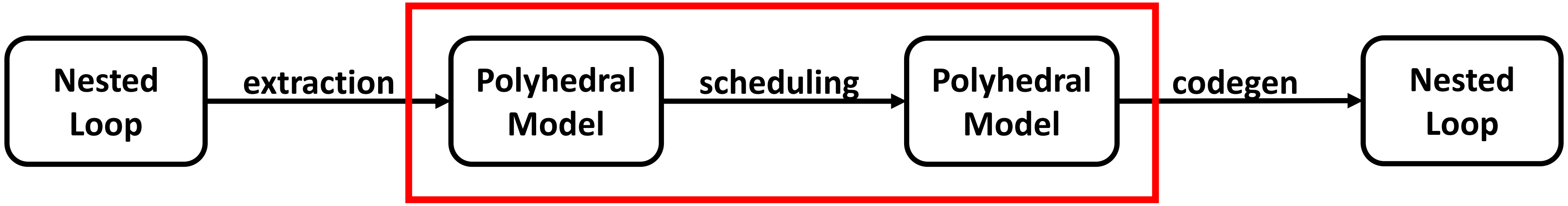
codegen

```
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

Nested Loop → extraction → **Polyhedral Model** → scheduling → **Polyhedral Model** → codegen → **Nested Loop**

Polyhedral Compilation



Our focus!
Scheduling can be done manually or by automatic algorithms, like Pluto[1].
We want to *ensure its correctness*.

---

[1] https://pluto-compiler.sourceforge.net/

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

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

  - 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*.

Compilation correctness

Why not directly verify the scheduling algorithm?

- **On the one hand, it contains complex heuristic with heavy mathematics. Hard/Impractical to verify.**
- **On the other hand, it has simple validation algorithm due to its simple correctness criterion: not breaking dependence.** (Also called *Bernstein's Condition*[2].)

- 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

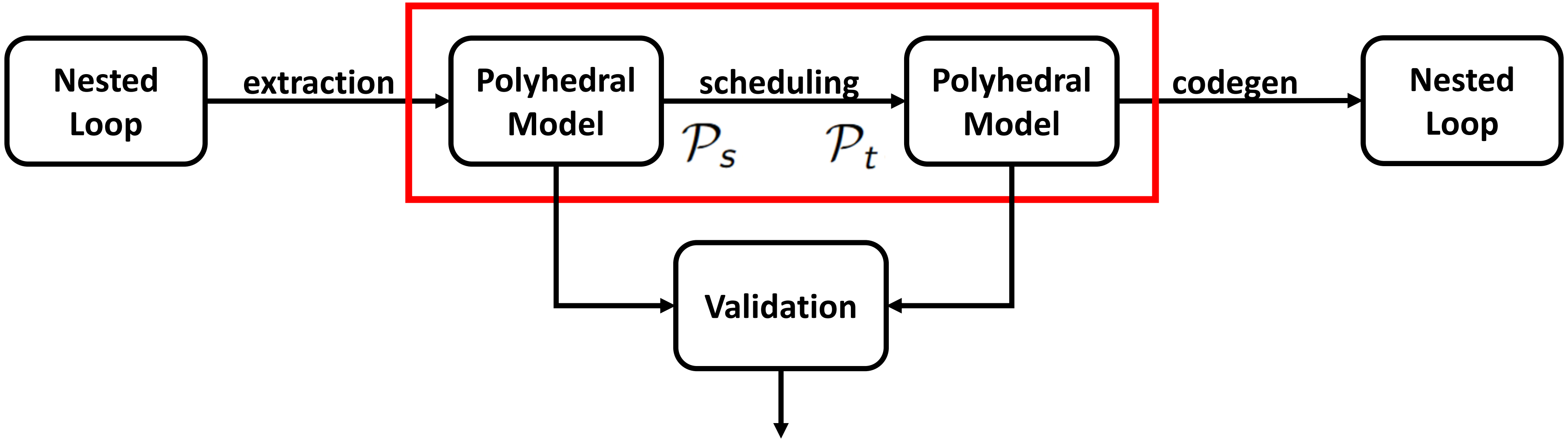
$$\forall \mathcal{P}_s, \mathcal{P}_t.\ \mathit{Validate}(\mathcal{P}_s, \mathcal{P}_t) = \texttt{true} \implies \mathcal{P}_t \sqsubseteq \mathcal{P}_s.$$

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

---

[2] https://link.springer.com/referenceworkentry/10.1007/978-0-387-09766-4_521

Validating Polyhedral Scheduling


Nested Loop

extraction

Polyhedral Model

scheduling

$\mathcal{P}_s$

$\mathcal{P}_t$

Polyhedral Model

codegen

Nested Loop

Validation

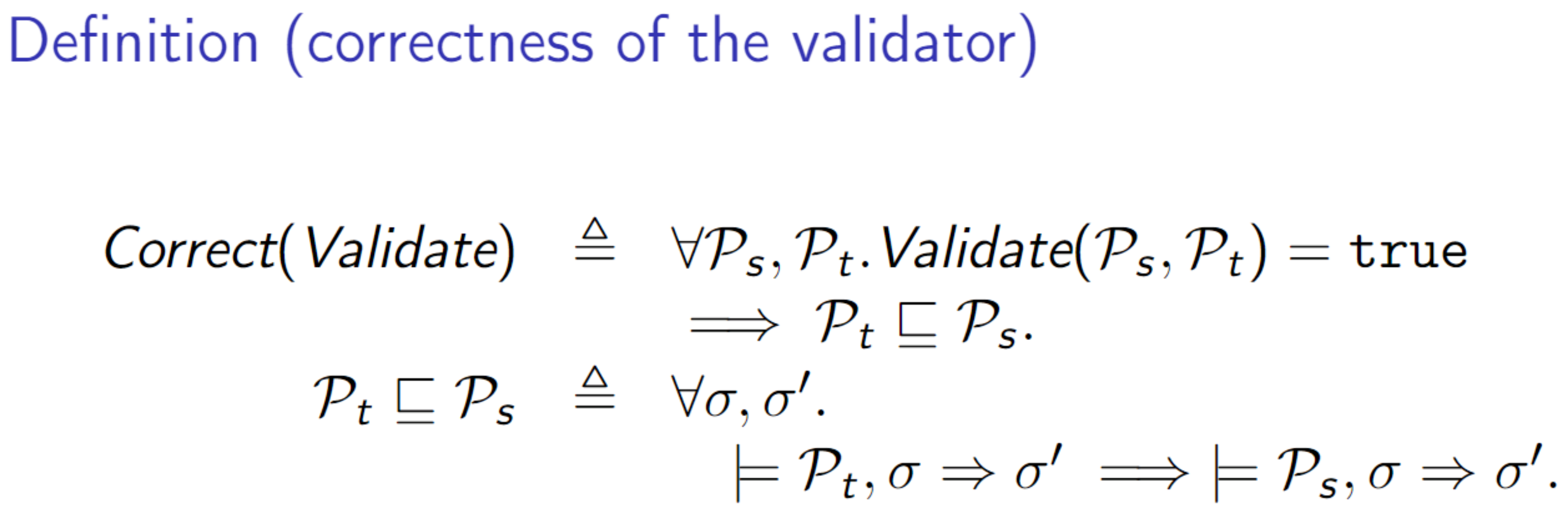
Output: true or false(unknown).

Our work:
Verified Validation for Polyhedral Scheduling

- We implement and verify a general validation function *Validate* for polyhedral scheduling in Coq proof assistant from scratch. It is parameterized by instruction language.
- We instantiate *Validate* with a variant of CompCert's instruction language, showing its practicality.
- We adapt *Validate* so that it works with the Pluto compiler. It successfully validates all its available test cases with reasonable overhead.

General Verified Validation function for Polyhedral Scheduling

- We define a validation function *Validate* that checks **Bernstein's condition** within polyhedral model, and mechanized its correctness with the Coq proof assistant.
- It is parametrized by instruction language to be reusable.
- Our top lemma



Definition (correctness of the validator)

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

Case study on CompCert

ACM
Software
System
Award
2021

ACM
SIGPLAN
Programming
Languages
Software
Award
2022

- What is CompCert[3]?
  - a high-assurance compiler for almost all of the C language (ISO C 2011), generating efficient code for the ARM, PowerPC, RISC-V and x86 processors.
  - It is not enough optimized[4] than production compiler like clang and gcc. **So aggressive optimizers like polyhedral-based ones do help!**
- We successfully instantiate *Validate* with CompCert's semantics model, showing the possibility towards a fully verified polyhedral extension to CompCert.

---

[3] https://compcert.org/

[4] https://doi.org/10.1145/3622799

Case study on Pluto compiler

- Loop optimizers like polyhedral-based ones are error prone[5] indeed! **So formal methods do help.**
- We evaluate on Pluto, one of the famous polyhedral compiler.
  - Pluto: ACM SIGPLAN PLDI Most Influential Paper award in 2018

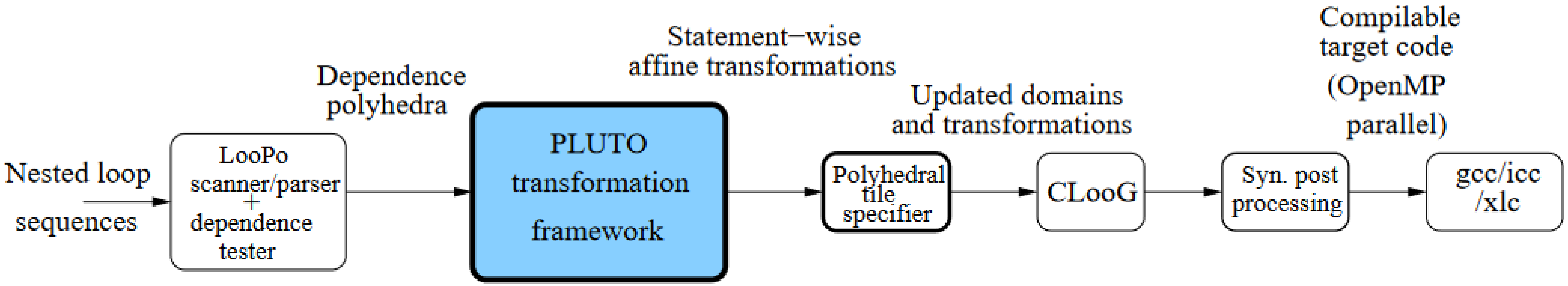


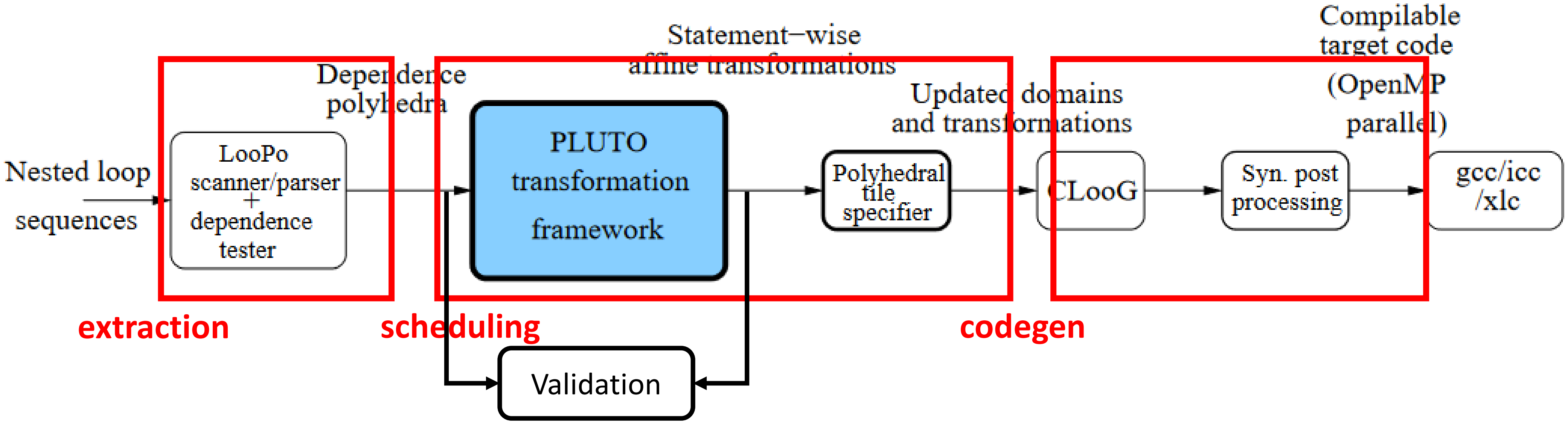
Figure from https://www.csa.iisc.ac.in/~udayb/publications/uday-thesis.pdf, Page 98

---

[5] https://doi.org/10.1145/3591295

Case study on Pluto compiler

- Loop optimizers like polyhedral-based ones are error prone[5] indeed! **So formal methods do help.**
- We evaluate on Pluto, one of the famous polyhedral compiler.
  - Pluto: ACM SIGPLAN PLDI Most Influential Paper award in 2018



---

[5] https://doi.org/10.1145/3591295

Case study on Pluto compiler

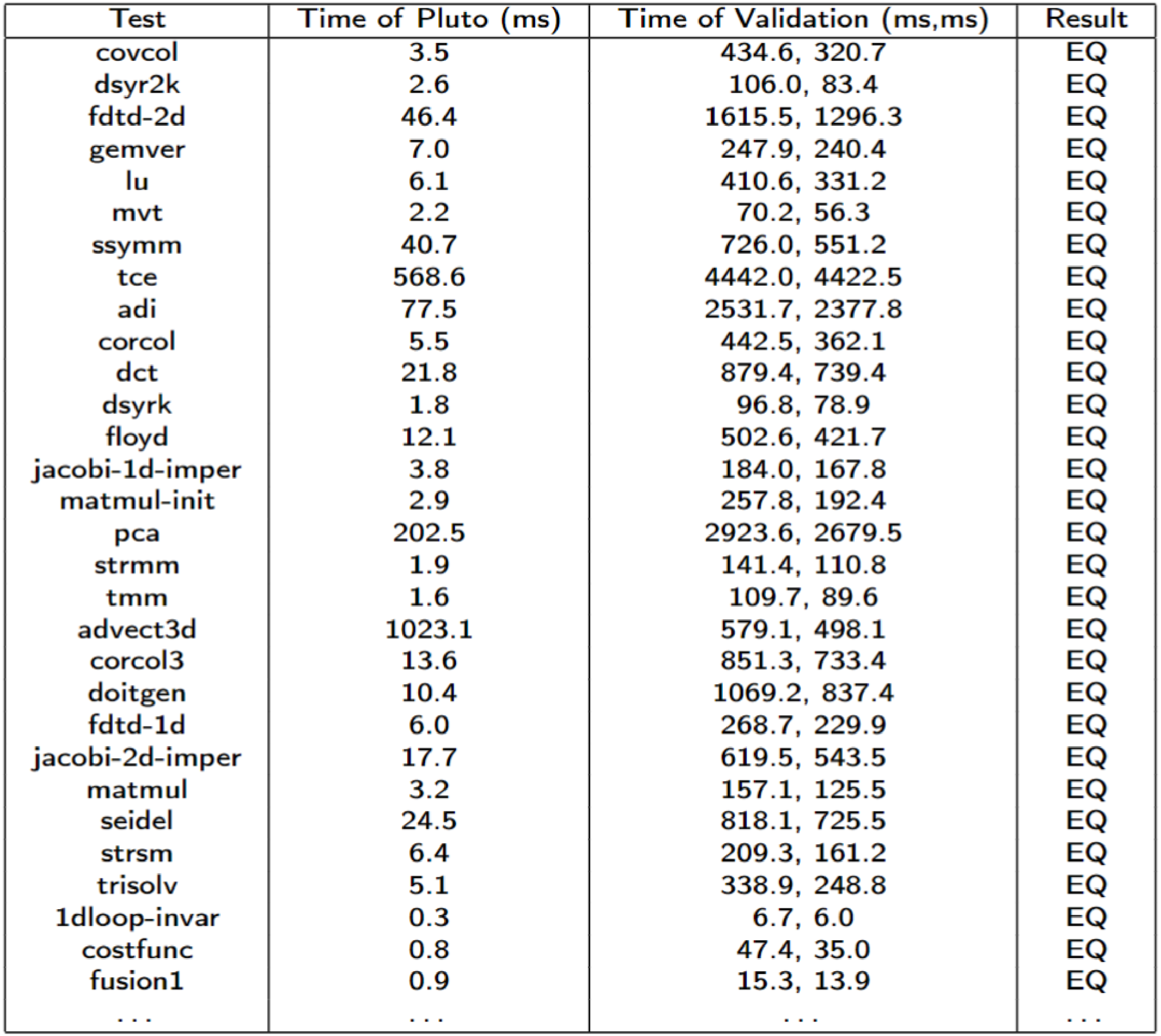
- Loop optimizers like polyhedral-based ones are error prone[5] indeed!
  So formal methods do help.
- We evaluate on Pluto, one of the famous polyhedral compiler.
  - Pluto: ACM SIGPLAN PLDI Most Influential Paper award in 2018
- Work and work well with Pluto compiler: successfully validate its all 62 available test cases.
  - Not just a research prototype: work as an executable on practical polyhedral compiler.
  - Soundness v.s. completeness:
    - Formal verification proves the algorithm ***soundness***: guarantee ***refinement*** whenever it outputs ***true***.
    - The evaluation shows validation's algorithm ***completeness****:* if refinement establishes on inputs*, the algorithm* tries its best to output ***true*** rather than ***unknown***!
  - Also, the algorithm has reasonable overhead.

---

[5] https://doi.org/10.1145/3591295

Case st

- Loop opti ed!
  So formal
- We evalu
  - Pluto: /
- Work and all 62 available
  - Not jus dral compil
  - Soundr
    - For enever it out
    - The ishes on inputs, *the*
  - Also, th



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

Coq development

- Around 18000 lines of Coq, 1000 lines of OCaml. Open source at https://github.com/verif-scop/PolCert.
- Our work bases on Verified Polyhedron Library[6] and is syntactically compatible to PolyGen[7] (Verified Polyhedral Code Generation) in POPL'21.

---

[6] https://ieeexplore.ieee.org/document/8750763

[7] https://dl.acm.org/doi/10.1145/3434321

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 …

---

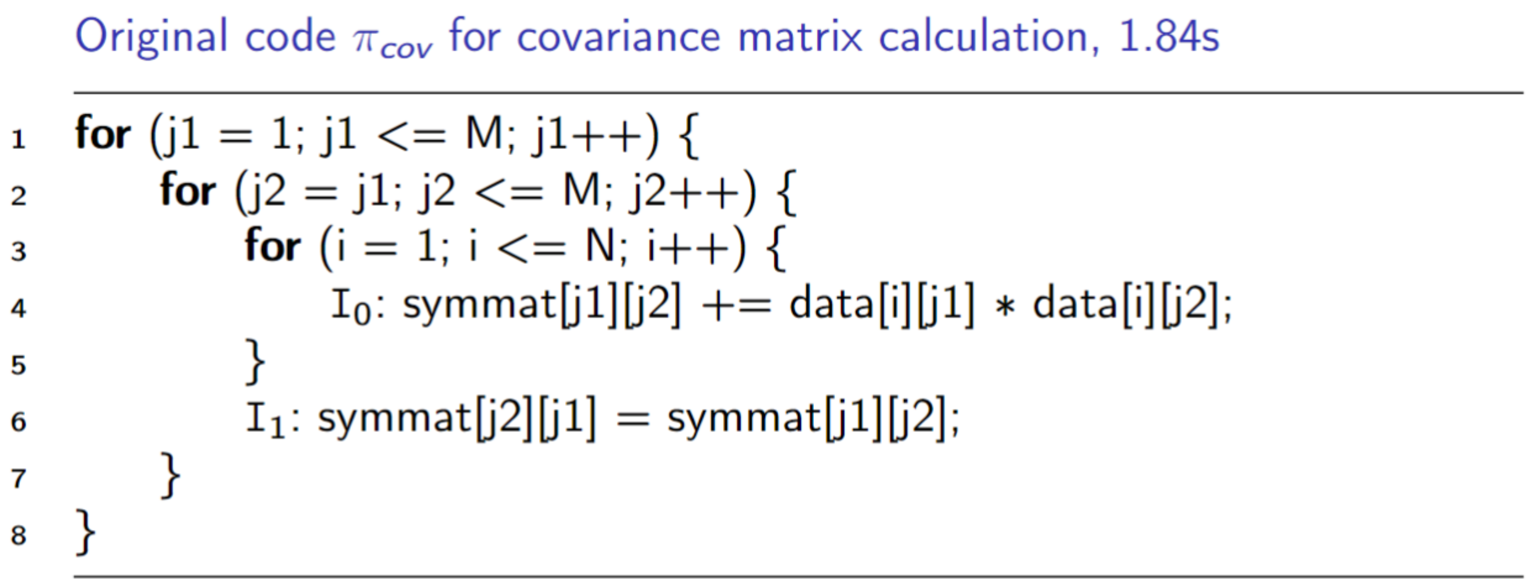
[8] https://dl.acm.org/doi/10.5555/3049832.3049864

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

Q&A

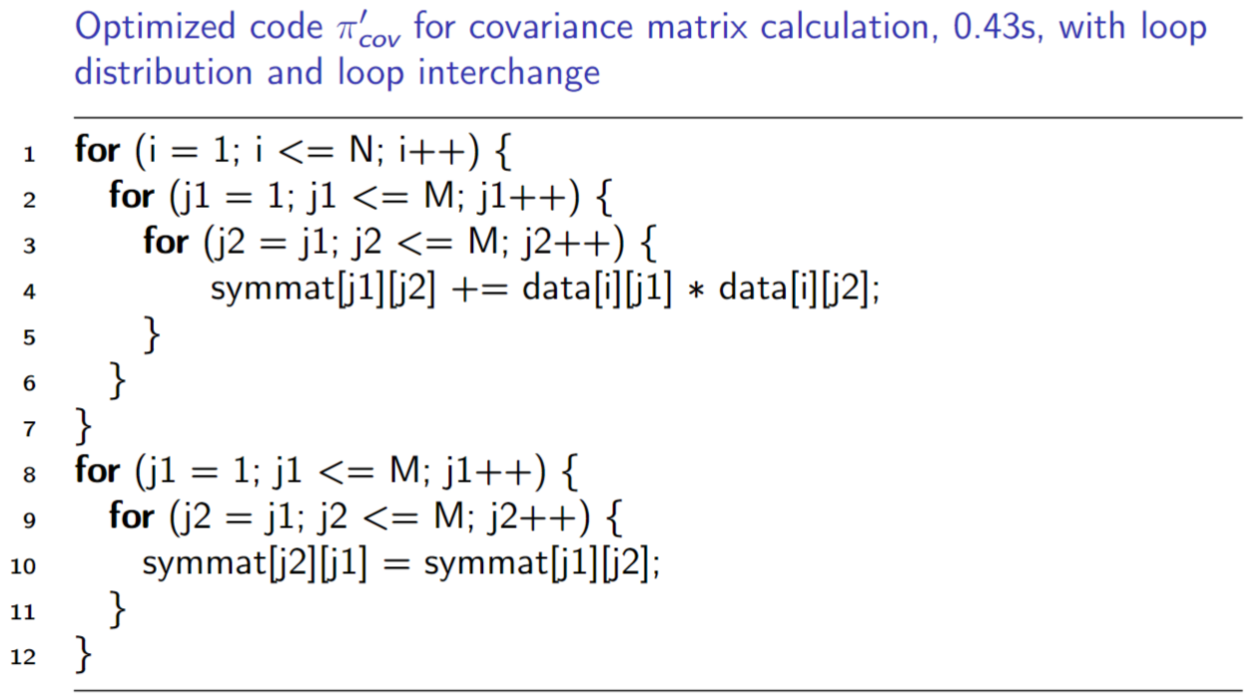
Our work:
Verified Validation for Polyhedral Scheduling

- We implement and verify a general validation function *Validate* for polyhedral scheduling in Coq proof assistant from scratch. It is parameterized by instruction language.
- We instantiate *Validate* with a variant of CompCert's instruction language, showing its practicality.
- We adapt *Validate* so that it works with the Pluto compiler. It successfully validates all its available test cases with reasonable overhead.



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

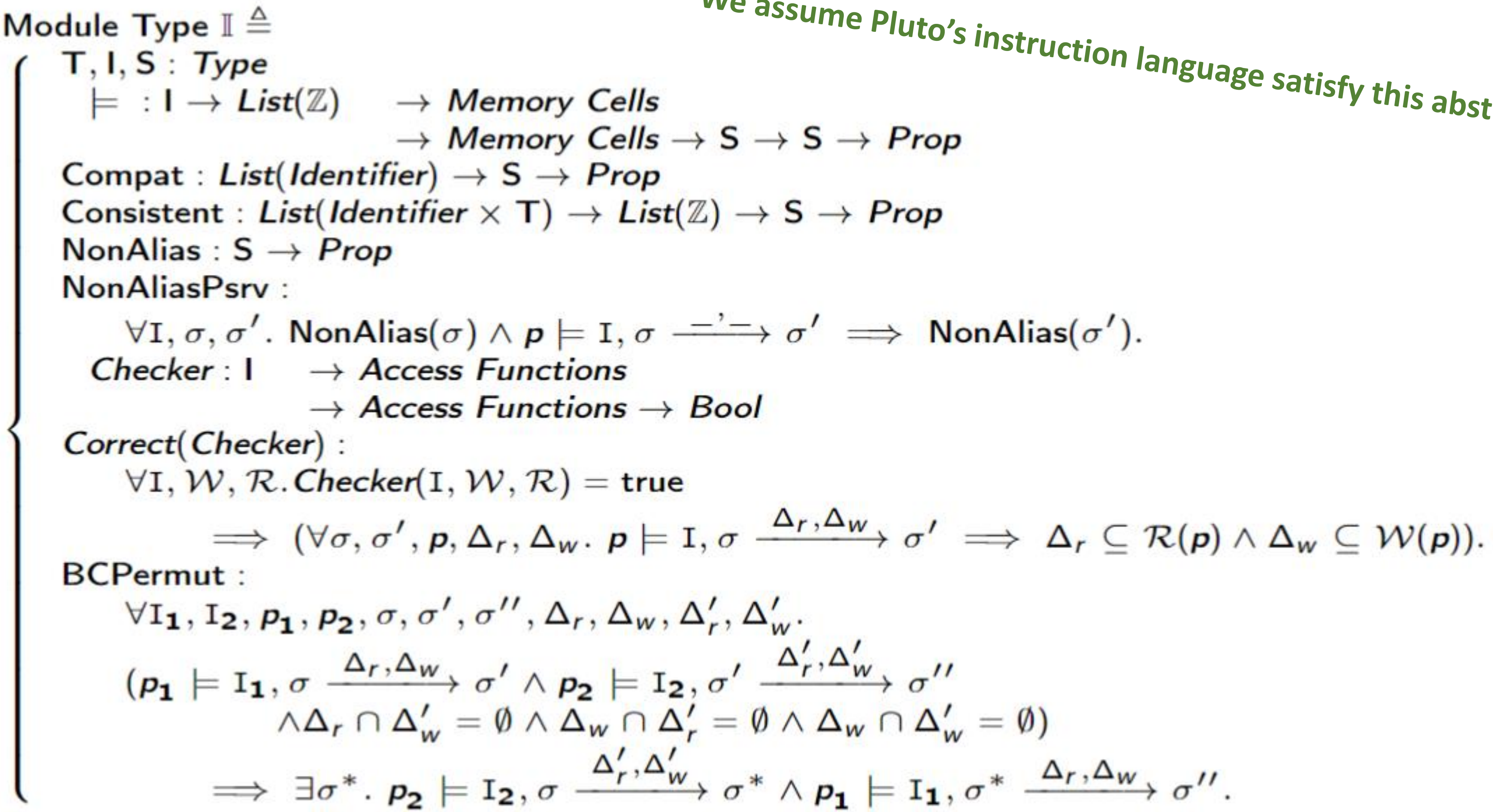
```
for (i = 1; i <= N; i++) {
  for (j1 = 1; j1 <= M; j1++) {
    for (j2 = j1; j2 <= M; j2++) {
        symmat[j1][j2] += data[i][j1] * data[i][j2];
    }
  }
}
for (j1 = 1; j1 <= M; j1++) {
  for (j2 = j1; j2 <= M; j2++) {
    symmat[j2][j1] = symmat[j1][j2];
  }
}
```

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 \mathtt{I}, \sigma, \sigma'.\ \mathsf{NonAlias}(\sigma) \wedge \boldsymbol{p} \models \mathtt{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 \mathtt{I}, \mathcal{W}, \mathcal{R}. \mathit{Checker}(\mathtt{I}, \mathcal{W}, \mathcal{R}) = \mathsf{true} \\
\qquad\qquad \implies (\forall \sigma, \sigma', \boldsymbol{p}, \Delta_r, \Delta_w.\ \boldsymbol{p} \models \mathtt{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 \mathtt{I}_1, \mathtt{I}_2, \boldsymbol{p}_1, \boldsymbol{p}_2, \sigma, \sigma', \sigma'', \Delta_r, \Delta_w, \Delta_r', \Delta_w'. \\
\qquad (\boldsymbol{p}_1 \models \mathtt{I}_1, \sigma \xrightarrow{\Delta_r, \Delta_w} \sigma' \wedge \boldsymbol{p}_2 \models \mathtt{I}_2, \sigma' \xrightarrow{\Delta_r', \Delta_w'} \sigma'' \\
\qquad\qquad\quad \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 \mathtt{I}_2, \sigma \xrightarrow{\Delta_r', \Delta_w'} \sigma^* \wedge \boldsymbol{p}_1 \models \mathtt{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.

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

Polyhedral Compilation

- Extraction -> Scheduling -> Codegen

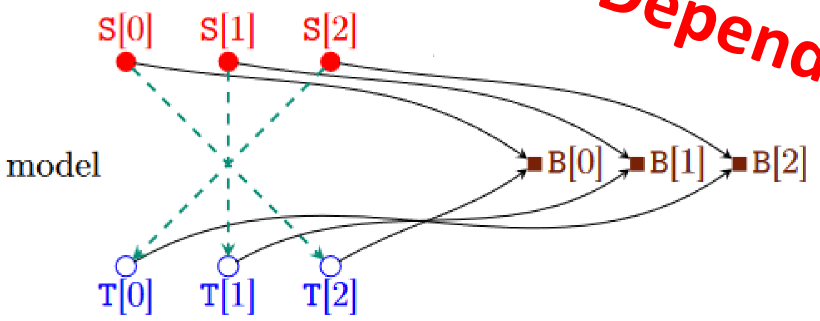
loop reverse + loop fusion

~>

```
for i in [0, 3]:
S:    B[i] = A[i] + 1
for i in [0, 3]:
T:    C[i] = B[2-i] * 3
```

```
for i in [0, 3]:
S:    B[i] = A[i] + 1
T:    C[2-i] = B[i] * 3
```

Polyhedral Model



Dependences

- An alternative integer-programming-based representation for nested loop. A **polyhedral program** is a multiset of **polyhedral instructions**, which consists of at least three parts: the **instruction**, the **domain**, and the **schedule**.
- “Extraction”

```
for i in [0, 3]:
S:    B[i] = A[i] + 1
for i in [0, 3]:
T:    C[i] = B[2-i] * 3
```

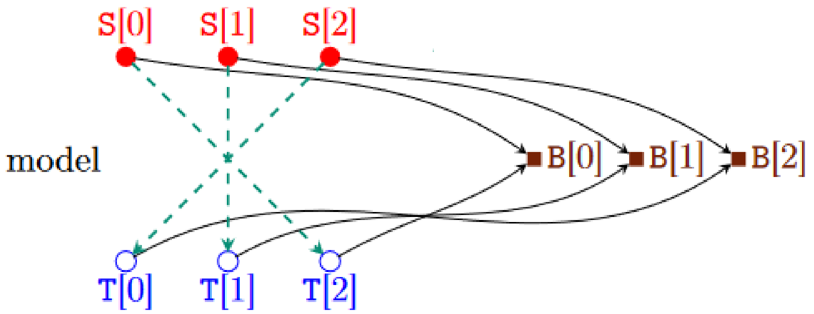
`~>`

```
{(S, 0<=i<3, [i]~>[0,i]),
 (T, 0<=i<3, [i]~>[1,i])}
```

Lexicographically ordered!

Execution order: `S[0];S[1];S[2];T[0];T[1];T[2]`

Polyhedral Model



Our work's focus!

- "Scheduling": do reordering transformation on polyhedral model, guided by dependence analysis.
- Well-designed automatic algorithms, like Pluto[2], serve for this purpose.
- Manual scheduling is also possible.
- Recall the correctness criterion: Bernstein's condition.

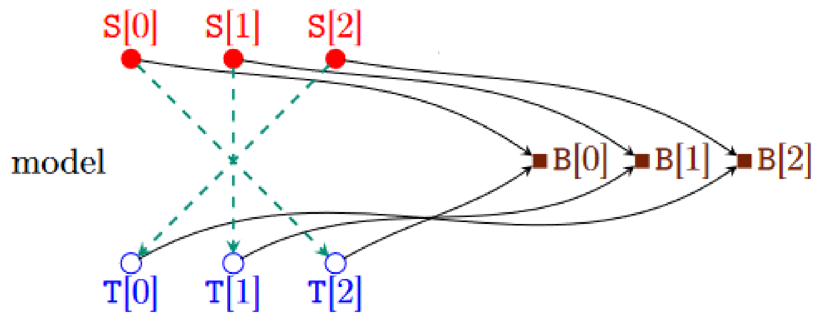
```
{(S, 0<=i<3, [i]~>[0,i]),  ~>  {(S, 0<=i<3, [i]~>...),
 (T, 0<=i<3, [i]~>[1,i])}        (T, 0<=i<3, [i]~>...)}
```

Old Execution order: `S[0];S[1];S[2];T[0];T[1];T[2]`

---

[2] https://pluto-compiler.sourceforge.net/

Polyhedral Model



Our work's focus!

- "Scheduling": do reordering transformation on polyhedral model, guided by dependence analysis.
- Well-designed automatic algorithms, like Pluto[2], serve for this purpose.
- Manual scheduling is also possible.

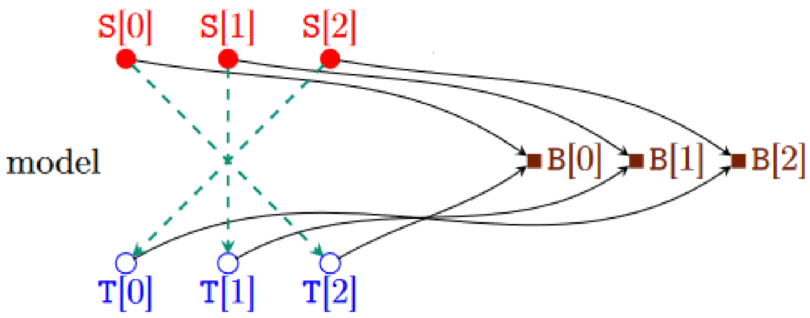
Average distance of dependence's satisfaction seems too long. Cache locality.

Old Execution order: `S[0];S[1];S[2];T[0];T[1];T[2]`

---

[2] https://pluto-compiler.sourceforge.net/

Polyhedral Model

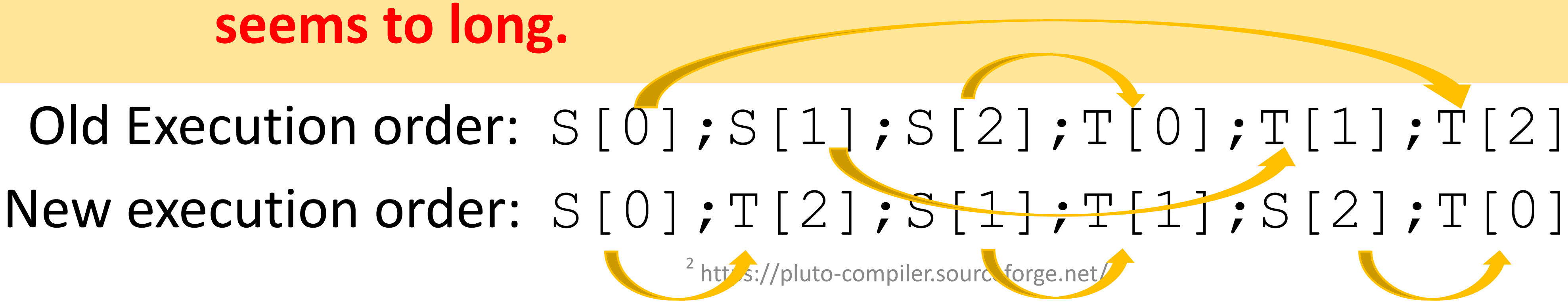


Our work's focus!

- "Scheduling": do reordering transformation on polyhedral model, guided by dependence analysis.
- Well-designed automatic algorithms, like Pluto[2], serve for this purpose.
- Manual scheduling is also possible.

Average distance of dependence's satisfaction seems to long.

Try to minimize the distance without breaking dependences!



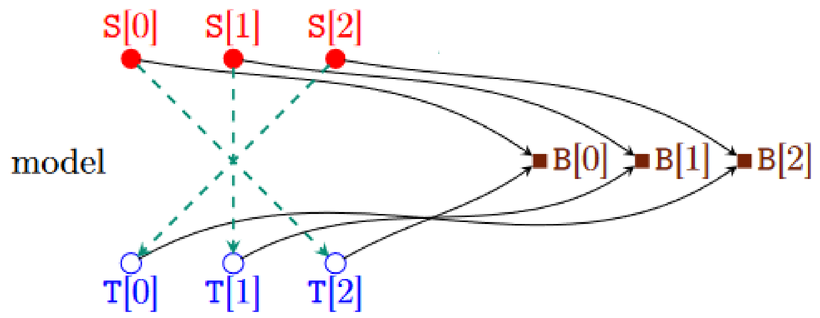
Old Execution order: `S[0];S[1];S[2];T[0];T[1];T[2]`

New execution order: `S[0];T[2];S[1];T[1];S[2];T[0]`

---

[2] https://pluto-compiler.sourceforge.net/

Polyhedral Model



Our work's focus!

- "Scheduling": do reordering transformation on polyhedral model, guided by dependence analysis.
- Well-designed automatic algorithms, like Pluto[2], serve for this purpose.
- Manual scheduling is also possible.
- Recall the correctness criterion: Bernstein's condition.

```
{(S, 0<=i<3, [i]~>[0,i]),  ~>  {(S, 0<=i<3, [i]~>[i,0]),
 (T, 0<=i<3, [i]~>[1,i])}        (T, 0<=i<3, [i]~>[2-i,1])}
```

Old Execution order: `S[0];S[1];S[2];T[0];T[1];T[2]`

New execution order: `S[0];T[2];S[1];T[1];S[2];T[0]`

[2] https://pluto-compiler.sourceforge.net/

Polyhedral Model

- “Codegen”: recover the imperative control structure from the optimized polyhedral model.

```
                                             ~>
{(S, 0<=i<3, [i]~>[i,0]),
 (T, 0<=i<3, [i]~>[2-i,1])}
```

```
for i in [0, 3]:
S:    B[i] = A[i] + 1
T:    C[2-i] = B[i] * 3
```