

# The Polyhedral Model

Cosmin E. Oancea  
`cosmin.oancea@diku.dk`

Department of Computer Science (DIKU)  
University of Copenhagen

December 2024 DPP Lecture Slides

# Agenda

Motivation: Dependency Graphs and Transformations

Polydral Analysis: Iteration Domain, Access Relations, Schedule

Presburger Sets, Relations and Associated Operations

Data-Flow Analysis and Dependency Graph

Examples: Checking Validity of Code Transformations

Proposed Exercises (Facultative)

# Acknowledgments

The material presented in these slides was taken from the tutorial "Presburger Formulas and Polyhedral Compilation" by Sven Verdoolaege, and associated slides, found online for example at <http://labexcompilation.ens-lyon.fr/wp-content/uploads/2013/02/Sven-slides.pdf> and

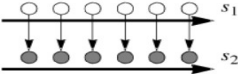

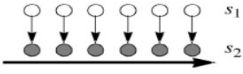
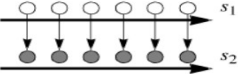
[https://www.researchgate.net/publication/291352331\\_Presburger\\_Form](https://www.researchgate.net/publication/291352331_Presburger_Form)

Additionally, we have used material from Andreas Kloeckner's "Languages and Abstractions for High-Performance Scientific Computing" Course, CS598 APK, available online at: <https://andreaks.cs.illinois.edu/cs598apk-f18/notes.pdf#page=214>

Polyhedral model provides a useful framework for reasoning about certain loop-based transformations. Questions to answer:

- How to compute the dependency graph of a loop nest?
- How to represent a code transformation?
- How to prove the legality of such a transformation?

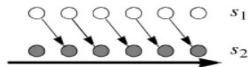
# Transformations: Fusion and Fission

SOURCE CODE	PARTITION	TRANSFORMED CODE
<pre>for (i=1; i&lt;=N; i++)   Y[i] = Z[i]; /*s1*/ for (j=1; j&lt;=N; j++)   X[j] = Y[j]; /*s2*/</pre> 	<p>Fusion</p> $s_1 : p = i$ $s_2 : p = j$	<pre>for (p=1; p&lt;=N; p++){   Y[p] = Z[p];   X[p] = Y[p]; }</pre> 
<pre>for (p=1; p&lt;=N; p++){   Y[p] = Z[p];   X[p] = Y[p]; }</pre> 	<p>Fission</p> $s_1 : i = p$ $s_2 : j = p$	<pre>for (i=1; i&lt;=N; i++)   Y[i] = Z[i]; /*s1*/ for (j=1; j&lt;=N; j++)   X[j] = Y[j]; /*s2*/</pre> 

[Aho/Ullman/Sethi '07]

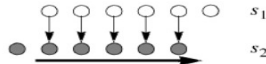
# Transformations: Reindexing and Scaling

```
for (i=1; i<=N; i++) {
    Y[i] = Z[i];    /*s1*/
    X[i] = Y[i-1]; /*s2*/
}
```

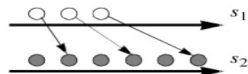


Re-indexing  
 $s_1 : p = i$   
 $s_2 : p = i - 1$

```
if (N>=1) X[1]=Y[0];
for (p=1; p<=N-1; p++){
    Y[p]=Z[p];
    X[p+1]=Y[p];
}
if (N>=1) Y[N]=Z[N];
```

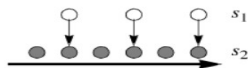


```
for (i=1; i<=N; i++)
    Y[2*i] = Z[2*i]; /*s1*/
for (j=1; j<=2N; j++)
    X[j]=Y[j];      /*s2*/
```



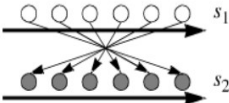
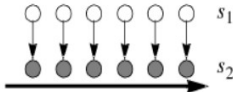
Scaling  
 $s_1 : p = 2 * i$   
 $(s_2 : p = j)$

```
for (p=1; p<=2*N; p++){
    if (p mod 2 == 0)
        Y[p] = Z[p];
    X[p] = Y[p];
}
```



[Aho/Ullman/Sethi '07]

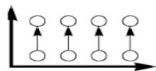
# Transformations: Partition

SOURCE CODE	PARTITION	TRANSFORMED CODE
<pre>for (i=0; i&lt;=N; i++)   Y[N-i] = Z[i]; /*s1*/ for (j=0; j&lt;=N; j++)   X[j] = Y[j]; /*s2*/</pre> 	<p>Reversal <math>s_1 : p = N - i</math> <math>(s_2 : p = j)</math></p>	<pre>for (p=0; p&lt;=N; p++){   Y[p] = Z[N-p];   X[p] = Y[p]; }</pre> 

[Aho/Ullman/Sethi '07]

# Transformations: Permutation

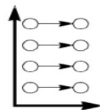
```
for (i=1; i<=N; i++)
  for (j=0; j<=M; j++)
    Z[i,j] =
      Z[i-1,j];
```



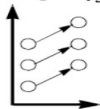
Permutation

$$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

```
for (p=0; p<=M; p++)
  for (q=1; q<=N; i++)
    Z[q,p] = Z[q-1,p]
```



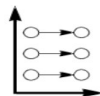
```
for (i=1; i<=N+M-1; i++)
  for (j=max(1,i+N);
       j<=min(i,M); j++)
    Z[i,j] =
      Z[i-1,j-1];
```



Skewing

$$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

```
for (p=1; p<=N; p++)
  for (q=1; q<=M; q++)
    Z[p,q-p] =
      Z[p-1,q-p-1]
```



[Aho/Ullman/Sethi '07]

Loop skewing example does not seem quite right (next slide)!



## Transformations: Loop Skewing

```
float X[N][N];  
for(int i=1; i<N; i++) {  
    for(int j=1; j < min(i+2, N); j++) {  
        X[i][j] = X[i-1][j] + X[i][j-1];  
    }  
}
```

Change of variables:  $p \leftarrow i+j, q \leftarrow j$

```
for(int p=2; p < 2*N-1; p++) {  
    int up_bd = ((p+2)/2) + (p%2);  
    for(int q=max(1,p-N+1); q<min(up_bd,N); q++) {  
        X[p-q][q] = X[p-q-1][q] + X[p-q][q-1];  
    }  
}
```

Polyhedral model provides a useful framework for reasoning about certain loop-based transformations. Questions to answer:

- How to compute the dependency graph of a loop nest?
- How to represent a code transformation?
- How to prove the legality of such a transformation?

Motivation: Dependency Graphs and Transformations

**Polydral Analysis: Iteration Domain, Access Relations, Schedule**

Presburger Sets, Relations and Associated Operations

Data-Flow Analysis and Dependency Graph

Examples: Checking Validity of Code Transformations

Proposed Exercises (Facultative)

# Main Components of Polyhedral Analysis

Key features:

- instance based:
  - ▶ statement instances
  - ▶ array elements
- compact representation
  - ▶ Presburger set and relations ...

Program Representation Uses:

- **Iteration Domain**: the set of all statement instances
- **Access Relations**: maps each statement instance to the array elements accessed (read/written) by that statement instance.
- **Schedule**: maps each statement instance to its execution time. (Execution time is abstractly represented by the total order of iterations in the target loop nest).

# Main Components of Polyhedral Analysis

Key features:

- instance based:
  - ▶ statement instances
  - ▶ array elements
- compact representation
  - ▶ Presburger set and relations ...

Program Representation Uses:

- **Iteration Domain**: the set of all statement instances
  - **Access Relations**: maps each statement instance to the array elements accessed (read/written) by that statement instance.
  - **Schedule**: maps each statement instance to its execution time. (Execution time is abstractly represented by the total order of iterations in the target loop nest).
- ⇒ Compute automatically the **Dependency Graph**: maps the source (statement instance) of a dependence to its sink.
- ⇒ Check automatically the **Validity of a desired transformation**.

Motivation: Dependency Graphs and Transformations

Polydral Analysis: Iteration Domain, Access Relations, Schedule

**Presburger Sets, Relations and Associated Operations**

Data-Flow Analysis and Dependency Graph

Examples: Checking Validity of Code Transformations

Proposed Exercises (Facultative)

## Illustrative Example (Naive)

```
R:   h(A[2]);  
      for(int i=0; i<2; i++)  
          for(int j=0; j<2; j++)  
S:      A[i+j] = f(i,j);  
      for(int k=0; k<2; k++)  
T:      g(A[k], A[0]);
```

- Iteration domain: set of all statement instances:  
 $I = \{R[]; S[0,0]; S[0,1]; S[1,0]; S[1,1]; T[0]; T[1]\}$
- Access relation (statement instance accesses array elements):  
 $W = \{S[0,0] \rightarrow A[0]; S[0,1] \rightarrow A[1]; S[1,0] \rightarrow A[1]; S[1,1] \rightarrow A[2]\}$   
 $R = \{R[] \rightarrow A[2]; T[0] \rightarrow A[0]; T[1] \rightarrow A[1]; T[1] \rightarrow A[0]\}$
- Schedule (total ordering of stmts modeling execution time):  
 $S = \{R[] \rightarrow 0; S[0,0] \rightarrow 1; S[0,1] \rightarrow 2; S[1,0] \rightarrow 3; S[1,1] \rightarrow 4;$   
 $T[0] \rightarrow 5; T[1] \rightarrow 6\}$

## Illustrative Example (Compact)

```
R:   h(A[2]);  
    for(int i=0; i<2; i++)  
        for(int j=0; j<2; j++)  
S:   A[i+j] = f(i,j);  
    for(int k=0; k<2; k++)  
T:   g(A[k], A[0]);
```

- Iteration domain: set of all statement instances:  
 $I = \{R[]; S[i,j]: 0 \leq i < 2 \wedge 0 \leq j < 2; T[k]: 0 \leq k < 2\}$
- Access relation (statement instance accesses array elements):  
 $W = \{S[i,j] \rightarrow A[i+j]: 0 \leq i < 2 \wedge 0 \leq j < 2\}$   
 $R = \{R[] \rightarrow A[2]; T[k] \rightarrow A[0]: 0 \leq k < 2; T[k] \rightarrow A[k]: 0 \leq k < 2\}$
- Schedule (total ordering of stmts modeling execution time):  
 $S = \{R[] \rightarrow [0,0,0]; S[i,j] \rightarrow [1,i,j]: 0 \leq i < 2 \wedge 0 \leq j < 2;$   
 $T[k] \rightarrow [2,k,0]: 0 \leq k < 2;\}$





## Parametric Example: Matrix Matrix Multiplication

```
    for(int i=0; i<M; i++)  
        for(int j=0; j<N; j++) {  
S1:      C[i,j] = 0.0;  
  
          for(int k=0; k<K; k++)  
S2:      C[i,j] = C[i,j] + A[i,k] * B[k,j];  
        }
```

- **Iteration domain** (set of all statement instances):

## Parametric Example: Matrix Matrix Multiplication

```
    for(int i=0; i<M; i++)  
        for(int j=0; j<N; j++) {  
S1:      C[i,j] = 0.0;  
  
        for(int k=0; k<K; k++)  
S2:      C[i,j] = C[i,j] + A[i,k] * B[k,j];  
        }
```

- Iteration domain (set of all statement instances):

$$I = \left\{ \begin{array}{ll} S1[i,j] & : 0 \leq i < M \wedge 0 \leq j < N; \\ S2[i,j,k] & : 0 \leq i < M \wedge 0 \leq j < N \wedge 0 \leq k < K \end{array} \right\}$$

- Access relation (R = Read, W = Write):

# Parametric Example: Matrix Matrix Multiplication

```
    for(int i=0; i<M; i++)  
        for(int j=0; j<N; j++) {  
S1:      C[i,j] = 0.0;  
  
        for(int k=0; k<K; k++)  
S2:      C[i,j] = C[i,j] + A[i,k] * B[k,j];  
        }
```

- Iteration domain (set of all statement instances):

$$I = \left\{ \begin{array}{ll} S1[i,j] & : 0 \leq i < M \wedge 0 \leq j < N; \\ S2[i,j,k] & : 0 \leq i < M \wedge 0 \leq j < N \wedge 0 \leq k < K \end{array} \right\}$$

- Access relation (R = Read, W = Write):

$$W = \{ S1[i,j] \rightarrow C[i,j]; S2[i,j,k] \rightarrow C[i,j] \}$$

$$R = \{ S2[i,j,k] \rightarrow C[i,j]; S2[i,j,k] \rightarrow A[i,k]; S2[i,j,k] \rightarrow B[k,j] \}$$

- Schedule (total ordering of stmts modeling execution time):

# Parametric Example: Matrix Matrix Multiplication

```
    for(int i=0; i<M; i++)  
        for(int j=0; j<N; j++) {  
S1:      C[i,j] = 0.0;  
  
        for(int k=0; k<K; k++)  
S2:      C[i,j] = C[i,j] + A[i,k] * B[k,j];  
        }
```

- Iteration domain (set of all statement instances):

$$I = \left\{ \begin{array}{ll} S1[i,j] & : 0 \leq i < M \wedge 0 \leq j < N; \\ S2[i,j,k] & : 0 \leq i < M \wedge 0 \leq j < N \wedge 0 \leq k < K \end{array} \right\}$$

- Access relation (R = Read, W = Write):

$$W = \{ S1[i,j] \rightarrow C[i,j]; S2[i,j,k] \rightarrow C[i,j] \}$$

$$R = \{ S2[i,j,k] \rightarrow C[i,j]; S2[i,j,k] \rightarrow A[i,k]; S2[i,j,k] \rightarrow B[k,j] \}$$

- Schedule (total ordering of stmts modeling execution time):

$$S = \{ S1[i,j] \rightarrow [i,j,0,0]; S2[i,j,k] \rightarrow [i,j,1,k] \}$$

# Presburger Sets and Relations

```
R:  h(A[2]);  
    for(int i=0; i<2; i++)  
        for(int j=0; j<2; j++)  
S:      A[i+j] = f(i,j);  
        for(int k=0; k<2; k++)  
T:      g(A[k], A[0]);
```

Examples:

$I = \{R[]; S[i,j]: 0 \leq i < 2 \wedge 0 \leq j < 2; T[k]: 0 \leq k < 2\}$

$R = \{R[] \rightarrow A[2]; T[k] \rightarrow A[0]: 0 \leq k < 2; T[k] \rightarrow A[k]: 0 \leq k < 2\}$

General Form:

- Sets:  $\{ S_1[i] : f_1(i); S_2[i] : f_2(i); \dots \}$   
with  $f_k$  Presburger formulas  
 $\Rightarrow$  set of elements of form  $S_k[i]$ , one for each  $i$  satisfying  $f_k(i)$ .
- Relations:  $\{ S_1[i] \rightarrow T_1[j] : f_1(i,j); S_2[i] \rightarrow T_2[j] : f_2(i,j); \dots \}$   
 $\Rightarrow$  set of pairs of elements of the form  $S_k[i] \rightarrow T_k[j]$ .  
(Not necessarily single-valued functions.)

# Presburger Formulas

Presburger arithmetic allows (quasi-)exact answers/solutions.

- **Language**  $\mathcal{L} = \{f_1/r_1, f_2/r_2, \dots, P_1/S_1, P_2/S_2, \dots\}$

$f_i$  function symbol with arity  $r_i \geq 0$ :

- ▶ addition, subtraction:  $+/2, -/2$
- ▶ constant  $d/0$ , for each integer  $d$
- ▶ integer division:  $\lfloor \cdot / d \rfloor / 1$ , for a fixed integer  $d > 0$
- ▶ set of symbolic constant  $c_i/0$

$P_i$  predicate symbol with arity  $s_i \geq 0$ , e.g.,  $\leq /2$ .

- **Terms** (inductive definition)

- ▶  $v$  is a term if  $v$  is a variable
- ▶  $f_i(t_1, \dots, t_{r_i})$  is a term if  $t_1, \dots, t_{r_i}$  are terms

- **Formulas** (inductive definition)

true	$F_1 \wedge F_2$ (conjunction)	quantification:
$P_i(t_1, \dots, t_{s_i})$	$F_1 \vee F_2$ (disjunction)	$\exists v : F_1(v)$ (existential)
$t_1 = t_2$	$\neg F_1$ (negation)	$\forall v : F_1(v)$ (universal)

$P_i/s_i$  are predicates,  $t_j$  are terms,  $v$  variable,  $F_k$  are formulas.

# Interpretation of Presburger Formulas

- **Domain of Discourse (Universe):** sets of integers in  $\mathbb{Z}$
- **Interpretation** function/predicate symbols  $\rightarrow$  functions/predicates
  - ▶  $+/2, -/2$  map to addition and subtraction on integers ...
  - ▶ symbolic constants  $c_i$  are “uninterpreted”,  
i.e., consider all possible interpretations as integers
- **Truth Values**
  - ▶ **true** is true;  $P_i(t_1, \dots, t_{r_i})$  is true if interpretation is true
  - ▶ ...
  - ▶  $\exists v : F(v)$  is true iff  $F(d)$  is true for **some** integer  $d$  in the universe ( $\mathbb{Z}$ ).
  - ▶  $\forall v : F(v)$  is true iff  $F(d)$  is true for **every** integers  $d$  in the universe ( $\mathbb{Z}$ ).



# Syntactic Sugar

**Notation:**  $\bar{i}^n \equiv i_1, \dots, i_n$ , and  $n$  can be left unspecified.

- $false$  is equal to  $\neg true$
- $a \Rightarrow b$  is equal to  $\neg a \vee b$
- $S[\bar{i}]$  is equal to  $S[\bar{i}] : true$
- $S[i_1, \dots, i_{k-1}, g(i_1, \dots, i_{k-1}), i_{k+1}, \dots, i_n] : f(\bar{i})$  is equal to  $S[i_1, \dots, i_{k-1}, i_k, i_{k+1}, \dots, i_n] : i_k = g(i_1, \dots, i_{k-1}) \wedge f(\bar{i})$   
e.g.,  $\{S[i] \rightarrow T(i+1)\}$  is equal to  $\{S[i] \rightarrow T(j) : j = i+1\}$
- $a < b$  is equal to  $a \leq b - 1 \dots$
- $\{S[i, j] : i, j \geq 0\}$  is equal to  $\{S[i, j] : i \geq 0 \wedge j \geq 0\}$
- $\{S[i] : 0 \leq i \leq 10\}$  is equal to  $\{S[i] : 0 \leq i \wedge i \leq 10\}$
- $-e$  is equal to  $0 - e$
- $n \cdot e$  is equal to  $e + \dots + e$ , with  $n$  positive integer constant
- $a_1, \dots, a_n \prec b_1, \dots, b_n$  equals  $\bigvee_{i=1}^n ((\bigwedge_{j=1}^{i-1} a_j = b_j) \wedge a_i < b_i)$   
e.g.,  $\{S[i_1, i_2] \rightarrow T[j_1, j_2] : i_1, i_2 \prec j_1, j_2\}$  equals  $\{S[i_1, i_2] \rightarrow T[j_1, j_2] : i_1 < j_1 \vee (i_1 = j_1 \wedge i_2 < j_2)\} \dots$

# Examples

- $\{S[i,j] \rightarrow [1,i,j] : 0 \leq i,j < 2; T[k] \rightarrow [2,k,0] : 0 \leq k < 2\}$   
is equal to  $\{S[0,0] \rightarrow [1,0,0]; S[0,1] \rightarrow [1,0,1]; S[1,0] \rightarrow [1,1,0]; S[1,1] \rightarrow [1,1,1]; T[0] \rightarrow [2,0,0]; T[1] \rightarrow [2,1,0]\}$
- $\{[i] : 0 \leq i \leq 10 \wedge \exists \alpha : i = 2 \cdot \alpha\}$  is equal to  $\{[0]; [2]; [4]; [6]; [8]; [10]\}$
- $\{[i] : 0 \leq i \leq 10 \wedge i = 2 \cdot \alpha\}$  is equal to
$$\begin{cases} \{[2 \cdot \alpha]\} & \text{if } 0 \leq \alpha \leq 5 \\ \emptyset & \text{otherwise} \end{cases}$$
- $\{[i] : \forall j : i + j \leq 10\}$  is equal to  $\emptyset$ .

# Spaces

Recall:

- Sets:  $\{ S_1[\bar{i}] : f_1(\bar{i}); S_2[\bar{i}] : f_2(\bar{i}); \dots \}$
- Relations:  $\{ S_1[\bar{i}] \rightarrow T_1[\bar{j}] : f_1(\bar{i}, \bar{j}); S_2[\bar{i}] \rightarrow T_2[\bar{j}] : f_2(\bar{i}, \bar{j}); \dots \}$

The **identifier** (e.g.,  $S_1, S_2, T_1, T_2$ ) together with the **dimension**, i.e., the number of elements in the subsequent tuple (e.g.,  $\bar{i}, \bar{j}$ ) will be called a **space**.

When we say  $S_2[\bar{i}] = T_1[\bar{j}]$  we mean:

- the **identifiers**  $S_2$  and  $T_1$  are the same, and
- the **dimensions (rank)** of  $\bar{i}$  and  $\bar{j}$  are the same.

For example:  $S[] \neq S[i]$ ;  $S[a] = S[b]$ ;  $S[] \neq T[]$ .

# Operations on Relations

**Union:**  $\{S_1[\bar{i}] \rightarrow T_1[\bar{j}] : f_1(\bar{i}, \bar{j}); \dots\} \cup \{S_2[\bar{i}] \rightarrow T_2[\bar{j}] : f_2(\bar{i}, \bar{j}); \dots\}$   
 $\Rightarrow \{S_1[\bar{i}] \rightarrow T_1[\bar{j}] : f_1(\bar{i}, \bar{j}); \dots; S_2[\bar{i}] \rightarrow T_2[\bar{j}] : f_2(\bar{i}, \bar{j}); \dots\}$

**Inverse:**  $R = \{S[\bar{i}] \rightarrow T[\bar{j}] : f(\bar{i}, \bar{j})\} \Rightarrow R^{-1} = \{T[\bar{j}] \rightarrow S[\bar{i}] : f(\bar{i}, \bar{j})\}$

**Dom:**  $R = \{S[\bar{i}] \rightarrow T[\bar{j}] : f(\bar{i}, \bar{j})\} \Rightarrow \text{dom } R = \{S[\bar{i}] : \exists \bar{j} : f(\bar{i}, \bar{j})\}$

**Range:**  $R = \{S[\bar{i}] \rightarrow T[\bar{j}] : f(\bar{i}, \bar{j})\} \Rightarrow \text{ran } R = \{T[\bar{j}] : \exists \bar{i} : f(\bar{i}, \bar{j})\}$

**UnivRel:**  $A = \{S[\bar{i}] : f(\bar{i})\}$  and  $B = \{T[\bar{j}] : g(\bar{j})\}$   
 $\Rightarrow A \rightarrow B = \{S[\bar{i}] \rightarrow T[\bar{j}] : f(\bar{i}) \wedge g(\bar{j})\}$

**Intersect**  $\{S_1[\bar{i}_1] \rightarrow T_1[\bar{j}_1] : f_1(\bar{i}_1, \bar{j}_1)\} \cap \{S_2[\bar{i}_2] \rightarrow T_2[\bar{j}_2] : f_2(\bar{i}_2, \bar{j}_2)\} \Rightarrow$

$$\begin{cases} \{S_1[\bar{i}] \rightarrow T_1[\bar{j}] : f_1(\bar{i}, \bar{j}) \wedge f_2(\bar{i}, \bar{j})\} & \text{if } S_1[\bar{i}_1] = S_2[\bar{i}_2] \text{ and} \\ & T_1[\bar{j}_1] = T_2[\bar{j}_2] \\ \emptyset & \text{otherwise} \end{cases}$$

# Examples: Operations on Relations

```
R:  h(A[2]);  
    for(int i=0; i<2; i++)  
        for(int j=0; j<2; j++)  
S:      A[i+j] = f(i,j);  
    for(int k=0; k<2; k++)  
T:      g(A[k], A[0]);
```

- Access relation (statement instance accesses array elements):

$$W = \{S[i,j] \rightarrow A[i+j] : 0 \leq i < 2 \wedge 0 \leq j < 2\}$$

$$R = \{R[] \rightarrow A[2]; T[k] \rightarrow A[0] : 0 \leq k < 2; T[k] \rightarrow A[k] : 0 \leq k < 2\}$$

- $R \cup W = \{ R[] \rightarrow A[2]; T[k] \rightarrow A[0] : 0 \leq k < 2; T[k] \rightarrow A[k] : 0 \leq k < 2; S[i,j] \rightarrow A[i+j] : 0 \leq i < 2 \wedge 0 \leq j < 2 \}$
- $R^{-1} = \{A[2] \rightarrow R[]; A[0] \rightarrow T[k] : 0 \leq k < 2; A[k] \rightarrow T[k] : 0 \leq k < 2\}$
- $\text{dom } R = \{R[]; T[k] : 0 \leq k < 2\}$
- $\text{ran } R = \{A[k] : 0 \leq k \leq 2\}$
- $\text{dom } R \rightarrow \text{ran } R = \{R[] \rightarrow A[j] : 0 \leq j \leq 2; T[k] \rightarrow A[j] : 0 \leq k < 2 \wedge 0 \leq j \leq 2\}$
- $\{T[k] \rightarrow A[k] : 0 \leq k < 2\} \cap \{T[k] \rightarrow A[0] : 0 \leq k < 2\} = \{T[0] \rightarrow A[0]\}$

# Domain/Range Restrictions

Assume  $A = \{S_1[i_1] : f(i_1)\}$ ,  $B = \{S_2[i_2] \rightarrow T_2[j_2] : g(i_2, j_2)\}$

- Domain Restrictions:  $R \cap_{\text{dom}} S = R \cap (S \rightarrow \text{ran } R)$

$$A \cap_{\text{dom}} B = \begin{cases} \{S_2[i] \rightarrow T_2[j] : f(i) \wedge g(i, j)\}, & \text{if } S_1(i_1) = S_2(i_2) \\ \emptyset & \text{otherwise} \end{cases}$$

- Range Restrictions:  $R \cap_{\text{ran}} S = R \cap ((\text{dom } R) \rightarrow S)$

$$B \cap_{\text{ran}} A = \begin{cases} \{S_2[i] \rightarrow T_2[j] : f(j) \wedge g(i, j)\}, & \text{if } S_1(i_1) = T_2(j_2) \\ \emptyset & \text{otherwise} \end{cases}$$

Example:

- $I = \{R[]; S[i, j] : 0 \leq i < 2 \wedge 0 \leq j < 2; T[k] : 0 \leq k < 2\}$   
 $S_0 = \{R[] \rightarrow [0, 0, 0]; S[i, j] \rightarrow [1, i, j]; T[k] \rightarrow [2, k, 0]; \}$

$$S = I \cap_{\text{dom}} S_0 = \{ R[] \rightarrow [0, 0, 0]; T[k] \rightarrow [2, k, 0] : 0 \leq k < 2; \\ S[i, j] \rightarrow [1, i, j] : 0 \leq i < 2 \wedge 0 \leq j < 2 \quad \}$$

# Relation Difference/Subtraction and Comparisons

$$A = \{S_1[i_1] \rightarrow T_1[j_1] : f(i_1, j_1)\},$$

$$B = \{S_2[i_2] \rightarrow T_2[j_2] : g(i_2, j_2)\}$$

$$A \setminus B = \begin{cases} \{S_1[i] \rightarrow T_1[j] : f(i, j) \wedge \neg g(i, j)\}, & \text{if } S_1(i_1) = S_2(i_2) \text{ and} \\ & T_1(j_1) = T_2(j_2) \\ \{S_1[i] \rightarrow T_1[j] : f(i, j)\} & \text{otherwise} \end{cases}$$

Example:

$$\{T[k] \rightarrow A[k] : 0 \leq k < 2\} \setminus \{T[k] \rightarrow A[0] : 0 \leq k < 2\} = \{T[1] \rightarrow A[1]\}$$

Comparisons:

- emptiness check (if the Preseburger formula reduces to false)
- $A \subseteq B$  is defined as  $A \setminus B = \emptyset$
- $A \supseteq B$  is defined as  $B \subseteq A$
- $A = B$  is defined as  $B \subseteq A \wedge A \subseteq B$
- $A \subset B$  is defined as  $A \subseteq B \wedge \neg(A = B)$
- $A \supset B$  is defined as  $B \subset A$

# Composition of Relations

Composition:

$$A = \{S_1[i_1] \rightarrow T_1[j_1] : f(i_1, j_1)\}, \quad B = \{S_2[i_2] \rightarrow T_2[j_2] : g(i_2, j_2)\}$$

$$B \circ A = \begin{cases} \{S_1[i] \rightarrow T_2[j] : \exists k : f(i, k) \wedge g(k, j)\}, & \text{if } T_1(j_1) = S_2(i_2) \\ \emptyset & \text{otherwise} \end{cases}$$

Example:

$$\text{Write Set: } W = \{S[i, j] \rightarrow A[i + j] : 0 \leq i < 2 \wedge 0 \leq j < 2\}$$

Inverse of Write set (i.e., written array elements to statements):

$$W^{-1} = \{A[a] \rightarrow S[i, j] : a = i + j \wedge 0 \leq i < 2 \wedge 0 \leq j < 2\}$$

Pairs of statement instances that write the same array element:

$$W^{-1} \circ W = \{S[i, j] \rightarrow S[i', j'] : 0 \leq i, j, i', j' < 2 \wedge i + j = i' + j'\} = \\ \{S[0, 0] \rightarrow S[0, 0]; S[0, 1] \rightarrow S[0, 1]; S[1, 0] \rightarrow S[1, 0]; S[1, 1] \rightarrow \\ S[1, 1]; S[0, 1] \rightarrow S[1, 0]; S[1, 0] \rightarrow S[0, 1]; \}$$





# Application of a Relation to a Set

Application:

$$A = \{S_1[i_1] : f(i_1)\}, \quad B = \{S_2[i_2] \rightarrow T_2[j_2] : g(i_2, j_2)\}$$

$$B(A) = \begin{cases} \{T_2[j] : \exists i : f(i) \wedge g(i, j)\}, & \text{if } S_1(i_1) = S_2(i_2) \\ \emptyset & \text{otherwise} \end{cases}$$

Example:

Read Set  $R$  (Statement instances reading array elements):

$$\{R[] \rightarrow A[2]; T[k] \rightarrow A[0] : 0 \leq k < 2; T[k] \rightarrow A[k] : 0 \leq k < 2\}$$

Instances of  $T$  statements:

$$S = \{T[k] : 0 \leq k < 2\}$$

Array elements read by the  $T$  statements:

$$R(S) = \{A[k] : 0 \leq k < 2\}$$

# Lexicographic Order on Sets

$$A = \{S[\bar{i}] : f(\bar{i})\}, \quad B = \{T[\bar{j}] : g(\bar{j})\}$$

$$A \prec B = \begin{cases} \{S[\bar{i}] \rightarrow S[\bar{j}] : f(\bar{i}) \wedge g(\bar{j}) \wedge \bar{i} \prec \bar{j}\}, & \text{if } S(\bar{i}) = T(\bar{j}) \\ \emptyset & \text{otherwise} \end{cases}$$

Example:

Iteration Domain:

$$I = \{R[]; S[i, j] : 0 \leq i < 2 \wedge 0 \leq j < 2; T[k] : 0 \leq k < 2\}$$

$I \prec I$  lexicographic order on pairs of statement instances:

$$\{S[i, j] \rightarrow S[i', j'] : 0 \leq i, j, i', j' < 2 \wedge i, j \prec i', j'; T[0] \rightarrow T[1]\} =$$

$$\{S[0, 0] \rightarrow S[0, 1]; S[0, 0] \rightarrow S[1, 0]; S[0, 0] \rightarrow S[1, 1]; S[0, 1] \rightarrow S[1, 0]; S[0, 1] \rightarrow S[1, 1]; S[1, 0] \rightarrow S[1, 1]; T[0] \rightarrow T[1]\}$$

# Lexicographic Order on Relations

Binary relation on domains reflect lexicographic order of images:

$$A = \{S_1[\bar{i}_1] \rightarrow T_1[\bar{j}_1] : f(\bar{i}_1, \bar{j}_1)\},$$

$$B = \{S_2[\bar{i}_2] \rightarrow T_2[\bar{j}_2] : g(\bar{i}_2, \bar{j}_2)\}$$

$$A \prec B = \begin{cases} \{S_1[\bar{i}_1] \rightarrow S_2[\bar{i}_2] : \exists j_1, j_2 : f(\bar{i}_1, \bar{j}_1) \wedge \\ g(\bar{i}_2, \bar{j}_2) \wedge \bar{j}_1 \prec \bar{j}_2\}, & \text{if } T_1(\bar{j}_1) = T_2(\bar{j}_2) \\ \emptyset & \text{otherwise} \end{cases}$$

# Lexicographic Optimizations: Last Write

Binary relation on domains reflect lexicographic order of images:

$$R = \{S[\bar{i}] \rightarrow T[\bar{j}] : f(\bar{i}, \bar{j})\},$$

$$\text{lexmax } R = \{S[\bar{i}] \rightarrow T[\bar{j}] : f(\bar{i}, \bar{j}) \wedge \forall j' : f(i, j') \Rightarrow j \succeq j'\}$$

Example:

$(R \cup W)^{-1}$ : statement instances accessing array element

$$\{A[2] \rightarrow R[]; A[a] \rightarrow S[i, j] : a = i + j \wedge 0 \leq i, j < 2; \\ A[0] \rightarrow T[k] : 0 \leq k < 2; A[k] \rightarrow T[k] : 0 \leq k < 2\}$$

$\text{lexmax } (R \cup W)^{-1}$ : last instance of statement accessing element

$$\{A[2] \rightarrow R[]; A[a] \rightarrow S[a, 0] : 0 \leq a < 2; A[2] \rightarrow S[1, 1]; \\ A[k] \rightarrow T[1] : 0 \leq k < 2\}$$

Motivation: Dependency Graphs and Transformations

Polydral Analysis: Iteration Domain, Access Relations, Schedule

Presburger Sets, Relations and Associated Operations

**Data-Flow Analysis and Dependency Graph**

Examples: Checking Validity of Code Transformations

Proposed Exercises (Facultative)

## Last-Write Analysis (see file last-write.py)

*Given a read from an array element, what was the last write to the same array element before the read?*

```
    for(int i=0; i<N; i++)  
        for(int j=0; j<N-i; j++)  
F:          A[i+j] = f(A[i+j]);  
  
    for(int i=0; i<N; i++)  
S:  X[i] = g(A[i]);
```

## Last-Write Analysis (see file last-write.py)

*Given a read from an array element, what was the last write to the same array element before the read?*

```
for(int i=0; i<N; i++)  
    for(int j=0; j<N-i; j++)
```

```
F:      A[i+j] = f(A[i+j]);
```

```
for(int i=0; i<N; i++)
```

```
S:      X[i] = g(A[i]);
```

- Access relations:

$$W_1 = \{F[i,j] \rightarrow A[i+j] : 0 \leq i < N \wedge 0 \leq j < N - i\}$$

$$R_2 = \{S[i] \rightarrow A[i] : 0 \leq i < N\}$$



## Last-Write Analysis (see file last-write.py)

*Given a read from an array element, what was the last write to the same array element before the read?*

```
for(int i=0; i<N; i++)  
    for(int j=0; j<N-i; j++)
```

```
F:      A[i+j] = f(A[i+j]);
```

```
for(int i=0; i<N; i++)
```

```
S:      X[i] = g(A[i]);
```

- Access relations:

$$W_1 = \{F[i,j] \rightarrow A[i+j] : 0 \leq i < N \wedge 0 \leq j < N-i\}$$

$$R_2 = \{S[i] \rightarrow A[i] : 0 \leq i < N\}$$

- Map each statement instance reading an element to all the statements that have written that element:

$$R = W_1^{-1} \circ R_2 = \{S[i] \rightarrow F[i', i-i'] : 0 \leq i' \leq i < N\}$$

# Last-Write Analysis (see file last-write.py)

*Given a read from an array element, what was the last write to the same array element before the read?*

```
for(int i=0; i<N; i++)  
    for(int j=0; j<N-i; j++)
```

```
F:      A[i+j] = f(A[i+j]);
```

```
for(int i=0; i<N; i++)
```

```
S:      X[i] = g(A[i]);
```

- Access relations:

$$W_1 = \{F[i,j] \rightarrow A[i+j] : 0 \leq i < N \wedge 0 \leq j < N-i\}$$

$$R_2 = \{S[i] \rightarrow A[i] : 0 \leq i < N\}$$

- Map each statement instance reading an element to all the statements that have written that element:

$$R = W_1^{-1} \circ R_2 = \{S[i] \rightarrow F[i', i-i'] : 0 \leq i' \leq i < N\}$$

- Last Write:  $\text{lexmax } R = \{S[i] \rightarrow F[i,0] : 0 \leq i < N\}$

# Dependency Graph and Code Transformations

Recall: iteration  $\bar{j}$  depends on iteration  $\bar{i}$  iff:

- $\bar{j}$  is executed before  $\bar{i}$  in the original program,
- $\bar{i}$  and  $\bar{j}$  may access the same memory location, and
- at least one of those two accesses is a write!

# Dependency Graph and Code Transformations

Recall: iteration  $\bar{j}$  depends on iteration  $\bar{i}$  iff:

- $\bar{j}$  is executed before  $\bar{i}$  in the original program,
- $\bar{i}$  and  $\bar{j}$  may access the same memory location, and
- at least one of those two accesses is a write!

Dependency Graph Computation:

$$D = ((W^{-1} \circ R) \cup (W^{-1} \circ W) \cup (R^{-1} \circ W)) \cap (S \prec S)$$

$W$ : write-access relation,  $R$ : read-access relation,

$S$ : original schedule.

# Dependency Graph and Code Transformations

Recall: iteration  $\bar{j}$  depends on iteration  $\bar{i}$  iff:

- $\bar{j}$  is executed before  $\bar{i}$  in the original program,
- $\bar{i}$  and  $\bar{j}$  may access the same memory location, and
- at least one of those two accesses is a write!

Dependency Graph Computation:

$$D = ((W^{-1} \circ R) \cup (W^{-1} \circ W) \cup (R^{-1} \circ W)) \cap (S \prec S)$$

$W$ : write-access relation,  $R$ : read-access relation,

$S$ : original schedule.

A code transformation corresponds to computing a new schedule  $S'$  that executes the same statements in a different order. The transformation is valid if  $S'$  respects the dependencies of  $D$ :

$$\bar{i} \rightarrow \bar{j} \in D \Rightarrow S'(\bar{i}) \prec S'(\bar{j})$$

## Validating New Schedules (see file common.py)

Dependency Graph Computation:

$$D = ((W^{-1} \circ R) \cup (W^{-1} \circ W) \cup (R^{-1} \circ W)) \cap (S \prec S)$$

Safe rescheduling  $S'$  iff

$$\bar{i} \rightarrow \bar{j} \in D \Rightarrow S'(\bar{i}) \prec S'(\bar{j})$$

How to implement the test above? Assume  $S'$  a new schedule (mapping original statement instances to new time abstraction).

$$T_{src \rightarrow sink} = (S' \circ D) \circ S'^{-1}$$

# Validating New Schedules (see file common.py)

Dependency Graph Computation:

$$D = ((W^{-1} \circ R) \cup (W^{-1} \circ W) \cup (R^{-1} \circ W)) \cap (S \prec S)$$

Safe rescheduling  $S'$  iff

$$\bar{i} \rightarrow \bar{j} \in D \Rightarrow S'(\bar{i}) \prec S'(\bar{j})$$

How to implement the test above? Assume  $S'$  a new schedule (mapping original statement instances to new time abstraction).

$$T_{src \rightarrow sink} = (S' \circ D) \circ S'^{-1}$$

- Maps the source stmt to the time of the dependence sink;
- Maps the time of the source stmt to the time of the sink.

$$S'_{desc} = (\text{ran } S') \succeq (\text{ran } S')$$

( $S'_{desc}$  denotes all illegal re-orderings)

Code Transformation is valid if  $T_{src \rightarrow sink} \cap S'_{desc} = \emptyset$

# Validating New Schedules (see file common.py)

Dependency Graph Computation:

$$D = ((W^{-1} \circ R) \cup (W^{-1} \circ W) \cup (R^{-1} \circ W)) \cap (S \prec S)$$

Safe rescheduling  $S'$  iff

$$\bar{i} \rightarrow \bar{j} \in D \Rightarrow S'(\bar{i}) \prec S'(\bar{j})$$

How to implement the test above? Assume  $S'$  a new schedule (mapping original statement instances to new time abstraction).

$$T_{src \rightarrow sink} = (S' \circ D) \circ S'^{-1}$$

- Maps the source stmt to the time of the dependence sink;
- Maps the time of the source stmt to the time of the sink.

$$S'_{desc} = (\text{ran } S') \succeq (\text{ran } S')$$

( $S'_{desc}$  denotes all illegal re-orderings)

Code Transformation is valid if  $T_{src \rightarrow sink} \cap S'_{desc} = \emptyset$

(if for all dependencies, in the new schedule, the time of the source is still smaller than the time of the sink statement!)



Motivation: Dependency Graphs and Transformations

Polydral Analysis: Iteration Domain, Access Relations, Schedule

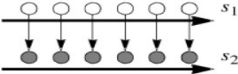

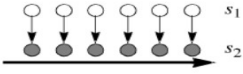
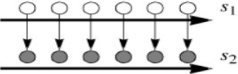
Presburger Sets, Relations and Associated Operations

Data-Flow Analysis and Dependency Graph

**Examples: Checking Validity of Code Transformations**

Proposed Exercises (Facultative)

# Transformations: Fusion and Fission

SOURCE CODE	PARTITION	TRANSFORMED CODE
<pre>for (i=1; i&lt;=N; i++)   Y[i] = Z[i]; /*s1*/ for (j=1; j&lt;=N; j++)   X[j] = Y[j]; /*s2*/</pre> 	<p>Fusion</p> $s_1 : p = i$ $s_2 : p = j$	<pre>for (p=1; p&lt;=N; p++){   Y[p] = Z[p];   X[p] = Y[p]; }</pre> 
<pre>for (p=1; p&lt;=N; p++){   Y[p] = Z[p];   X[p] = Y[p]; }</pre> 	<p>Fission</p> $s_1 : i = p$ $s_2 : j = p$	<pre>for (i=1; i&lt;=N; i++)   Y[i] = Z[i]; /*s1*/ for (j=1; j&lt;=N; j++)   X[j] = Y[j]; /*s2*/</pre> 

[Aho/Ullman/Sethi '07]

## Fusion Encoding (see fusion.py)

- Iteration Domain:

$$I = \{S1[i] : 1 \leq i \leq N; S2[j] : 1 \leq j \leq N\}$$

## Fusion Encoding (see fusion.py)

- Iteration Domain:

$$I = \{S1[i] : 1 \leq i \leq N; S2[j] : 1 \leq j \leq N\}$$

- Original Schedule:

$$S = I \cap_{dom} \{S1[i] \rightarrow [1, i]; S2[j] \rightarrow [2, j]\}$$

# Fusion Encoding (see fusion.py)

- Iteration Domain:

$$I = \{S1[i] : 1 \leq i \leq N; S2[j] : 1 \leq j \leq N\}$$

- Original Schedule:

$$S = I \cap_{dom} \{S1[i] \rightarrow [1, i]; S2[j] \rightarrow [2, j]\}$$

- Read and Write Access Relations:

$$W_{access}^{rel} = I \cap_{dom} \{S1[i] \rightarrow Y[i]; S2[j] \rightarrow X[j]\}$$

$$R_{access}^{rel} = I \cap_{dom} \{S1[i] \rightarrow Z[i]; S2[j] \rightarrow Y[j]\}$$

## Fusion Encoding (see fusion.py)

- Iteration Domain:

$$I = \{S1[i] : 1 \leq i \leq N; S2[j] : 1 \leq j \leq N\}$$

- Original Schedule:

$$S = I \cap_{dom} \{S1[i] \rightarrow [1, i]; S2[j] \rightarrow [2, j]\}$$

- Read and Write Access Relations:

$$W_{access}^{rel} = I \cap_{dom} \{S1[i] \rightarrow Y[i]; S2[j] \rightarrow X[j]\}$$

$$R_{access}^{rel} = I \cap_{dom} \{S1[i] \rightarrow Z[i]; S2[j] \rightarrow Y[j]\}$$

- Make Dependence Graph:

$$D = \text{mkDepGraph}(S, R_{access}^{rel}, W_{access}^{rel})$$

## Fusion Encoding (see fusion.py)

- Iteration Domain:

$$I = \{S1[i] : 1 \leq i \leq N; S2[j] : 1 \leq j \leq N\}$$

- Original Schedule:

$$S = I \cap_{dom} \{S1[i] \rightarrow [1, i]; S2[j] \rightarrow [2, j]\}$$

- Read and Write Access Relations:

$$W_{access}^{rel} = I \cap_{dom} \{S1[i] \rightarrow Y[i]; S2[j] \rightarrow X[j]\}$$

$$R_{access}^{rel} = I \cap_{dom} \{S1[i] \rightarrow Z[i]; S2[j] \rightarrow Y[j]\}$$

- Make Dependence Graph:

$$D = \text{mkDepGraph}(S, R_{access}^{rel}, W_{access}^{rel})$$

- Fused Schedule:

$$S' = I \cap_{dom} \{S1[i] \rightarrow [i, 1]; S2[i] \rightarrow [i, 2]\}$$

- Check Fusion Safety:

$$\text{checkTimeDepsPreserved}(S', D)$$

# Parallelism

Is the fused loop parallel?

Fused and Parallel Schedule:



Is the fused loop parallel?

Fused and Parallel Schedule:

$$S' = I \cap_{dom} \{S1[i] \rightarrow [1, 1]; S2[i] \rightarrow [1, 2]\}$$

## Fission Encoding (see fission-wrong.py)

Is it safe to distribute the loop across statements S1 and S2?

```
    for(p=1; p<=N; p++) {  
S1:      Y[p] = f(Z[p]);  
S2:      X[p] = g(Y[p+1]) }  
    ■ Iteration Domain:
```

## Fission Encoding (see fission-wrong.py)

Is it safe to distribute the loop across statements S1 and S2?

```
    for(p=1; p<=N; p++) {  
S1:      Y[p] = f(Z[p]);  
S2:      X[p] = g(Y[p+1]) }
```

- Iteration Domain:

$$I = \{S1[p] : 1 \leq p \leq N; S2[p] : 1 \leq p \leq N\}$$

- Original Schedule:

## Fission Encoding (see fission-wrong.py)

Is it safe to distribute the loop across statements S1 and S2?

```
    for(p=1; p<=N; p++) {  
S1:      Y[p] = f(Z[p]);  
S2:      X[p] = g(Y[p+1]) }
```

- Iteration Domain:

$$I = \{S1[p] : 1 \leq p \leq N; S2[p] : 1 \leq p \leq N\}$$

- Original Schedule:

$$S = I \cap_{dom} \{S1[p] \rightarrow [p, 1]; S2[p] \rightarrow [p, 2]\}$$

- Read and Write Access Relations:

## Fission Encoding (see fission-wrong.py)

Is it safe to distribute the loop across statements S1 and S2?

```
    for(p=1; p<=N; p++) {  
S1:      Y[p] = f(Z[p]);  
S2:      X[p] = g(Y[p+1]) }
```

- Iteration Domain:

$$I = \{S1[p] : 1 \leq p \leq N; S2[p] : 1 \leq p \leq N\}$$

- Original Schedule:

$$S = I \cap_{dom} \{S1[p] \rightarrow [p, 1]; S2[p] \rightarrow [p, 2]\}$$

- Read and Write Access Relations:

$$W_{access}^{rel} = I \cap_{dom} \{S1[p] \rightarrow Y[p]; S2[p] \rightarrow X[p]\}$$

$$R_{access}^{rel} = I \cap_{dom} \{S1[p] \rightarrow Z[p]; S2[p] \rightarrow Y[p+1]\}$$

- $D = \text{mkDepGraph}(S, R_{access}^{rel}, W_{access}^{rel})$
- Fissioned Schedule:

# Fission Encoding (see fission-wrong.py)

Is it safe to distribute the loop across statements S1 and S2?

```
    for(p=1; p<=N; p++) {  
S1:      Y[p] = f(Z[p]);  
S2:      X[p] = g(Y[p+1]) }
```

- Iteration Domain:

$$I = \{S1[p] : 1 \leq p \leq N; S2[p] : 1 \leq p \leq N\}$$

- Original Schedule:

$$S = I \cap_{dom} \{S1[p] \rightarrow [p, 1]; S2[p] \rightarrow [p, 2]\}$$

- Read and Write Access Relations:

$$W_{access}^{rel} = I \cap_{dom} \{S1[p] \rightarrow Y[p]; S2[p] \rightarrow X[p]\}$$

$$R_{access}^{rel} = I \cap_{dom} \{S1[p] \rightarrow Z[p]; S2[p] \rightarrow Y[p+1]\}$$

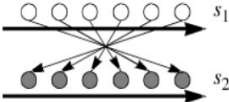
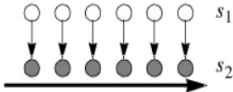
- $D = \text{mkDepGraph}(S, R_{access}^{rel}, W_{access}^{rel})$

- Fission Schedule:

$$S' = I \cap_{dom} \{S1[i] \rightarrow [1, i]; S2[j] \rightarrow [2, j]\}$$

- Is Fission Safe?  $\text{checkTimeDepsPreserved}(S', D)$

# Transformation: Reversal + Fusion

SOURCE CODE	PARTITION	TRANSFORMED CODE
<pre>for (i=0; i&lt;=N; i++)   Y[N-i] = Z[i]; /*s1*/ for (j=0; j&lt;=N; j++)   X[j] = Y[j]; /*s2*/</pre> 	<p>Reversal <math>s_1 : p = N - i</math> <math>(s_2 : p = j)</math></p>	<pre>for (p=0; p&lt;=N; p++){   Y[p] = Z[N-p];   X[p] = Y[p]; }</pre> 

BUG: should be for (i=0; i<=N; i++)!

[Aho/Ullman/Sethi '07]

## Reversal + Fusion Encoding (see fused-rev.py)

- Iteration Domain:



## Reversal + Fusion Encoding (see fused-rev.py)

- Iteration Domain:

$$I = \{S1[i] : 1 \leq i \leq N; S2[j] : 1 \leq j \leq N\}$$

- Original Schedule:

## Reversal + Fusion Encoding (see fused-rev.py)

- Iteration Domain:

$$I = \{S1[i] : 1 \leq i \leq N; S2[j] : 1 \leq j \leq N\}$$

- Original Schedule:

$$S = I \cap_{dom} \{S1[i] \rightarrow [1, i]; S2[j] \rightarrow [2, j]\}$$

- Read and Write Access Relations:

## Reversal + Fusion Encoding (see fused-rev.py)

- Iteration Domain:

$$I = \{S1[i] : 1 \leq i \leq N; S2[j] : 1 \leq j \leq N\}$$

- Original Schedule:

$$S = I \cap_{dom} \{S1[i] \rightarrow [1, i]; S2[j] \rightarrow [2, j]\}$$

- Read and Write Access Relations:

$$W_{access}^{rel} = I \cap_{dom} \{S1[i] \rightarrow Y[N - i]; S2[j] \rightarrow X[j]\}$$

$$R_{access}^{rel} = I \cap_{dom} \{S1[i] \rightarrow Z[i]; S2[j] \rightarrow Y[j]\}$$

- $D = \text{mkDepGraph}(S, R_{access}^{rel}, W_{access}^{rel})$

- Transformed Schedule:

# Reversal + Fusion Encoding (see fused-rev.py)

- Iteration Domain:

$$I = \{S1[i] : 1 \leq i \leq N; S2[j] : 1 \leq j \leq N\}$$

- Original Schedule:

$$S = I \cap_{dom} \{S1[i] \rightarrow [1, i]; S2[j] \rightarrow [2, j]\}$$

- Read and Write Access Relations:

$$W_{access}^{rel} = I \cap_{dom} \{S1[i] \rightarrow Y[N - i]; S2[j] \rightarrow X[j]\}$$

$$R_{access}^{rel} = I \cap_{dom} \{S1[i] \rightarrow Z[i]; S2[j] \rightarrow Y[j]\}$$

- $D = \text{mkDepGraph}(S, R_{access}^{rel}, W_{access}^{rel})$

- Transformed Schedule:

- ▶ the statements of the first loop are reversed;
- ▶ the two loops are fused, hence  $S2[j] \rightarrow [j, 2]$  instead of  $S2[j] \rightarrow [2, j]$

$$S' = I \cap_{dom} \{S1[p] \rightarrow [N - p, 1]; S2[j] \rightarrow [j, 2]\}$$

- Is Fission Safe?  $\text{checkTimeDepsPreserved}(S', D)$

## Transformation: Loop Skewing

```
float X[N][N];  
for(int i=1; i<N; i++) {  
    for(int j=1; j < min(i+2, N); j++) {  
S1:      X[i][j] = X[i-1][j] + X[i][j-1];  
    }  
}
```

Change of variables:  $p \leftarrow i+j, q \leftarrow j$

```
for(int p=2; p < 2*N-1; p++) {  
    int up_bd = ((p+2)/2) + (p%2);  
    for(int q=max(1,p-N+1); q<min(up_bd,N); q++)  
S1:      X[p-q][q] = X[p-q-1][q] + X[p-q][q-1];  
    }  
}
```

## Encoding Loop Skewing (see loop-skewing.py)

- Iteration Domain:

$$I = \{S1[i,j] : 1 \leq i < N \wedge 1 \leq j < \min(i + 2, N)\}$$

- Original Schedule:  $S = I \cap_{dom} \{ S1[i,j] \rightarrow [i,j] \}$
- Read and Write Access Relations:

# Encoding Loop Skewing (see loop-skewing.py)

- Iteration Domain:

$$I = \{S1[i, j] : 1 \leq i < N \wedge 1 \leq j < \min(i + 2, N)\}$$

- Original Schedule:  $S = I \cap_{dom} \{S1[i, j] \rightarrow [i, j]\}$

- Read and Write Access Relations:

$$W_{access}^{rel} = I \cap_{dom} \{S1[i, j] \rightarrow X[i, j];\}$$

$$R_{access}^{rel} = I \cap_{dom} \{S1[i, j] \rightarrow X[i - 1, j]; S1[i, j] \rightarrow X[i, j - 1]\}$$

- $D = \text{mkDepGraph}(S, R_{access}^{rel}, W_{access}^{rel})$

- Transformed Schedule:  $p \leftarrow i + j, q \leftarrow j$

► Original stmt  $S1[i, j] = S1[p - q, q]$  is rescheduled to iter  $[p, q]$ ;

# Encoding Loop Skewing (see loop-skewing.py)

- Iteration Domain:

$$I = \{S1[i, j] : 1 \leq i < N \wedge 1 \leq j < \min(i + 2, N)\}$$

- Original Schedule:  $S = I \cap_{dom} \{ S1[i, j] \rightarrow [i, j] \}$

- Read and Write Access Relations:

$$W_{access}^{rel} = I \cap_{dom} \{ S1[i, j] \rightarrow X[i, j]; \}$$

$$R_{access}^{rel} = I \cap_{dom} \{ S1[i, j] \rightarrow X[i - 1, j]; S1[i, j] \rightarrow X[i, j - 1] \}$$

- $D = \text{mkDepGraph}(S, R_{access}^{rel}, W_{access}^{rel})$

- Transformed Schedule:  $p \leftarrow i+j, q \leftarrow j$

► Original stmt  $S1[i, j] = S1[p - q, q]$  is rescheduled to iter  $[p, q]$ ;

► Hence,  $S1[x, q] \rightarrow [x + q, q]$

$$S' = I \cap_{dom} \{ S1[x, q] \rightarrow [x + q, q] \}$$

- Is Loop-Skewing Safe?  $\text{checkTimeDepsPreserved}(S', D)$

- Inner loop parallel? Try  $S'' = I \cap_{dom} \{ S1[x, q] \rightarrow [x + q, 1] \}$



Motivation: Dependency Graphs and Transformations

Polydral Analysis: Iteration Domain, Access Relations, Schedule

Presburger Sets, Relations and Associated Operations

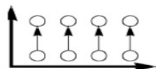
Data-Flow Analysis and Dependency Graph

Examples: Checking Validity of Code Transformations

**Proposed Exercises (Facultative)**

# Exercise: Permutation (Ignore Loop Skewing)

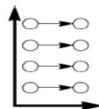
```
for (i=1; i<=N; i++)  
  for (j=0; j<=M; j++)  
    Z[i,j] =  
      Z[i-1,j];
```



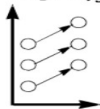
Permutation

$$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}$$

```
for (p=0; p<=M; p++)  
  for (q=1; q<=N; q++)  
    Z[q,p] = Z[q-1,p]
```



```
for (i=1; i<=N+M-1; i++)  
  for (j=max(1,i+N);  
       j<=min(i,M); j++)  
    Z[i,j] =  
      Z[i-1,j-1];
```



Skewing

$$\begin{bmatrix} p \\ q \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

```
for (p=1; p<=N; p++)  
  for (q=1; q<=M; q++)  
    Z[p,q-p] =  
      Z[p-1,q-p-1]
```

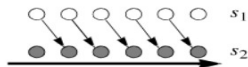


[Aho/Ullman/Sethi '07]

Ignore Loop Skewing

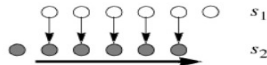
# Exercise: Reindexing and Scaling

```
for (i=1; i<=N; i++) {
    Y[i] = Z[i];    /*s1*/
    X[i] = Y[i-1]; /*s2*/
}
```

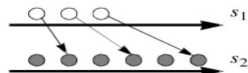


Re-indexing  
 $s_1 : p = i$   
 $s_2 : p = i - 1$

```
if (N>=1) X[1]=Y[0];
for (p=1; p<=N-1; p++){
    Y[p]=Z[p];
    X[p+1]=Y[p];
}
if (N>=1) Y[N]=Z[N];
```

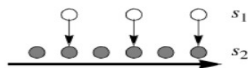


```
for (i=1; i<=N; i++)
    Y[2*i] = Z[2*i]; /*s1*/
for (j=1; j<=2N; j++)
    X[j]=Y[j];      /*s2*/
```



Scaling  
 $s_1 : p = 2 * i$   
 $(s_2 : p = j)$

```
for (p=1; p<=2*N; p++){
    if (p mod 2 == 0)
        Y[p] = Z[p];
    X[p] = Y[p];
}
```



[Aho/Ullman/Sethi '07]