High-Level Loop Transformations and Polyhedral Compilation

Sven Verdoolaege

Polly Labs and KU Leuven (affiliated researcher)

May 30, 2017

Outline

Loop Distribution

```
L: for (int i = 1; i < 100; ++i) {
        A[i] = f(i);
        B[i] = A[i] + A[i - 1];
}
```

Can this loop be parallelized?

Requirement:
writes of iteration do not conflict with reads/writes of other iteration

L[1]: W(A[1]) R(A[1]) R(A[0]) W(B[1])
L[2]: W(A[2]) R(A[2]) R(A[1]) W(B[2])

Loop distribution

```
L1: for (int i = 1; i < 100; ++i)
        A[i] = f(i);
L2: for (int i = 1; i < 100; ++i)
        B[i] = A[i] + A[i - 1];
```

No conflicts between iterations of L1 ⇒ can be run in parallel
No conflicts between iterations of L2 ⇒ can be run in parallel

Loop Distribution

```
L: for (int i = 1; i < 100; ++i) {
        A[i] = f(i);
        B[i] = A[i] + A[i + 1];
}
```

Can this loop be parallelized?

Requirement:
writes of iteration do not conflict with reads/writes of other iteration

L[1]: W(A[1]) R(A[1]) R(A[2]) W(B[1])
L[2]: W(A[2]) R(A[2]) R(A[3]) W(B[2])

Loop distribution changes meaning!

```
L1: for (int i = 1; i < 100; ++i)
        A[i] = f(i);
L2: for (int i = 1; i < 100; ++i)
        B[i] = A[i] + A[i + 1];
```

before distribution, L[1] reads A[2] value written before code fragment
after distribution, L2[1] reads A[2] value written by L1[2]

Loop Fusion

```
L1: for (int i = 0; i < 100; ++i)
        A[i] = f(i);
L2: for (int i = 0; i < 100; ++i)
        B[i] = g(A[i]);
```

Assume A does not fit in the cache
⇒ elements get evicted and reloaded for use in L2

Loop fusion (changes execution order ⇒ may not preserve meaning)

```
for (int i = 0; i < 100; ++i) {
        A[i] = f(i);
        B[i] = g(A[i]);
}
```

- ⇒ elements of A get reused immediately
- ⇒ better locality

Loop Transformations Loop Fusion May 30, 2017 6 / 82

Loop Fusion

```
L1: for (int i = 0; i < 100; ++i)
        A[i] = f(i);
L2: for (int i = 0; i < 100; ++i)
        B[i] = g(A[i]);
```

Assume A does not fit in the cache
⇒ elements get evicted and reloaded for use in L2

Loop fusion (changes execution order ⇒ may not preserve meaning)

```
for (int i = 0; i < 100; ++i) {
        A    = f(i);
        B[i] = g(A   );
}
```

- ⇒ elements of A get reused immediately
- ⇒ better locality

If A not needed outside code fragment

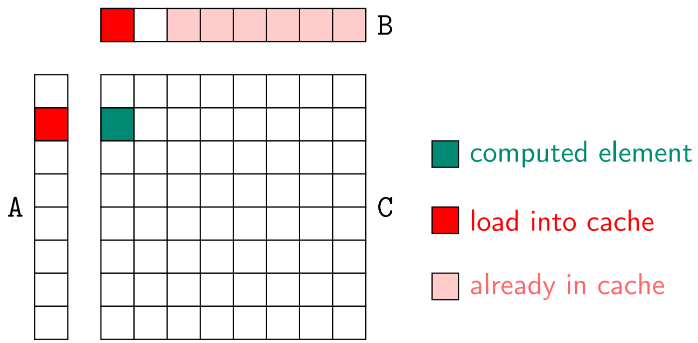
- ⇒ array can be replaced by a scalar
- ⇒ memory compaction

Loop Tiling

[17, 35]

```
L1: for (int i = 0; i < 8; ++i)
L2:     for (int j = 0; j < 8; ++j)
            C[i][j] = A[i] * B[j];
```

Assume B does not fit in the cache
⇒ elements get (re)loaded and evicted in every iteration of L1



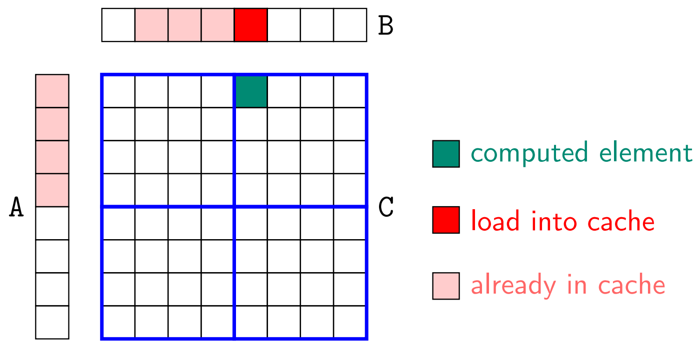
Loop Transformations Loop Tiling May 30, 2017 7 / 82

Loop Tiling

[17, 35]

```
L1: for (int i = 0; i < 8; ++i)
L2:     for (int j = 0; j < 8; ++j)
            C[i][j] = A[i] * B[j];
```

Assume B does not fit in the cache
⇒ elements get (re)loaded and evicted in every iteration of L1



⇒ compute C in tiles, e.g., 4×4

Loop Tiling

[17, 35]

```
L1: for (int i = 0; i < 8; ++i)
L2:     for (int j = 0; j < 8; ++j)
                C[i][j] = A[i] * B[j];
```

Assume `B` does not fit in the cache
⇒ elements get (re)loaded and evicted in every iteration of `L1`

Loop tiling (changes execution order ⇒ may not preserve meaning)

```
for (int ti = 0; ti < 8; ti += 4)
    for (int tj = 0; tj < 8; tj += 4)
        for (int i = ti; i < ti + 4; ++i)
            for (int j = tj; j < tj + 4; ++j)
                C[i][j] = A[i] * B[j];
```


Motivation

- Computer architectures are becoming more difficult to program efficiently
  - multiple levels of parallelism
  - non-uniform memory architectures

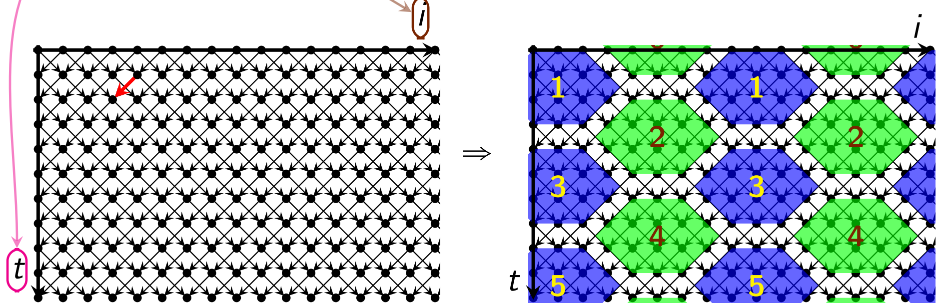
⇒ Advanced compiler optimizations are required
  - hierarchical partitioning and reordering of operations (e.g., parallelization, loop fusion, . . .)
  - mapping to different processing units
  - memory transfers between processing units

⇒ Global view of individual operations is required

⇒ Polyhedral Model

Polyhedral Compilation — Example

```
for (t = 0; t < T; t++)
  for (i = 1; i < N - 1; i++)
    A[(t+1)%2][i] = A[t%2][i-1] + A[t%2][i+1];
```



1. Extract polyhedral model
   ⇒ each dynamic instance represented by (t, i) pair
2. Compute dependences
   ⇒ iteration $t = 2, i = 3$ depends on iteration $t = 1, i = 4$
3. Compute schedule respecting dependences
   ⇒ tiles with same number can be executed in parallel
   ⇒ rows within tiles can be executed in parallel

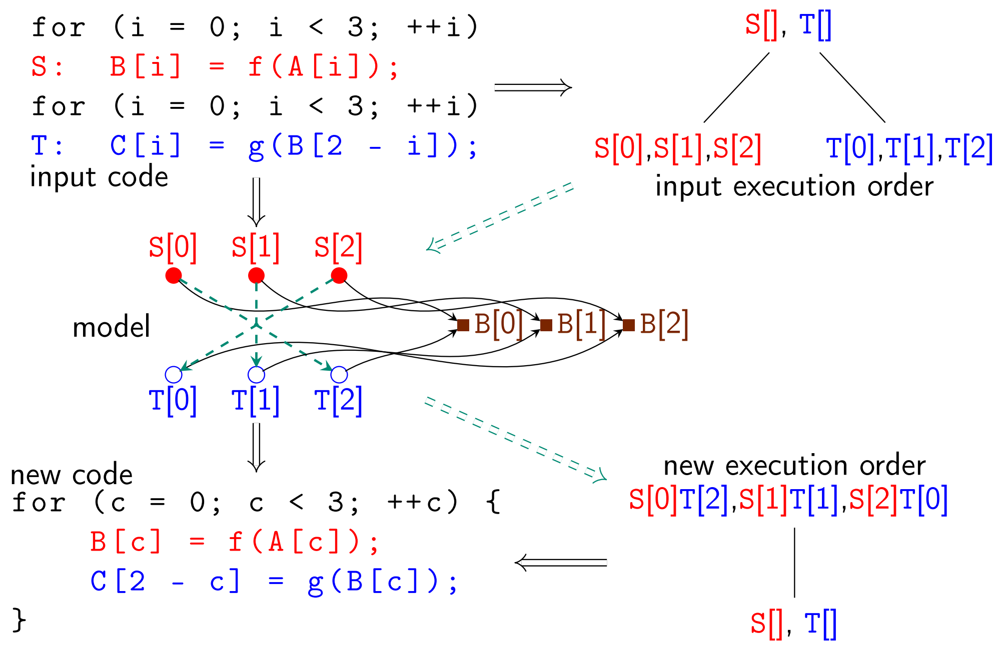
Polyhedral Model

[27]

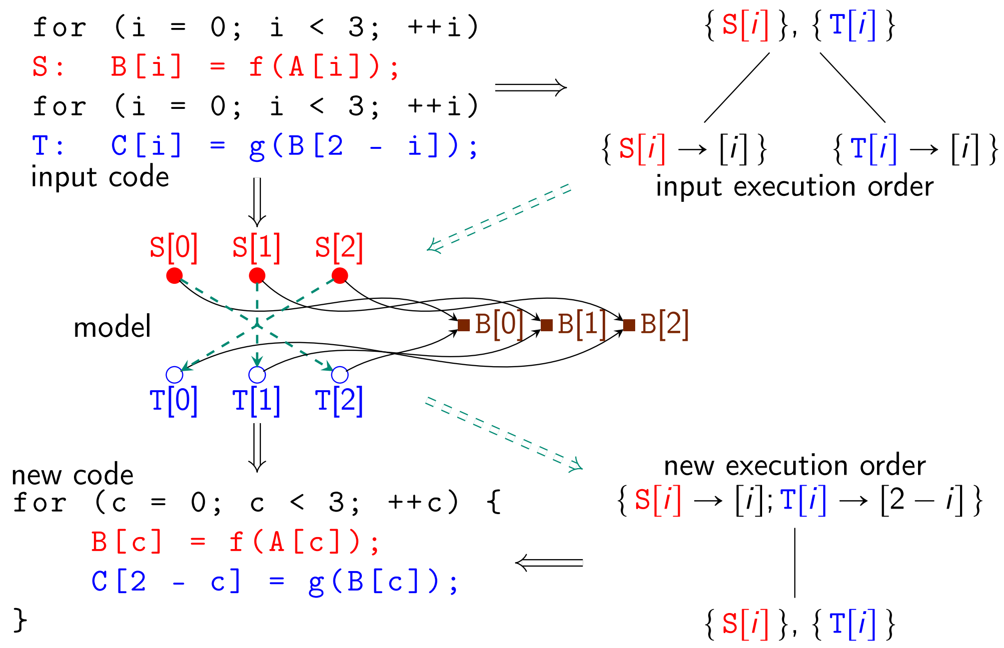
Key features
- instance based
  ⇒ statement *instances*
  ⇒ array *elements*
- compact representation based on polyhedra or similar objects
  ⇒ Presburger sets and relations
  ⇒ . . .

Main constituents of program representation
- Instance Set
  ⇒ the set of all statement instances
- Access Relations
  ⇒ the array elements accessed by a statement instance
- Dependences
  ⇒ the statement instances that depend on a statement instance
- Schedule
  ⇒ the relative execution order of statement instances
- Context
  ⇒ constraints on parameters

Polyhedral Model — Example



Polyhedral Model — Example



Polyhedral Model

[27]

Key features

- instance based
  - ⇒ statement *instances*
  - ⇒ array *elements*
- compact representation based on polyhedra or similar objects
  - ⇒ Presburger sets and relations defined by Presburger formula
  - ⇒ ...
- quasi-affine expression (no multiplication)
  - variable
  - constant integer number
  - constant symbol
  - addition (+), subtraction (−)
  - integer division by integer constant d ($\lfloor\cdot/d\rfloor$)
- Presburger formula
  - true
  - quasi-affine expression
  - less-than-or-equal relation ($\leqslant$)
  - equality ($=$)
  - first order logic connectives: $\wedge$, $\vee$, $\neg$, $\exists$, $\forall$

Parametric Example: Matrix Multiplication

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

- Instance Set (set of statement instances)

$$\{\, \mathtt{S1}[i,j] : 0 \leqslant i < M \wedge 0 \leqslant j < N;\ \mathtt{S2}[i,j,k] : 0 \leqslant i < M \wedge 0 \leqslant j < N \wedge 0 \leqslant k < K \,\}$$

- Access Relations (accessed array elements; W: write, R: read)

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

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

Schedule Representation

[30]

Schedule S keeps track of relative execution order of statement instances

$\Rightarrow$ for each pair of statement instances $\mathbf{i}$ and $\mathbf{j}$, schedule determines

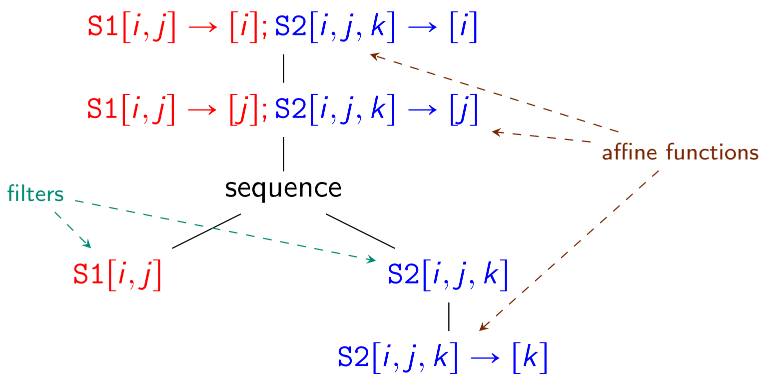
- $\mathbf{i}$ executed before $\mathbf{j}$ ($\mathbf{i} <_S \mathbf{j}$),
- $\mathbf{i}$ executed after $\mathbf{j}$ ($\mathbf{j} <_S \mathbf{i}$), or
- $\mathbf{i}$ and $\mathbf{j}$ may be executed simultaneously

Schedule trees form a combined hierarchical schedule representation

- Main constructs:
  - affine schedule: instances are executed according to affine function
  - *sequence*: partitions instances through child *filters* executed in order
- Order of instances determined by outermost node that separates them
- Deriving schedule tree from AST
  - `for` loop $\Rightarrow$ affine schedule corresponding to loop iterator
  - compound statement $\Rightarrow$ sequence

Parametric Example: Matrix Multiplication

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



Schedule Representation

[30]

Schedule S keeps track of relative execution order of statement instances

$\Rightarrow$ for each pair of statement instances $\mathbf{i}$ and $\mathbf{j}$, schedule determines

- $\mathbf{i}$ executed before $\mathbf{j}$ ($\mathbf{i} <_S \mathbf{j}$),
- $\mathbf{i}$ executed after $\mathbf{j}$ ($\mathbf{j} <_S \mathbf{i}$), or
- $\mathbf{i}$ and $\mathbf{j}$ may be executed simultaneously

Schedule trees form a combined hierarchical schedule representation

- Main constructs:
  - affine schedule: instances are executed according to affine function
  - *band*: nested sequence of affine functions called its *members*; combined multi-dimensional affine function is called the *partial schedule* of the band
  - *sequence*: partitions instances through child *filters* executed in order
- Order of instances determined by outermost node that separates them
- Deriving schedule tree from AST
  - `for` loop $\Rightarrow$ affine schedule corresponding to loop iterator
  - compound statement $\Rightarrow$ sequence

Named Presburger Relation Schedules

Schedule tree with single (band) node

Flattening a schedule tree

- two nested band nodes
  - $\Rightarrow$ replace by single band node with concatenated partial schedule
- sequence with as children either leaves or trees consisting of a single band node
  - $\Rightarrow$ treat leaves as zero-dimensional band nodes
  - $\Rightarrow$ pad lower-dimensional bands (e.g., with zero)
  - $\Rightarrow$ construct one-dimensional band assigning increasing values to children
  - $\Rightarrow$ combine one-dimensional band with children

Parametric Example: Matrix Multiplication

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

$$\texttt{S1}[i,j] \to [i,j,0,0]; \texttt{S2}[i,j,k] \to [i,j,1,k]$$

Loop Transformations and the Polyhedral Model

Loop transformations result in
different execution order of statement instances
⇒ different schedule

Polyhedral model can be used to

- evaluate a schedule and/or
- construct a schedule

Polyhedral schedules can represent (combinations of)

- loop distribution
- loop fusion
- loop tiling
- ...

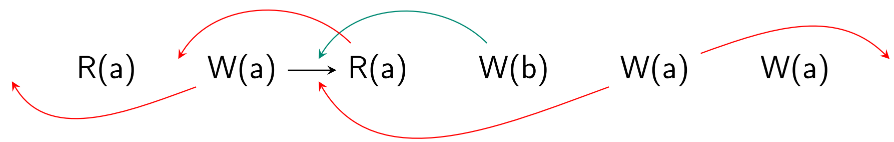
Schedule Properties

- Validity
  New schedule should preserve meaning
- Parallelism
  Can the iterations of a given loop be executed in parallel?
- Locality
  Statement instances scheduled closely to each other
- Tilability
  Can a given schedule band be tiled?

Schedule Validity

[3]

New schedule should preserve meaning



Internal restrictions

- No read of a value may be scheduled before the write of the value
- No other write to same memory location may be scheduled in between

External restrictions (on non-temporaries)

- No write may be scheduled before initial read from a memory location
- No write may be scheduled after last write to a memory location

Sufficient conditions:

- Every read of a memory location is scheduled after every preceding write to the same memory location
- Every write to a memory location is scheduled after every preceding read or write to the same memory location

Dependences

Sufficient conditions for validity of schedule S:

- Every read of a memory location is scheduled after every preceding write to the same memory location
- Every write to a memory location is scheduled after every preceding read or write to the same memory location

Dependence relation D: pairs of statement instances

- accessing the same memory location
- of which at least one is a write
- with the first executed before the second in original code

Sufficient condition:

$$\forall \mathbf{i} \to \mathbf{j} \in D : \mathbf{i} <_S \mathbf{j}$$

Dependence Analysis

Recall: sufficient conditions for validity of schedule S:

$$\forall \mathbf{i} \to \mathbf{j} \in D : \mathbf{i} <_S \mathbf{j}$$

Dependence relation D: pairs of statement instances

- accessing the same memory location
- of which at least one is a write
- with the first executed before the second in original code

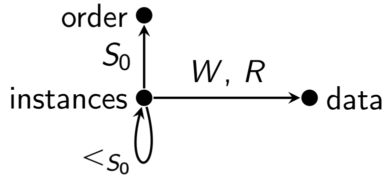
Computation:

$$D = ((W^{-1} \circ R) \cup (W^{-1} \circ W) \cup (R^{-1} \circ W)) \cap (<_{S_0})$$

W: write access relation

R: read access relation

S_0: original schedule



Local Validity

Schedule validity:

$$\forall \mathbf{i} \to \mathbf{j} \in D : \mathbf{i} <_S \mathbf{j}$$

Consider subset of *local* dependences L

At outermost node: $L = D$

Current node

- band node with partial schedule f

  $$\forall \mathbf{i} \to \mathbf{j} \in L : f(\mathbf{i}) \preccurlyeq_{lex} f(\mathbf{j})$$

  Carried dependences: $\mathbf{i} \to \mathbf{j} \in L : f(\mathbf{i}) \neq f(\mathbf{j})$
  $\Rightarrow$ no longer need to be considered in nested nodes
  Remaining dependences: $L' = \{\, \mathbf{i} \to \mathbf{j} \in L : f(\mathbf{i}) = f(\mathbf{j}) \,\}$
- sequence node with child position p and filters F_k

  $$\forall \mathbf{i} \to \mathbf{j} \in L : p(\mathbf{i}) \leqslant p(\mathbf{j})$$

  Carried dependences: $\mathbf{i} \to \mathbf{j} \in L : p(\mathbf{i}) \neq p(\mathbf{j})$
  Remaining dependences in child c: $L' = \{\, \mathbf{i} \to \mathbf{j} \in L : \mathbf{i}, \mathbf{j} \in F_c \,\}$
- leaf node: $L = \varnothing$

Loop Distribution Validity

```
for (int i = 1; i < 100; ++i) {
S:      A[i] = f(i);
T:      B[i] = A[i] + A[i - 1];
}
```

$\{\, \mathtt{S}[i] \to [i]; \mathtt{T}[i] \to [i] \,\}$

$\{\, \mathtt{S}[i] \,\}, \{\, \mathtt{T}[i] \,\}$

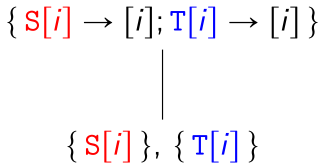
Dependences:

$$\{\, \mathtt{S}[i] \to \mathtt{T}[i] : 1 \leqslant i < 100; \mathtt{S}[i] \to \mathtt{T}[i+1] : 1 \leqslant i, i+1 < 100 \,\}$$

$\{\, \mathtt{S}[i] \to [i]; \mathtt{T}[i] \to [i] \,\}$
satisfied: $\{\, \mathtt{S}[i] \to \mathtt{T}[i] : 1 \leqslant i < 100; \mathtt{S}[i] \to \mathtt{T}[i+1] : 1 \leqslant i, i+1 < 100 \,\}$
carried: $\{\, \mathtt{S}[i] \to \mathtt{T}[i+1] : 1 \leqslant i, i+1 < 100 \,\}$
$\{\, \mathtt{S}[i] \,\}, \{\, \mathtt{T}[i] \,\}$
satisfied: $\{\, \mathtt{S}[i] \to \mathtt{T}[i] : 1 \leqslant i < 100 \,\}$
carried: $\{\, \mathtt{S}[i] \to \mathtt{T}[i] : 1 \leqslant i < 100 \,\}$

Loop Distribution Validity

```
for (int i = 1; i < 100; ++i) {
S:        A[i] = f(i);
T:        B[i] = A[i] + A[i - 1];
}
```

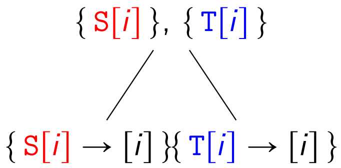


Dependences:

$$\{ \mathtt{S}[i] \to \mathtt{T}[i] : 1 \leqslant i < 100; \mathtt{S}[i] \to \mathtt{T}[i+1] : 1 \leqslant i, i+1 < 100 \}$$

Loop distribution

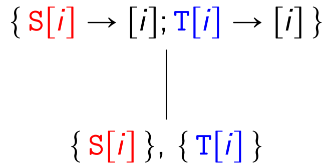
```
for (int i = 1; i < 100; ++i)
          A[i] = f(i);
for (int i = 1; i < 100; ++i)
          B[i] = A[i] + A[i - 1];
```



$\{ \mathtt{S}[i] \}, \{ \mathtt{T}[i] \}$
satisfied: $\{ \mathtt{S}[i] \to \mathtt{T}[i] : 1 \leqslant i < 100; \mathtt{S}[i] \to \mathtt{T}[i+1] : 1 \leqslant i, i+1 < 100 \}$
carried: $\{ \mathtt{S}[i] \to \mathtt{T}[i] : 1 \leqslant i < 100; \mathtt{S}[i] \to \mathtt{T}[i+1] : 1 \leqslant i, i+1 < 100 \}$

Loop Distribution Validity

```
for (int i = 1; i < 100; ++i) {
S:        A[i] = f(i);
T:        B[i] = A[i] + A[i + 1];
}
```

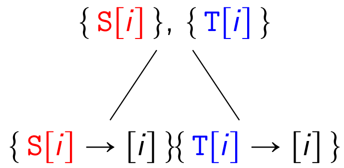


Dependences:

$$\{ \mathtt{S}[i] \to \mathtt{T}[i] : 1 \leqslant i < 100; \mathtt{T}[i] \to \mathtt{S}[i+1] : 1 \leqslant i, i+1 < 100 \}$$

Loop distribution

```
for (int i = 1; i < 100; ++i)
          A[i] = f(i);
for (int i = 1; i < 100; ++i)
          B[i] = A[i] + A[i + 1];
```



$\{ \mathtt{S}[i] \}, \{ \mathtt{T}[i] \}$
satisfied: $\{ \mathtt{S}[i] \to \mathtt{T}[i] : 1 \leqslant i < 100 \}$
violated: $\{ \mathtt{T}[i] \to \mathtt{S}[i+1] : 1 \leqslant i, i+1 < 100 \}$

Parallel Loops and Parallel Band Members

Recall:
Iterations of a given loop can be executed in parallel if
writes of iteration do not conflict with reads/writes of other iteration
iff there is no dependence between distinct iterations
(for any given iteration of the outer loops)

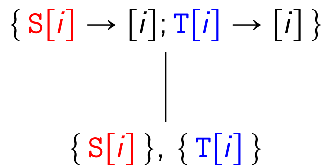
A band member with affine function f is parallel if

$$\forall \mathbf{i} \to \mathbf{j} \in L : f(\mathbf{i}) = f(\mathbf{j})$$

with L the local dependences

Loop Distribution and Parallelism

```
for (int i = 1; i < 100; ++i) {
S:        A[i] = f(i);
T:        B[i] = A[i] + A[i - 1];
}
```



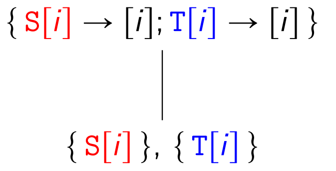
Dependences:

$$\{ \mathtt{S}[i] \to \mathtt{T}[i] : 1 \leqslant i < 100; \mathtt{S}[i] \to \mathtt{T}[i+1] : 1 \leqslant i, i+1 < 100 \}$$

$\{ \mathtt{S}[i] \to [i]; \mathtt{T}[i] \to [i] \}$
local: $\{ \mathtt{S}[i] \to \mathtt{T}[i] : 1 \leqslant i < 100; \mathtt{S}[i] \to \mathtt{T}[i+1] : 1 \leqslant i, i+1 < 100 \}$
conflict: $\{ \mathtt{S}[i] \to \mathtt{T}[i+1] : 1 \leqslant i, i+1 < 100 \}$
$\Rightarrow$ not parallel

Loop Distribution and Parallelism

```
for (int i = 1; i < 100; ++i) {
S:        A[i] = f(i);
T:        B[i] = A[i] + A[i - 1];
}
```

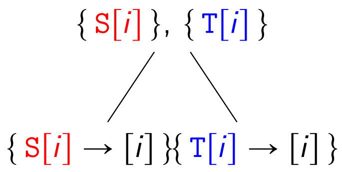


Dependences:

$$\{ \mathtt{S}[i] \to \mathtt{T}[i] : 1 \leqslant i < 100; \mathtt{S}[i] \to \mathtt{T}[i+1] : 1 \leqslant i, i+1 < 100 \}$$

Loop distribution

```
for (int i = 1; i < 100; ++i)
          A[i] = f(i);
for (int i = 1; i < 100; ++i)
          B[i] = A[i] + A[i - 1];
```



$\{ \mathtt{S}[i] \to [i] \}$
local: $\varnothing$
conflict: $\varnothing$
⇒ parallel

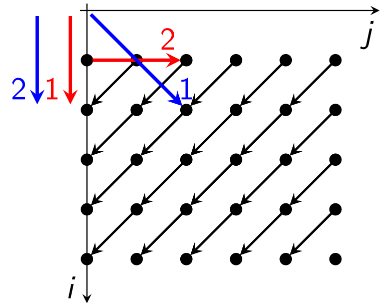
$\{ \mathtt{T}[i] \to [i] \}$
local: $\varnothing$
conflict: $\varnothing$
⇒ parallel

Parallelism Example

```
for (int i = 1; i < 6; ++i)
    for (int j = 0; j < 6; ++j)
S:        A[i][j] = f(A[i - 1][[j + 1]);
```

Dependences:

$$\{ \mathtt{S}[i,j] \to \mathtt{S}[i+1, j-1] : 1 \leqslant i, i+1 < 6 \wedge 0 \leqslant j, j-1 < 6 \}$$



original schedule:
$\mathtt{S}[i,j] \to [i,j]$
new schedule:
$\mathtt{S}[i,j] \to [i+j, i]$
$(i+j)$-direction is outer parallel

Decomposition: loop skewing + loop interchange

$[i,j] \to [i, i+j] \to [i+j, i]$

Locality

Statement instances **i** and **j** that reuse memory
⇒ scheduled closely to each other: $f(\mathbf{j}) - f(\mathbf{i})$ small

Types of locality:

- temporal locality
  ⇒ instances that access the same memory element
- spatial locality
  ⇒ instances that access adjacent memory elements

Sometimes further distinction made:

- self locality
  ⇒ pair of instances from same statement
- group locality
  ⇒ any pair of statement instances

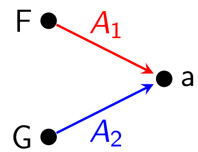
Temporal locality often restricted to
pairs of writes and reads that refer to the same value
⇒ dataflow

Array Dataflow Analysis

[14]

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

```
for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
F:        a[i+j] = f(a[i+j]);
for (i = 0; i < N; ++i)
G:  g(a[i]);
```



Access relations:
$A_1 = \{ \mathtt{F}[i,j] \to \mathrm{a}[i+j] : 0 \leqslant i < N \wedge 0 \leqslant j < N - i \}$
$A_2 = \{ \mathtt{G}[i] \to \mathrm{a}[i] : 0 \leqslant i < N \}$
Map to all writes: $R'' = A_1{}^{-1} \circ A_2 = \{ \mathtt{G}[i] \to \mathtt{F}[i', i - i'] : 0 \leqslant i' \leqslant i < N \}$
Map to all preceding writes:
$R' = R'' \cap (<_S)^{-1} = \{ \mathtt{G}[i] \to \mathtt{F}[i', i - i'] : 0 \leqslant i' \leqslant i < N \}$
Last preceding write: $R = \max_{<_S} R' = \{ \mathtt{G}[i] \to \mathtt{F}[i, 0] : 0 \leqslant i < N \}$

Tiling a Band

Input:

- band of affine schedule functions

$$f_1, f_2, \ldots, f_n$$

- tile sizes

$$T_1, T_2, \ldots, T_n$$

Steps (conceptually)

1. divide each direction into chunks of size T_i (strip-mining)

$$\lfloor f_1/T_1 \rfloor, f_1, \lfloor f_2/T_2 \rfloor, f_2, \ldots, \lfloor f_n/T_n \rfloor, f_n$$

   does not change execution order ⇒ always valid
2. combine the chunking (interchange)

$$\lfloor f_1/T_1 \rfloor, \lfloor f_2/T_2 \rfloor, \ldots, \lfloor f_n/T_n \rfloor, f_1, f_2, \ldots, f_n$$

   sufficient condition for interchange:
   all members are valid for local dependences at (top of) band
   ⇒ permutable band

Loop Tiling Example

```
for (int i = 0; i < 8; ++i)
    for (int j = 0; j < 8; ++j)
S:      C[i][j] = A[i] * B[j];
```

1. strip-mine

$$S[i,j] \to 4\lfloor i/4 \rfloor$$
$$S[i,j] \to i$$
$$S[i,j] \to 4\lfloor j/4 \rfloor$$
$$S[i,j] \to j$$

```
for (int ti = 0; ti < 8; ti += 4)
    for (int i = ti; i < ti + 4; ++i)
        for (int tj = 0; tj < 8; tj += 4)
            for (int j = tj; j < tj + 4; ++j)
                C[i][j] = A[i] * B[j];
```


Loop Tiling Example

```
for (int i = 0; i < 8; ++i)
    for (int j = 0; j < 8; ++j)
S:      C[i][j] = A[i] * B[j];
```

1. strip-mine
2. interchange

$$S[i,j] \to 4\lfloor i/4 \rfloor$$
$$S[i,j] \to 4\lfloor j/4 \rfloor$$
$$S[i,j] \to i$$
$$S[i,j] \to j$$

```
for (int ti = 0; ti < 8; ti += 4)
    for (int tj = 0; tj < 8; tj += 4)
        for (int i = ti; i < ti + 4; ++i)
            for (int j = tj; j < tj + 4; ++j)
                C[i][j] = A[i] * B[j];
```


Operations on Polyhedral Model

- Model Extraction
  - Input: AST
  - Output: instance set, access relations, original schedule
- Dependence analysis
  - Input: instance set, access relations, original schedule
  - Output: dependence relations
- Scheduling
  - Input: instance set, dependence relations
  - Output: schedule
- AST generation (polyhedral scanning, code generation)
  - Input: instance set, schedule
  - Output: AST
- Data layout transformations
  - Input: access relations, dependence relations
  - Output: transformed access relations

Polyhedral Model Requirements

Requirements for basic polyhedral model: "regular" code

- Static control
  ⇒ control does not depend on input data
- Affine
  ⇒ all relevant expressions are (quasi-)affine
- No Aliasing
  ⇒ essentially no pointer manipulations

Note:

- polyhedral model may be *approximation* of input that does not strictly satisfy all requirements
- many *extensions* are available

Aliasing

[1]

Some possible ways of handling aliasing:

- use an input language that does not permit aliasing
- pretend the problem does not exist
- require user to ensure absence of aliasing
  ⇒ e.g., use `restrict` keyword
- handle as may-write
  ⇒ may lead to too many dependences
- check aliasing at run-time
  ⇒ use original code in case of aliasing

Polyhedral Scheduling

[10, 15]

Polyhedral model can be used to

- evaluate a schedule and/or
- construct a schedule

Some popular polyhedral schedulers:

- Feautrier
  - maximal inner parallelism
    ⇒ carry as many dependences as possible at outer bands
- Pluto
  - tilable bands
  - locality: $f(\mathbf{j}) - f(\mathbf{i})$ small
    ⇒ parallelism as extreme case: $f(\mathbf{j}) - f(\mathbf{i}) = 0$

Many other scheduling algorithms have been proposed

Data layout transformations

[12, 13]

- Memory compaction
  Reuse memory locations to store different data
  - ⇒ apply non-injective mapping to array elements
  - ⇒ reduce memory requirements
  - ⇒ extreme case: replace array by scalar

```
for (int i = 0; i < 100; ++i) {
        A[i] = f(i);
        B[i] = g(A[i]);
}
```

- Expansion
  Use different memory locations to store different data
  - ⇒ map different accesses to memory element to distinct locations
  - ⇒ increase scheduling freedom (e.g., more parallelism)

False Dependences

```
for (int i = 0; i < n; ++i) {
S:      t = f1(A[i]);
T:      B[i] = f2(t);
}
```

Dependences

- read-after-write ("true"): $\{ S[i] \to T[i'] : i' \geqslant i \}$
- write-after-read ("anti"): $\{ T[i] \to S[i'] : i' > i \}$ } "false"
- write-after-write ("output"): $\{ S[i] \to S[i'] : i' > i \}$ } "false"

False dependences not from dataflow, but from reuse of memory location t

Possible solution: expansion/privatization

```
for (int i = 0; i < n; ++i) {
S:      t[i] = f1(A[i]);
T:      B[i] = f2(t[i]);
}
```

- dataflow (subset of "true" dependences): $\{ S[i] \to T[i] \}$

Expansion

Assume:

- instance sets and access relations are static and exact
  ⇒ each read has exactly one corresponding write
- single read and write per statement
  ⇒ expanded array indexed by statement instance of write

```
for (int i = 0; i < n; ++i) {
S:      t = f1(A[i]);
T:      B[i] = f2(t);
}
```

Dataflow: $\{ S[i] \to T[i] \}$

```
for (int i = 0; i < n; ++i) {
S:      S[i] = f1(A[i]);
T:      B[i] = f2(S[i]);
}
```

⇒ only remaining dependences are dataflow induced

Maximal Static Expansion

[5]

```
for (int i = 0; i < n; ++i) {
S1:     t = f1(i);                t1[i] = f1(i);
S2:     A[i] = t;                 A[i] = t1[i];
S3:     t = f2(i);                t2[i] = f2(i);
S4:     if (f3(i))                if (f3(i))
S5:             t = f4(i);                t2[i] = f4(i);
S6:     B[i] = t;                 B[i] = t2[i];
}
```

Dataflow cannot be determined independently of run-time information

⇒ approximate dataflow
$\{ S1[i] \to S2[i]; S3[i] \to S6[i]; S5[i] \to S6[i] \}$

⇒ a read may be associated to more than one write

⇒ corresponding equivalence classes should not be expanded apart

Approximate Dataflow Analysis

How to compute dataflow in presence of data dependent control?

Two approaches

- Direct computation
  - distinguish between may- and must-writes
- Derived from exact run-time dependent dataflow
  - compute exact dataflow in terms of run-time information
  - exploit properties of run-time information
  - project out run-time information

May Writes

Keep track of whether write is possible or definite

- Must-writes
  Array elements are definitely written by statement instance
- May-writes
  Array elements are possibly written by statement instance
  - statement instance not necessarily executed

```
for (i = 0; i < n; ++i)
     if (A[i] > 0)
S:        B[i] = A[i];
```

    May-write: $\{ \texttt{S}[i] \to \texttt{B}[i] \}$
  - array element not necessarily accessed

```
int A[N];
/* ... */
T:  A[B[0]] = 5;
```

    May-write: $\{ \texttt{T}[] \to \texttt{A}[a] : 0 \leqslant a < N \}$

Must-write access relation is subset of may-write access relation

Approximate Dataflow — Direct Computation

- Read-after-write dependences
  - write and read access same memory location
  - write executed before the read

  ⇒ Approximate dataflow analysis with no must-writes
- Dataflow dependences
  - write and read access same memory location
  - write executed before the read
  - no intermediate write to same memory location
    ⇒ intermediate write kills dependence
- Approximate dataflow dependences
  - may-write and read access same memory location
  - may-write executed before the read
  - no intermediate must-write to same memory location
    ⇒ intermediate must-write kills dependence

Approximate Dataflow Analysis

How to compute dataflow in presence of data dependent control?

Two approaches

- Direct computation
  - distinguish between may- and must-writes
- Derived from exact run-time dependent dataflow
  - compute exact dataflow in terms of run-time information
  - exploit properties of run-time information
  - project out run-time information

Run-time Dependent Dataflow Analysis

[6, 32]

Approaches

- "fuzzy array dataflow analysis"
- "on-demand-parametric array dataflow analysis"

```
for (int i = 0; i < n; ++i) {
S1:     t = f1(i);
S2:     A[i] = t;
S3:     t = f2(i);
S4:     if (f3(i))
S5:             t = f4(i);
S6:     B[i] = t;
}
```

- Run-time dependent dataflow
  $\{ \texttt{S1}[i] \to \texttt{S2}[i]; \texttt{S3}[i] \to \texttt{S6}[i] : \beta_{\texttt{S6}}^{\texttt{S5}} = 0; \texttt{S5}[i] \to \texttt{S6}[i] : \beta_{\texttt{S6}}^{\texttt{S5}} = 1 \}$
  β_C^P: any potential source instance P is executed for sink C
  $\boldsymbol{\lambda}_C^P$: last potential source instance P executed for sink C
- Approximate dataflow (project out β and $\boldsymbol{\lambda}$)
  $\{ \texttt{S1}[i] \to \texttt{S2}[i]; \texttt{S3}[i] \to \texttt{S6}[i]; \texttt{S5}[i] \to \texttt{S6}[i] \}$

Representing Dynamic Conditions

```
N1: n = f();
    for (int k = 0; k < 100; ++k) {
M:      m = g();
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < n; ++j)
A:              a[j][i] = g();
N2:     n = f();
    }
```

What is instance set (restricted to A statement)?
$\{ \texttt{A}[k,i,j] : 0 \leqslant k < 100 \wedge 0 \leqslant i < \texttt{m} \wedge 0 \leqslant j < \texttt{n} \}$?

⇒ no, m and n cannot be treated as symbolic constants
(they are modified inside k-loop)

$\{ \texttt{A}[k,i,j] : 0 \leqslant k < 100 \wedge 0 \leqslant i < \texttt{valueOf\_m}(k) \wedge 0 \leqslant j < \texttt{valueOf\_n}(k) \}$?

⇒ requires uninterpreted functions (of arity > 0)

Alternative: use overapproximation of instance set and keep track of which elements are executed

Representing Dynamic Conditions

```
N1: n = f();
    for (int k = 0; k < 100; ++k) {
M:      m = g();
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < n; ++j)
A:              a[j][i] = g();
N2:     n = f();
    }
```

- Instance set: $\{ \texttt{A}[k,i,j] : 0 \leqslant k < 100 \wedge 0 \leqslant i \wedge 0 \leqslant j \}$
- Filter:
  - Filter access relations: reader → [writer → array element]
    - $F_1^{\texttt{A}} = \{ \texttt{A}[k,i,j] \to [\texttt{M}[k] \to \texttt{m}[]] \}$
    - $F_2^{\texttt{A}} = \{ \texttt{A}[0,i,j] \to [\texttt{N1}[] \to \texttt{n}[]]; \texttt{A}[k,i,j] \to [\texttt{N2}[k-1] \to \texttt{n}[]] : k \geqslant 1 \}$
  - Filter value relation:
    $V^{\texttt{A}} = \{ \texttt{A}[k,i,j] \to [m,n] : 0 \leqslant k \leqslant 99 \wedge 0 \leqslant i < m \wedge 0 \leqslant j < n \}$

Statement instance is executed iff values written by corresponding write accesses (through filter access relations) satisfy filter value relation

Parametric Array Dataflow Analysis

```
while (1) {
N:  n = f();
    a = g();
    if (n < 100)
H:      a = h();
    if (n > 200)
T:      t(a);
}
```

potential source (H: a); sink (T: a)

$I = \{ \texttt{H}[i] : i \geqslant 0; \texttt{T}[i] : i \geqslant 0 \}$
$F^{\texttt{H}} = \{ \texttt{H}[i] \to [\texttt{N}[i] \to \texttt{n}[]] \}$
$V^{\texttt{H}} = \{ \texttt{H}[i] \to [n] : i \geqslant 0 \wedge n < 100 \}$
$F^{\texttt{T}} = \{ \texttt{T}[i] \to [\texttt{N}[i] \to \texttt{n}[]] \}$
$V^{\texttt{T}} = \{ \texttt{T}[i] \to [n] : i \geqslant 0 \wedge n > 200 \}$

Is there any dataflow between potential source and sink at inner level?

- $M = \{ \texttt{T}[i] \to \texttt{H}[i] \}$
- $F^{\texttt{H}} \circ M \subseteq F^{\texttt{T}}$
  - ⇒ filter elements accessed by any potential source instance associated to sink instance forms subset of filter elements accessed by sink instance
  - ⇒ constraints on filter values at sink also apply at corresponding potential source: $V^{\texttt{T}} \circ M^{-1} = \{ \texttt{H}[i] \to [n] : i \geqslant 0 \wedge n > 200 \}$
- $(V^{\texttt{T}} \circ M^{-1}) \cap V^{\texttt{H}} = \emptyset$
  - ⇒ there can be no dataflow at inner level

Polyhedral Process Networks

[24]

- Main purpose: extract task level parallelism from dataflow graph

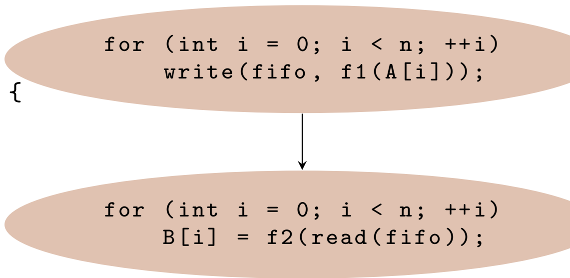
statement	→	process
flow dependence	→	communication channel

⇒ requires dataflow analysis

- Processes are mapped to parallel hardware (e.g., FPGA)

Example:

```
for (int i = 0; i < n; ++i) {
S:      t = f1(A[i]);
T:      B[i] = f2(t);
}
```



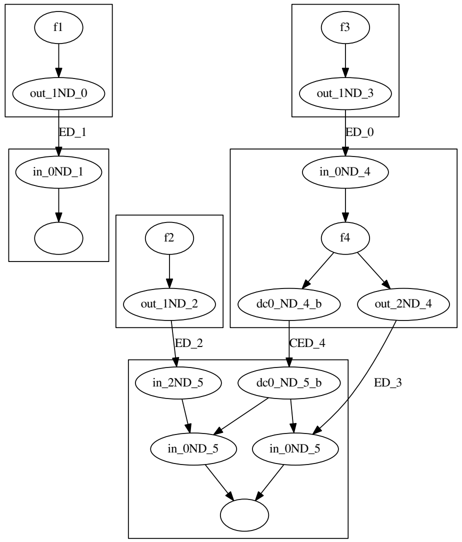
Process Networks with Dynamic Control

```
for (int i = 0; i < n; ++i) {
S1:     t = f1(i);
S2:     A[i] = t;
S3:     t = f2(i);
S4:     if (f3(i))
S5:             t = f4(i);
S6:     B[i] = t;
}
```

Run-time dependent dataflow:

$\{ \mathtt{S1}[i] \to \mathtt{S2}[i]; \mathtt{S3}[i] \to \mathtt{S6}[i] : \beta^{\mathtt{S5}}_{\mathtt{S6}} = 0;$

$\mathtt{S5}[i] \to \mathtt{S6}[i] : \beta^{\mathtt{S5}}_{\mathtt{S6}} = 1; \mathtt{S4}[i] \to \mathtt{S5}[i] \}$



Polyhedral Software

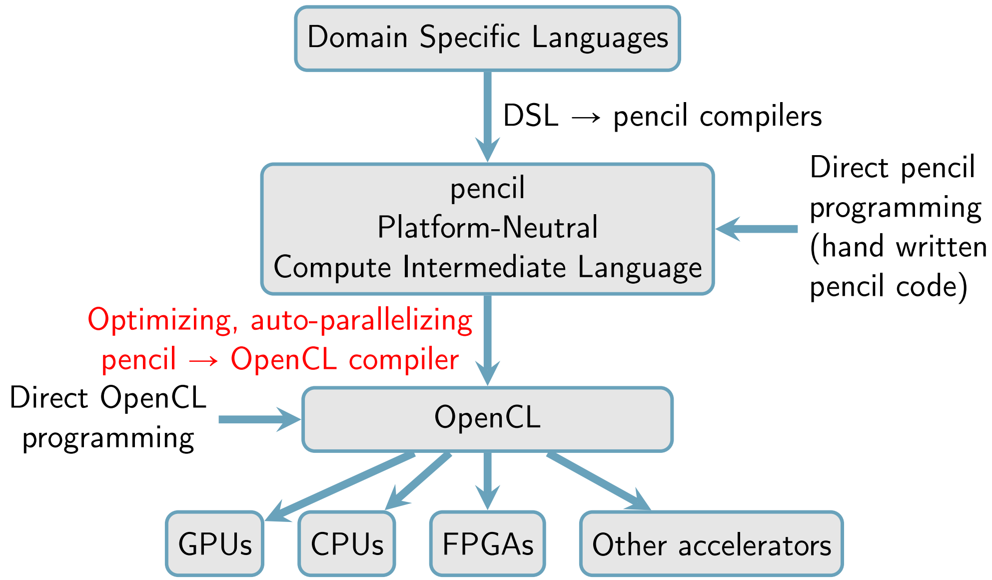
[4, 7, 8, 9, 10, 11, 16, 18, 19, 20, 21, 22, 23, 29, 31, 34]

http://polyhedral.info/software.html

- Core set manipulation libraries
  - integer sets: isl, omega(+), ...
  - rational sets: PolyLib, PPL, ...
- Model extraction
  - clan, pet, ...
- Dependence analysis
  - petit, candl, isl, FADA, ...
- Scheduler libraries
  - LetSee, isl, ...
- AST generation
  - omega(+), CLooG, isl, ...
- Source-to-source polyhedral compilers
  - Pluto, PoCC, PPCG, ...
- Compilers using polyhedral compilation
  - gcc/graphite, LLVM/Polly, ...

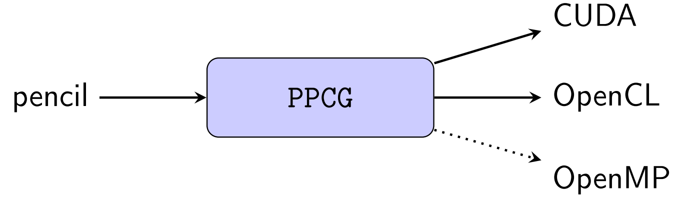
CARP Project (2011–2015)

Design tools and techniques to aid
Correct and Efficient Accelerator Programming



PPCG Overview

[31]



PPCG:

- detect/expose parallelism
- map parts of the code to an accelerator
- copy data to/from device
- introduce local copies of data

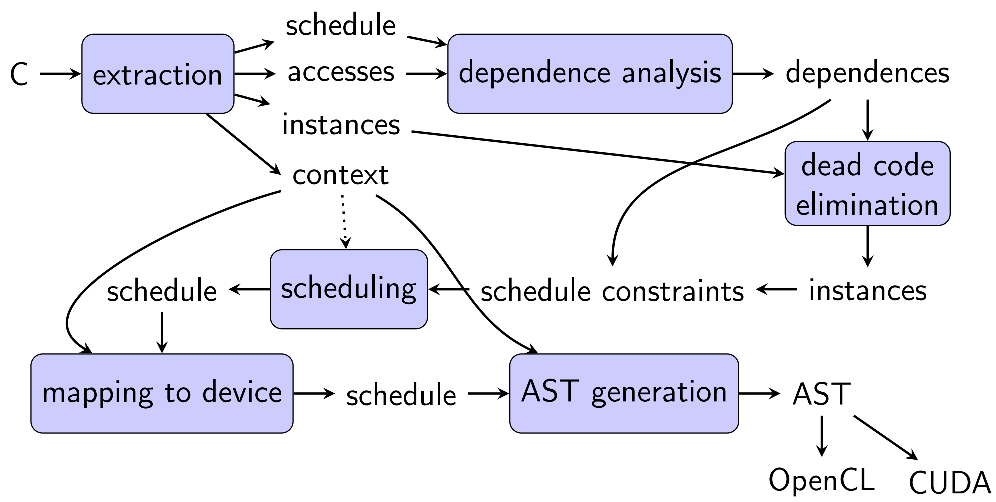
pencil:

- C99 with restrictions and some extra builtins and pragmas

pencil

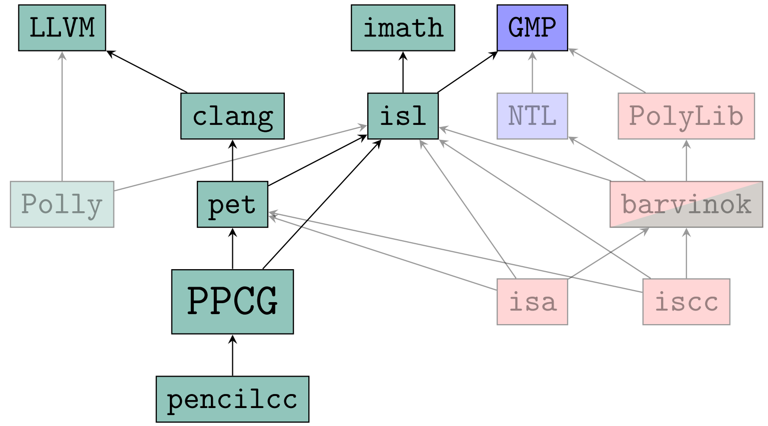
PPCG Internal Structure

[31]



Note: as currently implemented (version 0.07), not necessarily how it should be implemented

Connection with other Libraries and Tools



Licenses:
BSD/MIT
LGPL
GPL

`isl`: manipulates parametric affine sets and relations
`pet`: extracts polyhedral model from clang AST
`PPCG`: Polyhedral Parallel Code Generator
`pencilcc`: pencil compiler

Instance Set

Region that needs to be extracted may be

- marked by

```
#pragma scop
#pragma endscop
```

- autodetected (`--pet-autodetect`)

Internal structured dynamic control is encapsulated

```
for (int x = 0; x < n; ++x) {
A:      s = f();
B:      while (P(x, s))
                s = g(s);
C:      h(s);
}
```

Instance set: $\{ \mathtt{A}[x] : 0 \leqslant x < n; \mathtt{B}[x] : 0 \leqslant x < n; \mathtt{C}[x] : 0 \leqslant x < n \}$

Note: currently, internal order of accesses is lost
$\Rightarrow$ possible loss of accuracy in dependence analysis

Inlining

Enabled through C99 `inline` keyword on function definition

```
inline void set_diagonal(int n,
        float A[const restrict static n][n], float v)
{
        for (int i = 0; i < n; ++i)
U:              A[i][i] = v;
}

void f(int n, float A[const restrict static n][n])
{
#pragma scop
S:      set_diagonal(n, A, 0.f);
        for (int i = 0; i < n; ++i)
                for (int j = i + 1; j < n; ++j)
T:                      A[i][j] += A[i][j - 1] + 1;
#pragma endscop
}
```

Instance set: $\{ \mathtt{U}[i] : 0 \leqslant i < n; \mathtt{T}[i,j] : 0 \leqslant i < j < n \}$

Access Relations and Function Calls

```
void set_diagonal(int n,
        float A[const restrict static n][n], float v)
{
        for (int i = 0; i < n; ++i)
U:              A[i][i] = v;
}

void f(int n, float A[const restrict static n][n])
{
#pragma scop
S:      set_diagonal(n, A, 0.f);
        for (int i = 0; i < n; ++i)
                for (int j = i + 1; j < n; ++j)
T:                      A[i][j] += A[i][j - 1] + 1;
#pragma endscop
}
```

May-write: $\{ \texttt{S}[] \to \texttt{A}[i,i] : 0 \leqslant i < n; \texttt{T}[i,j] \to \texttt{A}[i,j] : 0 \leqslant i < j < n \}$
Must-write: $\{ \texttt{S}[] \to \texttt{A}[i,i] : 0 \leqslant i < n; \texttt{T}[i,j] \to \texttt{A}[i,j] : 0 \leqslant i < j < n \}$

Access Relations and Structures

[26]

```
struct s {
        int a;
        int b;
};

int f()
{
        struct s a, b[10];

S:      a.b = 57;
T:      a.a = 42;
        for (int i = 0; i < 10; ++i)
U:              b[i] = a;
}
```

Must-write: $\{ \texttt{S}[] \to \texttt{a\_b}[\texttt{a}[] \to \texttt{b}[]]; \texttt{T}[] \to \texttt{a\_a}[\texttt{a}[] \to \texttt{a}[]];$
$\texttt{U}[i] \to \texttt{b\_a}[\texttt{b}[i] \to \texttt{a}[]]; \texttt{U}[i] \to \texttt{b\_b}[\texttt{b}[i] \to \texttt{b}[]] \}$

Summary Functions

[2, 26]

Analysis of accesses in called function may be inaccurate or even infeasible

- missing body (library function without source)
- unstructured control
- aliasing
- pattern inside dynamic control is ignored
- additional information not explicitly expressed in code

⇒ explicitly specify accesses in summary function pencil

Summary Function Example

```
int f(int i); int maybe(); struct s { int a; };
void set_odd_summary(int n, struct s A[static n]) {
        for (int i = 1; i < n; i += 2)
                if (maybe())
                        A[i].a = 0;
}
__attribute__((pencil_access(set_odd_summary)))
void set_odd(int n, struct s A[static n])
{
        for (int i = 0; i < n; ++i)
                A[2 * f(i) + 1].a = i;
}
void foo(int n, struct s B[static 2 * n])
{
#pragma scop
S:      set_odd(2 * n, B);
#pragma endscop
}
```

May-write: $\{ \texttt{S}[] \to \texttt{B\_a}[\texttt{B}[i] \to \texttt{a}[]] : 0 \leqslant i < 2n \wedge i \bmod 2 = 1 \}$

Context

The context collects constraints on the symbolic constants

- derived by pet
  - exclude values that result in undefined behavior
    - ⋆ negative array sizes
    - ⋆ out-of-bounds accesses
    - ⋆ signed integer overflow
  - `__builtin_assume` or `__pencil_assume` pencil
    - ⇒ any constraint can be specified
    - ⇒ only quasi-affine constraints on symbolic constants are exploited
- specified on PPCG command line
  - `--ctx`
  - `--assume-non-negative-parameters`

Main purpose: simplify generated AST

Dependence analysis in isl

[27, 28]

isl contains generic dependence analysis engine
⇒ determines dependence relations between "sources" and "sinks"

Input:

- Sink $K : I \to D$
- May-source $Y : I \to D$
- Kill $L : I \to D$
- Schedule S on I ⇒ defines "before" and "intermediate"

Output:

- May-dependence relation: triples $(\mathbf{i}, \mathbf{k}, \mathbf{a})$
  - **i** has a may-source to **a**
  - **k** has a sink to **a**
  - **i** is scheduled before **k**
  - there is no intermediate kill to **a**
- May-no-source: sinks **k** → **a** with no kill to **a** before **k**

Dependence analysis in PPCG

[28]

isl:

- May-dependence relation: triples $(\mathbf{i}, \mathbf{k}, \mathbf{a})$
  - **i** has a may-source to **a**
  - **k** has a sink to **a**
  - **i** is scheduled before **k**
  - there is no intermediate kill to **a**
- May-no-source: sinks **k** → **a** with no kill to **a** before **k**

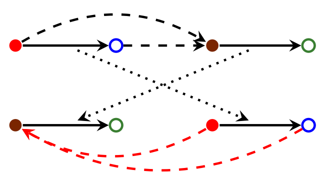
PPCG (without live-range reordering):

- flow dependences (without **a**) and live-in (may-no-source)
  - sink: may-read
  - may-source: may-write
  - kill: must-write
- false dependences (without **a**)
  - sink: may-write
  - may-source: may-read or may-write
  - kill: must-write
- killed writes (without **k**) (⇒ removed from may-write to get live-out)
  - sink: must-write
  - may-source: may-write

Live-Range Reordering

[26, 28]

```
a = f1();
f2(a);
a = f3();
f4(a);
```



⟶: flow
- - →: false

Reordering rejected due to false dependences

Live-range reordering

- allows such live-ranges to be reordered
- using somewhat different classification of dependences
- computed using different calls to the same dependence analysis engine

Pure Kills

[26]

Basic idea:

- Must-writes kill dependences to earlier writes
- Pure kills can also be useful
- Used only as kills during dependence analysis, not as source

Kills can be inserted

- automatically by pet
  - Variable declared within SCoP
    - ⇒ kill at declaration
    - ⇒ kill at end of enclosing block (if within SCoP)
  - Variable declared in scope that contains SCoP, only used inside
    - ⇒ kill at end of SCoP
- manually by the user
  - `__pencil_kill`

pencil

Dependence analysis in PPCG

[28]

`isl`:

- May-dependence relation: triples (**i**, **k**, **a**)
  - **i** has a may-source to **a**
  - **k** has a sink to **a**
  - **i** is scheduled before **k**
  - there is no intermediate kill to **a**
- May-no-source: sinks **k** → **a** with no kill to **a** before **k**

`PPCG` (without live-range reordering):

- flow dependences (without **a**) and live-in (may-no-source)
  - sink: may-read
  - may-source: may-write
  - kill: must-write or pure kill
- false dependences (without **a**)
  - sink: may-write
  - may-source: may-read or may-write
  - kill: must-write
- killed writes (without **k**) (⇒ removed from may-write to get live-out)
  - sink: must-write or pure kill
  - may-source: may-write

Kill Example

```
void f(int n, int A[restrict static n],
        int B[restrict static n])
{
        int t;
#pragma scop
        for (int i = 0; i < n; ++i) {
                t = A[i];
                B[i] = t;
        }
        __pencil_kill(t);
#pragma endscop
}
```

Without kill of `t`, compiler needs to assume `t` may be used after loop

⇒ last write needs to remain last

⇒ limited scheduling freedom (even with live-range reordering)

Note: kill inserted automatically by pet (if `t` not used after SCoP)

Absence of Loop Carried Dependences

[26]

```
void foo(int n, int A[restrict static n][n],
        int B[restrict static n][n])
{
        for (int i = 0; i < n; ++i)
                #pragma pencil independent
                for (int j = 0; j < n; ++j)
                        B[i][A[i][j]] = i + j;
}
```

Assume each row of `A` has distinct elements

⇒ no loop-carried dependences, but `PPCG` cannot tell

⇒ add `#pragma pencil independent`

pencil

Note: not handled very efficiently in current version of `PPCG`

⇒ only add when needed

Optimization Criteria for PPCG

- Two levels of parallelism
  - ⇒ blocks and threads (work groups and work items)
  - ⇒ parallelism

  In PPCG, second level obtained through tiling
  - ⇒ tilability
- Reduced working set for some arrays
  - ⇒ mapping to shared memory or registers

  Obtained through tiling
  - ⇒ tilability
- Reduced data movement
  - ⇒ locality
- Simple schedules
  - ⇒ schedule used in several subsequent steps, including AST generation
  - ⇒ simplicity

Scheduling Constraints

[28]

- Validity $\mathbf{a} \to \mathbf{b}$
  - ⇒ statement instance **b** needs to be executed after **a**
  - ⇒ $f(\mathbf{b}) \geqslant f(\mathbf{a})$
- Proximity $\mathbf{a} \to \mathbf{b}$
  - ⇒ statement instance **b** preferably executed close to **a**
  - ⇒ $f(\mathbf{b}) - f(\mathbf{a})$ as small as possible
- Coincidence $\mathbf{a} \to \mathbf{b}$
  - ⇒ statement instance **b** preferably executed together with **a**
  - ⇒ $f(\mathbf{b}) = f(\mathbf{a})$
  - ⇒ band member only considered "coincident" if it coschedules all pairs
- Conditional validity (live-range reordering)
  - condition $\mathbf{b} \to \mathbf{c}$ (⇜ flow dependences)
  - conditioned validity $\mathbf{a} \to \mathbf{b}$, $\mathbf{c} \to \mathbf{d}$ (⇜ order dependences)

Schedule constraints only relevant if coscheduled by outer nodes
Other schedule constraints are said to be *carried* by some outer node

Dependences and Schedule Constraints

[28]

Traditional dependences

- flow dependences
  - ⇒ validity constraints
  - ⇒ proximity constraints
  - ⇒ coincidence constraints (when parallelism is important)
- false dependences
  - ⇒ validity constraints
  - ⇒ coincidence constraints (when parallelism is important)
  - ⇒ proximity constraints (optional for memory reuse)
- pairs of reads with shared write ("input dependences")
  - ⇒ proximity constraints (optional)

Live-range reordering

- somewhat different classification of dependences
- slightly different mapping to schedule constraints

Current PPCG

- adds false dependences to proximity constraints for historical reasons
- does not consider input dependences
- uses live-range reordering by default

Forced Outer Coincidence Scheduler

Recall:

- Feautrier
  - maximal inner parallelism

    ⇒ carry as many dependences as possible at outer bands
- Pluto
  - tilable bands
  - locality: $f(\mathbf{j}) - f(\mathbf{i})$ small

    ⇒ parallelism as extreme case: $f(\mathbf{j}) - f(\mathbf{i}) = 0$

PPCG uses variant of Pluto-algorithm with Feautrier fallback

⇒ force outer coincidence in each band

⇒ locally fall back to Feautrier if infeasible (single step)

Members in bands constructed by Pluto-algorithm are permutable

⇒ if outer member cannot be coincident, then no member can be

Each step in Feautrier algorithm carries as many dependences as possible

⇒ subsequent application of Pluto more likely to find coincident member

Device Mapping

[31]

Input: schedule tree

If schedule tree contains no coincident band member

⇒ generate pure CPU code

Otherwise:

- select subtree for mapping to the device
  selected subtree is entire schedule tree, except
  - coincidence-free children of outer set node
  - coincidence-free initial children of outer sequence node
- within selected subtree, generate kernels for
  - outermost bands with coincident members
  - maximal coincidence-free subtrees
    ⇒ insert zero-dimensional band node
- add data copying to/from device around selected subtree
- add device initialization and clean-up around entire schedule tree

Data Copying to/from Device

Copy-out:

- take may-writes
- remove writes only needed for dataflow inside selected subtree
- approximate to entire array

May-persist:

- elements that may need to be preserved by selected subtree
- consists of
  - elements that may need to be preserved by entire SCoP
    ⇒ elements not definitely written and not definitely killed
  - elements in potential dataflow across selected subtree

May-not-written: (copy-out $\cap_{\text{ran}}$ may-persist) $\setminus$ must-write

Copy-in: live-in $\cup$ may-not-written

Note: if array elements are structures, then entire structures are copied

Data Copying Example

```
__pencil_kill(A);
for (int i = 0; i < n; i++)
     if (B[i] > 0)
          A[i] = B[i];
```

A may be written

⇒ A in copy-out

A may also *not* be written (completely), but no data can flow across kill

⇒ ~~parts of A may (be expected to) survive~~

⇒ ~~A also needs to be in copy-in~~

References I

[1] Péricles Alves, Fabian Gruber, Johannes Doerfert, Alexandros Lamprineas, Tobias Grosser, Fabrice Rastello, and Fernando Magno Quintão Pereira. "Runtime Pointer Disambiguation". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 589–606. doi: 10.1145/2814270.2814285.

[2] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Róbert Dávid, and Elnar Hajiyev. "PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming". In: *Proc. Parallel Architectures and Compilation Techniques (PACT'15)*. Oct. 2015. doi: 10.1109/PACT.2015.17.

References II

[3] Riyadh Baghdadi, Albert Cohen, Sven Verdoolaege, and Konrad Trifunovic. "Improved loop tiling based on the removal of spurious false dependences". In: *TACO* 9.4 (2013), p. 52. doi: 10.1145/2400682.2400711.

[4] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. "The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems". In: *Science of Computer Programming* 72.1–2 (2008), pp. 3–21.

[5] Denis Barthou, Albert Cohen, and Jean-François Collard. "Maximal static expansion". In: *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. San Diego, California, United States: ACM, 1998, pp. 98–106. doi: 10.1145/268946.268955.

References III

[6] Denis Barthou, Jean-François Collard, and Paul Feautrier. "Fuzzy Array Dataflow Analysis". In: *J. Parallel Distrib. Comput.* 40.2 (1997), pp. 210–226. doi: 10.1006/jpdc.1996.1261.

[7] Cédric Bastoul. *Generating loops for scanning polyhedra*. Tech. rep. 2002/23. Versailles University, 2002.

[8] Cédric Bastoul. *Extracting polyhedral representation from high level languages*. Tech. rep. LRI, Paris-Sud University, May 2008.

[9] Marouane Belaoucha, Denis Barthou, Adrien Eliche, and Sid-Ahmed-Ali Touati. "FADAlib: an open source C++ library for fuzzy array dataflow analysis". In: *Intl. Workshop on Practical Aspects of High-Level Parallel Programming*. Vol. 1. 1. Amsterdam, The Netherlands, May 2010, pp. 2075–2084. doi: DOI:10.1016/j.procs.2010.04.232.

References IV

[10] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model". In: *International Conference on Compiler Construction (ETAPS CC)*. Apr. 2008. doi: 10.1007/978-3-540-78791-4_9.

[11] *Candl*. http://icps.u-strasbg.fr/~bastoul/development/candl/.

[12] Alain Darte, Robert Schreiber, and Gilles Villard. "Lattice-Based Memory Allocation". In: *IEEE Trans. Comput.* 54.10 (2005), pp. 1242–1257. doi: 10.1109/TC.2005.167.

[13] Paul Feautrier. "Array expansion". In: *ICS '88: Proceedings of the 2nd international conference on Supercomputing*. St. Malo, France: ACM Press, 1988, pp. 429–441. doi: 10.1145/55364.55406.

References V

[14] Paul Feautrier. "Dataflow analysis of array and scalar references". In: *International Journal of Parallel Programming* 20.1 (1991), pp. 23–53. doi: 10.1007/BF01407931.

[15] Paul Feautrier. "Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time". In: *International Journal of Parallel Programming* 21.6 (Dec. 1992), pp. 389–420. doi: 10.1007/BF01379404.

[16] Tobias Grosser, Armin Größlinger, and Christian Lengauer. "Polly - Performing polyhedral optimizations on a low-level intermediate representation". In: *Parallel Processing Letters* 22.04 (2012). doi: 10.1142/S0129626412500107.

[17] François Irigoin and Rémi Triolet. "Supernode partitioning". In: *15th Annual ACM Symposium on Principles of Programming Languages*. San Diego, California, Jan. 1988, pp. 319–329.

References VI

[18] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *New user interface for Petit and other interfaces: user guide*. Tech. rep. Available as `petit/doc/petit.ps` in the `Omega` distribution. University of Maryland, Dec. 1996.

[19] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott. *The Omega Library*. Tech. rep. University of Maryland, Nov. 1996.

[20] *The Polyhedral Compiler Collection*. `http://www.cse.ohio-state.edu/~pouchet/software/pocc/`. 2012.

[21] Louis-Noël Pouchet, Cédric Bastoul, and Albert Cohen. *LetSee: the LEgal Transformation SpacE Explorator*. Third International Summer School on Advanced Computer Architecture and Compilation for Embedded Systems (ACACES'07), L'Aquila, Italia. Extended abstract, pp 247–251. July 2007.

References VII

[22] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. "GRAPHITE two years after: First lessons learned from real-world polyhedral compilation". In: *GCC Research Opportunities Workshop (GROW'10)*. 2010.

[23] Sven Verdoolaege. "isl: An Integer Set Library for the Polyhedral Model". In: *Mathematical Software - ICMS 2010*. Ed. by Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama. Vol. 6327. Lecture Notes in Computer Science. Springer, 2010, pp. 299–302. doi: `10.1007/978-3-642-15582-6_49`.

[24] Sven Verdoolaege. "Polyhedral process networks". In: *Handbook of Signal Processing Systems*. Ed. by Shuvra Bhattacharrya, Ed Deprettere, Rainer Leupers, and Jarmo Takala. Springer, 2010, pp. 931–965. doi: `10.1007/978-1-4419-6345-1_33`.

References VIII

[25] Sven Verdoolaege. "Counting Affine Calculator and Applications". In: *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*. Chamonix, France, Apr. 2011. doi: `10.13140/RG.2.1.2959.5601`.

[26] Sven Verdoolaege. *PENCIL support in pet and PPCG*. Tech. rep. RT-457, version 2. INRIA Paris-Rocquencourt, May 2015. doi: `10.13140/RG.2.1.4063.7926`.

[27] Sven Verdoolaege. *Presburger Formulas and Polyhedral Compilation*. 2016. doi: `10.13140/RG.2.1.1174.6323`.

[28] Sven Verdoolaege and Albert Cohen. "Live-Range Reordering". In: *Proceedings of the sixth International Workshop on Polyhedral Compilation Techniques*. Prague, Czech Republic, Jan. 2016. doi: `10.13140/RG.2.1.3272.9680`.

References IX

[29] Sven Verdoolaege and Tobias Grosser. "Polyhedral Extraction Tool". In: *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France, Jan. 2012. doi: `10.13140/RG.2.1.4213.4562`.

[30] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. "Schedule Trees". In: *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*. Vienna, Austria, Jan. 2014. doi: `10.13140/RG.2.1.4475.6001`.

[31] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. "Polyhedral parallel code generation for CUDA". In: *ACM Trans. Archit. Code Optim.* 9.4 (2013), p. 54. doi: `10.1145/2400682.2400713`.

References X

[32] Sven Verdoolaege, Hristo Nikolov, and Todor Stefanov. "On Demand Parametric Array Dataflow Analysis". In: *Third International Workshop on Polyhedral Compilation Techniques (IMPACT'13)*. Berlin, Germany, Jan. 2013. doi: 10.13140/RG.2.1.4737.7441.

[33] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. "Counting integer points in parametric polytopes using Barvinok's rational functions". In: *Algorithmica* 48.1 (June 2007), pp. 37–66. doi: 10.1007/s00453-006-1231-0.

[34] Doran K. Wilde. *A Library for doing polyhedral operations*. Tech. rep. 785. IRISA, Rennes, France, 1993, 45 p.

[35] Jingling Xue. *Loop tiling for parallelism*. Kluwer Academic Publishers, 2000.