CS 526

Advanced
Compiler
Construction

http://misailo.cs.Illinois.edu/courses/cs526

DEPENDENCE TRANSFORMS

The slides adapted from Vikram Adve

Polyhedral Compilation

Brief Introduction to Polyhedral Compilation Techniques:

Basic polyhedral concepts in program analysis

Iteration spaces; array references

Dependence analysis

Loop transformations: representation

Loop transformations: code generation

Polyhedra

k-tuple: A point in Z^k, e.g., (1,−4, 3) or J = (i1, i2, . . . , ik)

Tuple set: A set of tuple points (0,1,2), (2,3) …

Tagged tuple set: A set of tuple points A(1,2), C(3)

- Can be represented as a tuple, where e.g., map(A)=0, map(C)=2

Polyhedron: A tuple set defined by affine inequalities

General: $\{(i_1, i_2, \ldots, i_k) : A \cdot \vec{\imath} \leq \vec{U}\}$

e.g. $\{(i_1, i_2) : L_1 \leq i_1 < U_1 \wedge L_2 \leq i_2 < U_2\}$

- Focus on convex polyhedral
- Integer polyhedron: all in/out points are integers
- Integer hull: set of integer points that bounds rational polyhedron

Tuple Relations

Tuple relation (or *relation* or *mapping*:) A mapping from tuple sets to tuple sets, e.g.,

$$\{(i, j) \rightarrow (ii, jj) : 0 \leq i < N \wedge 0 \leq j < N \wedge ii = i \wedge jj = i + j - 1\}$$

A relation, R, "applied" to a tuple set, S, yields a new tuple set, R(S).

E.g., $S = \{(i) : 0 \leq i \leq N\}, R = \{(i) \rightarrow (ii) : 0 \leq i \leq N \wedge ii = 2i + 1\}$,

results in $R(S) = \{(ii) : \exists k : ii = 2k + 1 \wedge 1 \leq ii \leq 2N + 1\}$.

Analysis Steps

1. Extract model from the code
   - Affine iteration spaces as Polyhedra
   - Array references as polyhedral mappings

2. Dependence analysis:
   - Turn into polyhedral satisfaction problem

3. Transformations:
   - Permutations/transformations on the model, specified by tuple relations
   - Generate code from the model (original code and the transformed iteration spaces)

Affine Iteration Spaces as Polyhedra

```
do i1 = L1 to U1
  S1
  do i2 = L2 to U2
    S2
    . . .
      do ik = Lk to Uk
        Sk
      enddo
     . . .
  enddo
enddo
```

Every statement in the program has an associated iteration space, describing the enclosing loops:

$$L = \{(i_1, i_2, \dots i_k) : L_1 \le i_1 < U_1 \wedge L_2 \le i_2 < U_2 \wedge L_k \le i_k < U_k\}$$

- For polyhedral analysis, L_i, U_i must be affine functions of index variables (i), loop-invariant program variables and constants.

Array References as Polyhedral Mappings

```
do i1 = L1 to U1
  S1
  do i2 = L2 to U2
    S2
    . . .
      do ik = Lk to Uk
        A[i1,…,ik] = …
      enddo
     . . .
  enddo
enddo
```

Every array reference in the program is a mapping from the iteration space (of the statement) to array elements. E.g.,

$$
\begin{aligned}
L \to A : \{(\vec{\imath}, \sim a) : \vec{\imath} \in L \\
\wedge\ a_1 = f_1(\vec{\imath}) \ldots \\
\wedge\ a_r = f_r(\vec{\imath})\}
\end{aligned}
$$

- For polyhedral analysis, Li, Ui must be affine functions of index variables (i), loop-invariant program variables and constants.

Checking for Data Dependence

There is a data dependence between

$A(f_1(\vec{\imath}), f_2(\vec{\imath}), \ldots, f_r(\vec{\imath}))$ and $A(g_1(\vec{\imath}), g_2(\vec{\imath}), \ldots, g_r(\vec{\imath}))$

iff the following polyhedron contains integer points:

$$\{(i_1, i_2, \ldots, i_r, j_1, j_2, \ldots, j_r) : \vec{\imath} \in\ L \wedge \vec{\jmath} \in L \wedge$$
$$f1(\vec{\imath}) =\ g1(\vec{\jmath}) \wedge \ldots \wedge\ f_r(\vec{\imath})\ =\ g_r(\vec{\jmath})\}$$

Program Transformations

Program transformations as polyhedral mappings: Many program transformations can be represented as a mapping (for each original program statement) from its iteration space in the original program to its iteration space in the transformed program.

Loop reordering transformations:
a transformation on a perfect loop nest that reorders the loop iteration space but does not modify the relative order of statements within the innermost loop (sometimes called an atomic block).

$$L \rightarrow L : \{(\vec{\imath}) \rightarrow (\overrightarrow{ii}) : \vec{\imath} \in L \\ \wedge ii_1 = \varphi_1(\vec{\imath}) \wedge \ldots \wedge ii_k = \varphi_k(\vec{\imath})\}$$

Example Transformations

Loop reversal: $\Phi = \{(i) \rightarrow (ii) : L_1 \leq i \leq U_1 \wedge ii = U_1 - i + 1\}$

```
do i = L_1 to U_1
  A(i)= B(i) + C(i)
enddo
```

$\Longrightarrow$

```
do ii = U_1 to L_1 by -1
  A(ii) = B(ii) + C(ii)
enddo
```

Example Transformations

Loop reversal: $\Phi = \{(i) \rightarrow (ii) : L_1 \leq i \leq U_1 \wedge ii = U_1 - i + 1\}$

```
do i = L_1 to U_1                  do ii = U_1 to L_1 by -1
  A(i)= B(i) + C(i)       ==>        A(ii) = B(ii) + C(ii)
enddo                              enddo
```

Loop interchange: $\Phi = \{(i,j) \rightarrow (jj,ii) : L_1 \leq i \leq U_1 \wedge L_2 \leq j \leq U_2 \wedge ii = i \wedge jj = j\}$

```
do i = L_1 to U_1                  do jj = L_2 to U_2
  do j = L_2 to U_2                  do ii = L_1 to U_1
    A(i,j) = B(i+j,i-j) + 1  ==>       A(ii,jj) = B(ii+jj,ii-jj) + 1
  enddo                              enddo
enddo                              enddo
```

Example Transformations

Loop tiling: Tile sizes = (s_1, s_2)

$$\begin{aligned}\Phi \;=\; \{(i,j) \rightarrow (ti,tj,ii,jj) : L_1 \leq i \leq U_1 \wedge L_2 \leq j \leq U_2 \\ \wedge\, ti = s_1 \times \lfloor \tfrac{i-L_1}{s_1} \rfloor \;\wedge\, tj = s_2 \times \lfloor \tfrac{j-L_2}{s_2} \rfloor \\ \wedge\, ii = i \;\wedge\; ti \leq ii \leq min(ti + s_1 - 1, U_1) \\ \wedge\, jj = j \;\wedge\; tj \leq jj \leq min(tj + s_2 - 1, U_2)\}\end{aligned}$$

```
do i = L_1 to U_1
  do j = L_2 to U_2
    C[i,j] += A[i,k] * B[k,j]
  enddo
enddo
```

$\Longrightarrow$

```
do ti = L_1 to U_1 by s_1
  do tj = L_2 to U_2 by s_2
    do ii = ti to min(ti+s_1-1, U_1)
      do jj = tj to min(tj+s_2-1, U_2)
        C[ii,jj] += ...
enddo enddo enddo endddo
```

Imperfect Loop Nests

General approach: Add an extra ("sequencing") dimension in the iteration space to enforce ordering on individual statements:

```
do i = L_1 to U_1
    S1(i)
    do j = L_2 to U_2
        S2(i,j)
    enddo
    S3(i)
enddo
```

$L(S1) = \{(i, \mathbf{0}, j)\colon L1 \le i \le U1 \land j = L2\}$

$L(S2) = \{(i, \mathbf{1}, j)\colon L1 \le i \le U1 \land L2 \le j \le U2\}$

$L(S3) = \{(i, \mathbf{2}, j)\colon L1 \le i \le U1 \land j = U2\}$

Loop Transformations and Matrices

Alternate representation for loop transformations – as a matrix:
$\Phi(\vec{i}) = T \cdot \vec{i} + \vec{t}$

- The transformation is affine iff T is a constant matrix and $\vec{t}$ is a parametric vector consisting of loop-invariant program variables and constants.
- Each column in the matrix product represents a single input loop. Each row in the matrix product represents a single output loop.
- The transformation is called *unimodular* if T is unimodular (i.e., square integer matrix with determinant +1 or -1)

Loop Transformations and Matrices

A transformation is called *unimodular* if the matrix T is unimodular (i.e., square integer matrix with determinant +1 or -1)

Loop interchange: $T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \vec{t} = \vec{0}$

Loop reversal: $T = [-1], \vec{t} = (U_1 - 1)$

Pros and Cons

Pros:

- Principled representation
- Fine-grained optimization and analysis using mathematical programming
- Simplify loop transformations

Cons:

- In general, NP-complete problem:
  boils down to Integer programming
- Memory consuming, especially for irregular nests with control flow

References

Courses/Lectures:

- Louis-Noël Pouchet course: http://web.cse.ohio-state.edu/~pouchet/#lectures
- Pollylabs video and written tutorials: http://www.pollylabs.org/education.html

Tools: GCC Graphite, URUK, Omega, Loop…

Polly (LLVM):

- Tool: http://polly.llvm.org
- Interactive playground: http://playground.pollylabs.org/