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 $\mathbb{Z}^k$, e.g., $(1, -4, 3)$ or $J = (i_1, i_2, \ldots, i_k)$

**Tuple set:** A set of tuple points $(0, 1, 2), (2, 3) \ldots$

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

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

**Polyhedron:** A tuple set defined by affine inequalities

**General:** \[ \{(i_1, i_2, \ldots, i_k) : A \cdot \vec{i} \leq \vec{U}\} \]

- e.g. \[ \{(i_1, i_2) : L_1 \leq i_1 < U_1 \land L_2 \leq i_2 < U_2\} \]

- 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.,
\[
\{(i,j) \rightarrow (ii,jj) : 0 \leq i < N \land 0 \leq j < N \land ii = i \land jj = i + j - 1\}
\]

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

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

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

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

\[
L = \{(i_1, i_2, \ldots, i_k) : \ L_1 \leq i_1 < U_1 \\
\quad \wedge \ L_2 \leq i_2 < U_2 \\
\quad \wedge \ L_k \leq i_k < U_k\}
\]

- 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

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

\[ L \rightarrow A : \{(\vec{i}, \sim a) : \vec{i} \in L \land a_1 = f_1(\vec{i}) \ldots \land a_r = f_r(\vec{i})\} \]

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

There is a data dependence between
\[ A(f_1(\vec{i}), f_2(\vec{i}), \ldots, f_r(\vec{i})) \text{ and } A(g_1(\vec{i}), g_2(\vec{i}), \ldots, g_r(\vec{i})) \]

iff the following polyhedron contains integer points:

\[ \{(i_1, i_2, \ldots, i_r, j_1, j_2, \ldots, j_r) : \vec{i} \in L \land \vec{j} \in L \land f_1(\vec{i}) = g_1(\vec{j}) \land \ldots \land f_r(\vec{i}) = g_r(\vec{j})\} \]
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).

\[ L \rightarrow L : \{ (i) \rightarrow (\bar{i}) : \bar{i} \in L \] \[ \land ii_1 = \varphi_1(i) \land ... \land ii_k = \varphi_k(i) \} \]
Example Transformations

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

\[
\begin{align*}
\text{do } i &= L_1 \text{ to } U_1 \\
A(i) &= B(i) + C(i) \\
\text{enddo}
\end{align*}
\]

\[
\begin{align*}
\text{do } ii &= U_1 \text{ to } L_1 \text{ by } -1 \\
A(ii) &= B(ii) + C(ii) \\
\text{enddo}
\end{align*}
\]
Example Transformations

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

\[
\begin{align*}
\text{do } i & = L_1 \text{ to } U_1 \\
A(i) & = B(i) + C(i) \\
\text{endo}
\end{align*} \quad \implies \quad \begin{align*}
\text{do } ii & = U_1 \text{ to } L_1 \text{ by } -1 \\
A(ii) & = B(ii) + C(ii) \\
\text{endo}
\end{align*}
\]

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

\[
\begin{align*}
\text{do } i & = L_1 \text{ to } U_1 \\
\text{do } j & = L_2 \text{ to } U_2 \\
A(i, j) & = B(i+j, i-j) + 1 \\
\text{endo}
\end{align*} \quad \implies \quad \begin{align*}
\text{do } ii & = L_1 \text{ to } U_1 \\
\text{do } jj & = L_2 \text{ to } U_2 \\
A(ii, jj) & = B(ii+jj, ii-jj) + 1 \\
\text{endo}
\end{align*}
\]
Example Transformations

Loop tiling: Tile sizes = \((s_1, s_2)\)

\[\Phi = \{(i, j) \rightarrow (ti, tj, ii, jj) : L_1 \leq i \leq U_1 \land L_2 \leq j \leq U_2 \land ti = s_1 \times \left\lfloor \frac{i-L_1}{s_1} \right\rfloor \land tj = s_2 \times \left\lfloor \frac{j-L_2}{s_2} \right\rfloor \land ii = i \land ti \leq ii \leq \text{min}(ti + s_1 - 1, U_1) \land jj = j \land tj \leq jj \leq \text{min}(tj + s_2 - 1, U_2)\}\]

\[
\begin{align*}
do i = L_1 & \text{ to } U_1 \\
do j = L_2 & \text{ to } U_2 \\
C[i, j] & = A[i, k] \times B[k, j] \\
enddo & \implies \\
doi = L_1 & \text{ to } U_1 \text{ by } s_1 \\
dotj = L_2 & \text{ to } U_2 \text{ by } s_2 \\
C[ii, jj] & = \ldots \\
enddo & \text{ enddo enddo enddo enddo}
\]
Imperfect Loop Nests

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

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

\[ L(S1) = \{(i, 0, j): L_1 \leq i \leq U_1 \land j = L_2\} \]

\[ L(S2) = \{(i, 1, j): L_1 \leq i \leq U_1 \land L_2 \leq j \leq U_2\} \]

\[ L(S3) = \{(i, 2, j): L_1 \leq i \leq U_1 \land j = U_2\} \]
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)
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/