Dependence analysis and Transformations on Loops

Vivek Sarkar
Krishna Palem
Execution Instances of CFG nodes

• Consider CFG node $X$ with $d$ enclosing reducible (single-entry) intervals numbered $1 \ldots d$ from outermost to innermost. $X [i_1, i_2, \ldots, i_d]$ stands for the execution instance of $X$ corresponding to the $i^{th}$ iteration of interval $k$, $\forall 1 \leq k \leq d$.

• If interval $k$ is a normalized DO loop (i.e. with initial value and increment both $= 1$), then $i_k$ will equal the value of its index variable.
Data Dependence Direction Vectors: Definition

The direction vectors of a data dependence, $S_1 \rightarrow S_2$, is a set of $d$-tuples $\{ \Psi = (\Psi_1, \Psi_2, \ldots, \Psi_d) \}$, such that

1. $d$ is the number of intervals that enclose both $S_1$ and $S_2$, and
2. each $\Psi_k$ is one of $\{<, =, >, \leq, \geq, \neq, *\}$ and
3. for any pair of execution instances of $S_1$ and $S_2$, say $S_1 [i_1, \ldots, i_d, \ldots]$ and $S_2 [j_1, \ldots, j_d, \ldots]$, such that $S_1 [i_1, \ldots, i_d, \ldots] \rightarrow S_2 [j_1, \ldots, j_d, \ldots]$, we must have $i_k \Psi_j j_k \forall 1 \leq k \leq d$ for at least one direction vector, $\Psi$. 
Loop-independent and Loop-carried Data Dependences

A data dependence is said to be *loop-independent* if it has at least one direction vector, $\psi$, such that each element of $\psi$ contains “=“.

Loop-independent dependences constrain the reordering that an be performed within a loop iteration.

A data dependence is said to be loop-carried if it has at least one direction vector, $\psi$, such that at least one element of $\psi$ contains a direction other than ”=".
Loop-independent and Loop-carried Data Dependences (Contd.)

Loop-carried dependences constrain the reordering that can be performed across loop iterations.

A data dependence can be both loop-independent and loop-carried.
Execution Instances of CFG nodes

Consider CFG node $X$ with $d$ enclosing reducible (single-entry) intervals numbered $1\ldots d$ from outermost to innermost. $X[i_1,i_2,\ldots,i_d]$ stands for the execution instance of $X$ corresponding to the $i_k^{th}$ iteration of interval $k$, $\forall 1 \leq k \leq d$.

If interval $k$ is a normalized $\texttt{DO}$ loop (i.e. with initial value and increment both = 1), then $i_k$ will equal the value of its index variable.
### Examples of One-dimensional Loop-independent Loop-carried dependences

<table>
<thead>
<tr>
<th>Loop-independent:</th>
<th>Loop-carried:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DO</strong> i = 1, n</td>
<td><strong>DO</strong> i = 1, n</td>
</tr>
<tr>
<td>X(i) = X(i) + 1</td>
<td>X(i) = X(i-1) + i</td>
</tr>
<tr>
<td><strong>END DO</strong></td>
<td><strong>END DO</strong></td>
</tr>
</tbody>
</table>

Dir. vec. for X is (=)                     Dir. vec. for X is (<)

<table>
<thead>
<tr>
<th>X(1) =</th>
<th>X(2) =</th>
<th>X(1) =</th>
<th>X(2) =</th>
<th>X(3) =</th>
<th>X(0) =</th>
<th>X(1) =</th>
<th>X(2) =</th>
</tr>
</thead>
<tbody>
<tr>
<td>=X(1)</td>
<td>=X(2)</td>
<td>=X(1)</td>
<td>=X(2)</td>
<td>=X(0)</td>
<td>=X(1)</td>
<td>=X(2)</td>
<td>=X(0)</td>
</tr>
</tbody>
</table>
A direction vector, $\Psi = (\Psi_1, \Psi_2, \ldots, \Psi_d)$, identifies equality/inequality conditions under which the dependence holds, $i_k \Psi_k j_k \forall 1 \leq k \leq d$.

If, in addition, we know that the $k^{th}$ dependence condition only applies the case when $i_k + \Delta = j_k$, for some constant $\Delta$, then we can be more precise and set $\Psi_k = \Delta$, a distance value.

If $\Psi$ can contain both direction and distance values, we refer to it more generally as a dependence vector.
Example of Two-dimensional Loop-independent and Loop-carried dependences

\[
\begin{align*}
\text{DO} & \ 10 \ \text{i} = 2, \ n-1 \\
\text{DO} & \ 10 \ \text{j} = 2, \ n-1 \\
    & \quad \text{X(i,j)} = \text{X(i-1,j)} + \text{Y(i,j)} \\
    & \quad \text{A(i)} = \text{A(i)} + \text{j} \\
10 & \quad \text{if ( Y(i,j) .eq. 0 )} \quad \text{Z} = \text{Z} + \text{i*j}
\end{align*}
\]

- Flow dep. from \( X(i,j) \) to \( X(i-1,j) \) with dep. vector \((1,0)\)
- Flow dep. from \( A(i)_{\text{def}} \) to \( A(i)_{\text{use}} \) with dep. vector \((=,<)\) (can be refined to distance vector \((0,1))\)
- Anti dep. from \( A(i)_{\text{use}} \) to \( A(i)_{\text{def}} \) with dep. vector \((=,\leq_\_\_)\) (can be refined to distance vectors \{(0,0), (0,1)\})
- Output dep. from \( A(i)_{\text{def}} \) to itself with dep. Vector \((=,<)\) (can be refined to distance vector \((0,1))\)
Example of 2-D Loop-indep. And Loop-carried dependences (Contd.)

Flow dep. from $Z_{def}$ to $Z_{use}$ with dep. vectors \{$(=,<)$, $(<,*)$\} (set of all 2-D lexicographically positive vectors)

Anti dep. from $Z_{use}$ to $Z_{def}$ with dep. Vectors \{$(=,=)$, $(=,<)$, $(<,*)$\} (set of all 2-D lexicographically non-negative vectors)

Output dep. from $Z_{def}$ to $Z_{def}$ with dep. Vectors \{$(=,<),(<,*)$\} (set of all 2-D lexicographically positive vectors)

The dependence vectors for Z represent worst-case constraints and prevent any iteration-reordering transformation from being performed.
Plausible Dependence Vectors

A dependence vector is *plausible* if and only if it is lexicographically non-negative. All dependencies in sequential programs must be plausible (otherwise the input program would not satisfy its own data dependencies).

**Two-dimensional loop nest:**
DO i = 1,...  
  Do j = 1,...  
  . . .

Examples of plausible dependence vectors: (0,1), (1,-1), (1,*)  
Examples of implausible dependence vectors: (0,-1), (-1,0), (*,0)
## Identifying Plausible Output Dependences (Example)

<table>
<thead>
<tr>
<th>i = 1, n</th>
<th>i = 1</th>
<th>i = 2</th>
<th>i = 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1:</td>
<td>X(i) = ...</td>
<td>X(1)=_</td>
<td>X(2)=_</td>
</tr>
<tr>
<td></td>
<td>\</td>
<td>\</td>
<td>\</td>
</tr>
<tr>
<td>S2:</td>
<td>X(i-1) = ...</td>
<td>X(0)= _X(1)=</td>
<td>_X(2)=</td>
</tr>
<tr>
<td></td>
<td>END DO</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dependence vector from S1 to S2 is (+1)
⇒ *plausible output dependence*

Dependence vector from S2 to S1 is (-1)
⇒ *implausible output dependence* (can be ignored)
Identifying Plausible Flow and Anti Dependences (Example 2)

DO i = 1, n
  S1: X(i) = ...
  S2: ... = X(i+1)
END DO

⇒ implausible flow dependence (can be ignored)
→ plausible anti dependence

Dependence vector from S1 to S2 is (-1)
Dependence vector from S2 to S1 is (+1)
Data Dependence Analysis of Array Variables

Consider two references, $R_1$ and $R_2$, to an array variable $X$. Data dependence analysis proceeds as follows:

1. Construct a system of dependence equations, one for each array dimension, that models a dependence between the two array references.

2. Establish a context (constraints based on loop bounds and direction vectors) in which the dependence equations should be tested.

3. Invoke a decision algorithm to test the equations for existence of solutions in the provided context.
Consider two array references to an m-dimensional array variable $X$, $R_1 = X(f_1, \ldots, f_m)$ and $R_2 = X(g_1, \ldots, g_m)$, where $f_1, \ldots, f_m, g_m, \ldots, g_m$ are arbitrary subscript expressions.

In general, we need to test for a dependence between two arbitrary execution instances of the array references, $R_1[i_1, \ldots, i_d, \ldots]$ and $R_2[j_1, \ldots, j_d, \ldots]$, where $d$ is the number of common enclosing loops for $R_1$ and $R_2$. 
A dependence between the two references is simply modeled by a set of $m$ simulations equations:

$$f_1(i_1,\ldots,i_d,\ldots) = g_1(j_1,\ldots,j_d,\ldots)$$
$$\ldots = \ldots$$
$$f_m(i_1,\ldots,i_d,\ldots) = g_m g_1(j_1,\ldots,j_d,\ldots)$$

in which the free variables are $i_1,\ldots,i_d,\ldots,j_1,\ldots,j_d,\ldots$
Example of Data Dependence Analysis

Input Program:

```
DO 10 k=1, n
   DO 10 i=1, n
      DO 10 m=1, n
10      X(k, 2*1) = X(k-1, 2*1+1) + A(k, l, m)
```

Construction of Dependence Equations:
Consider execution instance \([k’,l’,m’]\) of def \(X(k,2*1)\)
and execution instance \([k’’,l’’,m’’]\) of use \(X(k-1,2*1+1)\).

The dependence equations for this data dependence test are

\[
k' = k'' - 1 \\
2l' = 2l'' + 1
\]
Context for Data Dependence Equations

Context = set of constraints on solutions to equations

1. The solution must be integer-valued
2. \( \forall 1 \leq k \leq d \), variables \( i_k \) and \( j_k \) must lie within the normalized loop bounds of the \( k \)th loop:
   \[
   \text{lower}_k(i_1, \ldots, i_{k-1}) \leq i_k \leq \text{upper}_k(i_1, \ldots, i_{k-1})
   \]
   \[
   \text{lower}_k(j_1, \ldots, j_{k-1}) \leq j_k \leq \text{upper}_k(j_1, \ldots, j_{k-1})
   \]
3. The constraint imposed by direction vector
   \( \Psi = (\Psi_1, \Psi_2, \ldots, \Psi_d) \) is \( i_k \Psi_k \) \( j_k \) \( \forall 1 \leq k \leq d \).
   When testing for a loop-independent dependence, we can eliminate \( d \) variables by setting \( i_1 = j_1, \ldots, i_d=j_d \).
4. The value of each subscript expression must lie within the array bounds for that dimension e.g. $lbound_1 \leq f_1 \leq ubound_1$. 
Decision Algorithm

**Goal:** determine whether or not a solution exists to the dependence equations, for the given constraints

- GCD test [Towle 1976, Banerjee 1976]
- Banerjee’s inequality [Banerjee 1976]
- Lambda test [Li, Ye, Zhu 1988]
- Rice U. tests
- Stanford U. tests
- Omega test
- ...
GCD test

Contexts considered: “=“ elements in the direction vector, integer-valued situations

Constraints ignored: inequalities arising from direction vectors, loop bounds, array bounds

Algorithm:
1. Eliminate one variable for every “=“ entry in the direction vector
2. Rewrite each remaining dependence equation in the form
   \[ \sum_{k=1}^{k=n} c_k v_k = c_0, \]
   where each \( c_k \) is a non-zero integer coefficient and each \( v_k \) is a variable.
3. If, for any equation, $\text{GCD}(c_1,\ldots,c_n)$ does not divide $c_0$ evenly, then the equation has no integer solutions
LOOP TRANSFORMATIONS

**Motivation:** restructure program so as to enable more effective back-end optimizations and hardware exploitation

Loop transformations are useful for enhancing

- register allocation
- instruction-level parallelism
- data-cache locality
- vectorization
- parallelization
Loop Distribution: Definition

Original loop body = $B$
Distributed loop bodies = $B_1, \ldots, B_k$
(need not appear in same order as in $B$)
Example of Loop Distribution for Eliminating Register Spills

Suppose only two floating-point registers were available for allocation:

\[
\begin{align*}
\text{Do } i &= 1, n \\
R &= R + X(i) \\
S &= S + Y(i) \\
T &= T + Z(i) \\
\text{END DO}
\end{align*}
\]

\[
\begin{align*}
\text{Do } i &= 1, n \\
R &= R + X(i) \\
\text{END DO}
\end{align*}
\]

\[
\begin{align*}
\text{Do } i &= 1, n \\
S &= S + Y(i) \\
\text{END DO}
\end{align*}
\]

\[
\begin{align*}
\text{Do } i &= 1, n \\
T &= T + Z(i) \\
\text{END DO}
\end{align*}
\]

\# f.p. regs needed = 4

\# f.p. regs needed = 2
Loop Distribution: Legality

Loop distribution is legal iff no pair of distinct distribution loop bodies, $B_i$ and $B_j$, is connected by a cycle of data and control dependencies in the original loop body, $B$. 
Example of Legal Loop Distribution

Do i = 1, n
    X(i) = X(i-1) + Y(i)
END DO

DO i = 1, n
    Y(i) = X(i) + 1
END DO
Example of Illegal Loop Distribution

Do i = 1, n

\[ X(i) = X(i-1) + Y(i-1) \]

\[ Y(i) = X(i) + 1 \]

END DO

Do i = 1, n

\[ X(i) = X(i-1) + Y(i-1) \]

END DO

\[ Y(i) = X(i) + 1 \]

END DO
Loop Distribution across Data Dependences

- Self-cycles on distributed loop bodies can be ignored
- Data dependences carried by outer loops can be ignored
- Anti and output data dependences can be eliminated by storage duplication and data copying
Loop Distribution across Control Dependences

• Distribution across *loop-carried* control dependence requires creation of temporary to store iteration count before loop exit

• Distribution across *loop-independent* control dependence requires storing predicate values in mask array or replicating the predicate computation in distributed loop bodies

• A conservative approach to deal with control dependences is to add dummy reverse control dependence edges when testing for loop distribution legality
Loop Distribution: Definition

DO i = . . .
B1(i)
END DO

.  DO i = . . .
.  ------------->  B(i)
.  END DO

DO i = . . .
Bk(i)
END DO

Original loop bodies = $B_1, \ldots B_k$
Fused loop body = $B$
Example of Loop Fusion for Improved Register Locality

\[\begin{align*}
X(i) &= Y(i) + Z(i) \\
Y(i) &= X(i) + Z(i) \\
\text{DO } i = 1, n \text{ END DO}
\end{align*}\]

\[\begin{align*}
\text{COST} &= 4n \text{ loads, 2n stores,} \\
\text{COST} &= 2n \text{ loads, 2n stores}
\end{align*}\]
Loop Fusion: Legality

Loop fusion is legal if and only if
1. Original loops are adjacent (or can be made adjacent), and
2. Original loops have conformable loop bounds, and
3. None of the original loops has a premature loop exit, and
4. Fusion does not create a lexicographically negative loop-carried data dependence vector between loop body $B_i$ and loop body $B_j$, $i<j$
Example of Legal Loop Fusion

DO i = 1, n
   X(i) = X(i-1) + Y(i)
END DO

DO i = 1, n
   Y(i) = X(i) + 1
END DO

DO i = 1, n
   X(i) = X(i-1) + Y(i)
   Y(i) = X(i) + 1
END DO
Example of Illegal Loop Fusion

DO i = 1, n
X(i) = X(i-1) + Y(i-1)
END DO

DO i = 1, n
---------->
X(i) = X(i-1) + Y(i-1)
Y(i) = X(i) + 1
END DO

DO i = 1, n
Y(i) = X(i) + 1
END DO
Loop Interchange: Definition

Before interchange:

\[
\text{DO } i_{1} = \ldots \text{ DO } i_{n} = \ldots \text{ B}(i_{1}, \ldots, i_{n})
\]

After interchange:

\[
\text{DO } i_{P(1)} = \ldots \text{ DO } i_{P(n)} = \ldots \text{ B}(i_{1}, \ldots, i_{n})
\]

where \( P \) is a permutation of \( 1 \ldots n \).

Original loop order = \( i_{1}, \ldots, i_{n} \)

Permutated loop order = \( i_{P(1)}, \ldots, i_{P(n)} \)
Example of Loop Interchange for Enabling Loop-Invariant Code Motion

Before: DO i=1, n
        DO j=1, n
            W(i,j) = X(i,j) + Y(j) + Z(j)
        END DO
    END DO

After:  DO j=1, n
        DO I=1,n
            W(i,j) = X(i,j) + Y(j) + Z(j)
        END DO
    END DO  <------- loop-variant expression
Loop Interchange: Legality

**Data Dependence:** Let \( D \) be the set of loop-carried dependence vectors for loops \( i_1, \ldots, i_n \), and \( D' = \{(d_{P(1)}, \ldots, d_{P(n)}) | (d_1, \ldots, d_n) \in D\} \) be the set of permuted dependence vectors.

Loop interchange is illegal if any vector in \( D' \) contains a lexicographically negative value.

**Control Dependence:** If loop body \( B \) contains a premature loop exit, then any non-identity permutation is illegal.

**Loop Bounds:** If the bounds for loop \( i_k \) contains an unanalyzable function of index variable \( i_j \), where \( j < k \), and if \( P(j) < P(k) \), then loop interchange in illegal.
Example of Legal Loop Interchange

DO i=1, n-1
    DO j=2, n-1
        X(i,j) = X(i-1,j) + Y(i,j)
    END DO
END DO

Consider loop interchange with permutation $P=[2,1]$
Original data dependence vectors = \{(0,1)\}
Permutated data dependence vectors = \{(0,1)\}
Example of Illegal Loop Interchange

DO i=2, n-1
    DO j=2, n-1
        X(i,j)=X(i-1,j+1)+Y(i,j)
    END DO
END DO

Consider loop interchange with permutation $P=[2,1]$
Original data dependence vectors = \{(1,-1)\}
Permutated data dependence vectors = \{(-1,1)\}
Example of Illegal Loop Interchange (contd.)

Original iteration space:

```
  i
  
  3  2  1
  
  3  [3,1] [3,2] [3,3]
  [2,1] [2,2] [2,3]
  [1,1] [1,2] [1,3]
```

Transfomed iteration space:

```
  j
  
  3  2  1
  
  3  [3,1] [3,2] [3,3]
  [2,1] [2,2] [2,3]
  [1,1] [1,2] [1,3]
```

Dependence Vector = (1,-1)

Dependence Vector = (1,1)
Loop Tiling: Definition

After tiling

\[ \text{DO } ii_1 = \ldots \]

Before tiling:

\[ \text{DO } i_1 = \ldots \]

\[ \text{DO } i_2 = \ldots \]

\[ \vdots \]

\[ \text{DO } i_n = \ldots \]

\[ B(i_1, \ldots, i_n) \]

Original loop nest has \( n \) loops, \( i_1, \ldots, i_n \)

Tiled loop nest has \( 2n \) loops, \( ii_1, \ldots, ii_n, i_1, \ldots, i_n \)
Loop Tiling: Definition

After tiling:
DO ii_1 = ...

Before tiling:
DO i_1 = ...

DO i_n =...
B(i_1,...,i_n)

DO ii_n =...
B(i_1,...,i_n)

Original loop nest has n loops, $i_1,...,i_n$
Tiled loop nest has 2n loops, $ii_1,...,ii_n,i_1,...,i_n$
Example of Loop Tiling for Improved Data Cache Locality

\[
\begin{align*}
&\text{DO } ii=1,n,B \\
&\text{DO } jj=1,n,B \\
&\text{DO } i=1,n \quad \text{DO } i=ii, \text{ min}(ii+B-1,n) \\
&\text{DO } j=1,n \quad \text{DO } j=jj, \text{ min } (jj+B-1,n) \\
&\quad X(i,j) = Y(j,i) \quad \text{------> } X(i,j) = Y(j,i) \\
&\text{END DO} \quad \text{END DO} \\
&\text{END DO} \quad \text{END DO} \\
&\text{END DO} \quad \text{END DO} \\
\end{align*}
\]

# cache misses \quad # Cache misses \sim 2*n^n/L

\sim n^n + n^n/L
Loop Tiling: Legality

Loop tiling is legal if and only if loops $i_1, ..., i_k$ are fully permutable.

Data Dependence: loop tiling is illegal if any element $d_j$ in a vector $(d_1, ..., d_n) \in D$ contains a negative value.

Control dependence: If loop body B contains a premature exit, then tiling for $n>1$ loops is illegal.

Loop bounds: if the bounds for loop $i_k$ contain an unanalyzable function of index variable $i_j$, where $j<k$, then loop tiling is illegal.
Loop Tiling Example
(original data access pattern)

\[ A(i,1:n) = B(1:n,i) \]
Loop Tiling Example
(transformed data access pattern)

\[ A(ii:ii+T_i-1,jj:jj+T_j-1)) = B(jj:jj+T_j-1,ii:ii+T_i-1) \]
Unrolling of Innermost Loop: Definition

DO i=lo,hi,1 (unroll R times)
  B(i) -------------------->
END DO

DO i=lo, hi-(R-1), R
  B(i)
  . . .
  B(i+R-1)
END DO

DO i=i, hi, 1
  B(i)
END DO

Original loop body = B(i)
R unrolled copies of loop body = B(i),...,B(i+R-1)

Unrolling of innermost loop is always legal
Example of Loop Unrolling for Improved Instruction-Level Parallelism

<table>
<thead>
<tr>
<th>Before unrolling:</th>
<th>After 2x unrolling:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{DO } i=1, n, 1 )</td>
<td>( \text{DO } i=1, n-1, 2 )</td>
</tr>
<tr>
<td>( W(i) = X(i) + Y(i) + Z(i) )</td>
<td>( W(i) = X(i) + Y(i) + Z(i) )</td>
</tr>
<tr>
<td>END DO</td>
<td>END DO</td>
</tr>
<tr>
<td>DO i=i, n, 1</td>
<td>DO i=i, n, 1</td>
</tr>
<tr>
<td>( W(i) = X(i) + Y(i) + Z(i) )</td>
<td>( W(i+1) = X(i+1) + Y(i+1) + Z(i+1) )</td>
</tr>
<tr>
<td>END DO</td>
<td>END DO</td>
</tr>
</tbody>
</table>
Unrolling of Multiple Loops: Definition

\[
\begin{align*}
\text{DO } i1=lo1, hi1-(R1-1), R1 \\
\text{DO } i2=lo2, hi2-(R2-1), R2 \\
\quad \ldots \\
\quad B(i1, i2, \ldots) \quad \longrightarrow \\
\quad \ldots \\
\text{END DO} \\
\end{align*}
\]

(\text{Unroll loop } i1 \text{ } R1 \text{ times})

\[
\begin{align*}
\text{DO } i1=lo1, hi1-(R1-1), R1 \\
\text{DO } i2=lo2, hi2-(R2-1), R2 \\
\quad \ldots \\
\quad B(i1, i2, \ldots) \quad \longrightarrow \\
\quad \ldots \\
\text{END DO} \\
\end{align*}
\]

(\text{Unroll loop } i2 \text{ } R2 \text{ times})

END DO
Example of Outer Loop Unrolling for Improved Common Subexpression Elimination

Before unrolling:

\[
\text{DO } j=1, n, 1 \\
\quad \text{DO } i=1, n \\
\quad \quad W(i,j) = X(i) + Y(j) \\
\quad \text{END DO} \\
\text{END DO}
\]

After 2x unrolling of loop j:

\[
\text{DO } i=1, n-1, 2 \\
\quad \text{DO } i=1, n \\
\quad \quad W(i,j) = X(i) + Y(j) \\
\quad \quad W(i,j+1) = X(i) + Y(j+1) \\
\quad \text{END DO} \\
\text{END DO} \\
\text{END DO} \\
\text{DO } j=j, n, 1 \\
\quad \text{DO } i=1, n \\
\quad \quad W(i,j) = X(i) + Y(j) \\
\quad \text{END DO} \\
\text{END DO}
\]
Unrolling of Multiple Loops: Legality

Given a set of $n$ perfectly nested loops $i_1, \ldots, i_n$, it is legal to unroll outer loop $i_j$ if it is legal to permute loop $i_j$ to the innermost position i.e. if permutation

$$P = [1, \ldots, j-1, j+1, \ldots, n, j]$$

is legal.