CS 526
Advanced Compiler Construction

http://misailo.cs.illinois.edu/courses/cs526
DEPENDENCE TRANSFORMS

The slides adapted from Vikram Adve
Motivation

**Memory hierarchy optimizations**
Goal 1: Improving reuse of data values within loop nest
Goal 2: Exploit reuse to reduce cache, TLB misses

**Tiling**
Goal 1: Exploit temporal reuse when data size > cache size
Goal 2: In parallel loops, reduce synchronization overhead

**Software Prefetching**
Goal: Prefetch predictable accesses k iterations ahead

**Software Pipelining**
Goal: Extract ILP from multiple consecutive iterations

**Automatic parallelization** Also, auto-vectorization
Goal 1: Enhance parallelism
Goal 2: Convert scalar loop to explicitly parallel
Goal 3: Improve performance of parallel code
Reordering Transformation

**Definition.** Legal Transformation preserves the meaning of that program, i.e., all externally visible outputs are identical to the original program, and in identical order.

- We consider two programs equivalent (i.e., the transformation preserving the program meaning) if on the same inputs both the original and transformed programs, after being executed, produce the same outputs.

**Theorem.** A reordering transformation that preserves all data dependences in a program is a legal transformation. 

*For discussion, see Allen and Kennedy book.*
Dependence Distance

**Dependence Distance**: If there is a dependence from statement S1 on iteration $I$ and statement S2 on iteration $I'$ then the corresponding dependence distance vector is

$$d_{I,I'} = [I'_1 - I_1, \ldots, I'_k - I_k]$$

**Note**: Computing distance vectors is harder than testing dependence
Dependence Distance

Direction Vector: For a distance vector of the form \( d_{I,I'} = [I'_1 - I_1, ..., I'_k - I_k] \) the corresponding direction vector is \( \delta_{I,I'} = [\delta_1, ..., \delta_k, ... \delta_m] \), where

\[
\delta_k = \begin{cases} 
- & \text{if } I'_k - I_k < 0 \\
+ & \text{if } I'_k - I_k > 0 \\
= & \text{if } I'_k - I_k = 0 \\
* & \text{if } \text{sign } +, -, = 
\end{cases}
\]

Note: \( I < J \) iff the leftmost non-’=’ entry in \( \delta(I,J) \) is ’+’.

• We use the property of lexicographical ordering
Loop-Carried Dependence

Statement S2 has a loop carried dependence on statement S1 iff S1 references location M on iteration I, S2 references M on iteration I’ and d(I,I’) > 0.

\[
\text{do } i = 1 \text{ to } N \\
A(i+1) = B(i) \\
B(i+1) = A(i) \\
\text{enddo}
\]

**Level** of loop-carried dependence is the leftmost non-“=” sign in the direction vector

- Forward dependence: S1 appears before S2 in the loop body
- Backward dependence: S2 appears before S1 in the loop body
# Reordering Transformations

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Preprocessing transformations</strong></td>
<td></td>
</tr>
<tr>
<td>Loop normalization</td>
<td>Make loops canonical</td>
<td>Simplify, improve dep. analysis</td>
</tr>
<tr>
<td>Ind. var. substitution</td>
<td>Identify aux. induction vars</td>
<td>Improve dependence information</td>
</tr>
<tr>
<td>Scalar expansion</td>
<td>Replace scalar with array</td>
<td>Eliminate spurious dependences</td>
</tr>
<tr>
<td>Scalar/array privatization</td>
<td>Treat var. as iteration-private</td>
<td>Eliminate spurious dependences</td>
</tr>
<tr>
<td>Variable renaming</td>
<td>Use multiple copies of vars</td>
<td>Eliminate anti- and output-dependences</td>
</tr>
<tr>
<td>Reduction recognition</td>
<td>Recognize reductions</td>
<td>Ignore special-case dependences</td>
</tr>
<tr>
<td></td>
<td><strong>Reordering transformations</strong></td>
<td></td>
</tr>
<tr>
<td>Loop interchange</td>
<td>Change loop nesting order</td>
<td>Cache, parallelism, vectorization</td>
</tr>
<tr>
<td>Loop strip-mining</td>
<td>Make 2 nested loops</td>
<td>”</td>
</tr>
<tr>
<td>Loop skewing</td>
<td>Change wavefront loop to parallel</td>
<td>Improve loop parallelism</td>
</tr>
<tr>
<td>Loop reversal</td>
<td>Run loop backwards</td>
<td>Reduce array storage</td>
</tr>
<tr>
<td>Index set splitting</td>
<td>Break loop by index space</td>
<td>Remove some deps.</td>
</tr>
<tr>
<td>Loop distribution</td>
<td>Break loop by statements</td>
<td>Simplify parallelization, vectorization</td>
</tr>
<tr>
<td>Loop alignment</td>
<td>Change carried to indep.</td>
<td>Simplify parallelization, vectorization</td>
</tr>
<tr>
<td>Loop fusion</td>
<td>Join loops by statements</td>
<td>Improve cache reuse</td>
</tr>
</tbody>
</table>
Math Intermezzo: Unimodular Matrix

A matrix $T$ is unimodular iff it is a square integer matrix with determinant $+1$ or $-1$

\[
\begin{bmatrix}
0 & 1 \\
1 & 0 \\
1 & 0 \\
1 & 1
\end{bmatrix}
\]

These properties will help us compose transformations:
- Product of two unimodular matrices is also unimodular
- Its inverse is also unimodular

For each integer vector $x$, a unimodular matrix $T$ maps it into a unique vector $y = Tx$
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: $T = [-1]$, $\vec{t} = (U_1 - 1)$

Legality of the transformation: $T \cdot \vec{t} \geq 0$
Examples of Unimodular Transformations

Interchange

for i=2 to N
    for j=2 to M-1
    end for
end for

for j=2 to M-1
    for i=2 to N
    end for
end for

Transform matrix

\[
\begin{bmatrix}
i' \\
j'
\end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}
\]

Reversal

for k=1 to L
end for

for k=L to 1 step -1
end for

\[
[k'] = [-1][k] + L
\]

Skew

for i=2 to N
    for j=2 to N
    end for
end for

for i=2 to N
    for jj=i+2 to i+N
    end for
end for

\[
\begin{bmatrix}
i' \\
j'
\end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix}
\]
Legality of Unimodular Transformations

A transformed loop nest is equivalent to the original if it preserves all dependencies. A transformation between these two nets is legal if the nests are equivalent.

Let D be the set of distance vectors of a loop nest. A unimodular transformation T is legal if and only if

$$\forall d \in D . \ T \cdot d \geq 0$$

Proof sketch (from Banerjee, Unimodular Transformations 2011):

Consider loop body S of the original nest and S’ of the transformed one. Two iterations S(I) and S(I’) in the original nest become S’(TI) and S’(TI’) in the transformed. S’(TI) precedes S’(TI’) iff T • I’ ≥ T • I.

“if part”: For each d, assume T • d ≥ 0. Consider that a statement S(I’) in iteration I’ depend on the statement S(I). Because d = I’ − I is the distance vector in the original loop, we get T • I’ − T • I = T(I’ − I) ≥ 0. With this we get that all dependencies are preserved in the transformed loop., i.e. the two loop nests are equivalent.

“only-if part”: Assume the transformation is legal. Let d=I’-I denote a distance in the original loop (and the statement in the iteration ’ depends on the one in iteration I. By hypothesis (the loop nests are equivalent), T • I’ ≥ T • I, so then T • I’ − T • I ≥ 0 and so T • (I’−I) = T • d ≥ 0
Loop Interchange

**Informal Definition:** Change nesting order of loops in a **perfect loop nest**, with no other changes.

for i=2 to N
    for j=2 to M-1
    end for
end for

for j=2 to M-1
    for i=2 to N
    end for
end for
Uses of Loop Interchange

1. Move independent loop innermost
2. Move independent loop outermost
3. Make accesses stride-1 in memory
4. Loop tiling (combine with strip-mining)
5. Unroll-and-jam (combine with unrolling)
Loop Interchange

Direction Vectors and Loop Interchange:
If $\delta$ is a direction vector of a particular dependence $S_1 \rightarrow S_2$ in a loop nest and the order of loops in the loop nest is permuted, then the same permutation can be applied to $\delta$ to obtain the new direction vector for the conflicting instances of $S_1$ and $S_2$.

Direction Matrix: A matrix where each row is the direction vector of a single dependence, i.e., each row $\leftrightarrow$ a dependence, each column $\leftrightarrow$ a loop
Loop Interchange Properties

Legality: A permutation of the loops in a perfect nest is legal iff the direction matrix, after the permutation is applied, has no “-” direction as the leftmost non-“=” direction in any row.

- Recall, for legality the vector after transformation should be lexicographically greater than “0” vector.

- Some more intuition: To preserve the dependencies, consider the cases before transformation of (=,=) [independent], (=,+), (+,=) [the dependence is still carried but by the outer (resp. inner loops)], (+,+) [Dependence is still carried]. But (+,-) is illegal since the antidependence turns into a true dependence.

Profitability: machine-dependent:

1. vector machines
2. parallel machines
3. caches with single outstanding loads
4. caches with multiple outstanding loads
Direction Matrix

Direction Matrix:
each row ↔ a dependence
each column ↔ a loop

for i = 2 to N
    for j = 2 to M-1
        Sp: \[ A[i,j] = B[i-1,j-1] \]
    endfor
endfor
Direction Matrix (Illegal)

Direction Matrix:
each row $\leftrightarrow$ a dependence
each column $\leftrightarrow$ a loop

for $i = 2$ to $N$
    for $j = 2$ to $M-1$
        Sp: $A[i,j] = B[i-1,j-1]$
    endfor
endfor

\[
\begin{align*}
\text{Sp} \rightarrow \text{Sq}: A[i,j]/A[i,j] &= = \\
\text{Sp} \rightarrow \text{Sq}: A[i,j]/A[i-1,j+1] &= + - \\
\text{Sq} \rightarrow \text{Sp}: B[i,j]/B[i-1,j-1] &= + +
\end{align*}
\]
Applying Loop Interchange

1. **Single ’+’ entry: a “serial loop”**
   - Move loop outermost for vectorization
   - Move loop innermost for parallelization

2. **Multiple ’+’ entries: Outermost one carries dependence**
   - Loop carrying the dependence *changes* after permutation!
   - May still benefit by moving carried-dependences to the outermost loop
Example

for \( i = 1 \) to \( n \)
    for \( j = 1 \) to \( m \)
    end for
end for

for \( i = 1 \) to \( n \)
    for \( j = 1 \) to \( m \)
        // vectorize
    end for
end for

parallel for \( j = 1 \) to \( m \)
    for \( i = 1 \) to \( n \)
    end for
end for
Loop Reversal

Informal Definition: Reverse the order of execution of the iterations of a loop

for $i=2$ to $N$
  for $j=2$ to $M-1$
    for $k=1$ to $L$
    endfor
  endfor
endfor

for $i=2$ to $N$
  for $j=2$ to $M-1$
    for $k=L$ to $1$ step $-1$
    endfor
  endfor
endfor
Legality of Loop Reversal

The loop that is reversed *should not carry dependence*

Recall, *Legality*: the vector after transformation should be lexicographically greater than “0” vector.

E.g., \((1, -1) \succ (0,0)\) but \((-1, 1) \prec (0,0)\)

In our case, two dependencies:

\[
\begin{align*}
(1) & \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} [+] = [+] \succ 0 \\
& \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} [-] = [+] \succ 0 \\
\end{align*}
\]
Uses of Loop Reversal

Convert a ‘-’ to a ’+’ in a direction vector to enable other transformations, e.g., loop interchange.

Scalarize a vector statement (e.g., in Fortran 90) by ensuring that values are read before being written.

• Scalarized code:

  ```
  for i = 64 to 2 step -1
  endfor
  ```
Loop Skewing

**Informal Definition:** Increase dependence distance by \( n \) by substituting loop index \( j \) with \( jj = j + n \times i \).

E.g., For \( n = 1 \), we use \( jj = j + 1 \)

```
for i=2 to N
    for j=2 to N
    end for
end for

for i=2 to N
    for jj=i+2 to i+N
    end for
end for
```

- Improve parallelism by converting '=' to '+' in a direction vector
- Improve vectorization in a similar way
- (Rarely) Could be used to simplify index expressions
Skewing: Full Example

for \( I_1 := 0 \) to 5 do
  for \( I_2 := 0 \) to 6 do
    \[ A[I_2 + 1] := \frac{1}{3} \times (A[I_2] + A[I_2 + 1] + A[I_2 + 2]) \; ; \]
    \[ D = \{(0,1),(1,0),(1,-1)\} \; . \]

\[
\begin{align*}
\text{for } I'_1 &:= 0 \text{ to } 5 \text{ do} \\
& \text{for } I'_2 := I'_1 \text{ to } 6+I'_1 \text{ do} \\
& \quad A[I'_2 - I'_1 + 1] := \frac{1}{3} \times (A[I'_2 - I'_1] \\
& \quad \quad + A[I'_2 - I'_1 + 1] + A[I'_2 - I'_1 + 2]) \; ;
\end{align*}
\]

\[ T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \]

\[ D' = TD = \{(0,1),(1,1),(1,0)\} \]
Loop Strip Mining

**Informal Definition** Convert a single loop into two nested loops for a specified “block size”

*Always safe.*

```plaintext
for i=1 to N
    A[i] = x + B[i] * 2
end for

for ii=1 to N step B
    for i=ii to min(ii+B-1, N)
        A[i] = x + B[i] * 2
    end for
end for
```
Loop Strip Mining Applications

- **Loop tiling:** *strip-mine* and then *interchange* multiple uses. Can be useful for increasing cache locality or blocking parallel loops;

  \[
  \text{for } j=1 \text{ to } N \\
  \quad \text{for } ii=1 \text{ to } N \text{ step } B \\
  \quad \quad \text{for } i=ii \text{ to } \min(ii+B-1, N) \\
  \quad \quad \quad A[i][j] = x + B[i][j]
  \]

  \[
  \text{for } ii=1 \text{ to } N \text{ step } B \\
  \quad \text{for } j=1 \text{ to } N \\
  \quad \text{for } i=ii \text{ to } \min(ii+B-1, N) \\
  \quad \quad A[i][j] = x + B[i][j]
  \]

  *When is it safe to do tiling?*

- **Prefetching:** strip-mine by cache line size; prefetch once per outer iteration

- **Instruction scheduling:** strip-mine and then unroll inner loop
Tiling Example

for $I'_1 := 0$ to $5$ do
  for $I'_2 := I'_1$ to $6+I'_1$ do
    $A[I'_2 - I'_1 + 1] := 1/3 \times (A[I'_2 - I'_1]$
    $+ A[I'_2 - I'_1 + 1] + A[I'_2 - I'_1 + 2])$

$T' = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$

$D' = T D = \{(0,1),(1,1),(1,0)\}$

for $II'_1 := 0$ to $5$ by $2$ do
  for $II'_2 := 0$ to $11$ by $2$ do
    for $I'_1 := II'_1$ to $\min(5, II'_1 + 1)$ do
      for $I'_2 := \max(I'_1, II'_2)$ to $\min(6+I'_1, II'_2+1)$ do
        $a[I'_2 + 1] := 1/3 \times (a[I'_2] + a[I'_2 + 1] + a[I'_2 + 2])$;
Loop Distribution

**Informal Definition:** Convert a loop nest containing two or more statements into two or more distinct loop nests so that each statement appears in only a single resulting loop nest.

```plaintext
for i = 2 to N
  S1: A[i] = B[i] + C[i]
  S2: D[i] = A[i] * 2.0
end for
```

```plaintext
for i = 2 to N
  S1: A[i] = B[i] + C[i]
  S2: D[i] = A[i] * 2.0
end for
```
Loop Distribution Applications

• Create perfect loops nests for other transformations like loop interchange
• Convert a loop-carried dependence within a loop into a loop-independent dependence crossing two loops:

```latex
\begin{align*}
  &\text{for } i=2 \text{ to } N \\
  &\text{S1: } \quad A[i] = B[i] + C[i] \\
  &\text{S2: } \quad D[i] = A[i-1] * 2.0 \\
  &\text{end for} \\
\end{align*}
\text{for } i=2 \text{ to } N \\
\text{S1: } \quad A[i] = B[i] + C[i] \\
\text{end for} \\
\text{for } i=2 \text{ to } N \\
\text{S2: } \quad D[i] = A[i-1] * 2.0 \\
\text{end for}
```
Maximal Loop Distribution

• Identify the SCCs of the data dependence graph, to group statements in an SCC in a single loop nest
• Sort the SCCs using a topological sort on the dependence graph
• Generate distinct loop nests, one for each SCC, in sorted order
• If we have control dependence between a statement $S_1$ is one SCC and the statement $S_2$ in another SCC, create an array ‘flags’ that contains the Boolean conditions, populate it in the first SCC that induce dependence and use them in the second SCC.

Reminder:
• **Strongly connected graph**: a directed graph in which there is a path between all pairs of vertices.
• **Strongly connected component (SCC)** is a maximal strongly connected subgraph
Loop Fusion

Informal Definition: Merge two or more distinct (perhaps non-adjacent) loops with identical loop bounds into a single loop.

for $i=1$ to $N$
    $A[i] = i*i$
end for

for $i=1$ to $N$
    $B[i] = A[i] + 1$
end for
Loop Fusion

for i=1 to M
  for j=1,N-1
    A[j,i] = i*i + j*j
  end for

  for j=1 to N
    B[j,i] = A[j,i] + i + j
  end for
end for

for i=1 to M
  for j=1 to N-1
    A[j,i] = i*i + j*j
    B[j,i] = A[j,i] + i + j
  end for
  // peel last iteration:
  j=N
  B[j,i] = A[j,i] + i + j
end for
Loop Fusion Motivation

• Increase cache reuse (if same array accessed in two loops) Fundamental optimization for array languages (e.g., Fortran 90, HPF, MATLAB, APL)

Example in F90:

\[
\begin{align*}
\end{align*}
\]

• Increase granularity of parallelism (work per iteration) Important for shared-memory parallelism (the model with parallel loop and barriers)
Legality of Loop Fusion

**Fusion-Preventing Dependence:** A loop-independent dependence from S1 to S2 in different loops is fusion-preventing if fusing the two loops causes the dependence to become a loop-carried dependence from S2 to S1.

**Legality of Loop Fusion:** Two loops can be fused if *all three* conditions are satisfied:

1. Both have identical bounds (*transform loops if needed*)
2. There is no fusion-preventing dependence between them.
3. There is no path of loop-independent dependences between them that contains a loop or statement that is not being fused with them.
Loop Fusion: Illegal Cases

for i=1 to M
    for j=2 to N
        A[j,i] = B[j-1,i] * 2
    end for
for j=2 to N
end for

for i=1 to M
    for j=2 to N
        t[j] = B[j-1,i]
    end for
for j=2 to N
    A[j,i] = t[j] * 2
end for
end for

Create temporary array to make fusion possible


**Loop Alignment**

**Informal Definition:** Eliminate a carried dependence by increasing the number of iterations and executing statements on different subsets of the iterations

*(Always safe)*

\[
\text{for } i = 2 \text{ to } N \\
\quad A[i] = B[i] + C[i] \\
\quad D[i] = A[i-1] \times 2.0 \\
\text{end for}
\]

\[
i = 1 \\
D[i+1] = A[i] \times 2 \\
\text{for } i = 2 \text{ to } N-1 \\
\quad A[i] = B[i] + C[i] \\
\quad D[i+1] = A[i] \times 2.0 \\
\text{end for}
\]

\[
i = N \\
A[i] = B[i] + C[i]
\]
Scalar Replacement

**Informal Definition:** Replace an array reference with a scalar temporary. (Use dependences to locate consistent re-use patterns)

```
for i = 1 to n
    for j = 2 to n
        x[j,i] = a[i] +
        x[j-1,i] +
        b[j,i]
    end for
end for

for i = 1 to n
    t1 = a[i];
    for j = 2 to n
        x[j,i] = t1 +
        x[j-1,i] +
        b[j,i]
    end for
end for
```
Unroll and Jam

**Informal Definition:** Unroll the outer loop by k, then fuse the resulting k inner loops into a single loop

```plaintext
for i = 1 to n
  for j = 1 to n
    a[i] = a[i] + b[j]
  end for
end for

for i = 1 to n step 2
  for j = 1 to n
    a[i] = a[i] + b[j]
    a[i+1] = a[i+1] + b[j]
  end for
end for
```
More details:

Optimizing Compilers for Modern Architectures

Allen and Kennedy

Academic Press