Acknowledgments

• Slides from previous offerings of COMP 515 by Prof. Ken Kennedy
  — http://www.cs.rice.edu/~ken/comp515/
Midterm Summary

Chapters 1-6 of Allen and Kennedy book
Compiler Challenges for High Performance Architectures

Allen and Kennedy, Chapter 1
Bernstein’s Conditions [1966]

- When is it safe to run two tasks R1 and R2 in parallel?
  - If none of the following holds:
    1. R1 writes into a memory location that R2 reads
    2. R2 writes into a memory location that R1 reads
    3. Both R1 and R2 write to the same memory location
- How can we convert this to loop parallelism?
  - Think of loop iterations as tasks
- Does this apply to sequential loops embedded in an explicitly parallel program?
  - Impact of memory model on ordering of read operations
Dependence: Theory and Practice

Allen and Kennedy, Chapter 2
Dependences

• Formally:

There is a data dependence from statement $S_1$ to statement $S_2$ ($S_2$ depends on $S_1$) if:

1. Both statements access the same memory location and at least one of them stores onto it, and
2. There is a feasible run-time execution path from $S_1$ to $S_2$
Load Store Classification

• Quick review of dependences classified in terms of load-store order:

  1. True dependences (RAW hazard)
     - $S_2$ depends on $S_1$ is denoted by $S_1 \delta S_2$

  2. Antidependence (WAR hazard)
     - $S_2$ depends on $S_1$ is denoted by $S_1 \delta^{-1} S_2$

  3. Output dependence (WAW hazard)
     - $S_2$ depends on $S_1$ is denoted by $S_1 \delta^0 S_2$
Formal Definition of Loop Dependence

- **Theorem 2.1 Loop Dependence:**
  There exists a dependence from statements $S_1$ to statement $S_2$ in a common nest of loops if and only if there exist two iteration vectors $i$ and $j$ for the nest, such that
  1. $i < j$ or $i = j$ and there is a path from $S_1$ to $S_2$ in the body of the loop,
  2. statement $S_1$ accesses memory location $M$ on iteration $i$ and statement $S_2$ accesses location $M$ on iteration $j$, and
  3. one of these accesses is a write.

- Follows from the definition of dependence
Reordering Transformations

• A reordering transformation is any program transformation that merely changes the order of execution of the code, without adding or deleting any executions of any statement.

• A reordering transformation does not eliminate dependences.

• A reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink of that dependence.

• Fundamental Theorem of Dependence:
  — Any reordering transformation that preserves every dependence in a program preserves the meaning of that program.
  — Proof by contradiction. Theorem 2.2 in the book.

• A transformation is said to be valid or legal for the program to which it applies if it preserves all dependences in the program.
Distance Vectors

• Consider a dependence in a loop nest of n loops
  — Statement $S_1$ on iteration $i$ is the source of the dependence
  — Statement $S_2$ on iteration $j$ is the sink of the dependence

• The distance vector is a vector of length $n$ $d(i,j)$ such that: $d(i,j)_k = j_k - i_k$

• We normalize distance vectors for loops in which the index step size is not equal to 1.
Implausible Distance & Direction Vectors

- A distance vector is implausible if its leftmost nonzero element is negative i.e., if the vector is lexicographically less than the zero vector.
- Likewise, a direction vector is implausible if its leftmost non "=" component is not "<".
- No dependence in a sequential program can have an implausible distance or direction vector as this would imply that the sink of the dependence occurs before the source.
Direction Vector Transformation

• Theorem 2.3. Direction Vector Transformation. Let T be a transformation that is applied to a loop nest and that does not rearrange the statements in the body of the loop. Then the transformation is valid if, after it is applied, none of the direction vectors for dependences with source and sink in the nest has a leftmost non-“=” component that is “>” i.e., none of the transformed direction vectors become implausible.

• Follows from Fundamental Theorem of Dependence:
  — All dependences exist
  — None of the dependences have been reversed
Loop-carried and Loop-independent Dependences

- If in a loop statement $S_2$ depends on $S_1$, then there are two possible ways of this dependence occurring:

1. $S_1$ and $S_2$ execute on different iterations
   - This is called a loop-carried dependence.

2. $S_1$ and $S_2$ execute on the same iteration
   - This is called a loop-independent dependence.
Loop-carried dependence

- Definition 2.11

- Statement $S_2$ has a loop-carried dependence on statement $S_1$ if and only if $S_1$ references location $M$ on iteration $i$, $S_2$ references $M$ on iteration $j$ and $d(i, j) > 0$ i.e., $D(i, j)$ contains a “<” as leftmost non “=” component and is lexicographically positive.

Example:

```fortran
DO I = 1, N
  S_1  A(I+1) = F(I)
  S_2  F(I+1) = A(I)
ENDDO
```
Loop-carried dependence

- Level of a loop-carried dependence is the index of the leftmost non-"=" of D(i,j) for the dependence.

For instance:

```plaintext
DO I = 1, 10
    DO J = 1, 10
        DO K = 1, 10
            S_1  A(I, J, K+1) = A(I, J, K)
        ENDDO
    ENDDO
ENDDO

- Direction vector for S1 is (=, =, <)
- Level of the dependence is 3
- A level-k dependence between S_1 and S_2 is denoted by S_1 \delta_k S_2
```
Loop-carried Transformations

- **Theorem 2.4** Any reordering transformation that (1) preserves the iteration order of the level-k loop, (2) does not interchange any loop at level $< k$ to a level $> k$, and (3) does not interchange any loop at level $> k$ to a position $< k$, must preserve all level-k dependences.

- **Proof:**
  - $D(i, j)$ has a “$<$” in the $k^{th}$ position and “=” in positions 1 through $k-1$
  - Source and sink of dependence are in the same iteration of loops 1 through $k-1$
  - Cannot change the sense of the dependence by a reordering of iterations of those loops

- As a result of the theorem, powerful transformations can be applied
Loop-independent dependences

- Definition 2.15. Statement $S_2$ has a loop-independent dependence on statement $S_1$ if and only if there exist two iteration vectors $i$ and $j$ such that:
  1) Statement $S_1$ refers to memory location $M$ on iteration $i$, $S_2$ refers to $M$ on iteration $j$, and $i = j$.
  2) There is a control flow path from $S_1$ to $S_2$ within the iteration.

Example:

```plaintext
DO I = 1, 10
S_1  A(I) = ... 
S_2  ... = A(I)
ENDDO
```
Loop-independent dependences

- **Theorem 2.5.** If there is a loop-independent dependence from $S_1$ to $S_2$, any reordering transformation that does not move statement instances between iterations and preserves the relative order of $S_1$ and $S_2$ in the loop body preserves that dependence.

- $S_2$ depends on $S_1$ with a loop independent dependence is denoted by $S_1 \delta_{\infty} S_2$

- Note that the direction vector will have entries that are all “=” for loop independent dependences
Simple Vectorization Algorithm

**procedure vectorize** (L, D)

// L is the maximal loop nest containing the statement.

// D is the dependence graph for statements in L.

find the set \{S_1, S_2, ..., S_m\} of maximal strongly-connected regions in the dependence graph D restricted to L (Tarjan);

construct \(L_p\) from \(L\) by reducing each \(S_i\) to a single node and compute \(D_p\), the dependence graph naturally induced on \(L_p\) by D;

let \(\{p_1, p_2, ..., p_m\}\) be the \(m\) nodes of \(L_p\) numbered in an order consistent with \(D_p\) (use topological sort);

**for** \(i = 1\) **to** \(m\) **do begin**

**if** \(p_i\) is a dependence cycle **then**

generate a DO-loop nest around the statements in \(p_i\);

**else**

directly rewrite \(p_i\) in Fortran 90, vectorizing it with respect to every loop containing it;

**end**

**end vectorize**
Problems With Simple Vectorization

DO I = 1, N
  DO J = 1, M
    \text{End DO}
  \text{End DO}

• Dependence from $S_1$ to itself with $d(i, j) = (1,0)$

• Key observation: Since dependence is at level 1, we can manipulate the other loop!

• Can be converted to:
  \text{DO I = 1, N}
  \text{S_1}
  A(I+1,1:M) = A(I,1:M) + B
  \text{End DO}

• The simple algorithm does not capitalize on such opportunities
Advanced Vectorization Algorithm

procedure codegen\((R, k, D)\);
// \(R\) is the region for which we must generate code.
// \(k\) is the minimum nesting level of possible parallel loops.
// \(D\) is the dependence graph among statements in \(R\).

find the set \(\{S_1, S_2, \ldots, S_m\}\) of maximal strongly-connected
regions in the dependence graph \(D\) restricted to \(R\);

construct \(R_p\) from \(R\) by reducing each \(S_i\) to a single node and
compute \(D_p\), the dependence graph naturally induced on \(R_p\) by \(D\);

let \(\{p_1, p_2, \ldots, p_m\}\) be the \(m\) nodes of \(R_p\) numbered in an order
consistent with \(D_p\) (use topological sort to do the numbering);

for \(i = 1\) to \(m\) do begin
  if \(p_i\) is cyclic then begin
    generate a level-\(k\) DO statement;
    let \(D_i\) be the dependence graph consisting of all dependence edges in \(D\) that are at level
    \(k+1\) or greater and are internal to \(p_i\);
    codegen \((p_i, k+1, D_i)\);
    generate the level-\(k\) ENDDO statement;
  end
  else
    generate a vector statement for \(p_i\) in \(r(p_i)\)-\(k+1\) dimensions, where \(r\) \((p_i)\) is the number of
    loops containing \(p_i\);
end
Dependence Testing

Allen and Kennedy, Chapter 3
Simple Dependence Testing

DO $i_1 = L_1, U_1, S_1$
DO $i_2 = L_2, U_2, S_2$
...
DO $i_n = L_n, U_n, S_n$

$S_1 \quad A(f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)) = \ldots$
$S_2 \quad \ldots = A(g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n))$

ENDDO

...
ENDDO
ENDDO

• A dependence exists from $S_1$ to $S_2$ if and only if there exist values of $\alpha$ and $\beta$ such that (1) $\alpha$ is lexicographically less than or equal to $\beta$ and (2) the following system of dependence equations is satisfied:

$$f_i(\alpha) = g_i(\beta) \text{ for all } i, 1 \leq i \leq m$$

• Direct application of Loop Dependence Theorem
The General Problem

DO \(i_1 = L_1, U_1\)
DO \(i_2 = L_2, U_2\)
...
DO \(i_n = L_n, U_n\)

\[S_1\] \[A(f_1(i_1, \ldots, i_n), \ldots, f_m(i_1, \ldots, i_n)) = \ldots\]

\[S_2\] \[\ldots = A(g_1(i_1, \ldots, i_n), \ldots, g_m(i_1, \ldots, i_n))\]

ENDDO

\ldots

ENDDO

ENDDO

Under what conditions is the following true for iterations \(\alpha\) and \(\beta\)?

\(f_i(\alpha) = g_i(\beta)\) for all \(i, 1 \leq i \leq m\)

Note that the number of equations equals the rank of the array, and the number of variables is twice the number of loops that enclose both array references (two iteration vectors)
A subscript equation is said to be
- ZIV if it contains no index (zero index variable)
- SIV if it contains only one index (single index variable)
- MIV if it contains more than one index (multiple index variables)

For Example:

\[ A(5,I+1,j) = A(1,I,k) + C \]

First subscript equation is ZIV
Second subscript equation is SIV
Third subscript equation is MIV
Dependence Testing: Overview

- Partition subscripts of a pair of array references into separable and coupled groups
- Classify each subscript as ZIV, SIV or MIV
- For each separable subscript apply single subscript test. If not done goto next step
- For each coupled group apply multiple subscript test
- If still not done, merge all direction vectors computed in the previous steps into a single set of direction vectors
Linear Diophantine Equations

- A basic result tells us that there are values for $x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n$ so that
  \[ a_1 x_1 - b_1 y_1 + \ldots + a_n x_n - b_n y_n = \gcd(a_1, \ldots, a_n, b_1, \ldots, b_n) \]

  What's more, $\gcd(a_1, \ldots, a_n, b_1, \ldots, b_n)$ is the smallest number this is true for.

- As a result, the equation has a solution iff $\gcd(a_1, \ldots, a_n, b_1, \ldots, b_n)$ divides $b_0 - a_0$
  - But the solution may not be in the region (loop iteration values) of interest

- Exercise: try this result on the $A(4*i+2) \& A(4*i+4)$ example
Real Solutions

• Unfortunately, the gcd test is less useful than it might seem.
• Useful technique is to show that the equation has no solutions in region of interest ==> explore real solutions for this purpose
• Solving $h(x) = 0$ is essentially an integer programming problem. Linear programming techniques are used as an approximation.
• Since the function is continuous, the Intermediate Value Theorem says that a solution exists iff:

$$\min_R h \leq 0 \leq \max_R h$$
Banerjee Inequality

- We need an easy way to calculate $\min_R h$ and $\max_R h$.

- Definitions:

\[ h_i^+ = \max_{R_i} h(x_i, y_i) \quad a^+ = \begin{cases} a & a \geq 0 \\ 0 & a < 0 \end{cases} \]

\[ h_i^- = \min_{R_i} h(x_i, y_i) \quad a^- = \begin{cases} |a| & a < 0 \\ 0 & a \geq 0 \end{cases} \]

- $a^+$ and $a^-$ are both $\geq 0$ and are called the positive part and negative part of $a$, so that $a = a^+ - a^-$
Theorem 3.3 (Banerjee). Let $D$ be a direction vector, and $h$ be a dependence function. $h = 0$ can be solved in the region $R$ iff:

$$\sum_{i=1}^{n} H_i^{-}(D_i) \leq b_0 - a_0 \leq \sum_{i=1}^{n} H_i^{+}(D_i)$$

Proof: Immediate from Lemma 3.3 and the IMV.
Preliminary Transformations

Chapter 4 of Allen and Kennedy
Loop Normalization

• Transform loop so that
  — The new stride becomes +1 (more important)
  — The new lower bound becomes +1 (less important)

• To make dependence testing as simple as possible

• Serves as information gathering phase
Enhancing Fine-Grained Parallelism

Chapter 5 of Allen and Kennedy
Chapter 2’s Codegen

- Codegen: tries to find parallelism using transformations of loop distribution and statement reordering

- If we deal with loops containing cyclic dependences early on in the loop nest, we can potentially vectorize more loops

- Goal in Chapter 5: To explore other transformations to exploit parallelism
Loop Interchange: Safety

- Theorem 5.1 Let $D(i,j)$ be a direction vector for a dependence in a perfect nest of loops. Then the direction vector for the same dependence after a permutation of the loops in the nest is determined by applying the same permutation to the elements of $D(i,j)$.

```fortran
DO I = 1, L
  DO J = 1, M
    DO K = 1, N
      A(I+1,J+1,K) = B(I,J,K)
    ENDDO
  ENDDO
ENDDO
```

Dependence: $(<, <, =)$

```fortran
DO I = 1, L
  DO K = 1, N
    DO J = 1, M
      A(I+1,J+1,K) = B(I,J,K)
    ENDDO
  ENDDO
ENDDO
```

Dependence: $(<, =, <)$
Loop Interchange: Safety

• Theorem 5.2 A permutation of the loops in a perfect nest is legal if and only if the direction matrix, after the same permutation is applied to its columns, has no ">" direction as the leftmost non-"=" direction in any row.

• Follows from Theorem 5.1 and Theorem 2.3

Example:

\[
\begin{array}{ccc}
  i & j & k \\
  < & < & = \\
  < & = & > \\
\end{array}
\quad \xrightarrow{\text{permutation}} \quad
\begin{array}{ccc}
  j & k & i \\
  < & = & < \\
  = & > & < \\
\end{array}
\]
Fully Permutable Loop Nest

- A contiguous set of $k \geq 1$ loops, $i_j, \ldots, i_{j+k-1}$ is fully permutable if all permutations of $i_j, \ldots, i_{j+k-1}$ are legal.

- Data dependence test: Loops $i_j, \ldots, i_{j+k-1}$ are fully permutable if for each dependence vector $(d_1, \ldots, d_n)$ carried at levels $j \ldots j+k-1$, each of $d_j, \ldots, d_{j+k-1}$ is non-negative.

- Fundamental result (to be discussed later in course): a set of $k$ fully permutable loops can be transformed using only Interchange, Reversal and Skewing transformations into an equivalent set of $k$ loops where $k-1$ of the loops are parallel.
Scalar Expansion and its use in Removing Anti and Output Dependences

DO I = 1, N
S1   T = A(I)
S2   A(I) = B(I)
S3   B(I) = T
ENDDO

• **Scalar Expansion:**

DO I = 1, N
S1   T$(I) = A(I)
S2   A(I) = B(I)
S3   B(I) = T$(I)
ENDDO
T = T$(N)

• **leads to:**

S1   T$(1:N) = A(1:N)
S2   A(1:N) = B(1:N)
S3   B(1:N) = T$(1:N)
Scalar Expansion: Covering Definitions

- A definition $D$ of a scalar $S$ is a covering definition for loop $L$ if a definition of $S$ placed at the beginning of $L$ reaches no uses of $S$ that occur past $D$.

```fortran
DO I = 1, 100
    T = X(I)
    Y(I) = T
ENDDO

DO I = 1, 100
    IF (A(I) .GT. 0) THEN
        T = X(I)
        Y(I) = T
    ENDIF
ENDDO
```
Scalar Renaming

- Renaming algorithm partitions all definitions and uses into equivalent classes, each of which can occupy different memory locations.

- Use the definition-use graph to:
  - Pick definition
  - Add all uses that the definition reaches to the equivalence class
  - Add all definitions that reach any of the uses...
  - ...until fixed point is reached

- Example:

  ```
  IF (...) THEN
    S_1  T = ...
    ELSE
      S_2  T = ...
    ENDIF
    S_3  ... = T
  ELSE
    T1 = ...
    ELSE
      T2 = ...
    ENDIF
    S_4  T = ...
    S_5  ... = T
  ENDIF
  ```
Array Renaming

DO I = 1, N
S_1 \quad A(I) = A(I-1) + X
S_2 \quad Y(I) = A(I) + Z
S_3 \quad A(I) = B(I) + C
ENDDO

- \(S_1 \delta_\infty S_2\) \quad \(S_2 \delta_\infty^{-1} S_3\) \quad \(S_3 \delta_1 S_1\) \quad \(S_1 \delta_\infty^0 S_3\)

- Rename \(A(I)\) to \(A'(I)\):
  DO I = 1, N
  S_1 \quad A'(I) = A(I-1) + X
  S_2 \quad Y(I) = A'(I) + Z
  S_3 \quad A(I) = B(I) + C
  ENDDO

- Dependences remaining: \(S_1 \delta_\infty^2 S_2\) and \(S_3 \delta_1 S_1\)
Node Splitting

• **Sometimes Renaming fails**

\[
\text{DO } I = 1, N \\
\quad S1: \quad A(I) = X(I+1) + X(I) \\
\quad S2: \quad X(I+1) = B(I) + 32 \\
\text{ENDDO}
\]

• **Recurrence kept intact by renaming algorithm**
Node Splitting

\[ \text{DO } I = 1, N \]
\[ \text{S1: } A(I) = X(I+1) + X(I) \]
\[ \text{S2: } X(I+1) = B(I) + 32 \]
\text{ENDDO} \]

• Break critical antidependence
• Make copy of read from which antidependence emanates

\[ \text{DO } I = 1, N \]
\[ \text{S1': } X$(I) = X(I+1) \]
\[ \text{S1: } A(I) = X$(I) + X(I) \]
\[ \text{S2: } X(I+1) = B(I) + 32 \]
\text{ENDDO} \]

• Recurrence broken
• Vectorized to
\[ \text{S1': } X$(1:N) = X(2:N+1) \]
\[ \text{S2: } X(2:N+1) = B(1:N) + 32 \]
\[ \text{S1: } A(1:N) = X$(1:N) + X(1:N) \]
Recognition of Reductions

- **Assuming commutativity and associativity**

\[
\begin{align*}
S &= 0.0 \\
&\text{DO } k = 1, 4 \\
&\quad \text{SUM}(k) = 0.0 \\
&\quad \text{DO } I = k, N, 4 \\
&\quad \quad \text{SUM}(k) = \text{SUM}(k) + A(I) \\
&\quad \text{ENDDO} \\
&\quad S = S + \text{SUM}(k) \\
&\text{ENDDO}
\end{align*}
\]

- **Distribute k loop**

\[
\begin{align*}
S &= 0.0 \\
&\text{DO } k = 1, 4 \\
&\quad \text{SUM}(k) = 0.0 \\
&\text{ENDDO} \\
&\text{DO } k = 1, 4 \\
&\quad \text{DO } I = k, N, 4 \\
&\quad \quad \text{SUM}(k) = \text{SUM}(k) + A(I) \\
&\quad \text{ENDDO} \\
&\quad \text{ENDDO} \\
&\text{DO } k = 1, 4 \\
&\quad S = S + \text{SUM}(k) \\
&\text{ENDDO}
\end{align*}
\]
Recognition of Reductions

- **After Loop Interchange**
  
  ```
  DO I = 1, N, 4
    DO k = I, min(I+3,N)
      SUM(k-I+1) = SUM(k-I+1) + A(I)
    ENDDO
  ENDDO
  ```

- **Vectorize**

  ```
  DO I = 1, N, 4
    SUM(1:4) = SUM(1:4) + A(I:I+3)
  ENDDO
  ```
Index-set Splitting

- Subdivide loop into different iteration ranges to achieve partial parallelization
  - Threshold Analysis [Strong SIV, Weak Crossing SIV]
  - Loop Peeling [Weak Zero SIV]
  - Section Based Splitting [Variation of loop peeling]
Threshold Analysis

**Threshold Analysis**

```
DO I = 1, 20
    A(I+20) = A(I) + B
ENDDO

Vectorize to..
A(21:40) = A(1:20) + B
```

```
DO I = 1, 100
    A(I+20) = A(I) + B
ENDDO

Strip mine to..
DO I = 1, 100, 20
    DO i = I, I+19
        A(i+20) = A(i) + B
    ENDDO
ENDDO
```

Vectorize this
Loop Peeling

- **Source of dependence is a single iteration**

  ```
  DO I = 1, N
    A(I) = A(I) + A(1)
  ENDDO
  
  Loop peeled to..
  A(1) = A(1) + A(1)
  DO I = 2, N
    A(I) = A(I) + A(1)
  ENDDO
  
  Vectorize to..
  A(1) = A(1) + A(1)
  A(2:N) = A(2:N) + A(1)
  ```
Run-time Symbolic Resolution

• “Breaking Conditions”

DO I = 1, N
       A(I+L) = A(I) + B(I)
ENDDO

Transformed to..

IF(L.LE.0 .OR. L.GT.N) THEN
       A(L+1:N+L) = A(1:N) + B(1:N)
ELSE
       DO I = 1, N
              A(I+L) = A(I) + B(I)
       ENDDO
ENDIF
Loop Skewing

- Reshape Iteration Space to uncover parallelism

\[
\begin{align*}
\text{DO I = 1, N} \\
\quad \text{DO J = 1, N} \\
\quad S: & \quad A(I,J) = A(I-1,J) + A(I,J-1) \\
& \quad (<,=) \\
\quad \text{ENDDO} \\
\text{ENDDO}
\end{align*}
\]

Parallelism not apparent
Loop Skewing

- Dependence Pattern before loop skewing
Loop Skewing

• Do the following transformation called loop skewing

\[ jj = J + I \text{ or } J = jj - I \]

\[ \text{DO I = 1, N} \]
  \[ \text{DO } jj = I + 1, I + N \]
  \[ J = jj - I \]
  \[ S: \ A(I, J) = A(I-1, J) + A(I, J-1) \]

ENDDO
ENDDO

Note: Direction Vector Changes, but statement body remains the same
(Examples in textbook usually copy propagate J=jj-I in all uses of J)
Loop Skewing

- Dependence pattern after loop skewing
Pipeline Parallelism with Strip Mining

POST (EV(1, 1))

DOACROSS I = 2, N-1

   K = 0

   DO J = 2, N-1, 2  ! CHUNK SIZE = 2

      K = K+1

      WAIT (EV(I-1,K))

      DO m = J, MIN(J+1, N-1)

         A(I, m) = .25 * (A(I-1, m) + A(I, m-1) + A(I+1, m) + A(I, m+1))

      ENDDO

   ENDDO

   POST (EV(I, K+1))

ENDDO

ENDDO
Coarse-Grain Parallelism

Chapter 6 of Allen and Kennedy
Single Loops

- The analog of scalar expansion is privatization.
- Temporaries can be given separate namespaces for each iteration.
Definition: A scalar variable $x$ in a loop $L$ is said to be privatizable if every path from the loop entry to a use of $x$ inside the loop passes through a definition of $x$.

Privatizability can be stated as a data-flow problem:

\[
up(x) = \text{use}(x) \cup (\neg \text{def}(x) \cap \bigcup_{y \in \text{succ}(x)} up(y))
\]

\[
\text{private}(L) = \neg up(\text{entry}) \cap (\bigcup_{y \in L} \text{def}(y))
\]

We can also do this by declaring a variable $x$ private if its SSA graph doesn't contain a phi function at the entry.
We need to privatize array variables.

For iteration J, upwards exposed variables are those exposed due to loop body without variables defined earlier.

\[
up(L_1) = \bigcup_{J=2}^N (\{T(J-1)\} \setminus \{T(n) : 2 \leq n \leq J\})
\]

So for this fragment, T(1) is the only exposed variable.
Array Privatization

- Using this analysis, we get the following code:

```
PARALLEL DO I = 1,100
   PRIVATE t(N)
S0   t(1) = X
L1   DO J = 2,N
S1       t(J) = t(J-1)+B(I,J)
S2       A(I,J)=t(J)
   ENDDO
   ENDDO
```
Loop Alignment

- Many carried dependencies are due to array alignment issues.
- If we can align all references, then dependencies would go away, and parallelism is possible.
- This is also related to Software Pipelining

\[
\begin{align*}
\text{DO } I &= 2,N \\
A(I) &= B(I)+C(I) \\
D(I) &= A(I-1)*2.0 \\
\text{ENDDO}
\end{align*} \quad \begin{align*}
\text{DO } I &= 1,N  \\
\text{IF } (I \text{ .GT. 1}) A(I) &= B(I)+C(I) \\
\text{IF } (I \text{ .LT. N}) D(I+1) &= A(I)*2.0 \\
\text{ENDDO}
\end{align*}
\]

\[
\begin{align*}
D(2) &= A(1)*2.0 \\
\text{DO } I &= 2,N-1  \\
A(I) &= B(I)+C(I) \\
D(I) &= A(I)*2.0 \\
\text{ENDDO}
\end{align*} \quad \begin{align*}
A(N) &= B(N)+C(N)
\end{align*}
\]
Code Replication

- If an array is involved in a recurrence, then alignment isn’t possible.
- If two dependencies between the same statements have different dependency distances, then alignment doesn’t work.
- We can fix the second case by replicating code:

```plaintext
DO I = 1,N
  A(I+1) = B(I)+C
  ! Replicated Statement
  IF (I .EQ 1) THEN
    t = A(I)
  ELSE
    t = B(I-1)+C
  END IF
  X(I) = A(I+1)+t
ENDDO
```
Strip Mining

- Converts available parallelism into a form more suitable for the hardware (assume \text{THRESHOLD} = \text{minimum iters for parallel loop})

\begin{align*}
\text{DO } I = 1, N \\
\ A(I) &= A(I) + B(I) \\
\text{ENDDO}
\end{align*}

\Rightarrow

\begin{align*}
\text{k} &= \text{MAX(THRESHOLD, CEIL(N / P))} \\
\text{PARALLEL DO } I = 1, N, k \\
\text{DO } i = I, \text{MIN}(I + k-1, N) \\
\ A(i) &= A(i) + B(i) \\
\text{ENDDO}
\end{align*}

\text{END PARALLEL DO}
Definition: A loop-independent dependence between statements $S1$ and $S2$ in loops $L1$ and $L2$ respectively is fusion-preventing if fusing $L1$ and $L2$ causes the dependence to be carried by the combined loop in the opposite direction.

```
DO I = 1,N
  S1  A(I) = B(I)+C
  ENDDO
DO I = 1,N
  S2  D(I) = A(I+1)+E
  ENDDO
```
Fusion Safety: Ordering Constraint

- We shouldn’t fuse loops if the fusing will violate ordering of the dependence graph.
- **Ordering Constraint:** Two loops can’t be validly fused if there exists a path of loop-independent dependencies between them containing a loop or statement not being fused with them i.e., if fusion will result in a cycle in the resulting loop-independent dependences.

Fusing L1 with L3 violates the ordering constraint. \{L1, L3\} must occur both before and after the node L2.
Loop Interchange

- Parallelization: move dependence-free loops to outermost level
- Theorem 6.3
  
  In a perfect nest of loops, a particular loop can be parallelized at the outermost level if and only if the column of the direction matrix for that nest contains only '=' entries
Motivation for Loop Interchange

DO I = 1, N
    DO J = 1, N
        A(I+1, J) = A(I, J) + B(I, J)
    ENDDO
ENDDO

• Parallelizing the J loop is OK for vectorization
• But inefficient for parallelization (N barriers)
Loop Interchange

PARALLEL DO J = 1, N
  DO I = 1, N
    A(I+1, J) = A(I, J) + B(I, J) (=, <)
  ENDDO
END PARALLEL DO
Loop Reversal

DO I = 2, N+1
  DO J = 2, M+1
    DO K = 1, L
      A(I, J, K) = A(I, J-1, K+1) + A(I-1, J, K+1)
    ENDDO
  ENDDO
ENDDO
ENDDO
ENDDO
Loop Reversal

DO I = 2, N+1
  DO J = 2, M+1
    DO K = L, 1, -1
      A(I, J, K) = A(I, J-1, K+1) + A(I-1, J, K+1)
    ENDDO
  ENDDO
ENDDO

\[
\begin{pmatrix}
I & J & K \\
< & > & = \\
<= & > & =<
\end{pmatrix}
\rightarrow
\begin{pmatrix}
I & J & K \\
<< & < & =<
\end{pmatrix}
\]
After Loop Reversal & Interchange

\[
\text{DO } K = L, 1, -1 \\
\text{PARALLEL DO } I = 2, N+1 \\
\text{PARALLEL DO } J = 2, M+1 \\
A(I, J, K) = A(I, J-1, K+1) + A(I-1, J, K+1) \\
\text{END PARALLEL DO} \\
\text{END PARALLEL DO} \\
\text{ENDDO}
\]

- Increase the range of options available for loop selection heuristics
Loop Skewing

DO I = 2, N+1
  DO J = 2, M+1
    DO K = 1, L
      A(I, J, K) = A(I, J-1, K) + A(I-1, J, K)
      B(I, J, K+1) = B(I, J, K) + A(I, J, K)
    ENDDO
  ENDDO
ENDDO
Loop Skewing

- Skewed using \( k = K + I + J \) yield:

\[
\begin{align*}
&\text{DO } I = 2, \ N+1 \\
&\quad \text{DO } J = 2, \ M+1 \\
&\quad \quad \text{DO } k = I+J+1, \ I+J+L \\
&\quad \quad \quad A(I, J, k-I-J) = A(I, J-1, k-I-J) + A(I-1, J, k-I-J) \\
&\quad \quad \quad B(I, J, k-I-J+1) = B(I, J, k-I-J) + A(I, J, k-I-J) \\
&\quad \quad \quad \text{ENDDO} \\
&\quad \text{ENDDO} \\
&\text{ENDDO}
\end{align*}
\]
Loop Skewing

\[ \text{DO } k = 5, N+M+1 \]
\[ \text{PARALLEL DO } I = \text{MAX}(2, k-M-L-1), \text{MIN}(N+1, k-L-2) \]
\[ \text{PARALLEL DO } J = \text{MAX}(2, k-I-L), \text{MIN}(M+1, k-I-1) \]
\[ \text{ENDDO} \]
\[ \text{ENDDO} \]
\[ \text{ENDDO} \]
Loop Skewing

- Transforms skewed loop into one that can be interchanged to the outermost position without changing the meaning of the program.
- Can be used to transform the skewed loop in such a way that, after outward interchange, it will carry all dependences formerly carried by the loop with respect to which it is skewed.

\[
\begin{align*}
k & = k \\
I & = I \\
J & = J
\end{align*}
\]
Pipeline Parallelism

POST (EV(1, 2))

DOACROSS I = 2, N-1

    DO J = 2, N-1

        WAIT (EV(I-1, J))

        A(I, J) = 0.25 * (A(I-1, J) + A(I, J-1) + A(I+1, J) + A(I, J+1))

        POST (EV(I, J))

    ENDDO

ENDDO
Pipeline Parallelism

Event synchronization

I = 2

J = 2

J = 3

J = 4

J = 5

J = 6

J = 7

I = 3

J = 2

J = 3

J = 4

J = 5

I = 4

J = 2

J = 3

J = 4

J = 5

I = 5

J = 2

J = 3

J = 4

J = 5

J = 4

77
Pipeline Parallelism with Strip Mining

POST (EV(1, 1))

DOACROSS I = 2, N-1
   
   K = 0
   
   DO J = 2, N-1, 2  ! CHUNK SIZE = 2
      
      K = K+1
      
      WAIT (EV(I-1,K))
      
      DO m = J, MIN(J+1, N-1)
         
         A(I, m) = .25 * (A(I-1, m) + A(I, m-1) + A(I+1, m) + A(I, m+1))
         
         ENDDO
         
      POST (EV(I, K+1))
      
   ENDDO
   
ENDDO

ENDDO
Pipeline Parallelism
Midterm exam

• Take-home exam (3 hours)
  — Open book: textbook only, no other resources
  — Will be made available on Monday, Oct 10th, and will be due by 5pm on Monday, Oct 17th