COMP 515: Advanced Compilation for Vector and Parallel Processors

Vivek Sarkar
Department of Computer Science
Rice University
vsarkar@rice.edu

http://www.cs.rice.edu/~vsarkar/comp515

COMP 515 Lecture 25 1 December, 2011

Acknowledgments

- Slides from previous offerings of COMP 515 by Prof. Ken Kennedy
  - http://www.cs.rice.edu/~ken/comp515/
- Other acknowledgments included with individual lectures

End-semester Summary

Chapters 7,8,9,11,13 of Allen and Kennedy book

Control Dependences

Chapter 7

Control Dependences

- **Constraints posed by control flow**

```
          DO 100 I = 1, N
S1            IF (A(I-1).GT. 0.0) GO TO 100
S2                A(I) = A(I) + B(I)*C
100   CONTINUE
```

$S_2 \; \delta_1 \; S_1$

If we vectorize by...

```
S2   A(1:N) = A(1:N) + B(1:N)*C
     DO 100 I = 1, N
S1        IF (A(I-1).GT. 0.0) GO TO 100
100 CONTINUE
```

...we get the wrong answer

- **We are missing dependences**
- **There is a dependence from S_1 to S_2 - a control dependence**

Branch removal for If-conversion

- **Basic idea:**
  - Make a pass through the program.
  - Maintain a Boolean expression cc that represents the condition that must be true for the current expression to be executed
  - On encountering a branch, conjoin the controlling expression into cc
  - On encountering a target of a branch, its controlling expression is disjoined into cc

Branch Removal: Forward Branches

- Remove forward branches by inserting appropriate guards

```
         DO 100 I = 1,N
C1           IF (A(I).GT.10) GO TO 60
20               A(I) = A(I) + 10
C2               IF (B(I).GT.10) GO TO 80
40                   B(I) = B(I) + 10
60               A(I) = B(I) + A(I)
80           B(I) = A(I) - 5
         ENDDO
==>
     DO 100 I = 1,N
             m1 = A(I).GT.10
20           IF(.NOT.m1) A(I) = A(I) + 10
             IF(.NOT.m1) m2 = B(I).GT.10
40           IF(.NOT.m1.AND..NOT.m2) B(I) = B(I) + 10
60           IF(.NOT.m1.AND..NOT.m2.OR.m1)A(I) = B(I) + A(I)
80           IF(.NOT.m1.AND..NOT.m2.OR.m1.OR..NOT.m1
                   .AND.m2) B(I) = A(I) - 5
     ENDDO
```

Branch Removal: Forward Branches

- **We can simplify to:**

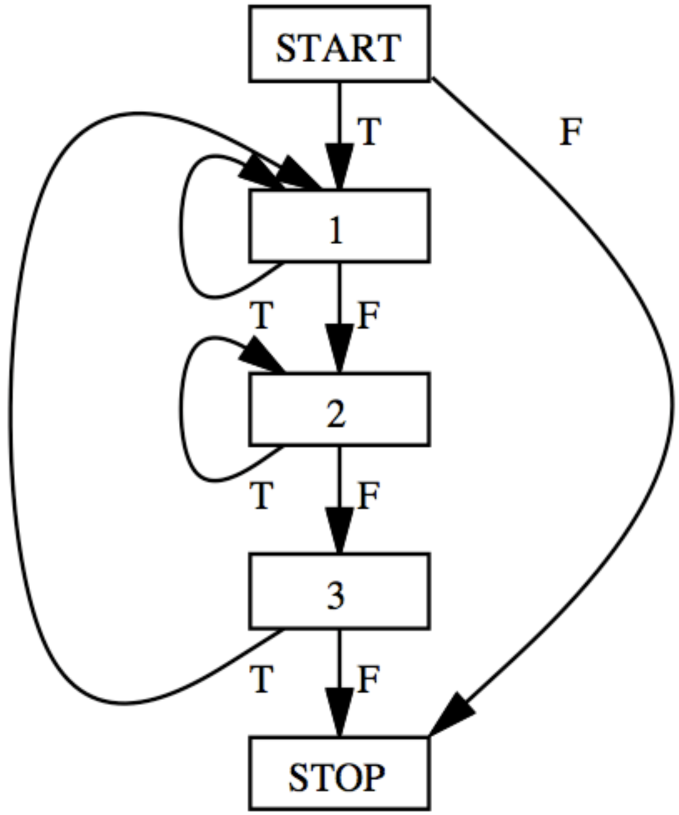
```
     DO 100 I = 1,N
                 m1 = A(I).GT.10
 20              IF(.NOT.m1) A(I) = A(I) + 10
                 IF(.NOT.m1) m2 = B(I).GT.10
 40              IF(.NOT.m1.AND..NOT.m2)
                         B(I) = B(I) + 10
 60              IF(m1.OR..NOT.m2)
                         A(I) = B(I) + A(I)
 80              B(I) = A(I) - 5
     ENDDO
```

- **and then vectorize to:**

```
     m1(1:N) = A(1:N).GT.10
 20  WHERE(.NOT.m1(1:N)) A(1:N) = A(1:N) + 10
     WHERE(.NOT.m1(1:N)) m2(1:N) = B(1:N).GT.10
 40  WHERE(.NOT.m1(1:N).AND..NOT.m2(1:N)) B(1:N) = B(1:N) + 10
 60  WHERE(m1(1:N).OR..NOT.m2(1:N)) A(1:N) = B(1:N) + A(1:N)
 80  B(1:N) = A(1:N) - 5
```

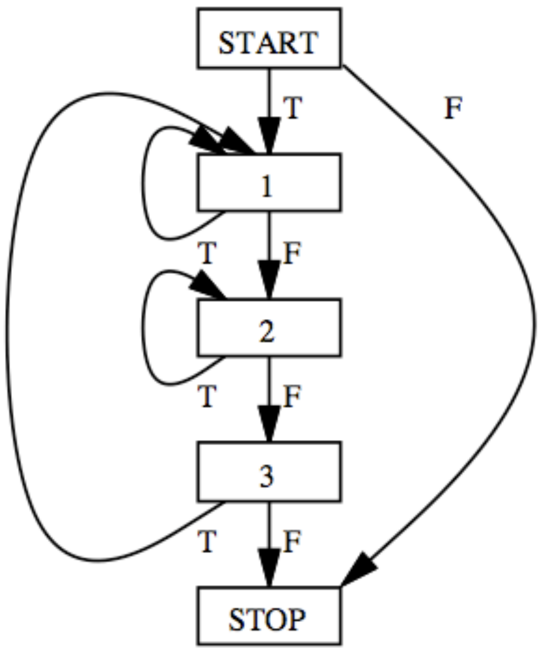
Control Flow Graph: Example

```
do {
  S1;
  if ( C1 ) continue;
  do {
    S2;
  } while ( C2 );
  S3;
} while ( C3 );
```

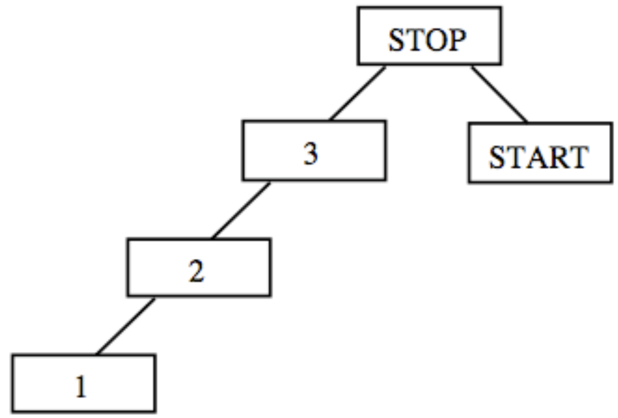


CONTROL FLOW GRAPH

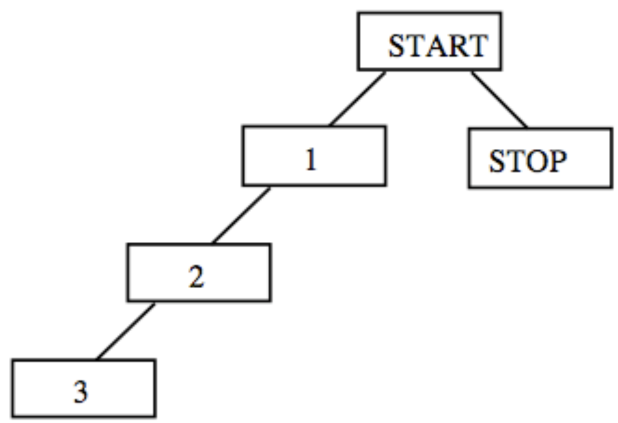
Examples of Dominator and Postdominator Trees



CONTROL FLOW GRAPH



POST–DOMINATOR TREE



DOMINATOR TREE

Control Dependence: Definition

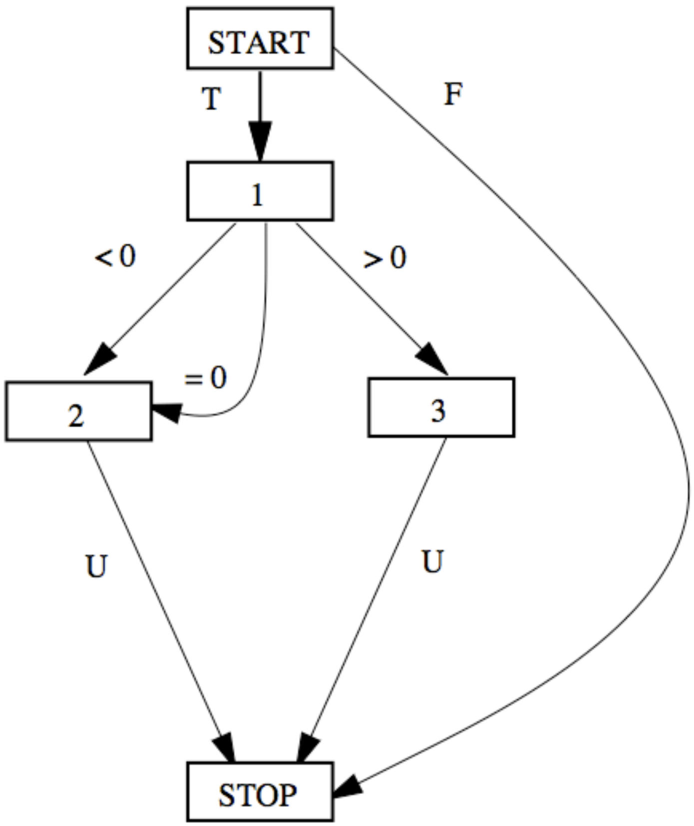
Node Y is *control dependent* on node X with label L in *CFG* if and only if

1. there exists a nonnull path $X \longrightarrow Y$, starting with the edge labeled L, such that Y post-dominates every node, W, strictly between X and Y in the path, and

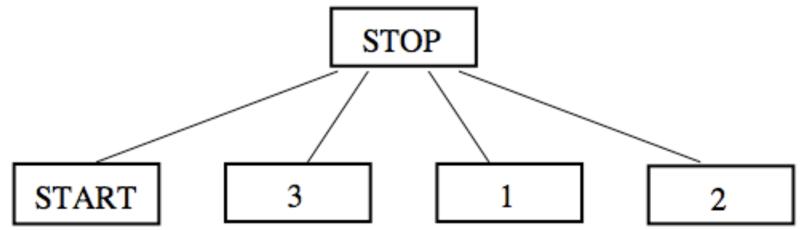
2. Y does not post-dominate X.

Reference: "The Program Dependence Graph and its Use in Optimization", J. Ferrante et al, *ACM TOPLAS, 1987*

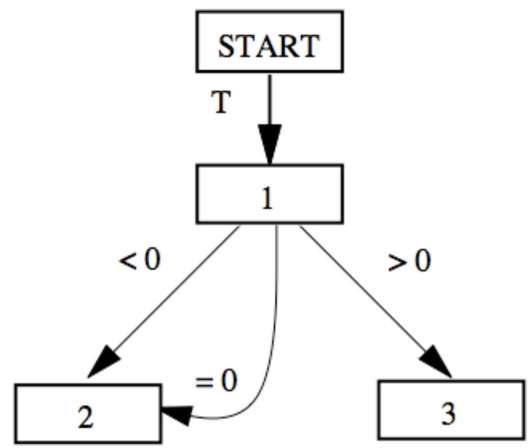
Example: Acyclic CFG and its Control Dependence Graph (CDG)



CONTROL FLOW GRAPH



POSTDOMINATOR TREE

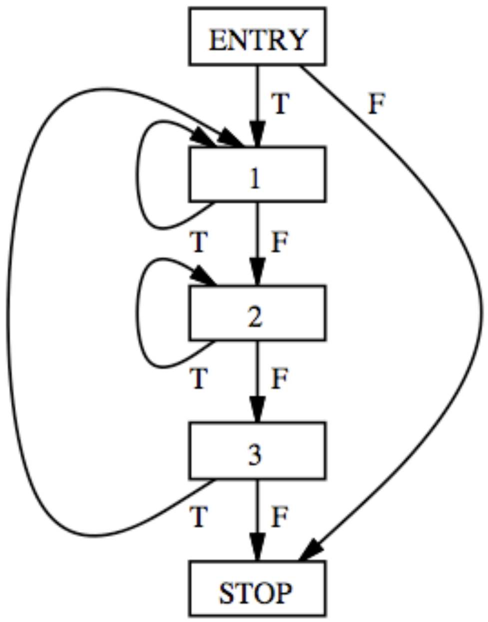


CONTROL DEPENDENCE GRAPH

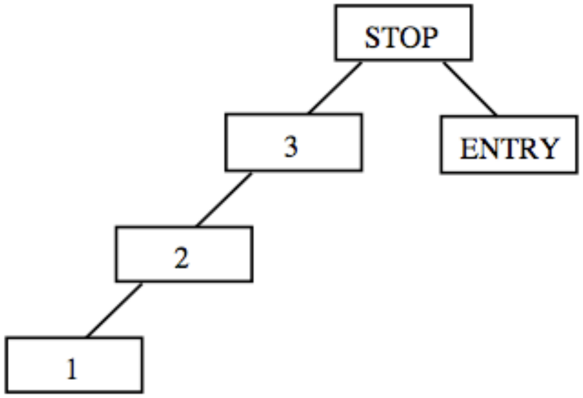
Control Dependence: Discussion

- A node x in directed graph G with a single exit node postdominates node y in G if any path from y to the exit node of G must pass through x.
- A statement y is said to be control dependent on another statement x if:
  - there exists a non-trivial path from x to y such that every statement $z \neq x$ in the path is postdominated by y and
  - x is not postdominated by y.
- In other words, a control dependence exists from S1 to S2 if one branch out of S1 forces execution of S2 and another doesn't
- Note that control dependences also can be seen at as a property of basic blocks (depends on CFG granularity)

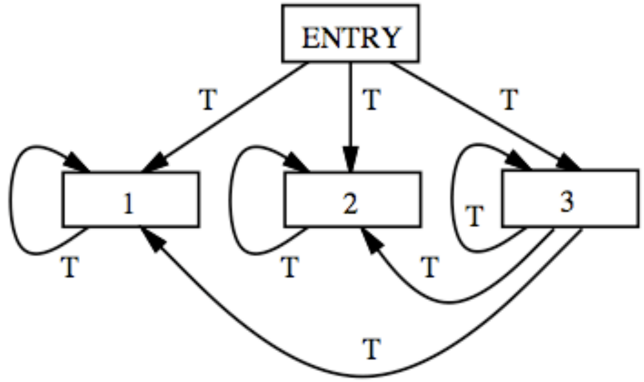
Example: Cyclic CFG and its CDG



CONTROL FLOW GRAPH



POST-DOMINATOR TREE



CONTROL DEPENDENCE GRAPH

CDG for a Cyclic CFG

Problem: CFG and CDG can have different loop/interval structures, in general

Solution: Compute CDG only for acyclic CFG's e.g.

1. Restrict construction and use of CDG's to innermost intervals with acyclic CFG's.

2. Compute CDG for acyclic Forward Control Flow Graph), which captures CFG's loop structure by insertion of pseudo nodes and edges. [Cytron, Ferrante, Sarkar 1990]

3. Compute CDG for each interval with an acyclic CFG, treating subintervals as atomic nodes.

Conclusion

- Idea behind control flow dependences
- If-conversion
  - Types of branches and branch removal
  - Iterative dependences (append range to each statement)
- Control Dependence Procedure as alternative to if-conversion
- Execution model for control dependence graphs
- Loop Distribution (selective if-conversion)
- Code Generation

Compiler Improvement of Register Usage

Chapter 8

Scalar Replacement

- Example: Scalar Replacement in case of loop independent dependence

```
DO I = 1, N
     A(I) = B(I) + C
     X(I) = A(I)*Q
ENDDO
```

```
DO I = 1, N
     t = B(I) + C
     A(I) = t
     X(I) = t*Q
ENDDO
```

- One fewer load for each iteration for reference to *A*

Scalar Replacement

- Example: Scalar Replacement in case of loop carried dependence spanning single iteration

```
DO I = 1, N
    A(I) = B(I-1)
    B(I) = A(I) + C(I)
ENDDO
```

```
tB = B(0)
DO I = 1, N
    tA = tB
    A(I) = tA
    tB = tA + C(I)
    B(I) = tB
ENDDO
```

- One fewer load for each iteration for reference to B which had a loop carried true dependence spanning 1 iteration
- Also one fewer load per iteration for reference to A

Scalar Replacement

- Example: Scalar Replacement in case of loop carried dependence spanning multiple iterations

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

```
t1 = B(0)
t2 = B(1)
DO I = 1, N
    t3 = B(I+1)
    A(I) = t1 + t3
    t1 = t2
    t2 = t3
ENDDO
```

- One fewer load for each iteration for reference to B which had a loop carried input dependence spanning 2 iterations
- Invariants maintained were

  `t1=B(I-1);t2=B(I);t3=B(I+1)`

Eliminate Scalar Copies

```
t1 = B(0)
t2 = B(1)
DO I = 1, N
    t3 = B(I+1)
    A(I) = t1 + t3
    t1 = t2
    t2 = t3
ENDDO
```

- Unnecessary register-register copies
- Unroll loop 3 times

```
t1 = B(0)
t2 = B(1)
mN3 = MOD(N,3)
DO I = 1, mN3
    t3 = B(I+1)
    A(I) = t1 + t3
    t1 = t2
    t2 = t3
ENDDO
DO I = mN3 + 1, N, 3
    t3 = B(I+1)
    A(I) = t1 + t3
    t1 = B(I+2)
    A(I+1) = t2 + t1
    t2 = B(I+3)
    A(I+2) = t3 + t2
ENDDO
```

Preloop

Main Loop

Scalar Replacement: Putting it together

1. Prune dependence graph; Apply typed fusion
2. Select a set of name partitions using register pressure moderation
3. For each selected partition
   A) If non-cyclic, replace using set of temporaries
   B) If cyclic replace reference with single temporary
   C) For each inconsistent dependence
      Use index set splitting or insert loads and stores
4. Unroll loop to eliminate scalar copies

Scalar Replacement: Case A

```
DO I = 1, N

      A(I+1) = A(I-1) + B(I-1)

      A(I) = A(I) + B(I) + B(I+1)

ENDDO
```

```
t0A = A(0); t1A0 = A(1);
tB1 = B(0); tB2 = B(1)
DO I = 1, N
        t1A1 = t0A + tB1
        tB3 = B(I+1)
        t0A = t1A0 + tB3 + tB2
        A(I) = t0A
        t1A0 = t1A1
        tB1 = tB2
        tB2 = tB3
ENDDO
A(N+1) = t1A1
```

Scalar Replacement: Case B

```
DO I = 1, N

      A(J) = B(I) + C(I,J)

      C(I,J) = A(J) + D(I)

ENDDO
```

replace with single temporary...

```
DO I = 1, N
      tA = B(I) + C(I,J)
      C(I,J) = tA + D(I)
ENDDO
A(J) = tA
```

Scalar Replacement: Case C

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

- Split this loop into three separate parts
  - A loop up to J
  - Iteration J
  - A loop after iteration J to N

```
tAI = A(0); tAJ = A(J)
JU = MAX(J-1,0)
DO I = 1, JU
   tAI = tAI + B(I); A(I) = tAI
   tAJ = tAJ + tAI
ENDDO
IF(J.GT.0.AND.J.LE.N) THEN
   tAI = tAI + B(I); A(I) = tAI
   tAJ = tAJ + tAI
   tAI = tAJ
ENDIF
DO I = JU+2, N
   tAI = tAI + B(I); A(I) = tAI
   tAJ = tAJ + tAI
ENDDO
A(J) = tAJ
```

Conditions for legality of unroll-and-jam

- **Definition:** Unroll-and-jam to factor n consists of unrolling the outer loop n-1 times and fusing those copies together.
- **Theorem:** An unroll-and-jam to a factor of n is legal iff there exists no dependence with direction vector (<,>) such that the distance for the outer loop is less than n.

Unroll-and-jam Algorithm

1. Create preloop
2. Unroll main loop m(the unroll-and-jam factor) times
3. Apply typed fusion to loops within the body of the unrolled loop
4. Apply unroll-and-jam recursively to the inner nested loop

Unroll-and-jam example

```
DO I = 1, N
 DO K = 1, N
  A(I) = A(I) + X(I,K)
 ENDDO
 DO J = 1, M
   DO K = 1, N
        B(J,K) = B(J,K) + A(I)
   ENDDO
 ENDDO
 DO J = 1, M
  C(J,I) = B(J,N)/A(I)
 ENDDO
ENDDO
```

```
DO I = mN2+1, N, 2
 DO K = 1, N
  A(I) = A(I) + X(I,K)
  A(I+1) = A(I+1) + X(I+1,K)
 ENDDO
 DO J = 1, M
   DO K = 1, N
     B(J,K) = B(J,K) + A(I)
     B(J,K) = B(J,K) + A(I+1)
   ENDDO
   C(J,I) = B(J,N)/A(I)
   C(J,I+1) = B(J,N)/A(I+1)
 ENDDO
ENDDO
```

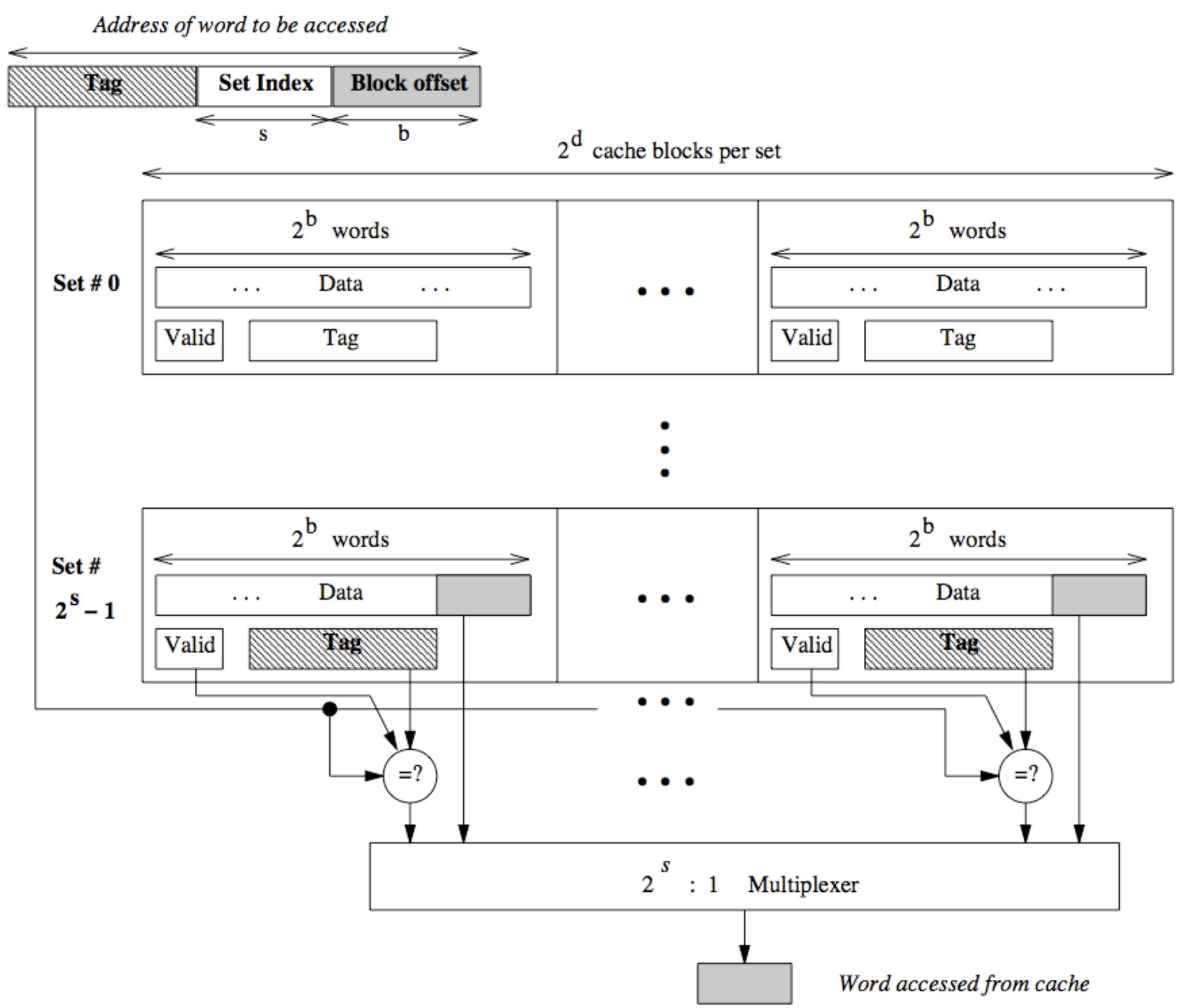
Conclusion

- We have learned two memory hierarchy transformations:
  - scalar replacement
  - unroll-and-jam
- They reduce the number of memory accesses by maximum use of processor registers

Managing Cache

Allen and Kennedy, Chapter 9

Review: How do set-associative caches work?



Cost Assignment

- **Consider cost analysis for an innermost loop with N iterations, for arrays with element size = s, and a cache with line size = l**
- **Cost is 1 for references that do not depend on loop induction variables**
- **Cost is N for references based on induction variables over a non-contiguous space**
- **Cost is Ns/l for induction variables based references over contiguous space**
- **Multiply the cost by the loop trip count if the reference varies with the loop index**

Loop Blocking (Tiling)

```
DO J = 1, M
  DO I = 1, N
    D(I) = D(I) + B(I,J)
  ENDDO
ENDDO
```

NM/b misses for each of arrays B and D

==> total of 2NM/b misses

b = block (line) size in words (elements)

Assume that N is large enough for elements of D to overflow cache

Blocking loop I

- After strip-mine-and-interchange

```
DO II = 1, N, S
  DO J = 1, M
    DO I = II, MIN(II+S-1, N)
      D(I) = D(I) + B(I,J)
    ENDDO
  ENDDO
ENDDO
```

$NM/b + N/b = (1 + 1/M)\, NM / b$ misses

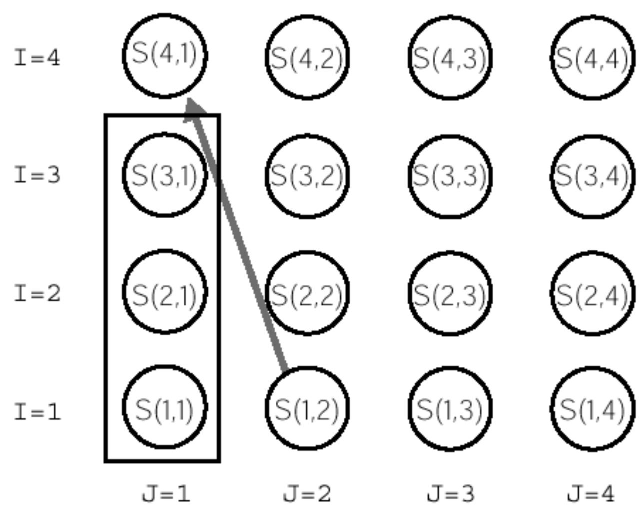
Assume that S is >= b and is also small enough to allow S elements of D to be held in cache for all iterations of the J loop

Blocking Loop J

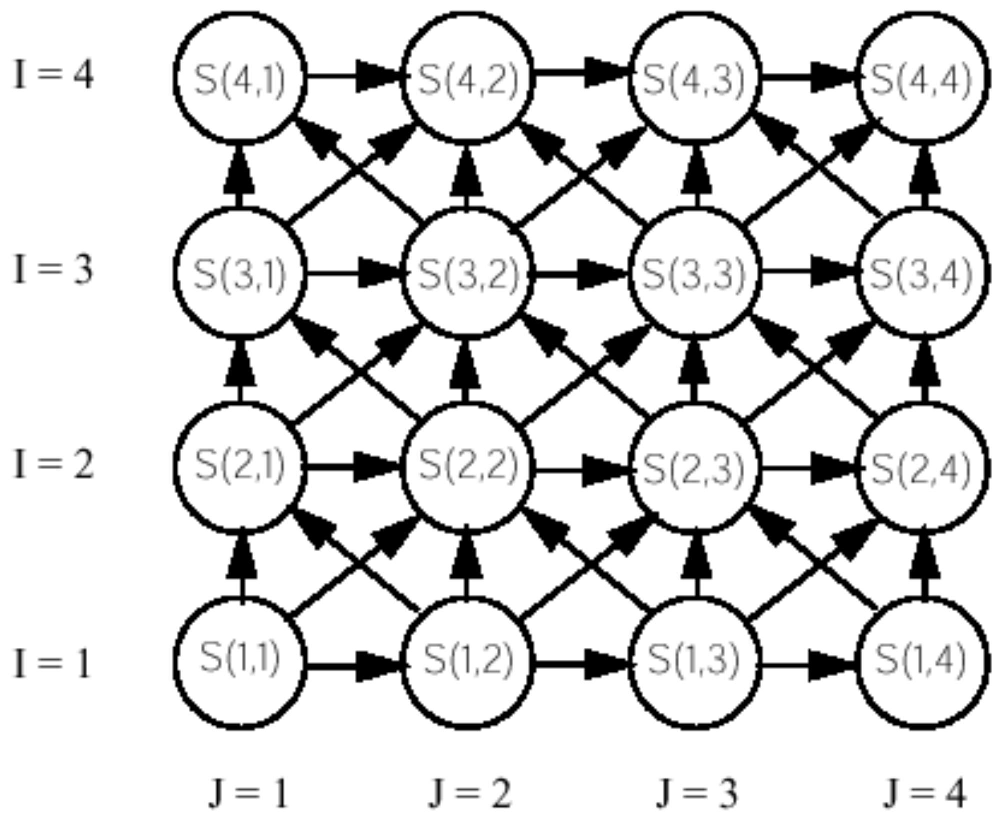
- ```
  DO J = 1, M, T
     DO I = 1, N
        DO jj = J, MIN(J+T-1, M)
           D(I) = D(I) + B(I, jj)
        ENDDO
     ENDDO
  ENDDO
  ```

NM/b misses for array B (if T is small enough)

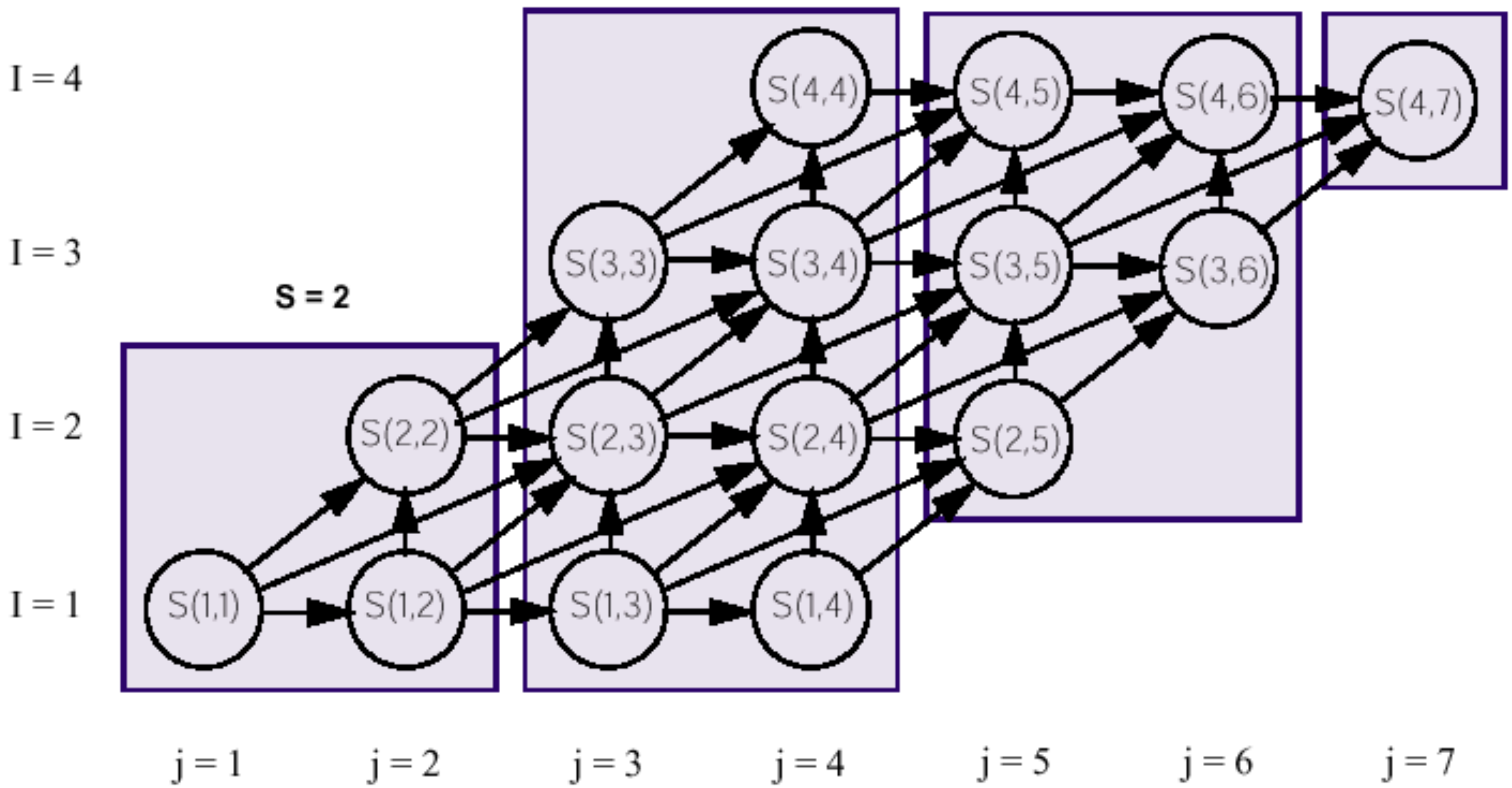
(N/b)*(M/T) misses for array D

==> Total of (1 + 1/T) NM/b misses
  ```

Legality of Blocking

- Every direction vector for a dependence carried by any of the loops $L_0 \ldots L_{k+1}$ has either an "=" or a "<" in the kth position
- Conservative testing



Profitability of Blocking

- Profitable if there is reuse between iterations of a loop that is not the innermost loop
- Reuse occurs when:
  - There's a small-threshold dependence of any type, including input, carried by the loop (temporal reuse), or
  - The loop index appears, with small stride, in the contiguous dimension of a multidimensional array and in no other dimension (spatial reuse)

Blocking with Skewing

- For cases where interchange is not possible
- ```
  DO I = 1, M
     DO J = 1, N
        A(J+1) = (A(J) + A(J+1))/2
     ENDDO
  ENDDO
  ```
  ```

Blocking with Skewing



Blocking with Skewing



Prefetch Analysis

- Identify where misses may happen
- Make use of dependence analysis strategy
  - Build on generator-based partitioning idea from scalar replacement
- First, ensure that every edge that is unlikely to correspond to reuse is eliminated from the graph
- Assume that the loop nest has been strip-mined and interchanged to increase locality
- Traverses the loop and mark 'ineffective' for loops without reuse

Prefetch Analysis

- **Identify where prefetching is required**
- **Two cases:**
  - If the group generator is not contained in a dependence cycle, a miss is expected on each iteration unless references to the generator on subsequent iterations display temporal locality
  - If the group generator is contained in a dependence cycle, then a miss is expected only on the first few iterations of the carrying loop, depending on the distance of the carrying dependence. In this case, a prefetch to the reference can be placed before the loop carrying the dependence

Insertion for Acyclic Partitions

- ```
  DO I = 1, M
     A(I, J) = A(I, J) + A(I-1, J)
  ENDDO
  ```

Assuming cache line of length four, then $i_o = 5$

and $l = 4$
  ```

Insertion for Acyclic Partitions

```
prefetch(A(0,J))
DO I = 1, 3
   A(I, J) = A(I, J) + A(I-1, J)
ENDDO
DO I = 4, M, 4
   IU = MIN(M, I+3)
   prefetch(A(IU, J))
   DO ii = I, IU
      A(ii, J) = A(ii, J) + A(ii-1, J)
   ENDDO
ENDDO
```

Insertion for Cyclic Name Partitions

- **Insert prefetch instructions prior to the loop carrying the cycle**
- **In the case where loop carrying the dependence is an outer loop, the prefetch can be vectorized**
  - Place prefetch loop nest outside the loop carrying the backward dependence of a cyclic name partition
  - Rearrange the loop nest so that the loop iterating sequentially over cache lines is innermost
  - Split the innermost loop into two –
    - Preloop to the first iteration of the innermost loop contaning a generator reference beginning on a new cache line and
    - Main loop that begins with the iteration containing the new cache reference.
  - Replace the preloop by a prefetch of the first generator reference. Set the stride of the main loop to the interval between new cache references.

Insertion for Cyclic Name Partitions

- ```
  DO J = 1, M
      DO I = 2, 33
          A(I, J) = A(I, J) * B(I)
      ENDDO
  ENDDO
  ```
  ```

Summary

- Two different kind of reuse
  - Temporal reuse
  - Spatial reuse
- Strategies to increase the two reuse
  - Loop Interchange
  - Cache Blocking
- Software prefetching

Interprocedural Analysis and Optimization

Chapter 11

Interprocedural Problem Classification

- May and Must problems
  - MOD, REF and USE are 'May' problems
  - KILL is a 'Must' problem
- Flow sensitive and flow insensitive problems
  - Flow sensitive: control flow info included in analysis
  - Flow insensitive: control flow info is (conservatively) ignored
- May and Must classification can apply to call graph edges as well

Flow Insensitive Side-effect Analysis

- Assumptions
  - No procedure nesting i.e., no inner functions
  - All parameters passed by reference
  - Size of the parameter list bounded by a constant,
- We will formulate and solve MOD(s) problem

Solving MOD

$$MOD(s) = DMOD(s) \cup \bigcup_{x \in DMOD(s)} ALIAS(p,x)$$

- **DMOD(s):** set of variables which are directly modified as side-effect of call at s (ignoring aliases)

$$DMOD(s) = \{v \mid s \Rightarrow p, v \xrightarrow{s} w, w \in GMOD(p)\}$$

- **GMOD(p):** set of global variables and formal parameters w of p that are modified, either directly or indirectly as a result of invocation of p
  - Global variables are modeled as special "parameters" in this formulation

Example: DMOD and GMOD

```
S0:     CALL P(A,B,C)

            …

            SUBROUTINE P(X,Y,Z)

                INTEGER X,Y,Z

                X = X*Z

                Y = Y*Z

            END
```

- GMOD(P)={X,Y}
- DMOD(S0)={A,B}

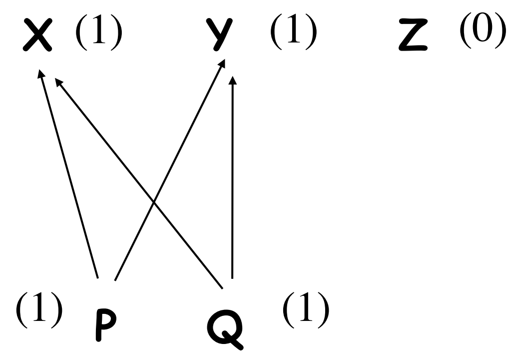
Solving for RMOD

- **RMOD(p):** set of formal parameters in p that may be modified in p, either directly or by assignment to a reference formal parameter of q as a side effect of a call of q in p
- Binding Graph $G_B=(N_B,E_B)$
  - One vertex for each formal parameter of each procedure
  - Directed edge from formal parameter, f1 of p to formal parameter, f2 of q if there exists a call site s=(p,q) in p such that f1 is bound to f2
- Use a marking algorithm to compute RMOD(p) (Figure 11.2)
  - Mark each vertex as false initially
  - Mark formals of P in IMOD(p) as true
  - Perform a closure operation (propagate bits)
    - Mark f1 as true if G_B has an edge from f1 to f2 and f2 is marked true
    - Use worklist algorithm (or reverse DFS, if you prefer)

  $O(N_B + E_B)$ running time

Solving for RMOD

```
SUBROUTINE A(X,Y,Z)
   INTEGER X,Y,Z
   X = Y + Z
   Y = Z + 1
END

SUBROUTINE B(P,Q)
   INTEGER P,Q,I
   I = 2
   CALL A(P,Q,I)
   CALL A(Q,P,I)
END
```



- **RMOD(A)={X,Y}**
- **RMOD(B)={P,Q}**
- **Complexity:** $O(N_B + E_B)$

  $N_B \leq \mu N$ $E_B \leq \mu E$

  $O(N + E)$

Solving for IMOD$^+$

- After gathering RMOD(p) for all procedures, update RMOD(p) to IMOD$^+$(p) using this equation

$$IMOD^+(p) = IMOD(p) \cup \bigcup_{s=(p,q)} \{z \mid z \xrightarrow{s} w, w \in RMOD(q)\}$$

- This can be done in O(NV+E) time

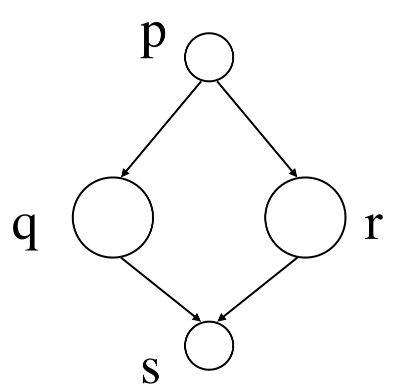
Solving for GMOD

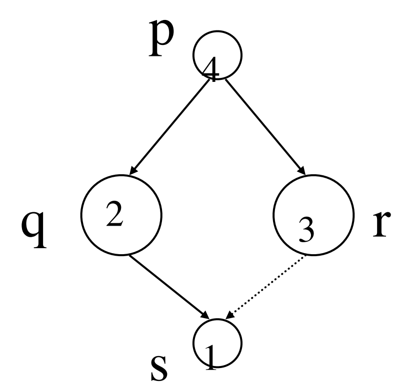
- **After gathering IMOD+(p) for all procedures, calculate GMOD (p) according to the following equation**

$$GMOD(p) = IMOD^{+}(p) \cup \bigcup_{s=(p,q)} GMOD(q) \cap \neg LOCAL$$

- **This can be solved using a DFS algorithm based on Tarjan's SCR algorithm on the Call Graph**

Solving for GMOD

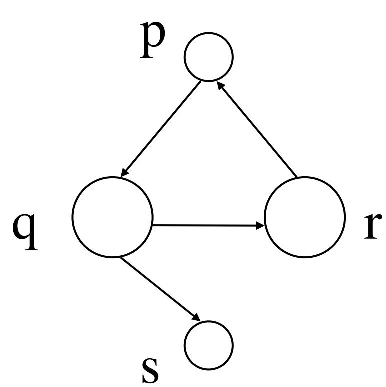


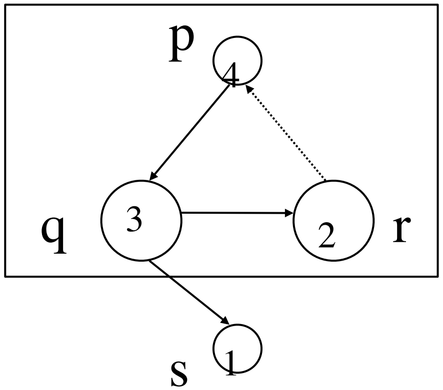


Initialize GMOD(p) to $IMOD^+(p)$ on discovery

Update GMOD(p) computation while backing up

Solving for GMOD





Initialize GMOD(p) to $IMOD^{+}$(p) on discovery

Update GMOD(p) computation while backing up

For each node u in a SCR update GMOD(u) in a cycle

O((N+E)V) Algorithm

Compiling Array Assignments

Allen and Kennedy, Chapter 13

Safe Scalarization

- Naive algorithm for safe scalarization: Use temporary storage to make sure scalarization dependences are not created
- Consider:

```
A(2:201) = 2.0 * A(1:200)
```

- can be split up into:

```
T(1:200) = 2.0 * A(1:200)
A(2:201) = T(1:200)
```

- Then scalarize using SimpleScalarize

```
DO I = 1, 200
        T(I) = 2.0 * A(I)
ENDDO
DO I = 2, 201
        A(I) = T(I-1)
ENDDO
```

Loop Reversal

```
A(2:256) = A(1:255) + 1.0
```

- A scalarization approach using loop reversal that avoids the need for a temporary:

```
DO I = 256, 2, -1
      A(I) = A(I-1) + 1.0
ENDDO
```

Input Prefetching

```
A(2:257) = ( A(1:256) + A(3:258) ) / 2.0
```

- Causes a scalarization fault when naively scalarized to:

```
DO I = 2, 257
   A(I) = ( A(I-1) + A(I+1) ) / 2.0
ENDDO
```

- Problem: Stores into first element of the LHS in the previous iteration
- Input prefetching: Use scalar temporaries to store elements of input and output arrays

Input Prefetching

```
T1 = A(1)
DO I = 2, 256
   T2 = ( T1 + A(I+1) ) / 2.0
   T1 = A(I)
   A(I) = T2
ENDDO
T2 = ( T1 + A(257) ) / 2.0
A(I) = T2
```

- Note: We are using scalar replacement, but the motivation for doing so is different than in Chapter 8

General Multidimensional Scalarization

- **Goal: To vectorize a single statement which has m vector dimensions**
  - Given an ideal order of scalarization $(l_1, l_2, ..., l_m)$
  - $(d_1, d_2, ..., d_n)$ be direction vectors for all plausible and implausible true dependences of the statement upon itself
  - The scalarization matrix is a $n \times m$ matrix of these direction vectors
- **For instance:**

```
A(1:N, 1:N, 1:N) = A(0:N-1, 1:N, 2:N+1) +
                          A(1:N, 2:N+1, 0:N-1)
```

$$\begin{pmatrix} > & = & < \\ < & > & = \end{pmatrix}$$

General Multidimensional Scalarization

- Once a loop has been selected for scalarization, the dependences carried by that loop, any dependence whose direction vector does not contain a = in the position corresponding to the selected loop may be eliminated from further consideration.
- In our example, if we move the second column to the outside, we get:

$$\begin{pmatrix} > & = & < \\ < & > & = \end{pmatrix} \Rightarrow \begin{pmatrix} = & > & < \\ > & < & = \end{pmatrix}$$

- Scalarization in this way will reduce the matrix to:

$$\begin{pmatrix} > & < \end{pmatrix}$$

Final exam

- **Take-home exam (3 hours)**
  - Open book: textbook only, no other resources
  - Scope of exam is limited to chapters 7, 8, 9, 11, 13
  - Exam will be made available on Monday, Dec 5th, and will be due by 5pm on Friday, Dec 16th