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 22 3 December, 2015

End-semester Summary

Exam 2 scope: Chapters 7, 8, 9, 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:**

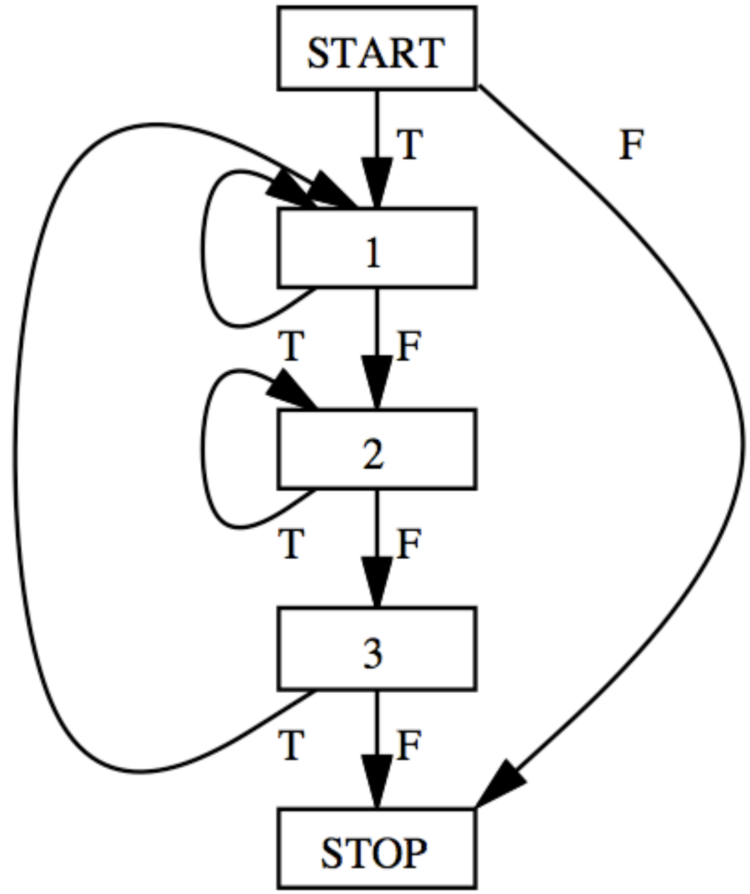
```
   DO 100 I = 1,N
                m1 = A(I).GT.10
20     IF(.NOT.m1) A(I) = A(I) + 10
       IF(.NOT.m1) m2 = B(I).GT.10
40     IF(.NOT.m1.AND..NOT.m2)
              B(I) = B(I) + 10
60     IF(m1.OR..NOT.m2)
              A(I) = B(I) + A(I)
80     B(I) = A(I) - 5
   ENDDO
```

- **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
```

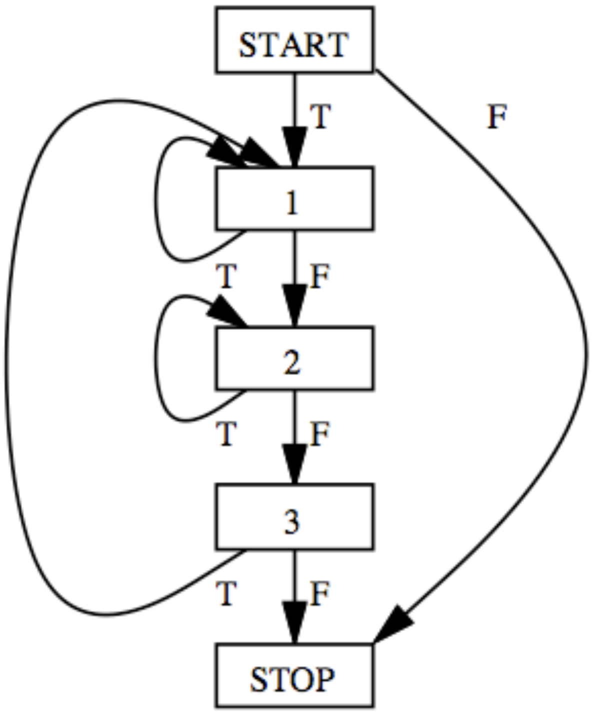
Control Flow Graph: Example

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

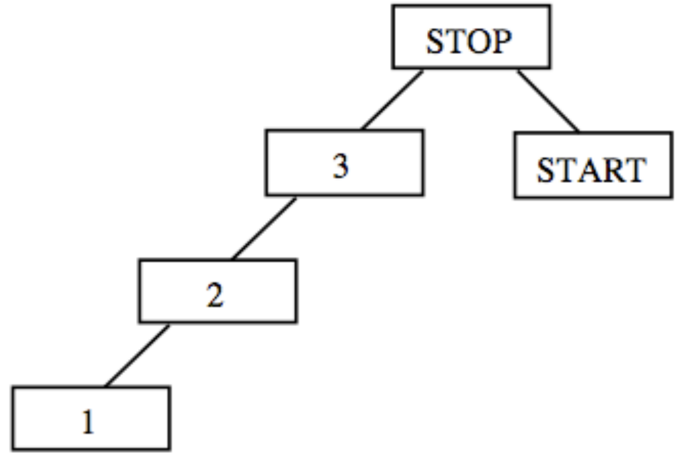


CONTROL FLOW GRAPH

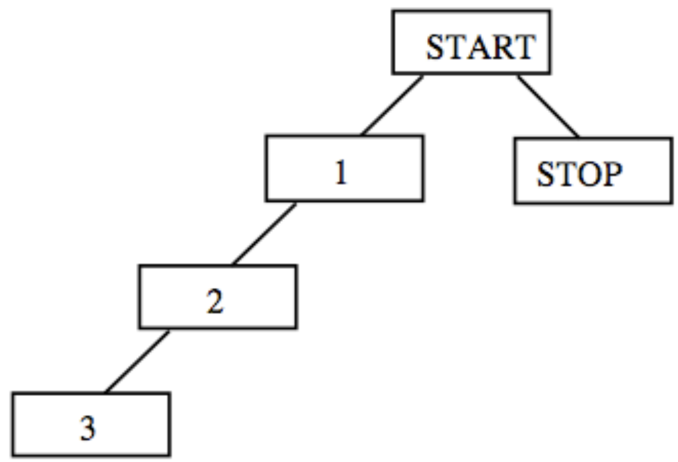
Examples of Dominator and Postdominator Trees



CONTROL FLOW GRAPH



POST–DOMINATOR TREE



DOMINATOR TREE

Control Dependence: Definition

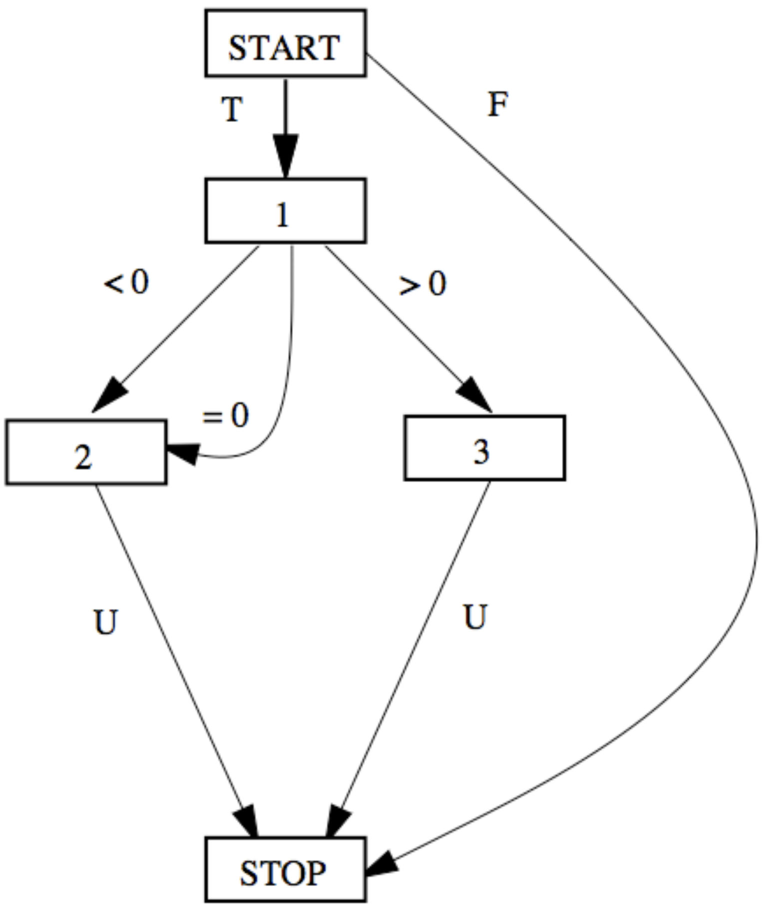
Node Y is *control dependent* on node X with label L in *CFG* if and only if

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

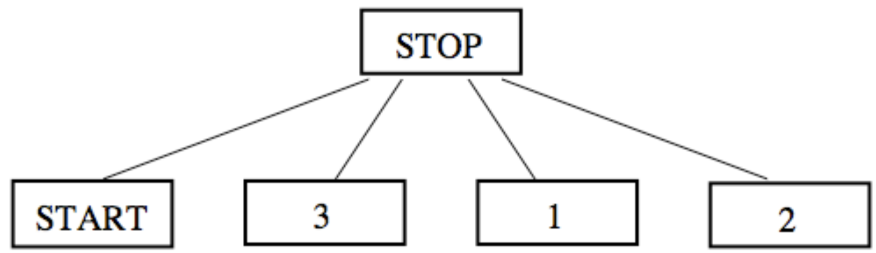
2. Y does not post-dominate X.

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

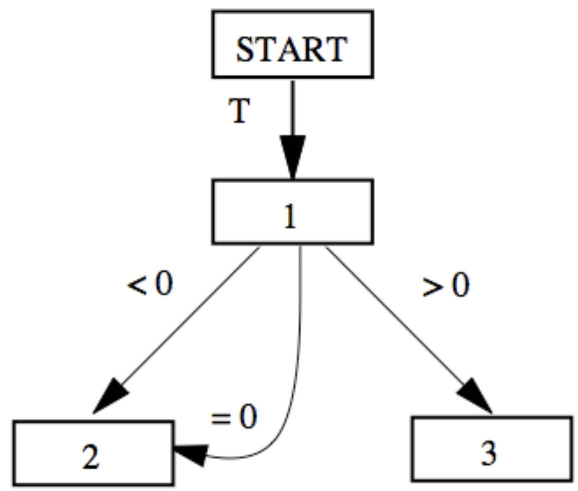
Example: Acyclic CFG and its Control Dependence Graph (CDG)



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)

Compiler Improvement of Register Usage

Chapter 8

Scalar Replacement (Recap)

- 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 by unrolling

```
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

Preloop

```
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
```

Main Loop

```
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
```

Pruning the dependence graph

- Prune all anti dependence edges
- Prune flow and input dependence edges that do not represent a potential reuse
- Prune redundant input dependence edges
- Prune output dependence edges after rest of the pruning is done

Pruning the dependence graph

- Example: Eliminate killed dependences
  - When killed dependence is a flow dependence

```
S1: A(I+1) = ...
S2: A(I)   = ...
S3: ...   = A(I)
```

  - Store in S2 is a killing store. Flow dependence from S1 to S3 is pruned
  - When killed dependence is an input dependence

```
S1: ... = A(I+1)
S2: A(I)   = ...
S3: ...   = A(I-1)
```

  - Store in S2 is a killing store. Input dependence from S1 to S3 is pruned

Unroll-and-Jam

```
DO I = 1, N*2
   DO J = 1, M
        A(I) = A(I) + B(J)
   ENDDO
ENDDO
```

- Can we achieve reuse of references to B ?
- Use transformation called Unroll-and-Jam

```
DO I = 1, N*2, 2
   DO J = 1, M
        A(I) = A(I) + B(J)
        A(I+1) = A(I+1) + B(J)
   ENDDO
ENDDO
```

- Unroll outer loop twice and then fuse the copies of the inner loop
- Brought two uses of B(J) together

Unroll-and-Jam

```
DO I = 1, N*2, 2
   DO J = 1, M
        A(I) = A(I) + B(J)
        A(I+1) = A(I+1) + B(J)
   ENDDO
ENDDO
```

- Apply scalar replacement on this code

```
DO I = 1, N*2, 2
   s0 = A(I)
   s1 = A(I+1)
   DO J = 1, M
        t = B(J)
        s0 = s0 + t
        s1 = s1 + t
   ENDDO
   A(I) = s0
   A(I+1) = s1
ENDDO
```

- Half the number of loads as the original program

Legality of Unroll-and-Jam

- Is unroll-and-jam always legal?

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

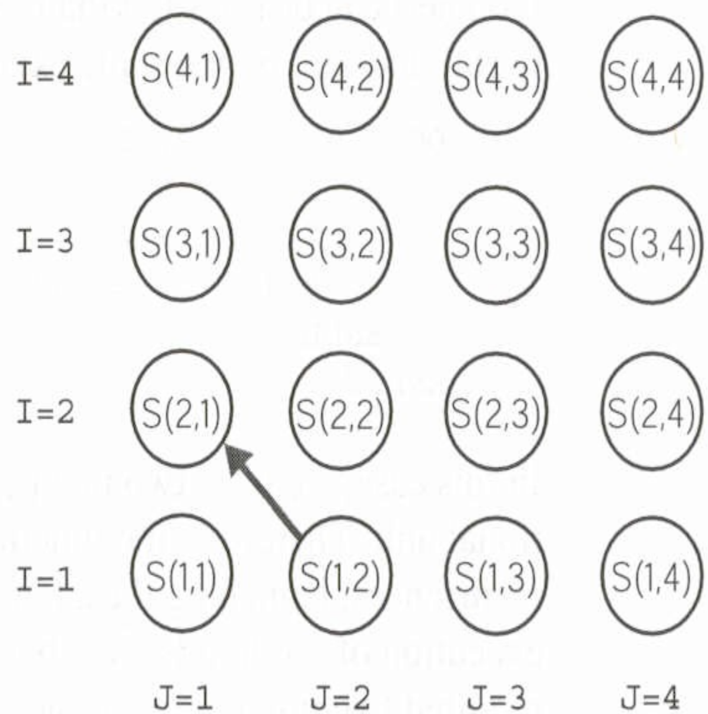
- Apply unroll-and-jam

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

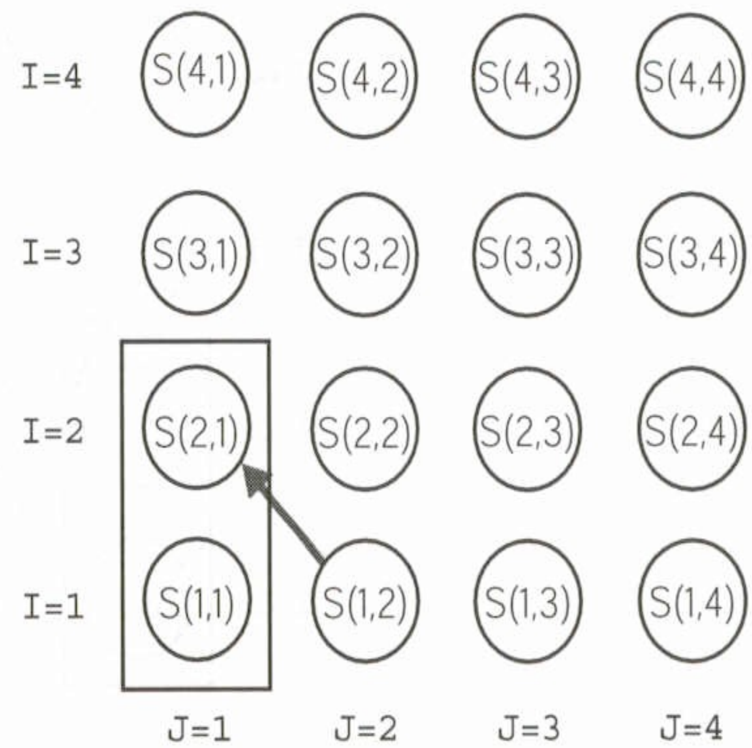
- This is wrong!!!

Legality of Unroll-and-Jam

Legality of unroll-and-jam



Legality of unroll-and-jam.



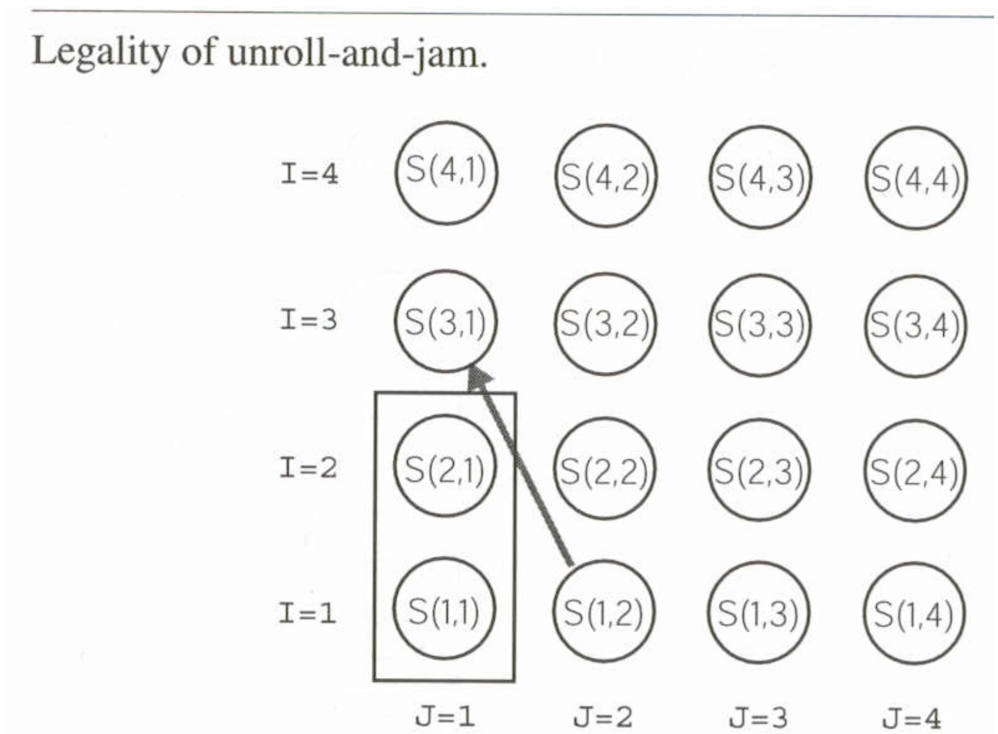
Legality of Unroll-and-Jam

- Direction vector in this example was (<,>)
  - This makes loop interchange illegal
  - Unroll-and-Jam is loop interchange followed by unrolling inner loop followed by another loop interchange
- But does loop interchange illegal imply unroll-and-jam illegal ? NO

Legality of Unroll-and-Jam

- Consider this example

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



- Direction vector is (<,>); still unroll-and-jam possible because of distances involved

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.

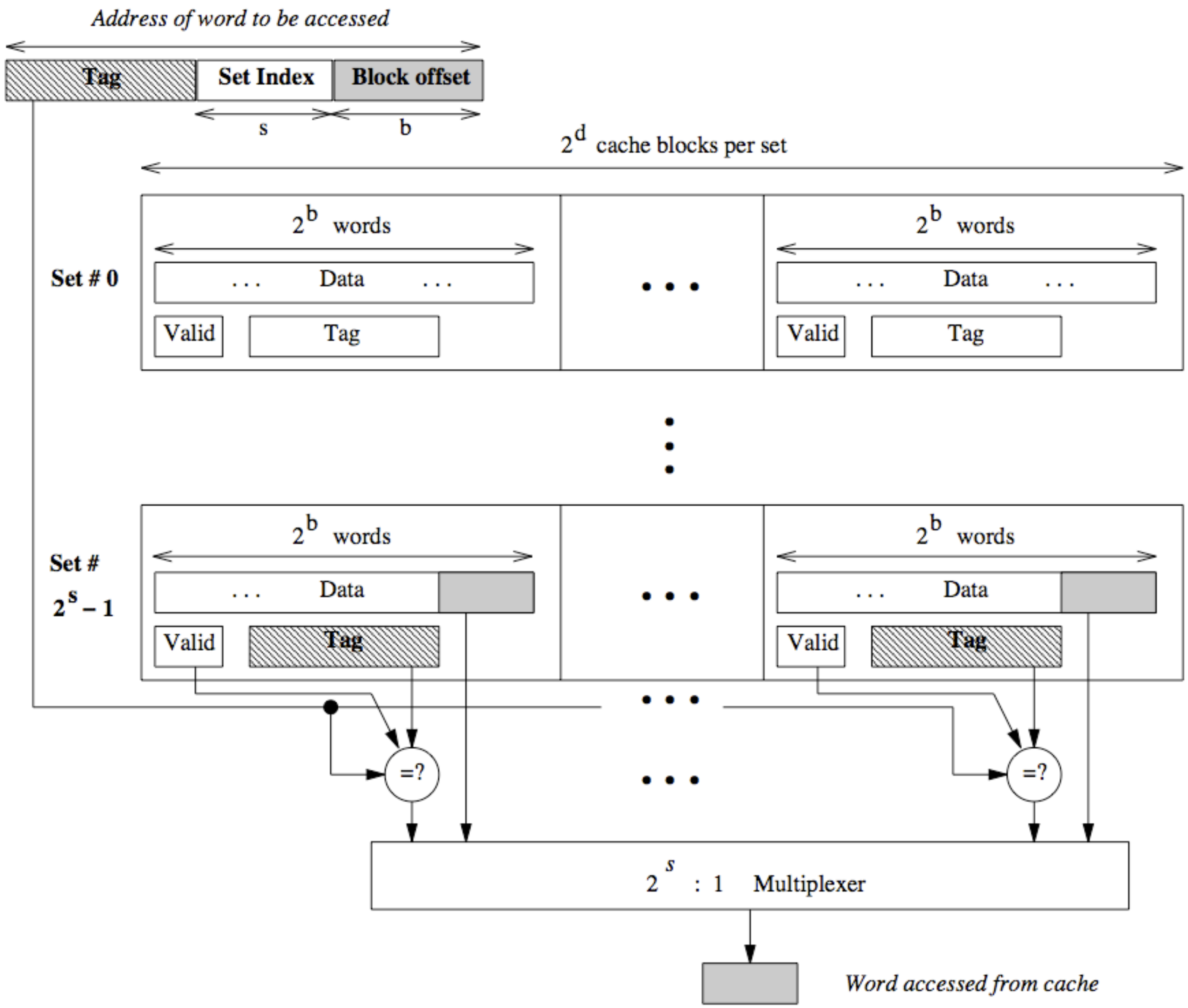
Conclusion

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

Managing Cache

Allen and Kennedy, Chapter 9

Review: How do set-associative caches work?



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
  ```

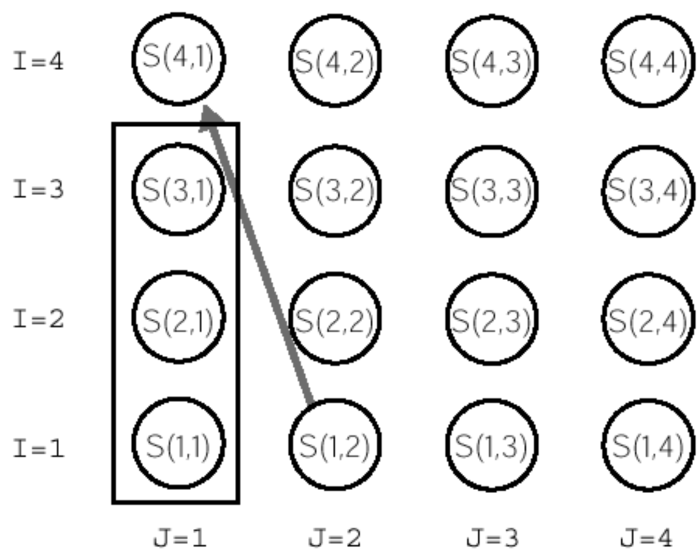
NM/b misses for array B (if T is small enough)

(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



Summary

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

Compiling Array Assignments

Allen and Kennedy, Chapter 13

Fortran 90

- Range of a vector operation in Fortran 90 denoted by a triplet: <lower bound: upper bound: increment>

```
A(1:100:2) = B(2:51:1) + 3.0
```

- Semantics of Fortran 90 require that for vector statements, all inputs to the statement are fetched before any results are stored

- As with DO loops, the default value of the increment is 1, i.e., B(2:51) is equivalent to B(2:51:1)

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
```

Loop Reversal

- When can we use loop reversal?
  - Loop reversal maps true dependences into antidependences
  - But may also map antidependences into true dependences

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

- After scalarization:

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

- Loop Reversal gets us:

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

- Thus, cannot use loop reversal in presence of antidependences
- Goal: ensure that scalarized loop has no loop-carried true dependences

Multidimensional Scalarization

- Vector statements in Fortran 90 in more than 1 dimension:

```
A(1:100, 1:100) = B(1:100, 1, 1:100)
```

- corresponds to:

```
DO J = 1, 100
  A(1:100, J) = B(1:100, 1, J)
ENDDO
```

- Scalarization in multiple dimensions:

```
A(1:100, 1:100) = 2.0 * A(1:100, 1:100)
```

- Obvious Strategy: convert each vector iterator into a loop:

```
DO J = 1, 100, 1
  DO I = 1, 100
    A(I,J) = 2.0 * A(I,J)
  ENDDO
ENDDO
```

Multidimensional Scalarization

- What should the order of the loops be after scalarization?
  - Familiar question: We dealt with this issue in Loop Selection/ Interchange in Chapter 5
- Profitability of a particular configuration depends on target architecture
  - For simplicity, we shall assume shorter strides through memory are better
  - Thus, optimal choice for innermost loop is the leftmost vector iterator

Loop Interchange

- Sometimes, there is a tradeoff between scalarization and optimal memory hierarchy usage

```
A(2:100, 3:101) = A(3:101, 1:201:2)
```

- If we scalarize this using the prescribed order:

```
DO I = 3, 101
  DO 100 J = 2, 100
    A(J,I) = A(J+1,2*I-5)
  ENDDO
ENDDO
```

- Direction vectors for true dependences:
  - (<, >) (for I = 3, 4) and (>, >) (for I = 6, 7)
- Cannot use loop reversal, input prefetching
- Can use temporaries

Loop Interchange

- However, we can use loop interchange to get:

```
DO J = 2, 100
  DO I = 3, 101
    A(J,I) = A(J+1,2*I-5)
  ENDDO
ENDDO
```

- Not optimal memory hierarchy usage, but reduction of temporary storage

- Loop interchange is useful to reduce size of temporaries
- It can also eliminate scalarization dependences

Scalarization Example

```
DO J = 2, N-1
   A(2:N-1,J) = A(1:N-2,J) + A(3:N,J) +
                A(2:N-1,J-1) + A(2:N-1,J+1)/4.
ENDDO
```

- Loop carried true dependence, antidependence
- Naive compiler could generate:

```
DO J = 2, N-1
  DO i = 2, N-1
    T(i-1) = (A(i-1,J) + A(i+1,J) + A(i,J-1) + A(i,J+1) )/4
  ENDDO
  DO i = 2, N-1
    A(i,J) = T(i-1)
  ENDDO
ENDDO
```

- $2 \cdot (N-2)^2$ accesses to memory due to array T

Scalarization Example

- However, can use input prefetching to get:

```
DO J = 2, N-1
  tA0 = A(1, J)
  DO i = 2, N-2
    tA1 = (tA0+A(i+1,J)+A(i,J-1)+A(i,J+1))/4
    tA0 = A(i-1, J)
    A(i,J) = tA1
  ENDDO
  tA1 = (tA0+A(N,J)+A(N-1,J-1)+A(N-1,J+1))/4
  A(N-1,J) = tA1
ENDDO
```

- If temporaries are allocated to registers, no more memory accesses than original Fortran 90 program

Exam 2

- Take-home exam (3 hours)
  - Open book: open book, open notes, no other resources
  - Scope of exam is limited to chapters 7, 8, 9, 13
  - Exam will be made available today, and will be due by 4pm on Friday, Dec 10th