COMP 515: Advanced Compilation for Vector and Parallel Processors

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

Announcements

- **Next week's classes**
  - Make-up class on Feb 20th, 2pm – 3:20pm, Mudd Lab 254
- **Take-home midterm**
  - Will be given in class on Tuesday, Feb 24th
  - No class on Thursday, Feb 26th
  - Due by 5pm on Tuesday, March 3rd

Acknowledgments

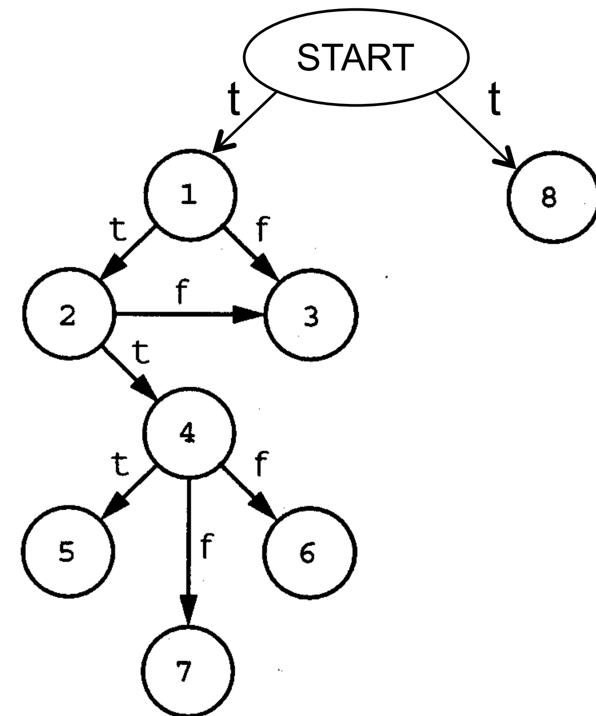
- Slides from previous offerings of COMP 515 by Prof. Ken Kennedy
  - http://www.cs.rice.edu/~ken/comp515/

Loop Distribution

- More complex example:

```
 DO I = 1, N
1      IF (A(I).NE.0) THEN
2            IF (B(I)/A(I).GT.1) GOTO 4
       ENDIF
3      A(I) = B(I)
       GOTO 8
4      IF (A(I).GT.T) THEN
5            T = (B(I) - A(I)) + T
       ELSE
6            T = (T + B(I)) - A(I)
7            B(I) = A(I)
       ENDIF
8      C(I) = B(I) + C(I)
       ENDDO
```

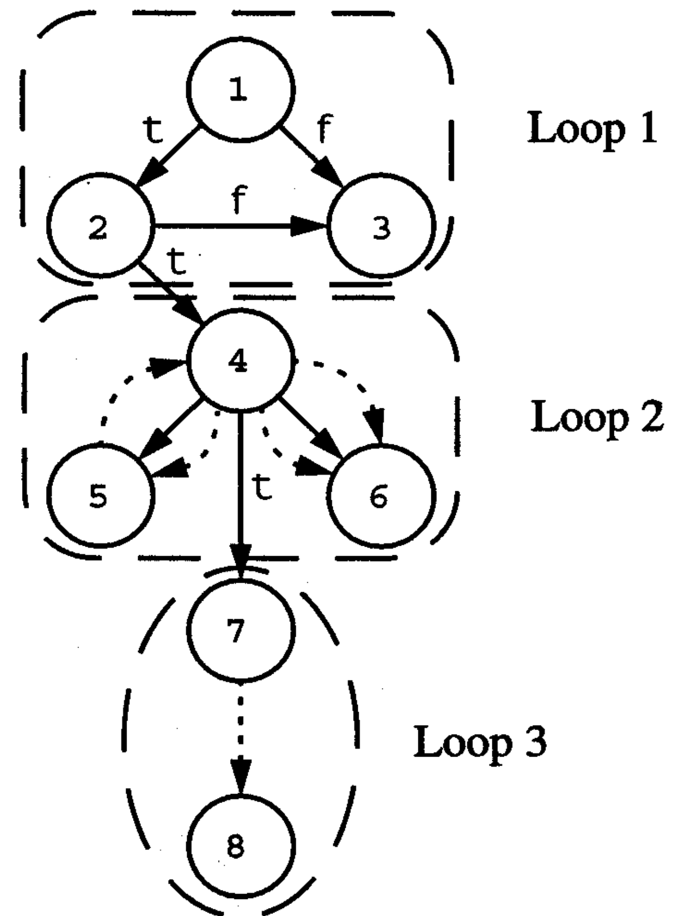
Control Dependence Graph for loop body



Loop Distribution

- Fusion into "like" regions
  - Loop 1 is parallel
  - Loop 2 is sequential
  - Loop 3 is parallel

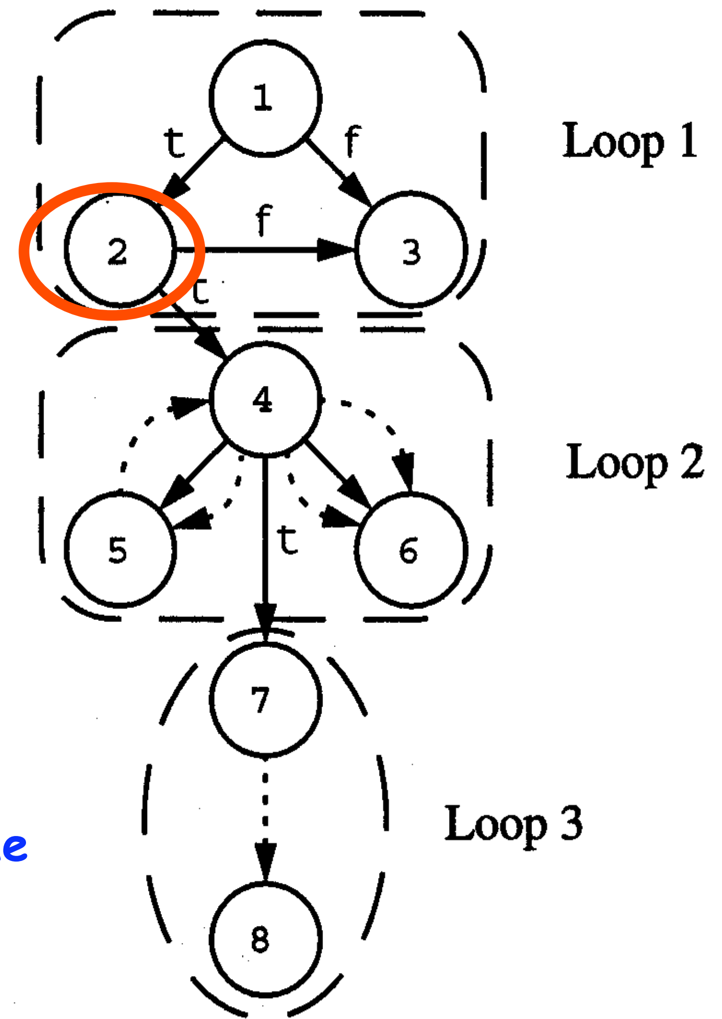
```
 DO I = 1, N
1      IF (A(I).NE.0) THEN
2            IF (B(I)/A(I).GT.1) GOTO
   4
     ENDIF
3      A(I) = B(I)
     GOTO 8
4      IF (A(I).GT.T) THEN
5            T = (B(I) - A(I)) + T
     ELSE
6            T = (T + B(I)) - A(I)
7            B(I) = A(I)
     ENDIF
8      C(I) = B(I) + C(I)
     ENDDO
```



Need execution variables E2(I) and E4(I) to hold result of branches at statement 2 and 4

Loop Distribution

- Consider branch at node 2:
- 3 cases may hold
  - Statement 2 is executed and the true branch to statement 4 is taken
  - Statement 2 is executed and the false branch to statement 3 is taken
  - Statement 2 is never executed because the false branch is taken at statement 1
- Corresponds to condition for doit variable to be set:
  - A control dependence exists from S_0 to S.
  - S_0 has its doit flag set
  - Value of the conditional expression is the label on the branch



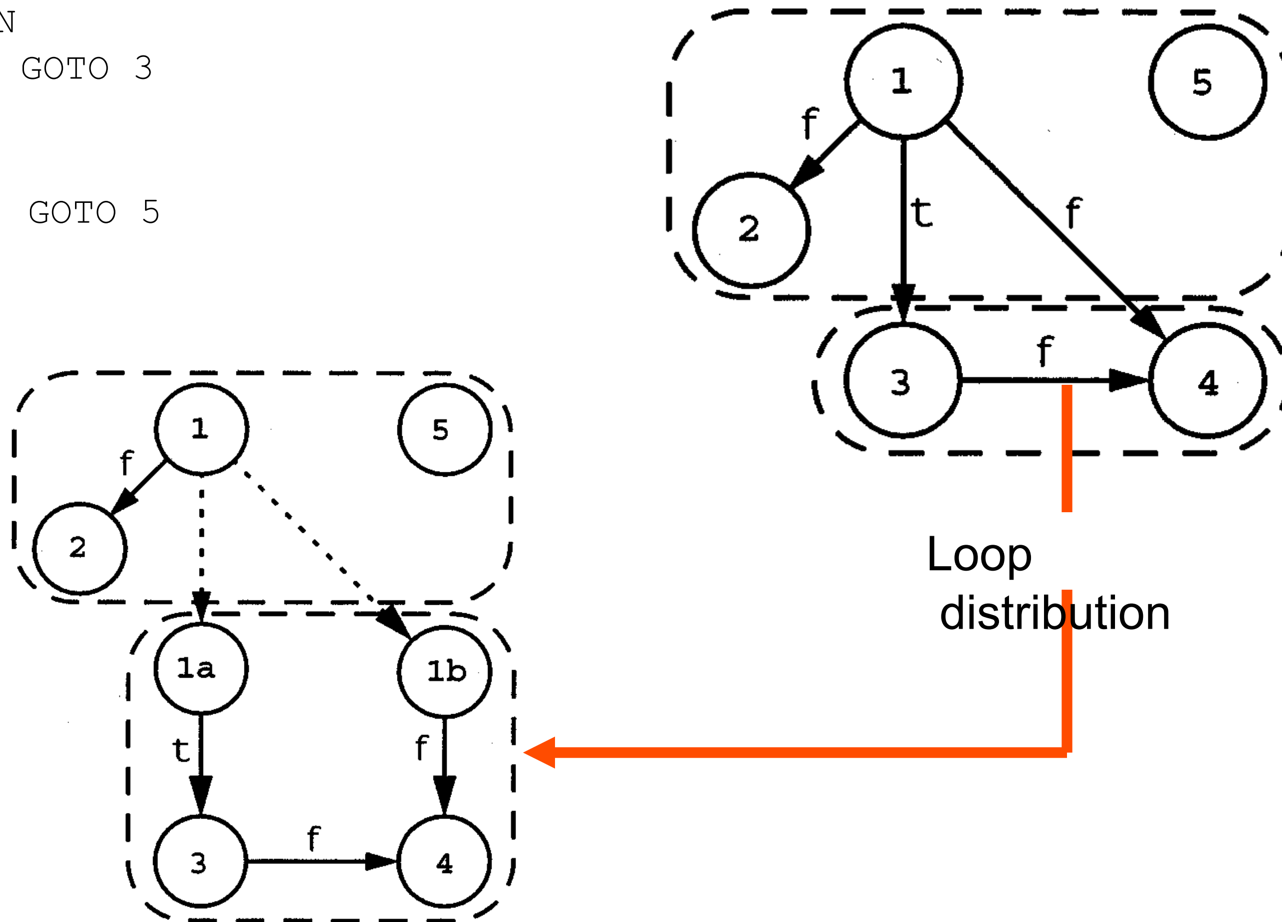
Loop Distribution

- Use three corresponding values: True, False, Undefined
- Procedure DistributeCDG implements these ideas. It inserts execution variables at appropriate places in the code and selectively converts control dependences to data dependences

Code Generation

- **Problem: Mapping the arbitrary control flow represented in the control dependence graph to real machines**

```
      DO I = 1, N
S1        IF (p1) GOTO 3
S2        ...
          GOTO 4
3         IF (p3) GOTO 5
4         S4
5         S5
      ENDDO
```



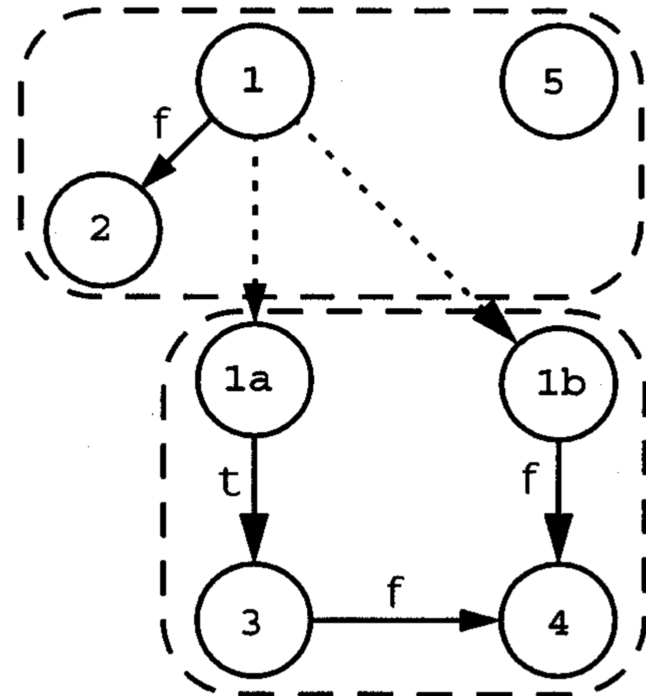
Code Generation

- **Code generated for first partition:**

```
DO I = 1, N
   E1(I) = p1
   IF (E1(I).EQ.FALSE) THEN
S2   ...
   ENDIF
S5   ...
ENDDO
```

- **For second partition:**

```
DO I = 1, N
   IF((E1(I).EQ..TRUE.).AND..NOT.p3).OR.
      (E1(I).EQ..FALSE.)) THEN
S4    ...
   ENDIF
ENDDO
```



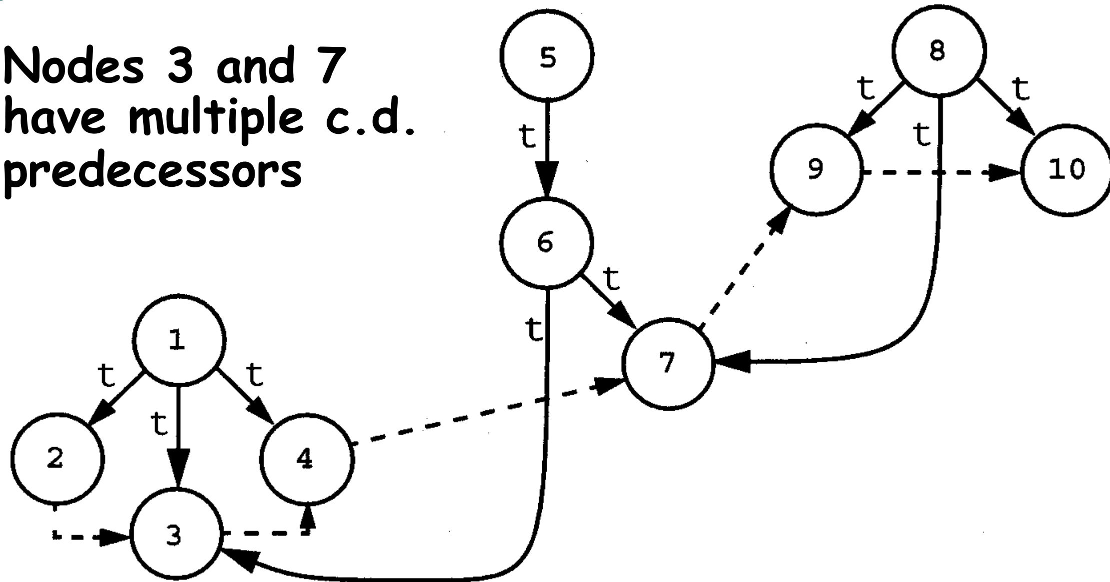
Code Generation

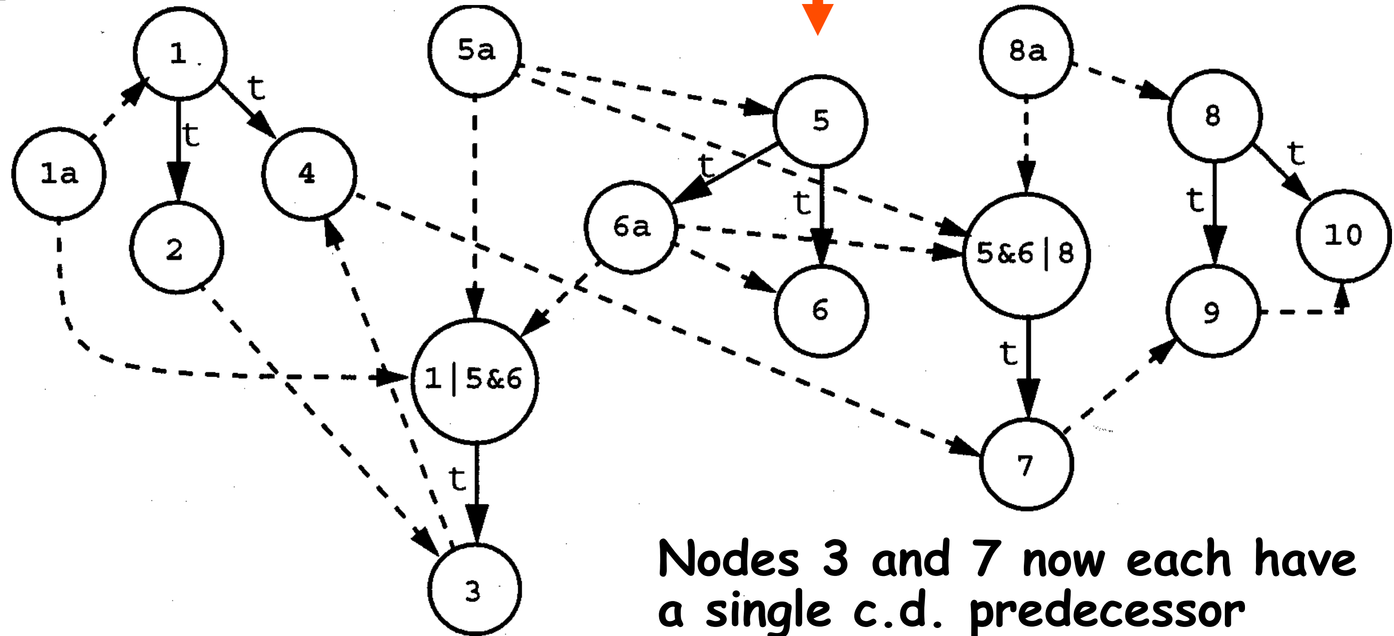
- Observation: generating code for graphs in which every vertex has at most one control dependence predecessor is relatively easy

- Thus, transform graph into canonical form consisting of a set of control dependence trees with the following properties:
  - each statement is control dependent on at most one other statement, i.e., each statement is a member of at most one tree
  - the trees can be ordered so that all data dependences between trees flow from trees earlier in the order to trees that are later in the order i.e., there should be no non-trivial cycle of data dependence edges among control dependence trees

Code Generation

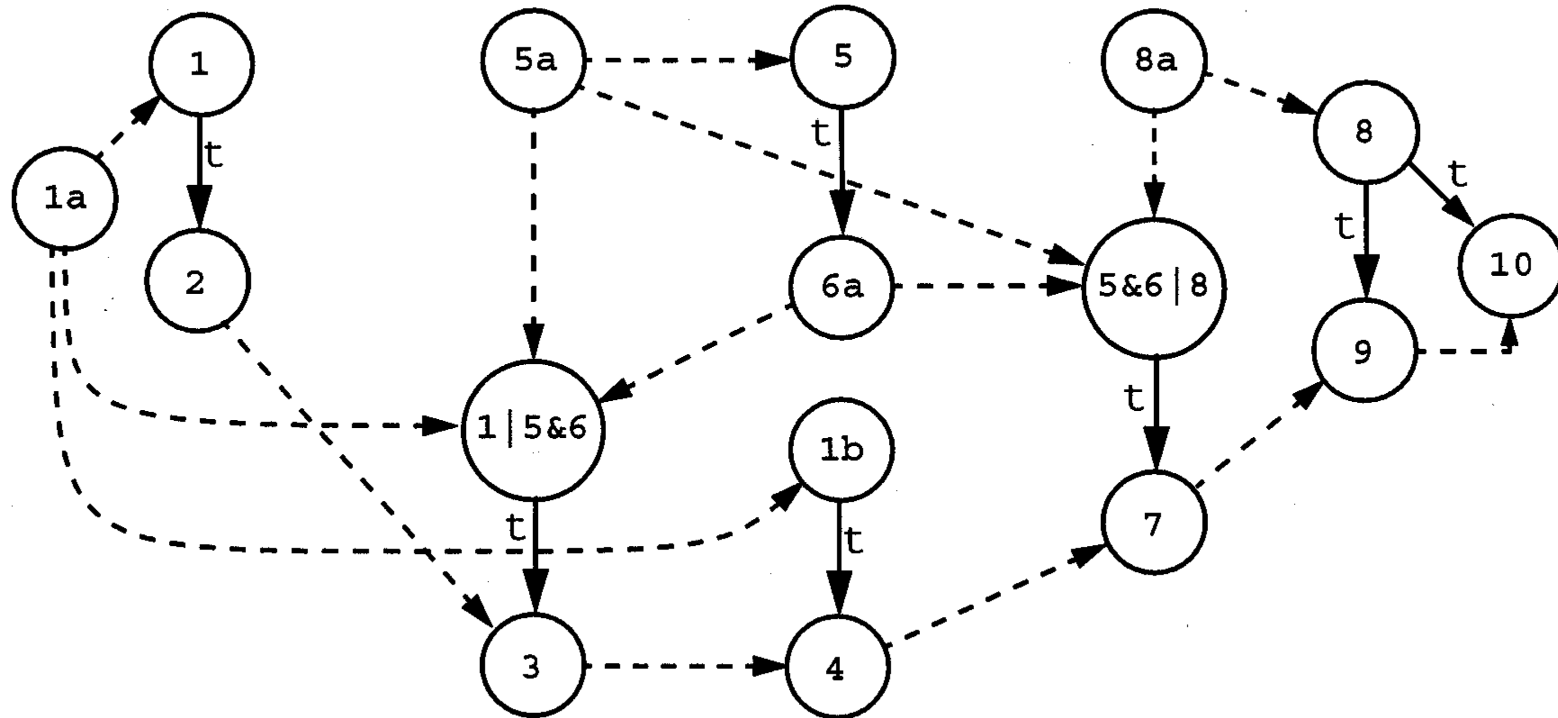
Nodes 3 and 7 have multiple c.d. predecessors





Nodes 3 and 7 now each have a single c.d. predecessor

Code Generation



Code Generation

- **How can the statements be organized into groups of statements that are part of the same conditional statement?**
  - Statements can be grouped together if there is no dependence path between them that passes through a statement that is not a child of the same conditional node with the same label
  - Typed Fusion!
  - Each statement typed by (p, l) where
  - p: its unique control dependence predecessor
  - l: the truth label of the edge from p to the statement

Code Generation

- Simple recursive procedure
- Generate code for each of the subtree in an order consistent with the data dependences
- Roughly linear in size of the original dependence graph

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

Overview

- Improving memory hierarchy performance by compiler transformations
  - Scalar Replacement
  - Unroll-and-Jam
- Saving memory loads & stores
- Make good use of the processor registers

Motivating Example

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

- A(I) can be left in a register throughout the inner loop
- Coloring based register allocation fails to recognize this

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

- All loads and stores to A in the inner loop have been saved
- High chance of T being allocated a register by the coloring algorithm

Scalar Replacement

- Convert array reference to scalar reference to improve performance of the coloring based allocator
- Our approach is to use dependences to achieve these memory hierarchy transformations

Dependence and Memory Hierarchy

- True or Flow - save loads and cache miss
- Anti - save cache miss
- Output - save stores
- Input - save loads

```
A(I) = ... + B(I)
...  = A(I) + k
A(I) = ...
...  = B(I)
```

Dependence and Memory Hierarchy

- **Loop Carried dependences - Consistent dependences most useful for memory management purposes**
- **Consistent dependences - dependences with constant threshold (dependence distance)**

Dependence and Memory Hierarchy

- Problem of overcounting optimization opportunities. For example

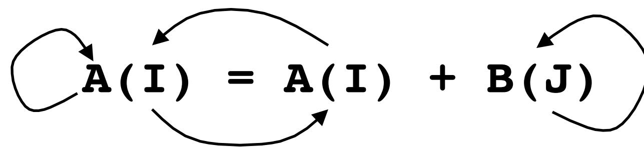
```
S1: A(I) = ...
S2: ...  = A(I)
S3: ...  = A(I)
```

- But we can save only two memory references not three
- Solution - Prune edges from dependence graph which don't correspond to savings in memory accesses

Using Dependences

- In the reduction example

```
DO I = 1, N
   DO J = 1, M
```




```
   ENDDO
ENDDO
```

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

- True dependence - replace the references to A in the inner loop by scalar T
- Output dependence - store can be moved outside the inner loop
- Antidependence - load can be moved before the inner loop

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