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 15 20 October, 2011

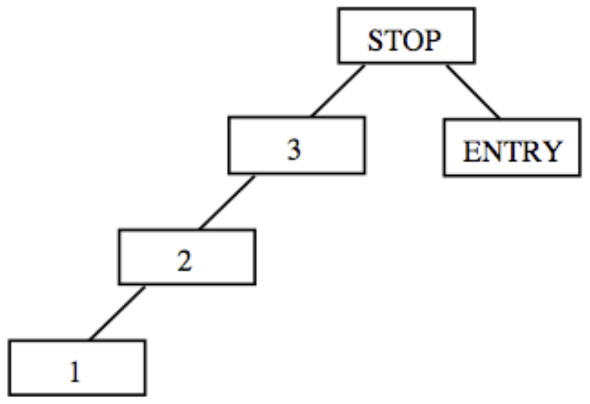
Acknowledgments

- Slides from previous offerings of COMP 515 by Prof. Ken Kennedy
  - http://www.cs.rice.edu/~ken/comp515/
- POPL 1996 tutorial by Krishna Palem & Vivek Sarkar

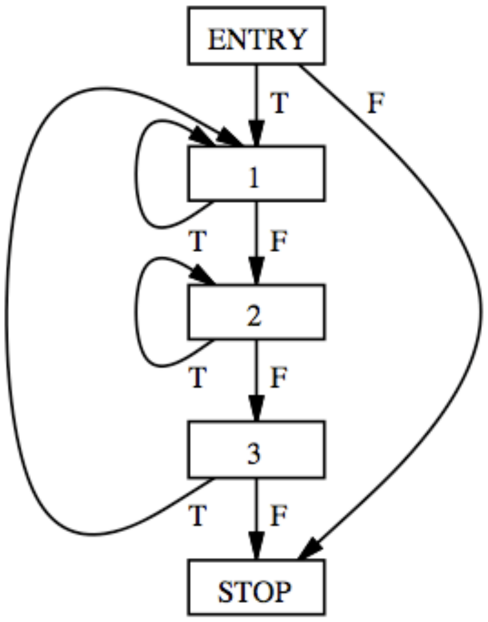
Control Dependences

Chapter 7 (contd)

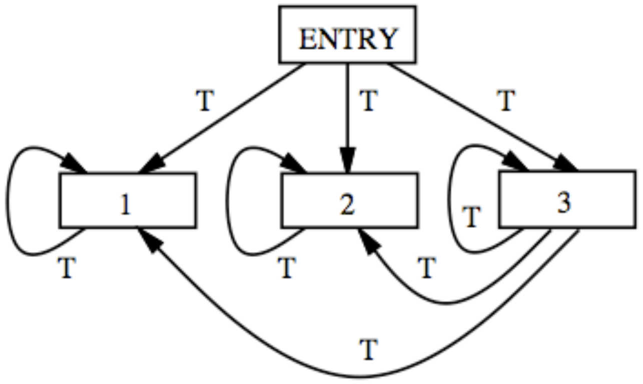
Example: Cyclic CFG and its CDG



POST-DOMINATOR TREE



CONTROL FLOW GRAPH



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.

Control Dependence and Parallelization

- From Chapter 2: Most loop transformations are unaffected by loop-independent dependences
  - A forward-branch need not inhibit coarse-grain parallelization
- Iteration-reordering transformations like loop reversal, loop skewing, strip mining, index-set splitting, loop interchange do not affect loop-independent dependences
- Statement reordering transformations might be problematic: loop fusion, loop distribution
  - Distribution can be performed by including control dependences in recurrence analysis, and performing scalar expansion on branch condition
  - Fusion of loops that do not contain exit branches is also possible

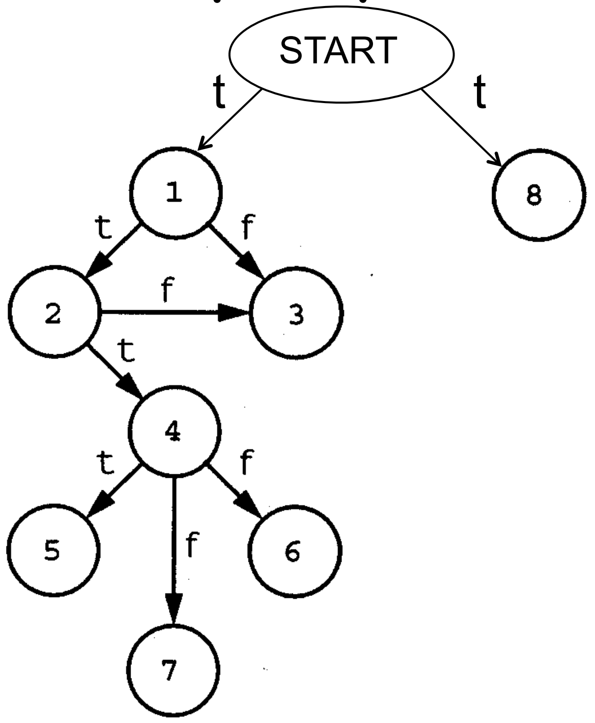
Loop Distribution

- 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

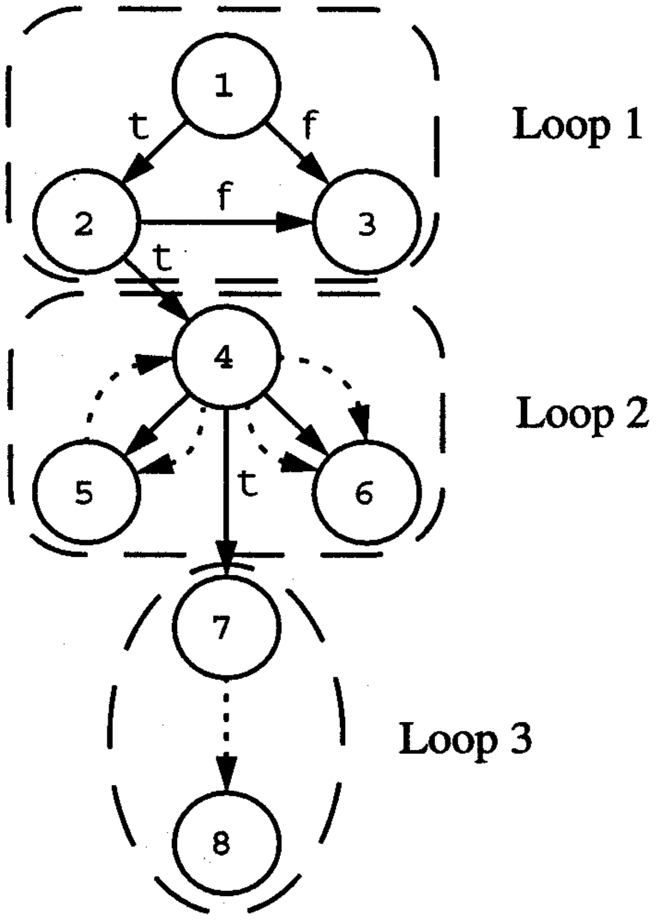
for loop body



Loop Distribution

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

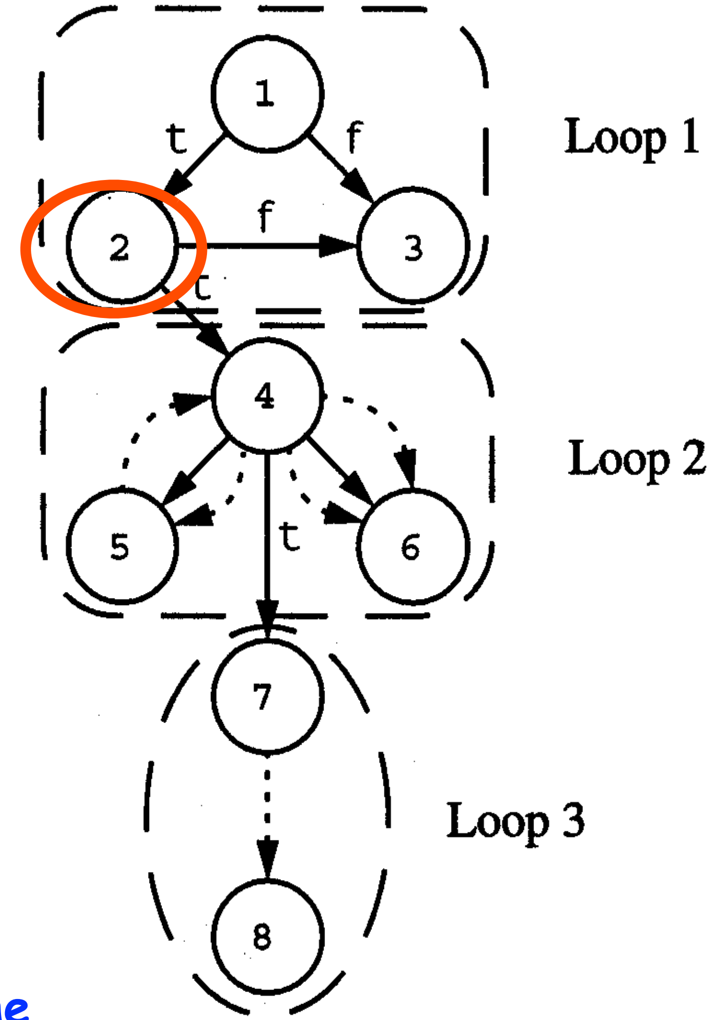
```
DO I = 1, N
 1      IF (A(I).NE.0) THEN
 2        IF (B(I)/A(I).GT.1) GOTO 4
        ENDIF
 3      A(I) = B(I)
        GOTO 8
 4      IF (A(I).GT.T) THEN
 5        T = (B(I) - A(I)) + T
        ELSE
 6        T = (T + B(I)) - A(I)
 7        B(I) = A(I)
        ENDIF
 8      C(I) = B(I) + C(I)
  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



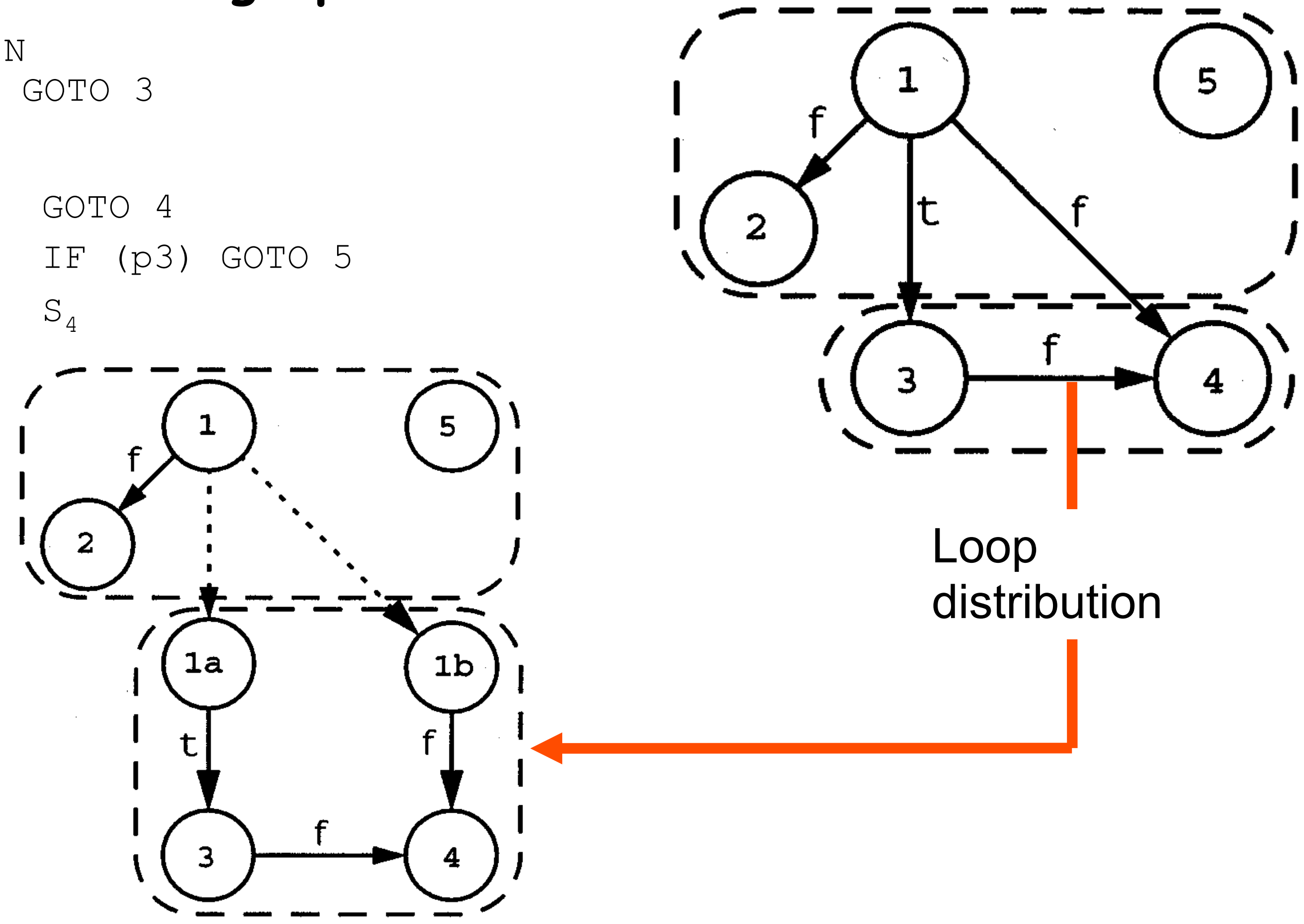
Loop Distribution

- 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**

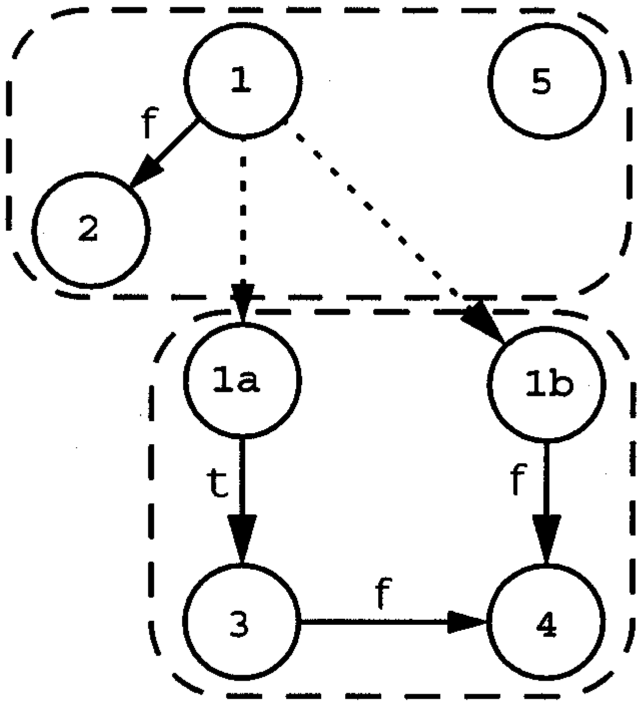
```
     DO I = 1, N
S1     IF (p1) GOTO 3
S2     ...
                GOTO 4
3               IF (p3) GOTO 5
4               S4
5
     ENDDO
```



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

- 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
- Standard 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 standard register allocation

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 dependence - save loads and cache misses
- Anti dependence - save cache misses
- Output dependence - save stores and cache misses
- Input "dependence" - save loads and cache misses
  - Read-read control flow path with no intervening write

```
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

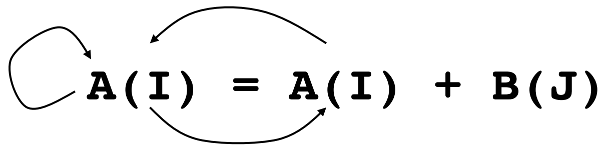
- Problem of overcounting optimization opportunities. For example

```
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
        A(I) = A(I) + B(J)
   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
- Anti dependence - 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 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

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
```