COMP 515: Advanced Compilation for Vector and Parallel Processors

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP515

Midterm exam reminder (Exam 1)

- Take-home exam (3 hours)
  - Open book: textbook only, no other resources
  - Made available on Thursday, Oct 15th, and needs to be returned to Annepha Pemberton in Duncan Hall room 3080 by Oct 22nd
  - Scope of exam is Chapters 1-6 of textbook

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 only considering data dependences ...

```
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

Control Dependences

- Two strategies to deal with control dependences:

  1) <u>If-conversion</u>: expose by converting control dependences to data dependences. Used for vectorization

     - Also supported in SIMT hardware (e.g., GPGPUs) which automatically masks out statements with control conditions = false

  2) Explicitly compute control dependences. Used for coarse-grained parallelism, or in cases where guarded execution is inefficient for vectorization.

If-conversion

- **Underlying Idea: Convert statements affected by branches to conditionally executed statements**

```
     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
```

can be converted to:

```
    DO I = 1, N
        IF (A(I-1).LE. 0.0) A(I) = A(I) + B(I)*C
    ENDDO
```

If-conversion

```
      DO 100 I = 1, N
S1    IF (A(I-1).GT. 0.0) GO TO 100
S2    A(I) = A(I) + B(I) * C
S3    B(I) = B(I) + A(I)
100 CONTINUE
```

- **can be converted to:**

```
        DO 100 I = 1, N
S2        IF (A(I-1).LE. 0.0) A(I) = A(I) + B(I) * C
S3        IF (A(I-1).LE. 0.0) B(I) = B(I) + A(I)
100   CONTINUE
```

- **And then vectorized using the Fortran WHERE statement:**

```
         DO 100 I = 1, N
S2          IF (A(I-1).LE. 0.0) A(I) = A(I) + B(I) * C
100    CONTINUE
S3     WHERE (A(0:N-1).LE. 0.0) B(1:N) = B(1:N) + A(1:N)
```

If-conversion

- **If-conversion assumes a target notation of guarded execution in which each statement implicitly contains a logical expression controlling its execution**

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

- **with guarded execution instead:**

```
S1   M = A(I-1).GT. 0.0
S2   IF (.NOT. M) A(I) = A(I) + B(I)*C
100 CONTINUE
```

Branch Classification

- **Forward Branch:** transfers control to a target that occurs lexically after the branch but at the same level of nesting

- **Backward Branch:** transfers control to a statement occurring lexically before the branch but at the same level of nesting

- **Exit Branch:** terminates one or more loops by transferring control to a target outside a loop nest
  - The break and return statements in C are examples of exit branches, when they occur inside a loop

If-conversion

- **If-conversion is a composition of two different transformations:**
  1. Branch relocation
  2. Branch removal

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
```

Removal of Forward Branches: Correctness

- To show correctness we must establish:
  - the guard for statement instance in the new program is true if and only if the corresponding statement in the old program is executed,
    - unless the statement has been introduced by the compiler to capture a guard variable value, which must be executed at the point the conditional expression would have been evaluated
  - the order of execution of statements in the new program with true guards is the same as the order of execution of those statements in the original program
  - Any expression with side effects is evaluated exactly as many times in the new program as in the old program

Control Dependences

- Two strategies to deal with control dependences:

  1) If-conversion: expose by converting control dependences to data dependences. Used for vectorization

    - Also supported in SIMT hardware (e.g., GPGPUs) which automatically masks out statements with control conditions = false

  2) Explicitly compute control dependences. Used for coarse-grained parallelism, or in cases where guarded execution is inefficient for vectorization.

Control Flow Graph Definition (Recap)

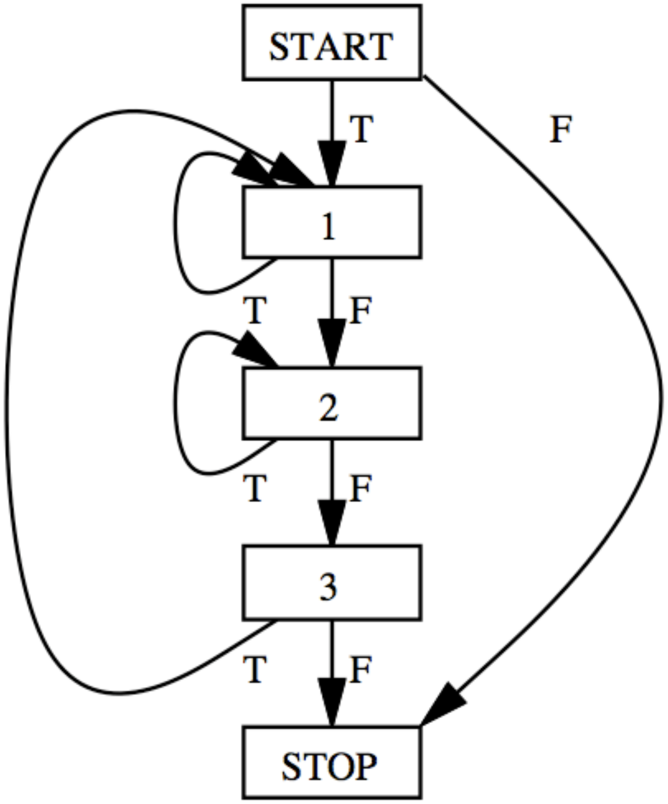
A *control flow graph* $CFG = (N_c, E_c, T_c)$ consists of

- N_c, a set of nodes. A node represents a straight-line sequence of operations with no intervening control flow i.e a basic block.
- $E_c \subseteq N_c \times N_c \times$ *Labels*, a set of *labeled* edges.
- T_c, a node type mapping. $T_c(n)$ identifies the type of node n as one of: *START*, *STOP*, *OTHER*.

We assume that *CFG* contains a unique *START* node and a unique *STOP* node, and that for any node N in *CFG*, there exist directed paths from *START* to N and from N to *STOP*.

Control Flow Graph: Example

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



CONTROL FLOW GRAPH

Dominators: Definition

Node V *dominates* another node $W \neq V$ if and only if every directed path from *START* to W in *CFG* contains V.

Define $dom(W) = \{V \mid V \text{ dominates } W\}$, the set of *dominators* of node W.

Consider any simple path from *START* to W containing W's dominators in the order $V_1, \ldots, V_k$. Then all simple paths from *START* to W must contain W's dominators in the same order. The element closest to W, $V_k = idom(W)$, is called the *immediate dominator* of W.

The $idom$ relation can be represented as a directed tree with root = *START*, and *parent*$(W) = idom(W)$.

Postdominators: Definition

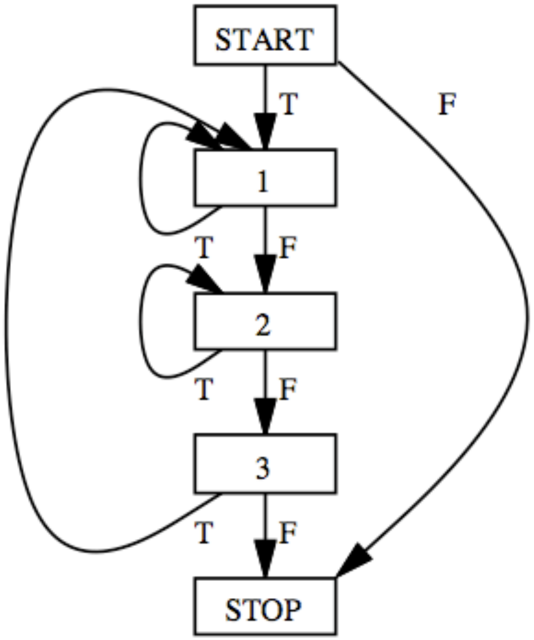
Node W *postdominates* another node $V \neq W$ if and only if every directed path from V to *STOP* in *CFG* contains W.

Define $pdom(V) = \{W \mid W$ postdominates $V\}$, the set of *postdominators* of node V.

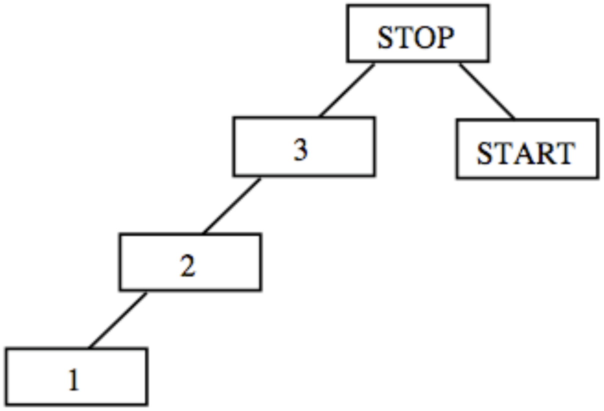
Consider any simple path from V to *STOP* containing V's postdominators in the order $W_1, \ldots, W_k$. Then all simple paths from V to *STOP* must contain V's postdominators in the same order. The element closest to V, $W_1 = ipdom(V)$, is called the *immediate postdominator* of V.

The $ipdom$ relation can be represented as a directed tree with root $=$is *STOP* and *parent*$(V) = ipdom(V)$.

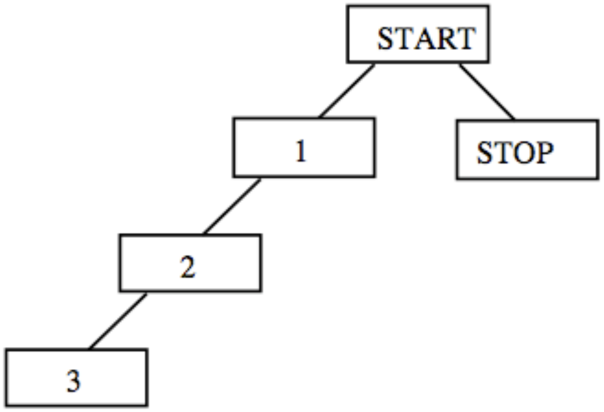
Examples of Dominator and Postdominator Trees



CONTROL FLOW GRAPH



POST–DOMINATOR TREE



DOMINATOR TREE

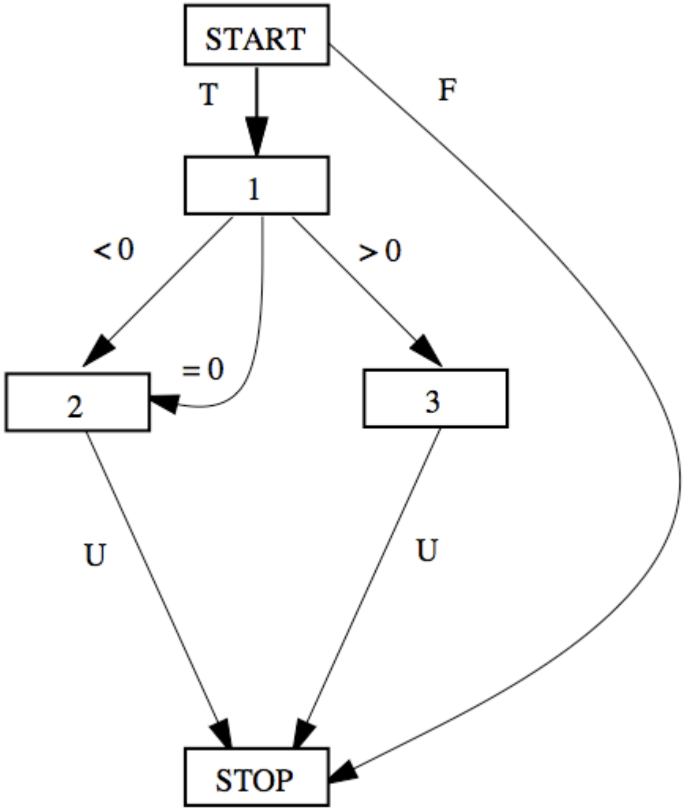
Control Dependence: Definition

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

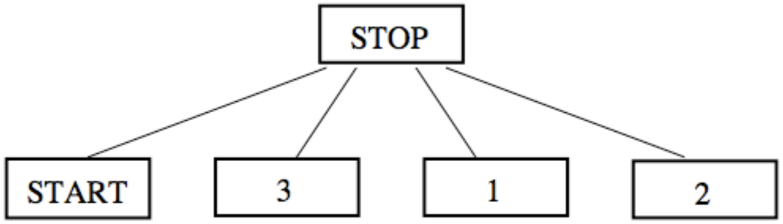
1. there exists a nonnull path $X \longrightarrow Y$, starting with the edge labeled L, such that Y post-dominates every node, W, strictly between X and Y in the path, and
2. Y does not post-dominate X.

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

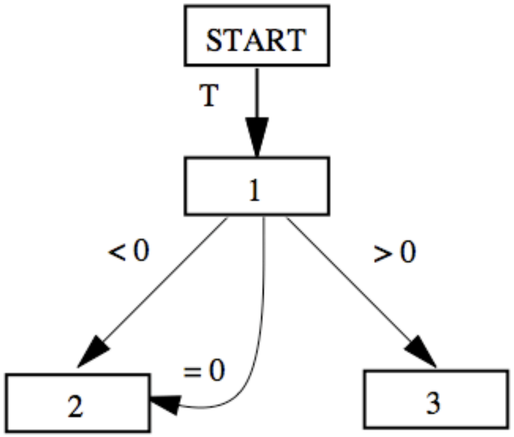
Example: Acyclic CFG and its Control Dependence Graph (CDG)



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)

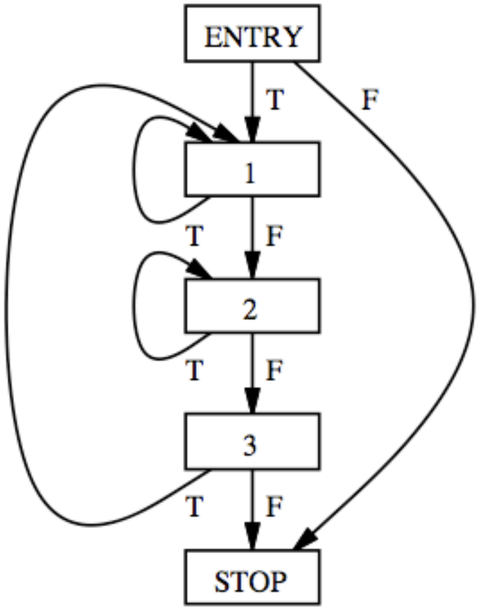
Program Dependence Graph

Program Dependence Graph (PDG) consists of

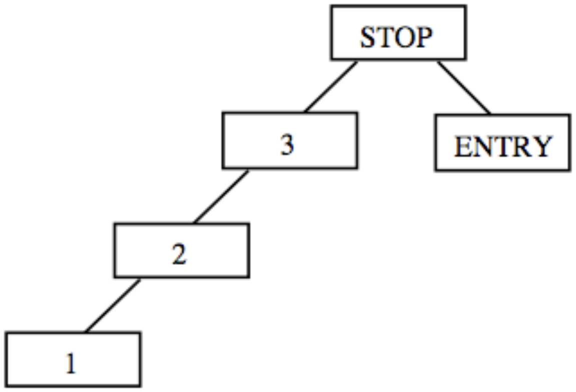
1. Set of nodes, as in the CFG
2. Control dependence edges
3. Data dependence edges

Together, the control and data dependence edges dictate whether or not a proposed code transformation is legal.

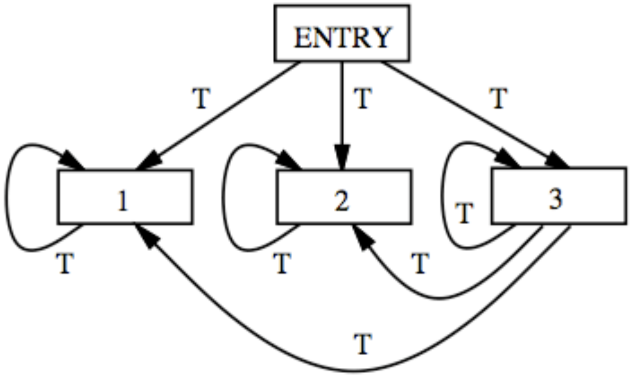
Example: Cyclic CFG and its CDG



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.

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

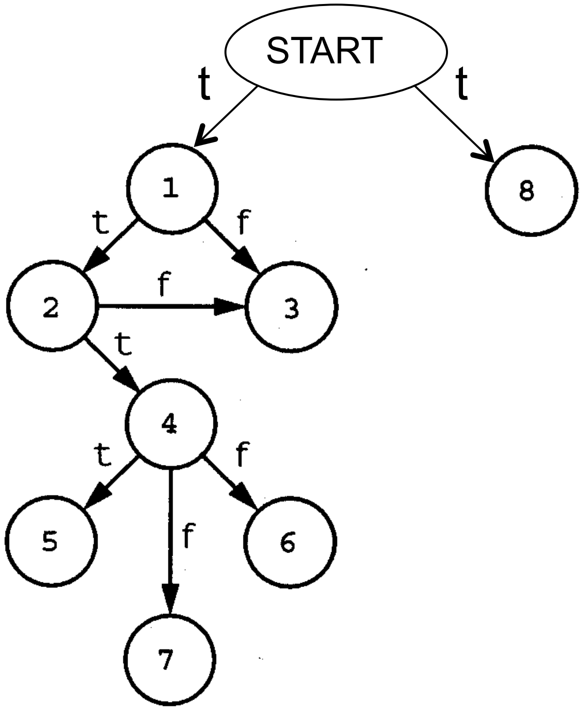
Loop Distribution

- **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

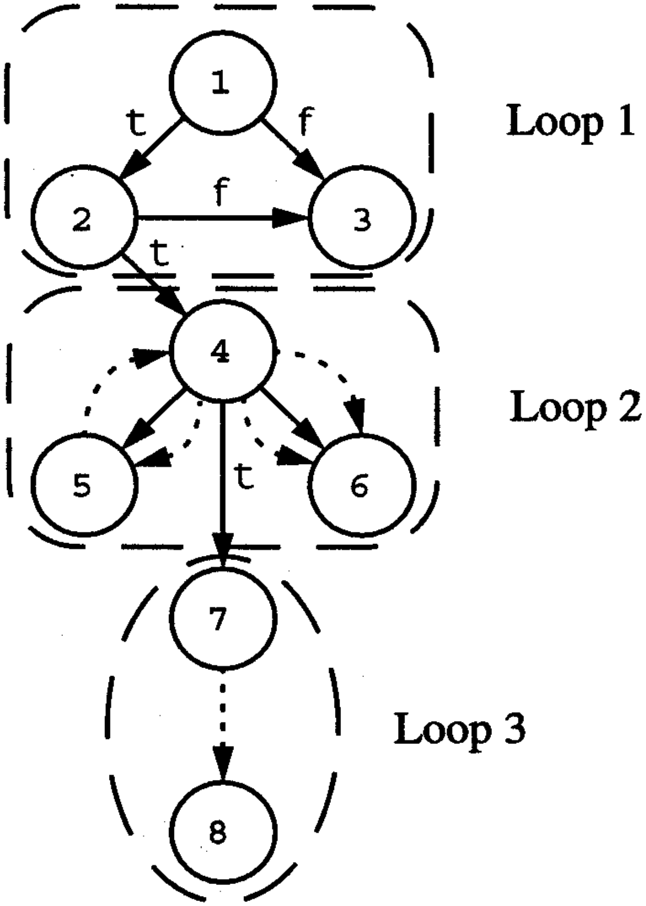
for loop body



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