Lecture 9:

Loop Invariant Computation and Code Motion

I. Finding loops

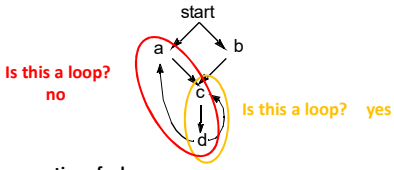
II. Loop-invariant computation

III. Algorithm for code motion

[ALSU 9.6]

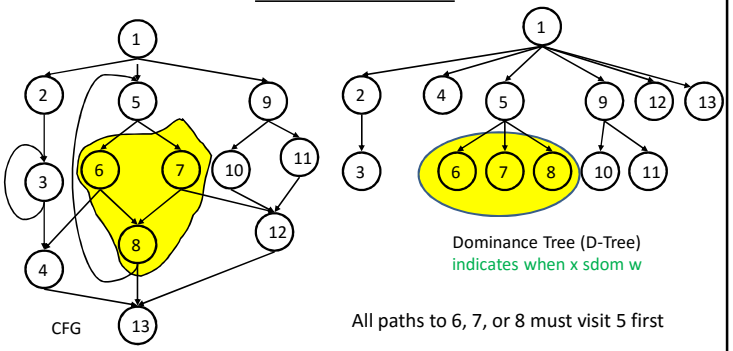
What is a Loop?

- **Goals:**
  - Define a loop in graph-theoretic terms (control flow graph)
  - Independent of specific programming language constructs used
  - A uniform treatment for all loops: DO, while, for, goto's
- **Not every cycle is a "loop" from an optimization perspective**



- **Intuitive properties of a loop**
  - single entry point
  - edges must form at least a cycle

Review: Dominance



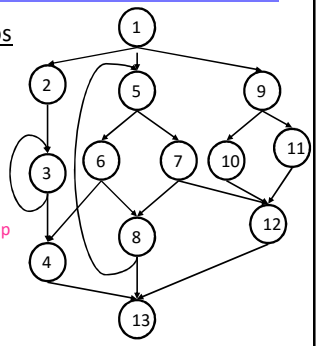
All paths to 6, 7, or 8 must visit 5 first

x strictly dominates w (x sdom w) iff impossible to reach w without passing through x first

x dominates w (x dom w) iff x sdom w OR x = w

Natural Loops

- Single entry-point: ***header***
  - a header dominates all nodes in the loop
- A ***back edge*** is an arc whose head dominates its tail (tail -> head)
  - a back edge must be a part of at least one loop
- The **natural loop of a back edge** is the smallest set of nodes that includes the head and tail of the back edge, and has no predecessors outside the set, except for the predecessors of the header.



Algorithm to Find Natural Loops

Step 1. Find the dominator relations in a flow graph

Step 2. Identify the back edges

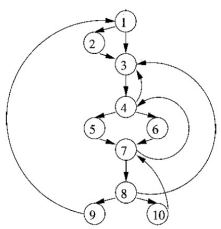
Step 3. Find the natural loop associated with the back edge

Step 1. Finding Dominators

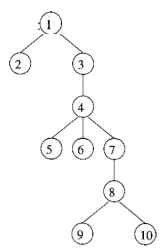
- **Definition**
  - Node *d* dominates node *n* in a graph (*d* dom *n*) if every path from the start node to *n* goes through *d*
- **Formulated as MOP problem:**
  - node *d* lies on all possible paths reaching node *n* (d dom p for all pred p of n)
    - Direction: forward
    - Values: basic blocks
    - Meet operator: $\cap$
    - Top: all basic blocks
    - Bottom: {}
    - Boundary condition for entry node: OUT[entry]= {entry}
    - Initialization for internal nodes OUT[b]= Top
    - Finite descending chain? Yes (depth=number of basic blocks)
    - Transfer function: $f_b(x) = \{b\} \cup x$
- **Speed:**
  - With rPostorder, most flow graphs (reducible flow graphs) converge in 1 pass

Example: Finding Dominators

OUT[b]={b} U ($\cap_{\{p=pre\ (b)\}}$OUT[p])



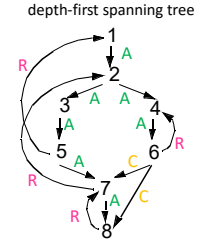
OUT[1] = {1}
OUT[2] = {1,2}
OUT[3] = {1,3}
OUT[4] = {1,3,4}
OUT[5] = {1,3,4,5}
OUT[6] = {1,3,4,6}
OUT[7] = {1,3,4,7}
OUT[8] = {1,3,4,7,8}
OUT[9] = {1,3,4,7,8,9}
OUT[10] = {1,3,4,7,8,10}
(No change in second iteration)



[ALSU, p.657-659]

Step 2. Finding Back Edges

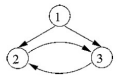
- **Depth-first spanning tree**
  - Edges traversed in a depth-first search of the flow graph form a depth-first spanning tree



- **Categorizing edges in graph**
  - Advancing edges (A): from ancestor to proper descendant
  - Cross edges (C): from right to left
  - Retreating edges (R): from descendant to ancestor (not necessarily proper)

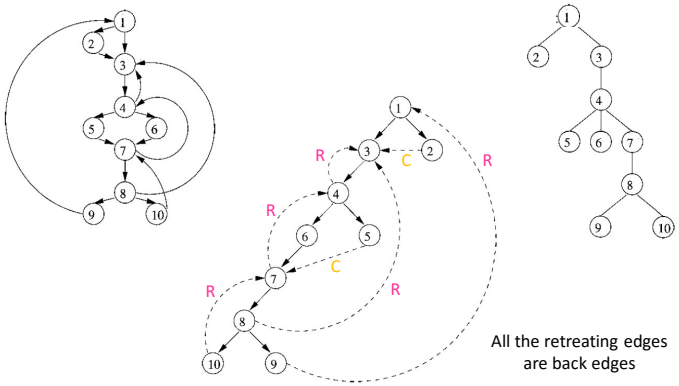
Back Edges

- **Definition**
  - **Back edge**: t->h, h dominates t
- **Relationships between graph edges and back edges**
- **Algorithm**
  - Perform a depth first search
  - For each retreating edge t->h, check if h is in t's dominator list
- **Most programs (all structured code, and most GOTO programs) have reducible flow graphs**
  - retreating edges = back edges



A nonreducible flow graph

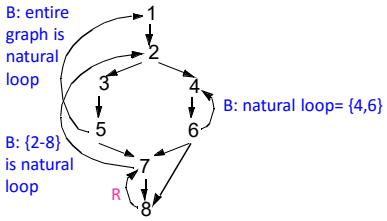
Example: Cross Edges, Retreating Edges, Back Edges



All the retreating edges are back edges

Step 3. Constructing Natural Loops

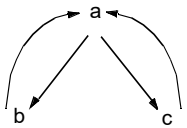
- The **natural loop of a back edge (B)** is the smallest set of nodes that includes the head and tail of the back edge, and has no predecessors outside the set, except for the predecessors of the header.



- **Algorithm**
  - delete *h* from the flow graph
  - find those nodes that can reach *t* (those nodes plus *h* form the natural loop of *t* -> *h*)

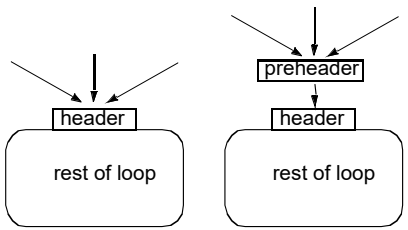
Inner Loops

- **If two loops do not have the same header:**
  - they are either disjoint, or
  - one is entirely contained (nested within) the other
    - inner loop: one that contains no other loop.
- **If two loops share the same header:**
  - Hard to tell which is the inner loop
  - Combine as one



Preheader

- **Optimizations often require code to be executed once before the loop**
- **Create a preheader basic block for every loop**

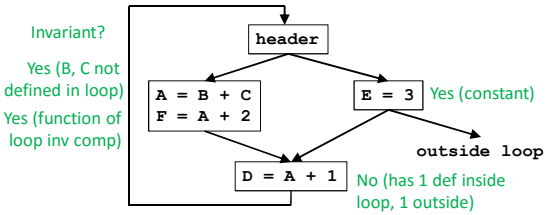


Finding Loops: Summary

- **Define loops in graph theoretic terms**
- **Definitions and algorithms for:**
  - Dominators
  - Back edges
  - Natural loops

II. Loop-Invariant Computation and Code Motion

- **A loop-invariant computation:**
  - a computation whose value does not change as long as control stays within the loop
- **Code motion:**
  - to move a statement within a loop to the preheader of the loop



Algorithm

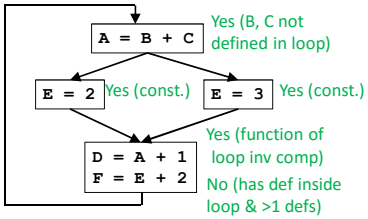
- **Observations**
  - Loop invariant
    - operands are defined outside loop or invariant themselves
  - Code motion
    - not all loop invariant instructions can be moved to preheader
- **Algorithm**
  - Find invariant expressions
  - Conditions for code motion
  - Code transformation

Detecting Loop Invariant Computation

- Compute reaching definitions
- Mark INVARIANT if
  all the definitions of B and C that reach a statement A=B+C
  are outside the loop
  - constant B, C?
- Repeat: Mark INVARIANT if
  - all reaching definitions of B are outside the loop, or
  - there is exactly one reaching definition for B, and it is from a loop-invariant statement inside the loop
  - similarly for C

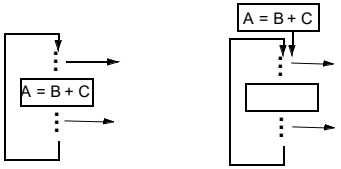
  until no changes to set of loop-invariant statements occur.

Which Statements are Loop Invariant?



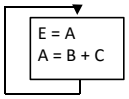
III. Conditions for Code Motion

- **Correctness: Movement does not change semantics of program**
- **Performance: Code is not slowed down**



OK to move?
No (moved past exit)

- **Basic idea: defines once and for all**
  - control flow: once?
    - Code dominates all exits
  - other definitions: for all?
    - No other definition
  - uses of the definition: for all?
    - Dominates use or no other reaching defs to use



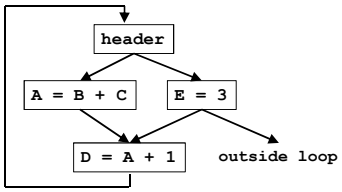
OK to move?
Not loop inv.
No (doesn't dominate use)

Code Motion Algorithm

Given: a set of nodes in a loop

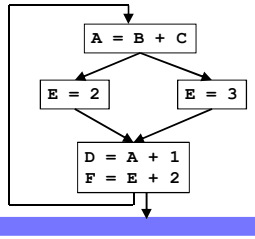
- **Compute reaching definitions**
- **Compute loop invariant computation**
- **Compute dominators**
- **Find the exits of the loop (i.e. nodes with successor outside loop)**
- **Candidate statement for code motion:**
  - loop invariant
  - in blocks that dominate all the exits of the loop
  - assign to variable not assigned to elsewhere in the loop
  - in blocks that dominate all blocks in the loop that use the variable assigned
- **Perform a depth-first search of the blocks**
  - Move candidate to preheader if all the invariant operations it depends upon have been moved

Code Motion Examples



Which statements can be moved to loop preheader?

Only E=3: only statement dominating all exits

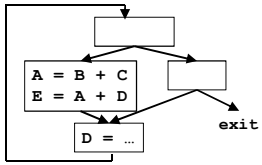


A=B+C
D=A+1

(Although E=2, E=3 are invariant, neither is only def of E)

More Aggressive Optimizations

- **Gamble on: most loops get executed**
  - When can we relax constraint of dominating all exits?



Can relax if destination not live after loop & can compute in preheader w/o causing an exception

- **Landing pads**

```
While p do loop-body  ➔  if p {
                            preheader
                            repeat
                               loop-body
                            until not p;
                         }
```

Ensures preheader executes only if enter loop

Summary

- **Precise definition and algorithm for loop invariant computation**
- **Precise algorithm for code motion**
- **Use of reaching definitions and dominators in optimizations**

What's Next

- Homework #1 due midnight
- No class on Friday
- Monday's Lecture: Induction Variables, Strength Reduction [ALSU 9.1.8]