Program Loops

- **Loop**: a computation repeatedly executed until a terminating condition is reached.

- High-level loop constructs:
  - While loop: `while(E) S`
  - Do-while loop: `do S while(E)`
  - For loop: `for(i=1;i<=n;i++) C`

- Why are loops important:
  - Most of the execution time is spent in loops
  - Typically: 90/10 rule, 10% code is a loop

- Therefore, loops are important targets of optimizations

Detecting Loops

- Need to identify loops in the program:
  - Easy to detect loops in high-level constructs
  - Difficult to detect loops in low-level code or in general control-flow graphs

- Examples where loop detection is difficult:
  - Languages with unstructured "goto" constructs: structure of high-level loop constructs may be destroyed
  - Optimizing Java bytecodes (without high-level source program): only low-level code is available

Control-Flow Analysis

- **Goal**: identify loops in the control flow graph

- A loop in the CFG:
  - Is a set of CFG nodes (basic blocks)
  - Has a loop header such that control to all nodes in the loop always goes through the header
  - Has a back edge from one of its nodes to the header

Dominateds

- Use concept of dominators to identify loops:
  "CFG node d dominates CFG node n if all the paths from entry node to n go through d"

- Intuition:
  - Header of a loop dominates all nodes in loop body
  - Back edges = edges whose heads dominate their tails
  - Loop identification = back edge identification

Outline

- Control flow analysis
  - Detect loops in control flow graphs
  - Dominators

- Loop optimizations
  - Code motion
  - Strength reduction for induction variables
  - Induction variable elimination
Immediate Dominators

- Properties:
  1. CFG entry node $n_0$ dominates all CFG nodes
  2. If $d_1$ and $d_2$ dominate $n$, then either
     - $d_1$ dominates $d_2$ or $d_2$ dominates $d_1$
- Immediate dominator $\text{idom}(n)$ of node $n$:
  - $\text{idom}(n) = n$
  - $\text{idom}(n)$ dominates $n$
  - If $m$ dominates $n$, then $m$ dominates $\text{idom}(n)$
- Immediate dominator $\text{idom}(n)$ exists and is unique because of properties 1 and 2

Dominator Tree

- Build a dominator tree as follows:
  - Root is CFG entry node $n_0$
  - $m$ is child of node $n$ iff $n \text{idom}(m)$
- Example:

Computing Dominators

- Formulate problem as a system of constraints:
  - $\text{dom}(n)$ is set of nodes who dominate $n$
  - $\text{dom}(n_0) = \{n_0\}$
  - $\text{dom}(n) = \cap \{ \text{dom}(m) | m \in \text{pred}(n) \}$
- Can also formulate problem in the dataflow framework:
  - What is the dataflow information?
  - What is the lattice?
  - What are the transfer functions?
  - Use dataflow analysis to compute dominators

Natural Loops

- Back edge: edge $n \rightarrow h$ such that $h$ dominates $n$
- Natural loop of a back edge $n \rightarrow h$:
  - $h$ is loop header
  - Loop nodes is set of all nodes that can reach $n$ without going through $h$
- Algorithm to identify natural loops in CFG:
  - Compute dominator relation
  - Identify back edges
  - Compute the loop for each back edge

Disjoint and Nested Loops

- Property: for any two natural loops in the flow graph, one of the following is true:
  1. They are disjoint
  2. They are nested
  3. They have the same header
- Eliminate alternative 3: if two loops have the same header and none is nested in the other, combine all nodes into a single loop

Loop Preheader

- Several optimizations add code before header
- Insert a new basic block (called preheader) in the CFG to hold this code
Loop optimizations

- Now we know the loops in the program
- Next: optimize loops
  - Loop invariant code motion
  - Strength reduction of induction variables
  - Induction variable elimination

Loop Invariant Code Motion

- Idea: if a computation produces same result in all loop iterations, move it out of the loop
- Example: for (i=0; i<10; i++)
  a[i] = 10*i + x*y;
- Expression x*y produces the same result in each iteration; move it of the loop:
  t = x*y;
  for (i=0; i<10; i++)
    a[i] = 10*i + t;

Loop Invariant Computation

- An instruction \( a = b \ OP \ c \) is loop-invariant if each operand is:
  - Constant, or
  - Has all definitions outside the loop, or
  - Has exactly one definition, and that is a loop-invariant computation

- Reaching definitions analysis computes all the definitions of \( x \) and \( y \) which may reach \( t = x \ OP \ y \)

Algorithm

\[ INV = \emptyset \]
Repeat:
  for each instruction \( i \in INV \)
    if operands are constants, or
    have definitions outside the loop, or
    have exactly one definition \( d \in INV \)
      then \( INV = INV \cup \{d\} \)
Until no changes in \( INV \)

Code Motion

- Next: move loop-invariant code out of the loop
- Suppose \( a = b \ OP \ c \) is loop-invariant
- We want to hoist it out of the loop

- Code motion of a definition \( d: a = b \ OP \ c \) in pre-header is valid if:
  1. Definition \( d \) dominates all loop exits where \( a \) is live
  2. There is no other definition of \( a \) in loop
  3. All uses of \( a \) in loop can only be reached from definition \( d \)

Other Issues

- Preserve dependencies between loop-invariant instructions when hoisting code out of the loop:
  for (i=0; i<N; i++)
    \( x = y+z; \)
    \( t = x*y; \)
    \( a[i] = 10*i + x*y; \)
  for (i=0; i<N; i++)
    \( a[i] = 10*i + t; \)
- Nested loops: apply loop invariant code motion algorithm multiple times
  \( t1 = x*y; \)
  for (i=0; i<N; i++)
    \( t2 = t1 + 10*i; \)
  for (j=0; j<M; j++)
    \( a[j][i] = x*y + 10*i + 100*j; \)
  for (j=0; j<M; j++)
    \( a[j][i] = t2 + 100*j; \)