CSCI 742 - Compiler Construction

Lecture 37

Loop Optimizations

Instructor: Hossein Hojjat

April 25, 2018

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<=u; i+=c) S`
- 90/10 rule:
  90% of any computation is normally spent in 10% of the code (loops)
- Control-flow graph can help give us useful information
- How to analyze the control-flow graph to detect loops?
- Some techniques to optimize loops

Some Loop Optimizations

- Loop-invariant code motion
  - Pre-compute before entering the loop
- Strength Reduction
  - Replace expensive operations (multiplications) with cheaper ones (additions)
- Elimination of induction variables
  - Induction variable: variable whose value on each loop iteration is a linear function of the iteration index
  - In most cases induction variables can be removed (if not used after loop)
- Elimination of null and array-bounds checks
  - Use data-flow analysis to prove integer range
- Loop unrolling to reduce number of control transfers

Detecting Loops

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

Examples where loop detection is difficult:

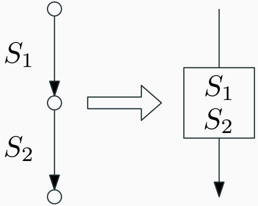
- Languages with unstructured `goto` constructs:
  structure of high-level loop constructs may be destroyed
- Java bytecode level (without high-level source program):
  only low-level code is available

Basic Blocks

- In some applications (e.g. loop detection) control-flow graph of basic block is more convenient
- Basic block is a sequence of instructions
  - no branches out from the middle of basic block
  - no branches into the middle of basic block
- Basic block should be maximal
- Execution of basic block
  - starts with first instruction
  - includes all instructions in basic block

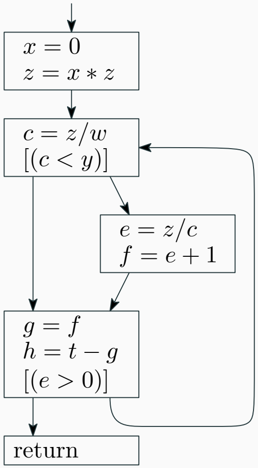
Basic Block Construction

- Start with control-flow graph of instructions
- Visit all edges in graph
- Merge adjacent edges



Basic Blocks Example

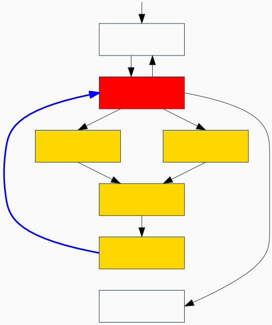
```
    x = 0;
    z = x * z;
L1: c =  z / w;
    if (c < y) goto L2;
    e = z / c;
    f = e + 1;
L2: g = f;
    h = t - g;
    if (e > 0) goto L3;
    goto L1;
L3: return
```



Control-Flow Analysis

- Goal: identify loops in the control flow graph

A loop in the CFG:

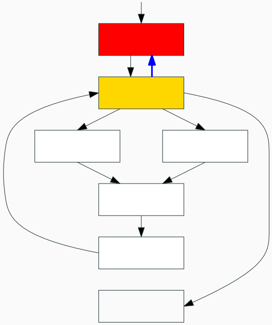


- Is a set of basic blocks
- Has a loop header: node in a loop that has no immediate predecessors in the loop
- Has a back edge from one of its nodes to the header

Control-Flow Analysis

- Goal: identify loops in the control flow graph

A loop in the CFG:

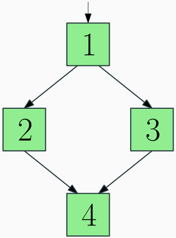


- Is a set of basic blocks
- Has a loop header: node in a loop that has no immediate predecessors in the loop
- Has a back edge from one of its nodes to the header

Dominators

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

- Every node dominates itself
- 1 dominates 1, 2, 3, 4
- 2 does not dominate 4
- 3 does not dominate 4



Intuition:

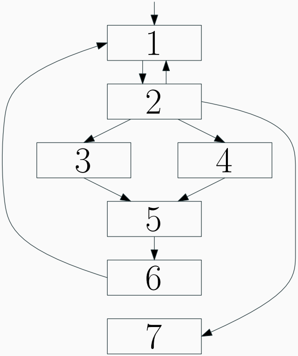
- Header of a loop dominates all nodes in loop body
- Back edges = edges whose heads dominate their tails
- Loop identification = back edge identification

Immediate Dominators

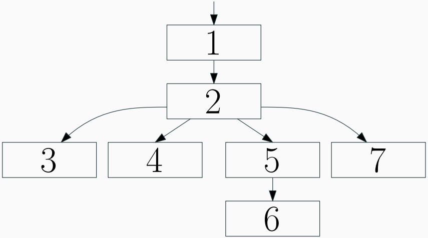
- CFG entry node dominates all CFG nodes
- If d_1 and d_2 dominate n, then either
  - d_1 dominates d_2, or
  - d_2 dominates d_1
- d strictly dominates n if d dominates n and $d \neq n$
- **Immediate dominator** *idom*(n) of a node n: the unique last strict dominator of n on any path from entry node

Dominator Tree

- Build a dominator tree as follows:
- Nodes are nodes of control flow graph
- Root is CFG entry node
- Edge from d to n if d immediate dominator of n



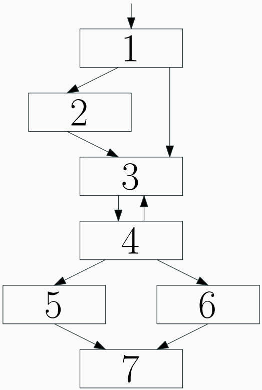
Control Flow Graph



Dominator Tree

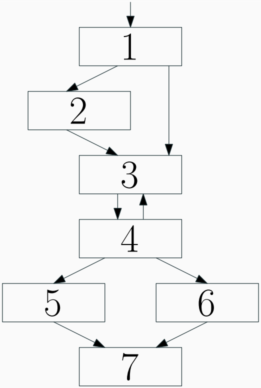
Exercise

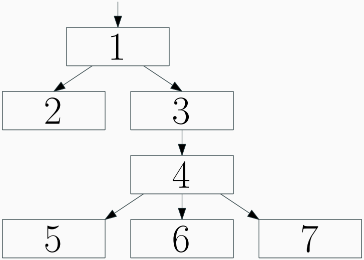
- Build the dominator tree for the following control flow graph



Exercise

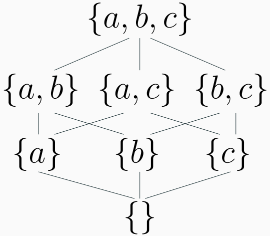
- Build the dominator tree for the following control flow graph





Data-flow-like Algorithm for Computing Dominators

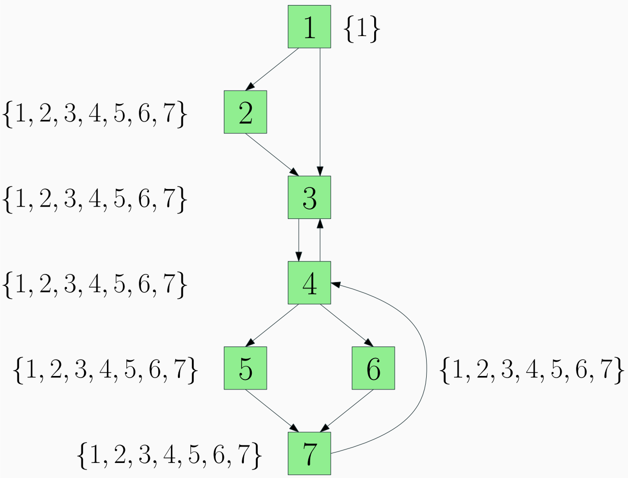
- Let N = set of all basic blocks
- Lattice: $(2^N, \subseteq)$
- Has finite height
- Meet is set intersection, top element is N



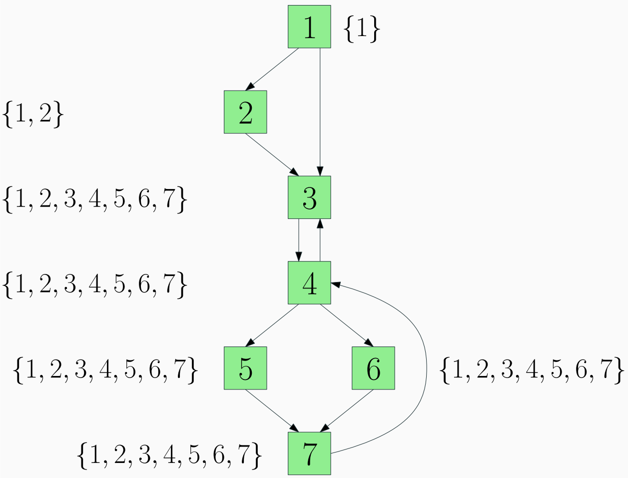
Formulate problem as a system of constraints

- Define *dom*(n) = set of nodes that dominate n
- *dom*$(n_0) = \{n_0\}$ where n_0 is the entry node
- *dom*$(n) = \bigcap\{$*dom*$(m) \mid m \in$ *pred*$(n)\} \cup \{n\}$
  i.e, the dominators of n are the dominators of all of n's predecessors and n itself

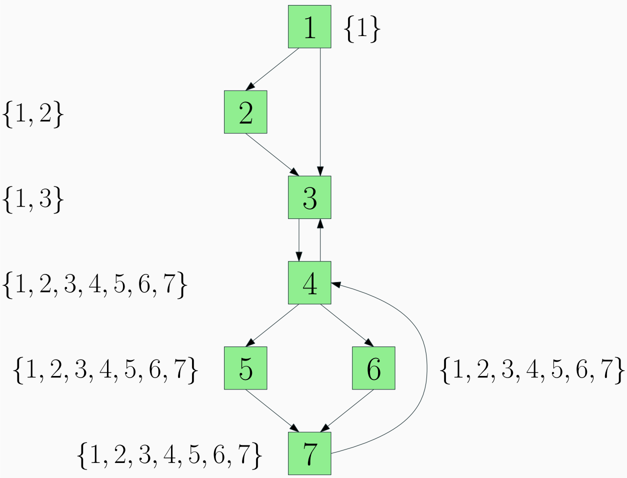
Dominator Computation



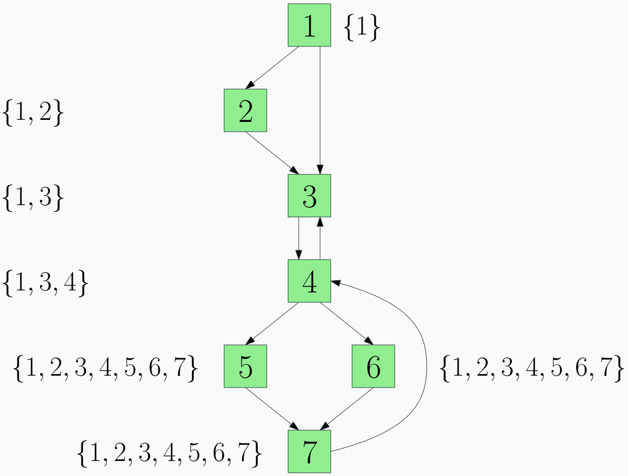
Dominator Computation



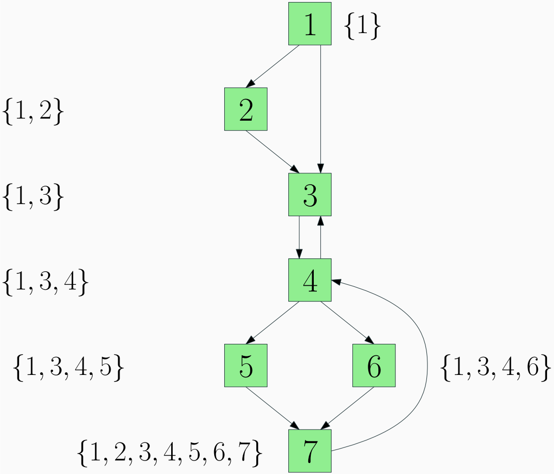
Dominator Computation



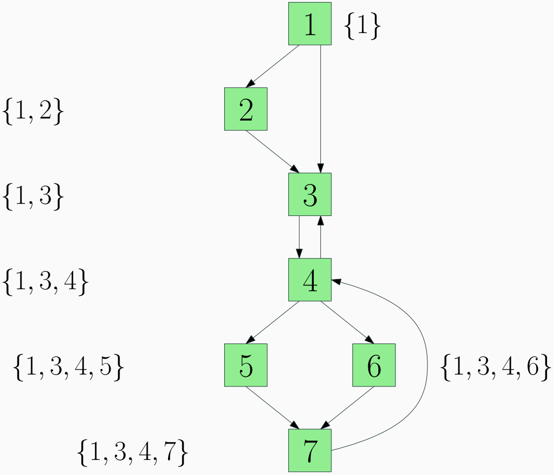
Dominator Computation



Dominator Computation



Dominator Computation



- $7 \to 4$ is a back edge: head $\boxed{4}$ dominates tail $\boxed{7}$
  - $4 \in \mathit{dom}(7)$

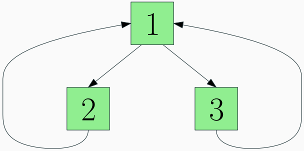
Natural Loops

- Back edge: edge $n \to h$ such that h dominates n
- **Natural loop** of a back edge $n \to h$:
  - h is loop header
  - Set of 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

Nested Loops

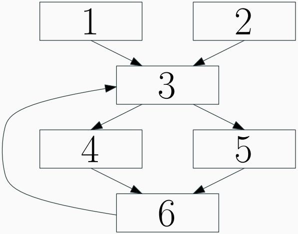
- If two loops do not have same header then
  - Either one loop (inner loop) contained in other (outer loop)
  - Or two loops are disjoint
- If two loops have same header, typically unioned and treated as one loop

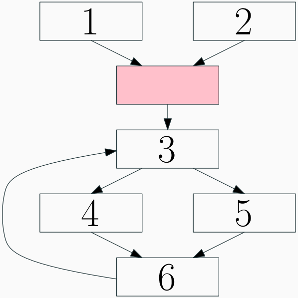
Two loops: $\{1, 2\}$ and $\{1, 3\}$
Unioned: $\{1, 2, 3\}$



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

Next: optimize these loops

- Loop invariant code motion (this lecture)
- Strength reduction of induction variables
- Induction variable elimination

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
for( i = 1; i <= N; i++) {
  x = x + 1;
  // inner loop
  for( j = 1; j <= N; j++)
    a[i][j] = 100*N + 10*i + j + x;
}
```

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N;
for( i = 1; i <= N; i++) {
  x = x + 1;
  // inner loop
  for( j = 1; j <= N; j++)
    a[i][j] = 100*N + 10*i + j + x;
}
```

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N;
for( i = 1; i <= N; i++) {
  x = x + 1;
  // inner loop
  for( j = 1; j <= N; j++)
    a[i][j] = t1    + 10*i + j + x;
}
```

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N;
for( i = 1; i <= N; i++) {
  x = x + 1;
  t2 = 10*i + x;
  for( j = 1; j <= N; j++)
    a[i][j] = t1    + 10*i + j + x;
}
```

Loop Invariant Code Motion

- If a computation produces the same value in every loop iteration, move it out of the loop

```
t1 = 100*N;
for( i = 1; i <= N; i++) {
  x = x + 1;
  t2 = 10*i + x;
  for( j = 1; j <= N; j++)
    a[i][j] = t1    + t2   + j + x;
}
```

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