Lecture 2: Basic Control Flow Analysis

COS 598C – Advanced Compilers

Prof. David August
Department of Computer Science
Princeton University

Compiler Backend IR

- Variable home location
  - Front-end – every variable in memory
  - Back-end – maximal but safe register promotion
    - All temporaries put into registers
    - All local scalars put into registers, except those accessed via &
    - All globals, local arrays/structs, unpromotable local scalars put in memory. Accessed via load/store.
- Backend IR (intermediate representation)
  - machine independent assembly code – really resource indep!
  - AKA: RTL (register transfer language) or 3-address code
  - `r1 = r2 + r3` or equivalently `add r1, r2, r3`
    - Opcode – not machine independent
    - Operands
      - Virtual registers – infinite number of these
      - Special registers – stack pointer (SP), PC, etc. (AKA Macro Regs)
      - Literals – compile-time constants

Control Flow

- Control transfer = branch (taken or fall-through)
- Control flow
  - Branching behavior of an application
  - What sequences of instructions can be executed
- Execution → Dynamic control flow
  - Direction of a particular instance of a branch
  - Predict, speculate, squash, etc.
- Compiler → Static control flow
  - Not executing the program
  - Input not known, all outcomes possible (conservative)
- Control Flow Analysis
  - Determining properties of the program branch structure
  - Determining instruction execution conditions

Basic Block (BB)

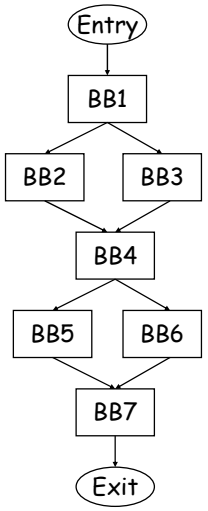
- Main Idea: Group operations into units with equivalent execution conditions
- Basic block – a sequence of consecutive operations in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end
  - Straight-line sequence of instructions
  - If one operation is executed in a BB, they all are
- Finding BB's
  - The first operation in a program/function starts a BB
  - Any operation that is the target of a branch starts a BB
  - Any operation that immediately follows a branch starts a BB

Identifying BBs – Example

```
L1: r7 = load(r8)
L2: r1 = r2 + r3
L3: beq r1, 0, L10
L4: r4 = r5 * r6
L5: r1 = r1 + 1
L6: beq r1 100 L2
L7: beq r2 100 L10
L8: r5 = r9 + 1
L9: r7 = r7 & 3
L10: r9 = load (r3)
L11: store(r9, r1)
```

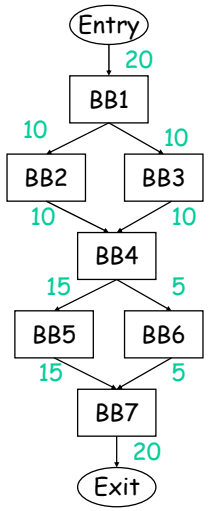

Control Flow Graph (CFG)

- Control Flow Graph – Directed graph, G = (V,E) where each vertex V is a basic block and there is an edge E, v1 (BB1) → v2 (BB2) if BB2 can immediately follow BB1 in some execution sequence
  - A BB has an edge to all blocks it can target
  - Standard representation used by many compilers
  - Often have 2 pseudo V's
    - entry node
    - exit node



Weighted CFG

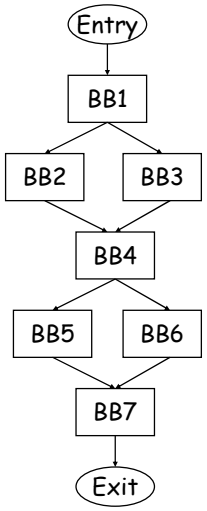
- Profiling – Run the application on 1 or more sample inputs, record some behavior
  - Control flow profiling**
    - edge profile
    - block profile
    - path profiling
  - Cache profiling
  - Memory dependence profiling
- Annotate control flow profile onto a CFG → weighted CFG
- Key idea: optimize more effectively with profile info
  - Optimize for the common case
  - Make educated guesses

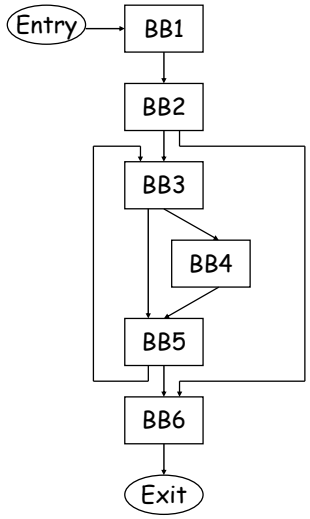


Dominator

- Dominator – Given a CFG(V, E, Entry, Exit), a node x dominates a node y, if every path from the Entry block to y contains x
- 3 properties of dominators
  - Each BB dominates itself
  - If x dominates y, and y dominates z, then x dominates z
  - If x dominates z and y dominates z, then either x dominates y or y dominates x
- Intuition
  - Given some BB, DOM blocks are guaranteed to have executed prior to executing the BB

Dominator Examples





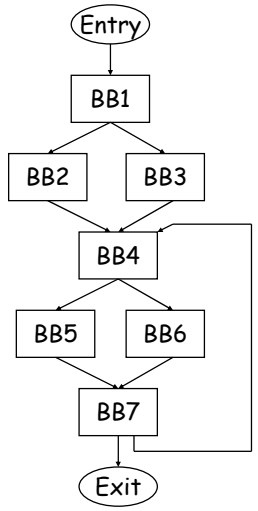
Dominator Analysis

- Compute DOM(BB_i) = set of BBs that dominate BB_i

Algorithm:

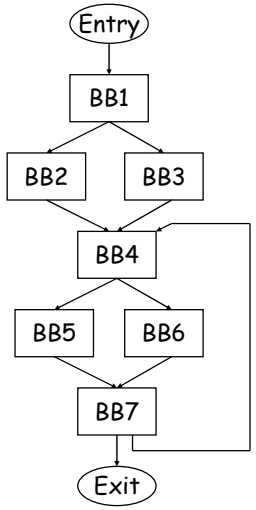
```
DOM(entry) = entry
DOM(everything else) = all nodes
change = true

while change, do
    Change = false
    for each BB (except the entry BB)
       TMP(BB) = BB + {intersect of DOM
                 of all predecessor BB's}
       if (TMP(BB) != DOM(BB))
          DOM(BB) = TMP(BB)
          change = true
```



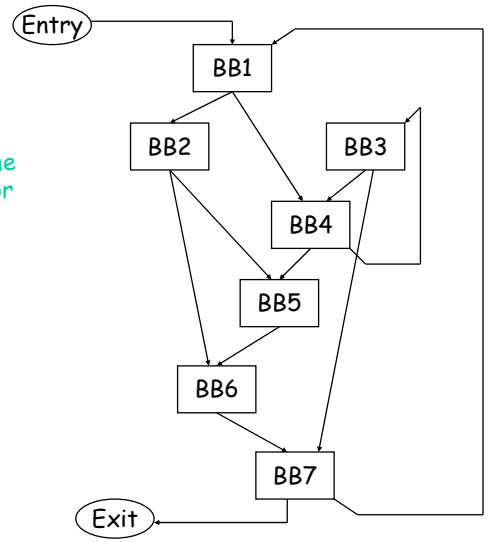
Immediate Dominator

- Immediate Dominator (IDOM)– Each node *n* has a unique immediate dominator *m* that is the last dominator of *n* on any path from the initial node to *n*
  - Closest node that dominates



Class Problem 1

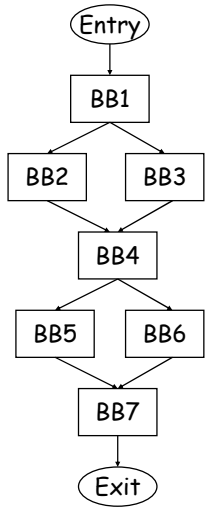
Calculate the DOM set for each BB

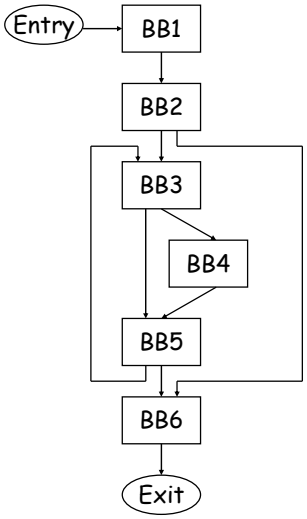


Post Dominator

- Post Dominator – Given a CFG(V, E, Entry, Exit), a node x post dominates a node y, if every path from y to the Exit contains x
- Reverse of dominator
- Intuition
  - Given some BB, which blocks are guaranteed to have executed after executing the BB

Post Dominator Examples



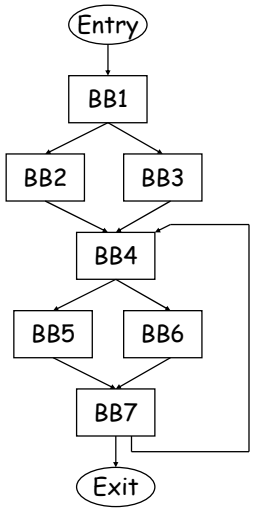


Post Dominator Analysis

- Compute PDOM(BB_i) = set of BBs that post dominate BB_i

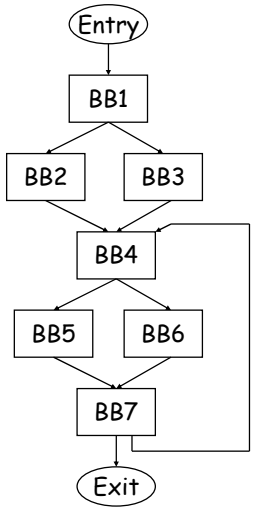
```
PDOM(exit) = exit
PDOM(everything else) = all nodes
change = true

while change, do
    change = false
    for each BB (except the exit BB)
       TMP(BB) = BB + {intersect of
         PDOM of all successor BB's}
       if (TMP(BB) != PDOM(BB))
          PDOM(BB) = TMP(BB)
          change = true
```



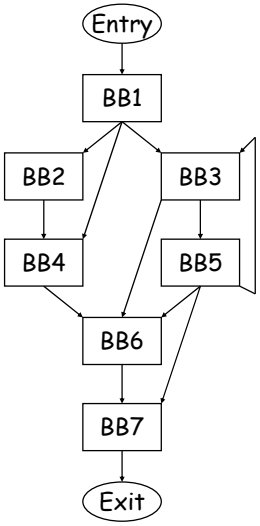
Immediate Post Dominator

- Immediate post dominator (IPDOM) – Each node *n* has a unique immediate post dominator *m* that is the first post dominator of *n* on any path from *n* to the Exit
  - Closest node that post dominates
  - First breadth-first successor that post dominates a node



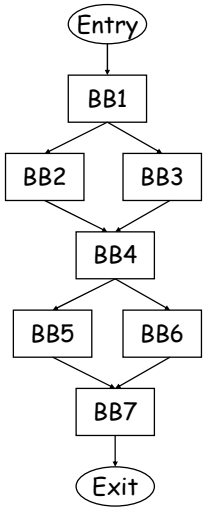
Class Problem 2

Calculate the PDOM set for each BB



Why Do We Care About Dominators?

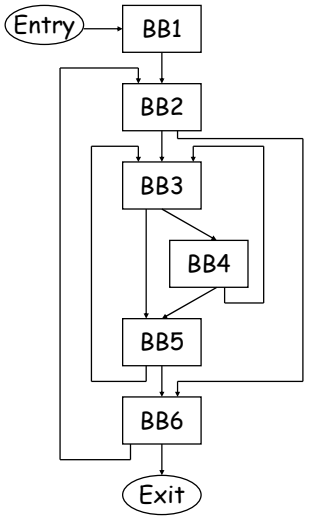
- For Loop detection (next subject)
- Dominator
  - Guaranteed to execute before
  - Redundant computation – a result is redundant if it is computed in a dominating BB
  - Most global optimizations use dominance info
- Post dominator
  - Guaranteed to execute after
  - Make a guess (ie 2 stores do not access the same location)
  - Check they really do not point to one another in the post dominating BB



Natural Loops

- Cycle suitable for optimization
  - Discuss optimization later
- 2 properties
  - Single entry point called the header
    - Header dominates all blocks in the loop
  - Must be one way to iterate the loop (ie at least 1 path back to the header from within the loop) called a backedge
- Backedge detection
  - Edge, x→ y where the target (y) dominates the source (x)

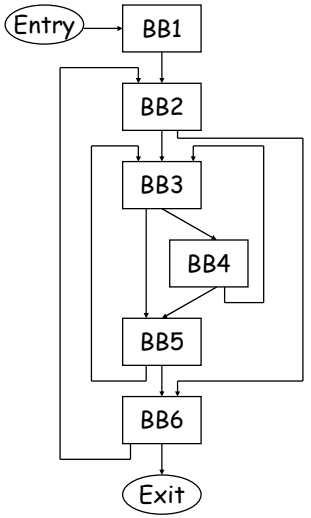
Backedge Example



Loop Detection

- Identify all backedges using DOM info
- Each backedge (x → y) defines a loop
  - Loop header is the backedge target (y)
  - Loop BB – basic blocks that comprise the loop
    - All predecessor blocks of x for which control can reach x without going through y are in the loop
- Common: Merge loops with the same header
  - For example, a loop with 2 continues
  - LoopBackedge = LoopBackedge1 + LoopBackedge2
  - LoopBB = LoopBB1 + LoopBB2
- Important property maintained
  - Header dominates all LoopBB
  - All backedges target header

Loop Detection Example

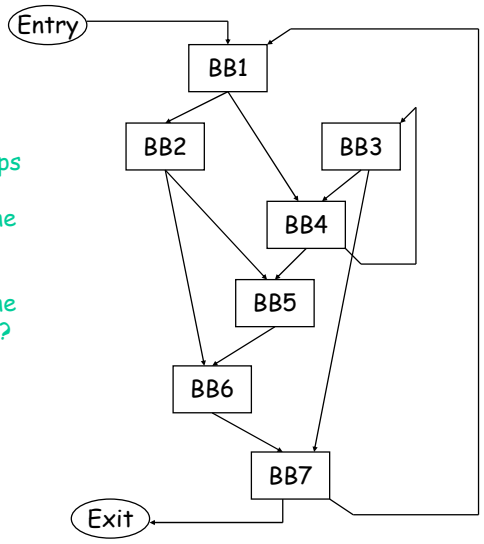


Class Problem 3

Find the loops

What are the header(s)?

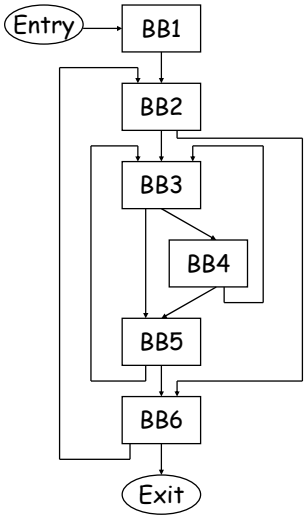
What are the backedge(s)?



Important Parts of a Loop

- Header, LoopBB
- Backedges, BackedgeBB
- Exitedges, ExitBB
  - For each LoopBB, examine each outgoing edge
  - If the edge is to a BB not in LoopBB, then its an exit
- Preheader (Preloop)
  - New block before the header (falls through to header)
  - Whenever you invoke the loop, preheader executed
  - Whenever you iterate the loop, preheader NOT executed
  - All edges entering header
    - Backedges – no change
    - All others, retarget to preheader
- Postheader (Postloop) - analogous

ExitBB/Preheader Example



Characteristics of a Loop

- Nesting (generally within a procedure scope)
  - Inner loop – Loop with no loops contained within it
  - Outer loop – Loop contained within no other loops
  - Nesting depth
    - depth(outer loop) = 1
    - depth = depth(parent or containing loop) + 1
- Invocation count
  - How many times the loop is activated (loop header weight)
- Trip count (average trip count)
  - How many times (on average) does the loop iterate
  - for (I=0; I<100; I++) → trip count = 100
  - Average trip count = weight(header) / weight(preheader)

Class Problem 4: Trip Count Calculation

Calculate the invocation and trip counts for all the loops in the graph