EECS 583 – Class 2
Control Flow Analysis

University of Michigan

September 10, 2018

Announcements & Reading Material

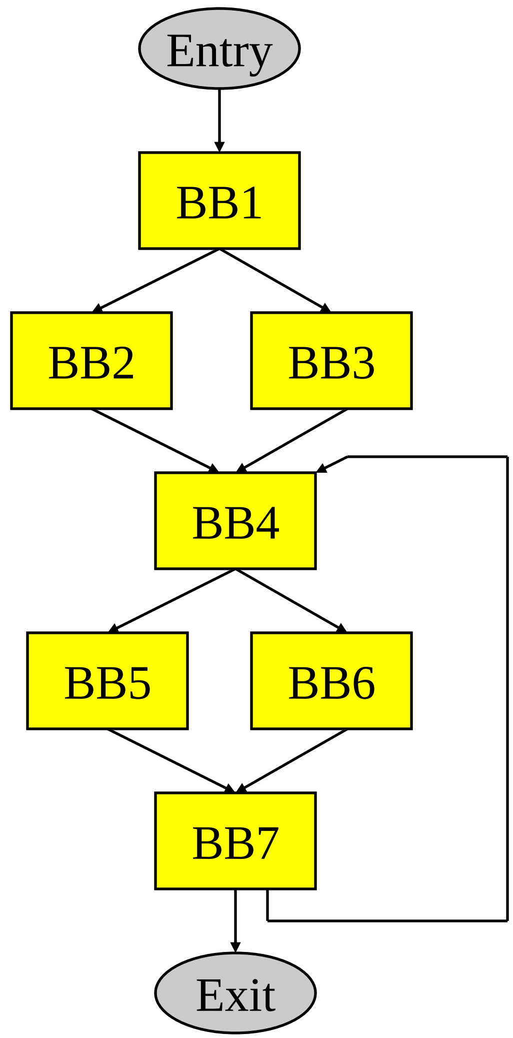
- eecs583a,eecs583b.eecs.umich.edu servers are ready
  - Everyone has home directory and login
- HW 0 – Nominally due on Wednes, but nothing to turn in
  - Please get this done ASAP, talk to Ze if you have problems
  - Needed for HW 1 which goes out on Wednes
- Reading
  - Today's class
    - Ch 9.4, 10.4 (6.6, 9.6) from Compilers: Principles, Techniques Tools Ed 1 (Ed 2)
    - "Trace Selection for Compiling Large C Applications to Microcode", Chang and Hwu, MICRO-21, 1988.
  - Next class
    - "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", Hwu et al., Journal of Supercomputing, 1993

From Last Time: Dominator (DOM)

- Defn: 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, which blocks are guaranteed to have executed prior to executing the BB

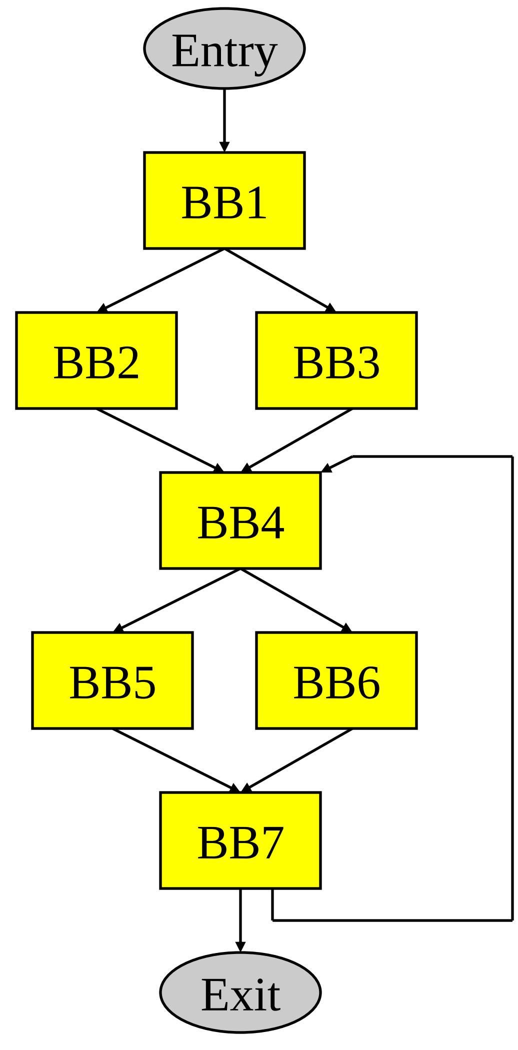
Dominator Analysis

- Compute dom(BBi) = set of BBs that dominate BBi
- Initialization
  - Dom(entry) = entry
  - Dom(everything else) = all nodes
- Iterative computation
  - 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

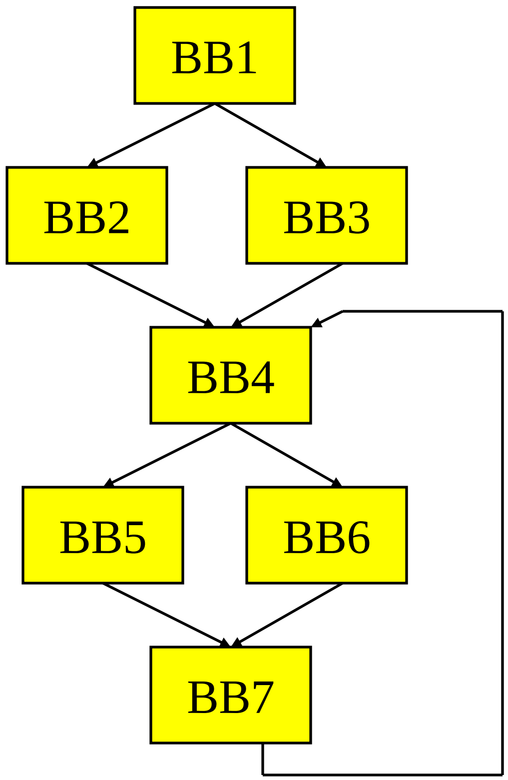
- Defn: Immediate dominator (idom) – Each node n has a unique immediate dominator m that is the last dominator of n on any path from the initial node to n
  - Closest node that dominates

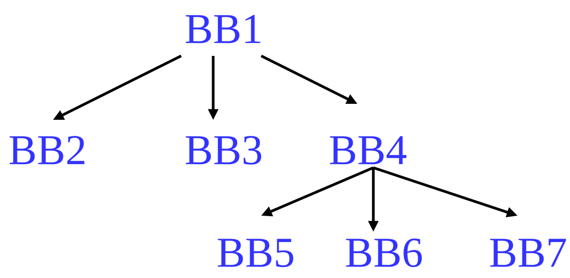


Dominator Tree

First BB is the root node, each node dominates all of its descendants

BB	DOM	BB	DOM
1	1	5	1,4,5
2	1,2	6	1,4,6
3	1,3	7	1,4,7
4	1,4		

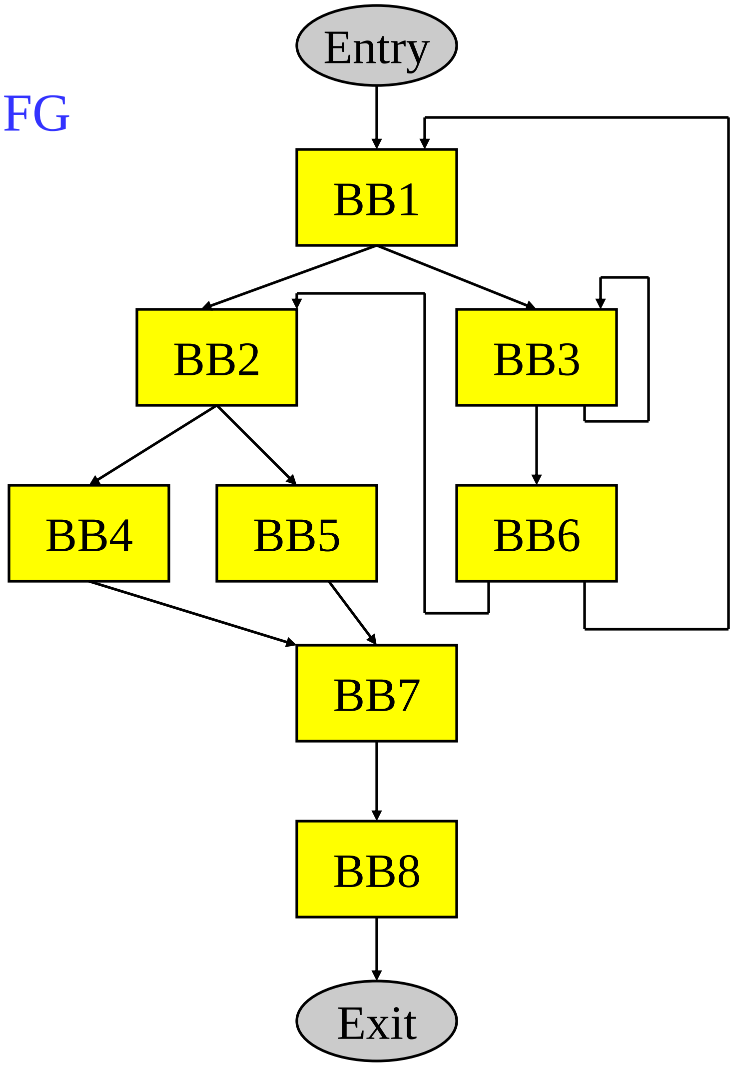




Dom tree

Class Problem

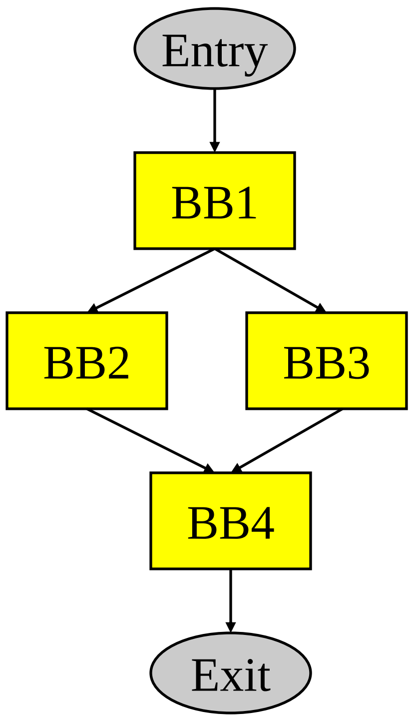
Draw the dominator
tree for the following CFG

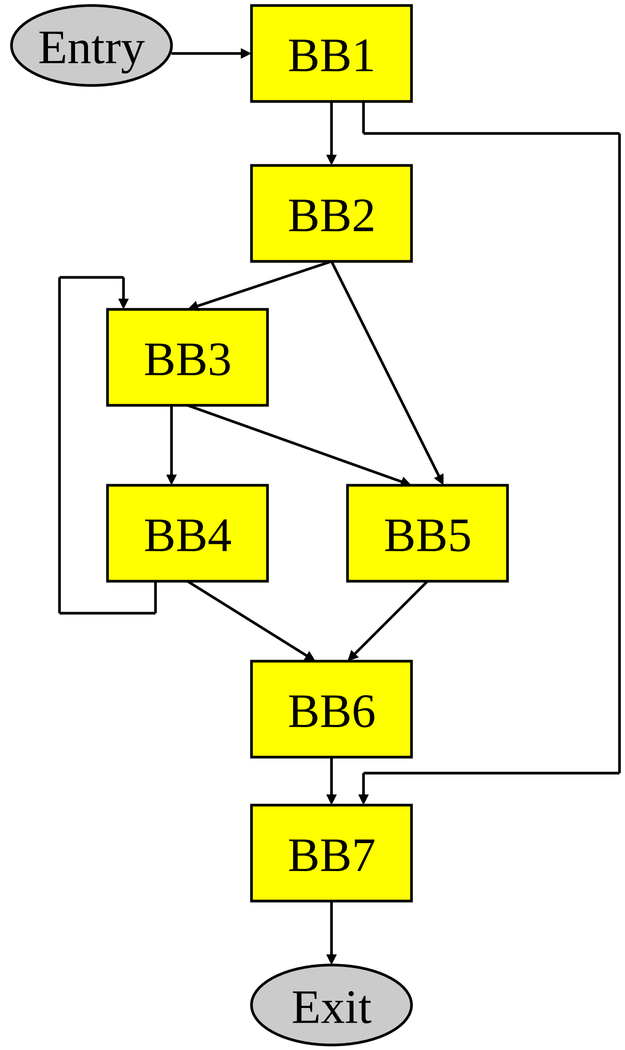


Post Dominator (PDOM)

- Reverse of dominator
- <u>Defn: Post Dominator</u> – 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
- Intuition
  - Given some BB, which blocks are guaranteed to have executed after executing the BB
- pdom(BBi) = set of BBs that post dominate BBi
- Initialization
  - Pdom(exit) = exit
  - Pdom(everything else) = all nodes
- Iterative computation
  - 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

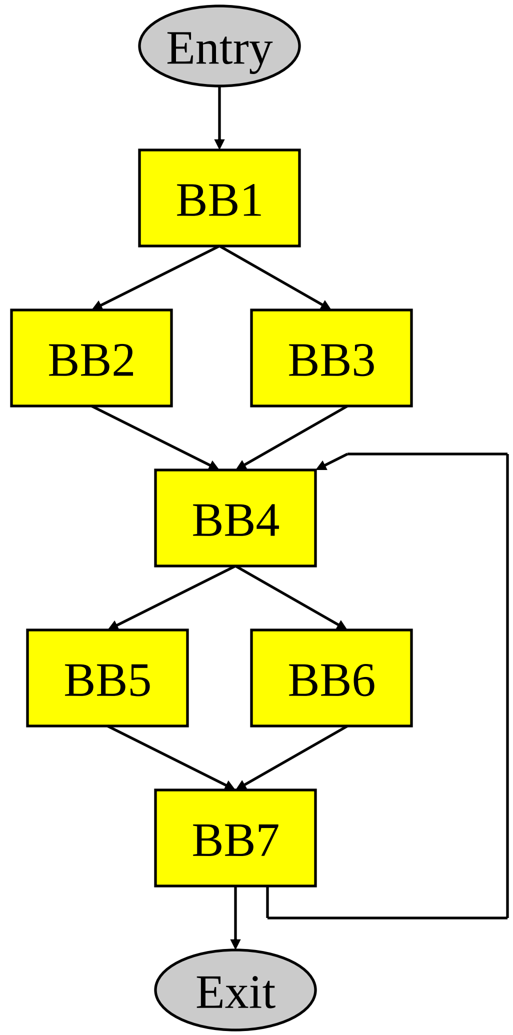
Post Dominator Examples





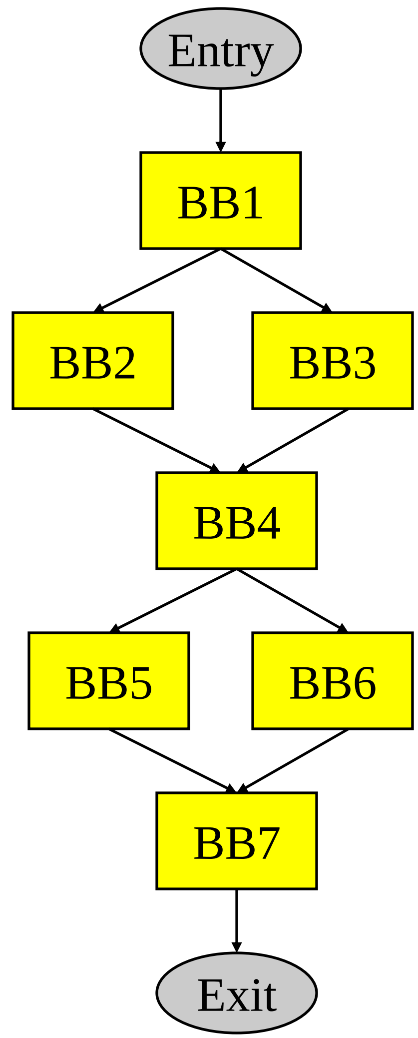
Immediate Post Dominator

- Defn: 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



Why Do We Care About Dominators?

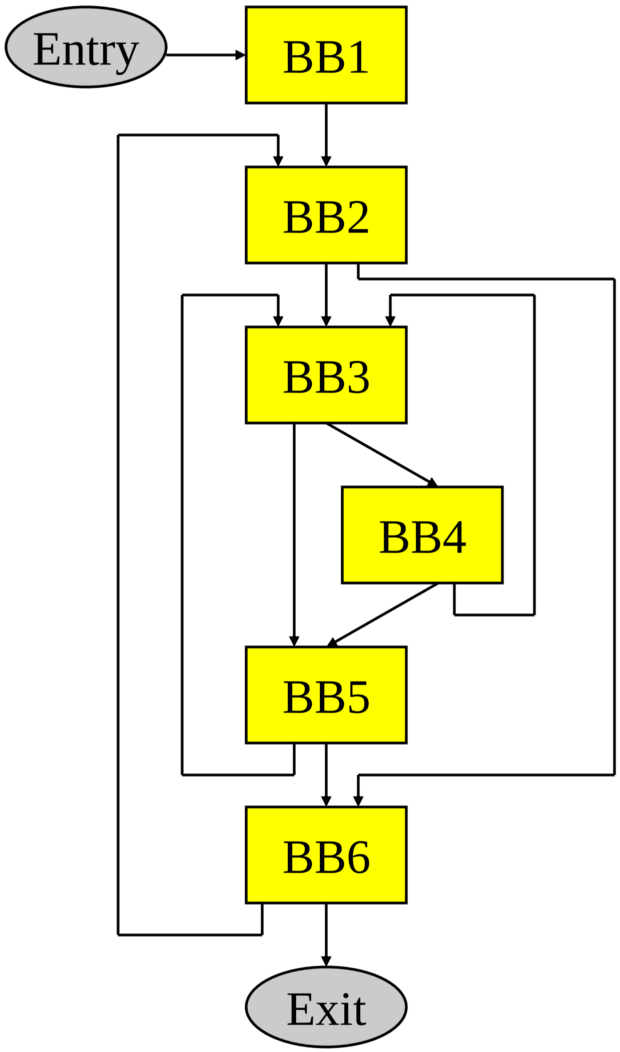
- Loop detection – next subject
- Dominator
  - Guaranteed to execute before
  - Redundant computation – an op is redundant if it is computed in a dominating BB
  - Most global optimizations use dominance info
- Post dominator
  - Guaranteed to execute after
  - Make a guess (ie 2 pointers do not point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



Natural Loops

- Cycle suitable for optimization
  - Discuss optimizations 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)

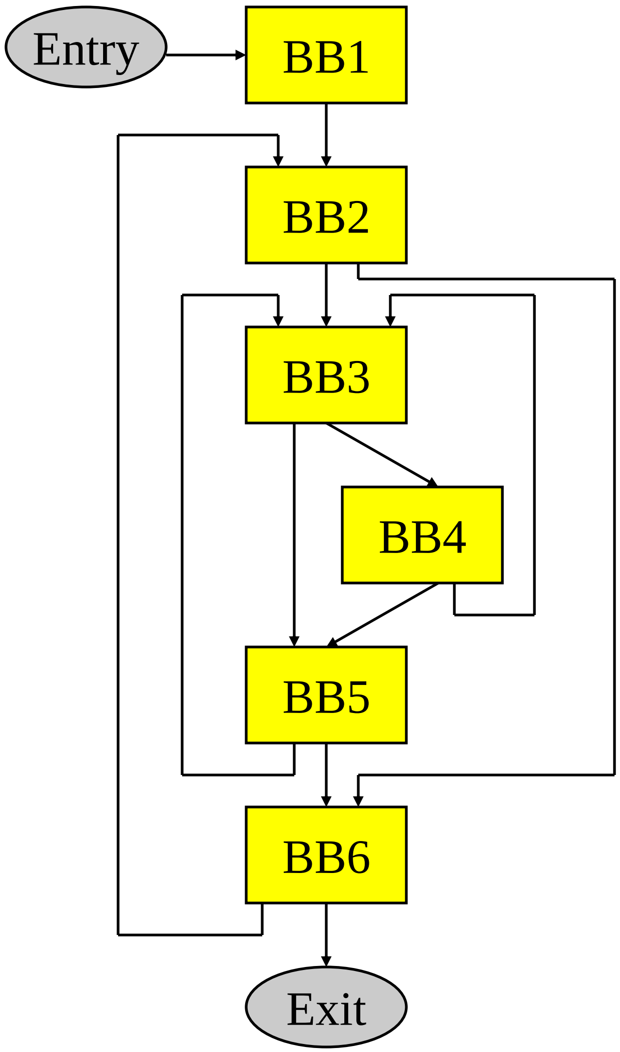
Backedge Example



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
- Merge loops with the same header
  - » I.e., a loop with 2 continues
  - » LoopBackedge = LoopBackedge1 + LoopBackedge2
  - » LoopBB = LoopBB1 + LoopBB2
- Important property
  - » Header dominates all LoopBB

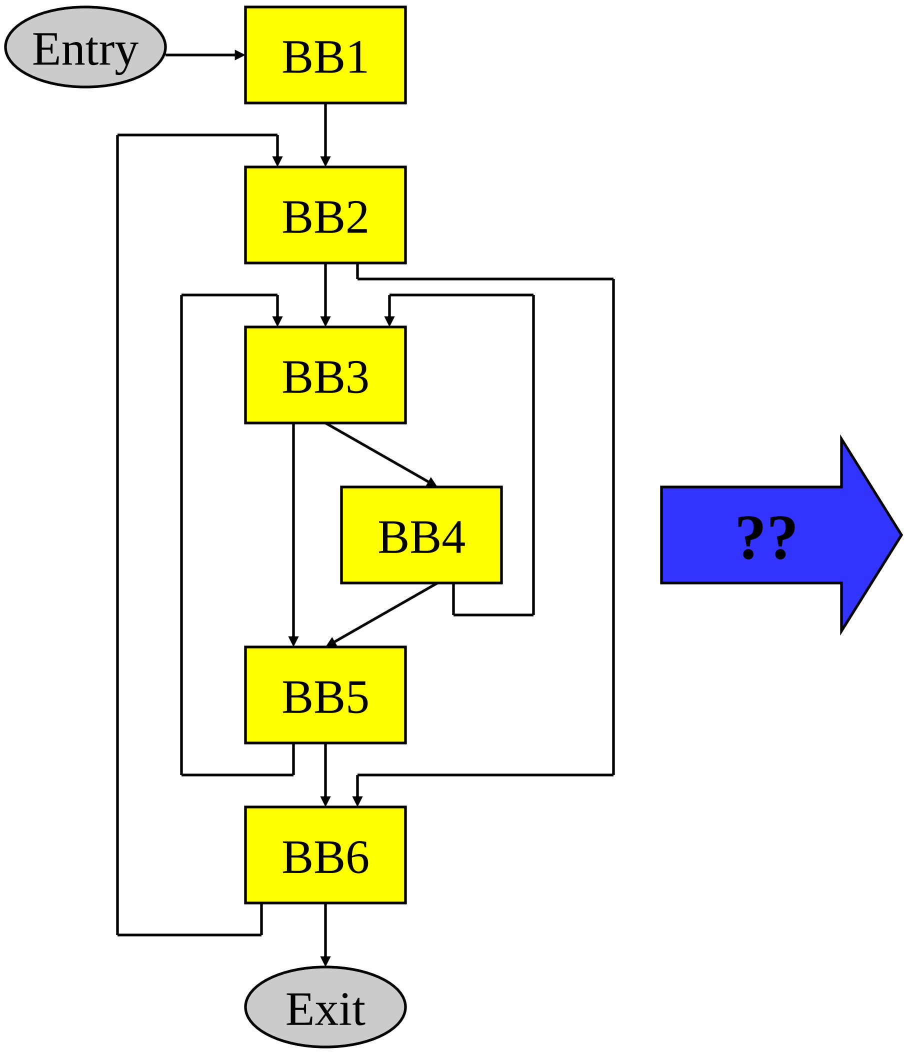
Loop Detection Example



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

Find the Preheaders for each Loop

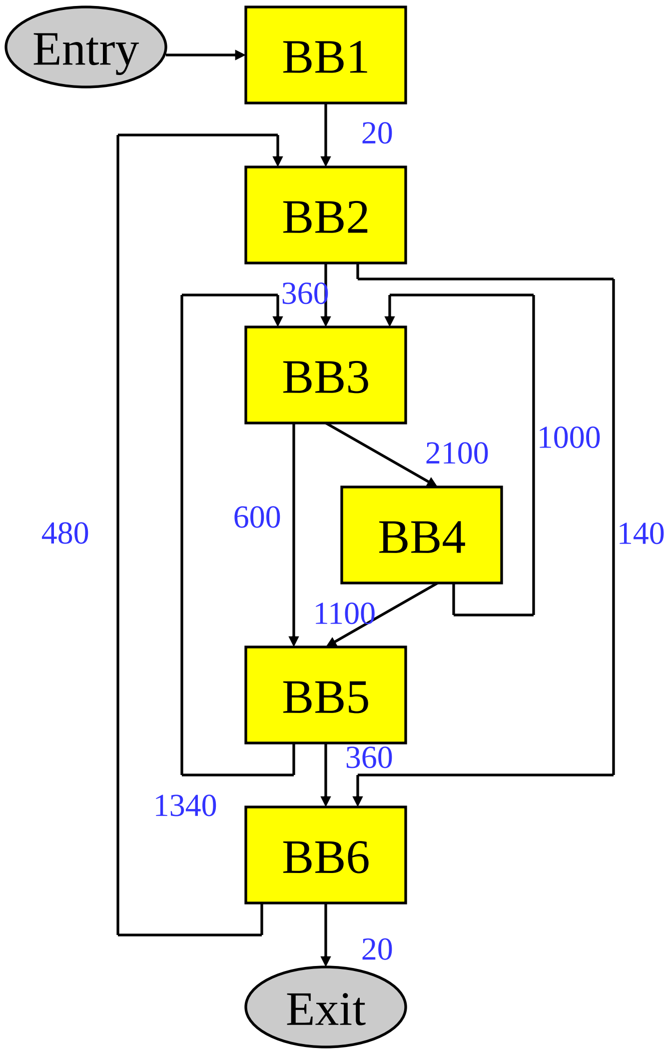


Characteristics of a Loop

- 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
- Trip count (average trip count)
  - How many times (on average) does the loop iterate
  - for (I=0; I<100; I++) → trip count = 100
  - With profile info:
    - Ave trip count = weight(header) / weight(preheader)

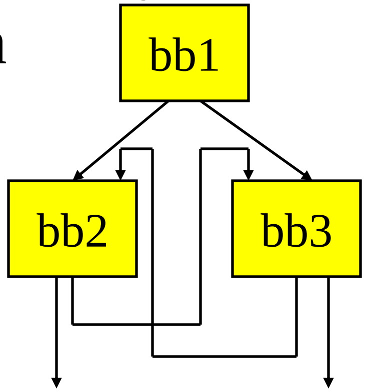
Trip Count Calculation Example

Calculate the trip counts for all the loops in the graph



Reducible Flow Graphs

- A flow graph is reducible if and only if we can partition the edges into 2 disjoint groups often called forward and back edges with the following properties
  - » The forward edges form an acyclic graph in which every node can be reached from the Entry
  - » The back edges consist only of edges whose destinations dominate their sources
- More simply – Take a CFG, remove all the backedges ($x \rightarrow y$ where y dominates x), you should have a connected, acyclic graph



Non-reducible!

Regions

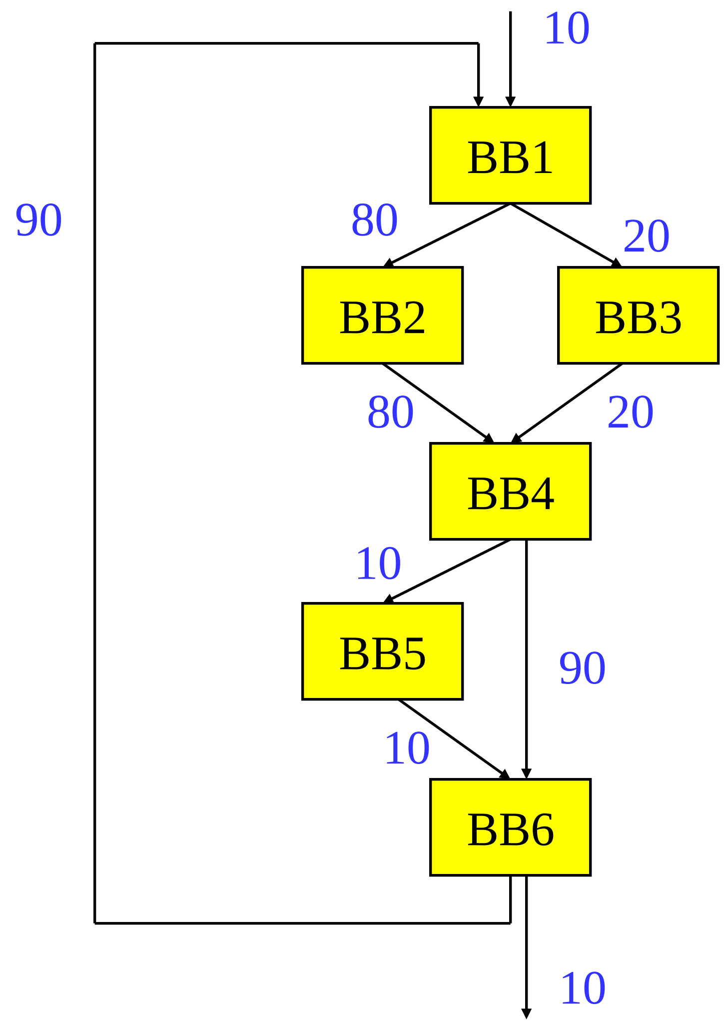
- Region: A collection of operations that are treated as a single unit by the compiler
  - Examples
    - Basic block
    - Procedure
    - Body of a loop
  - Properties
    - Connected subgraph of operations
    - Control flow is the key parameter that defines regions
    - Hierarchically organized
- Problem
  - Basic blocks are too small (3-5 operations)
    - Hard to extract sufficient parallelism
  - Procedure control flow too complex for many compiler xforms
    - Plus only parts of a procedure are important (90/10 rule)

Regions (2)

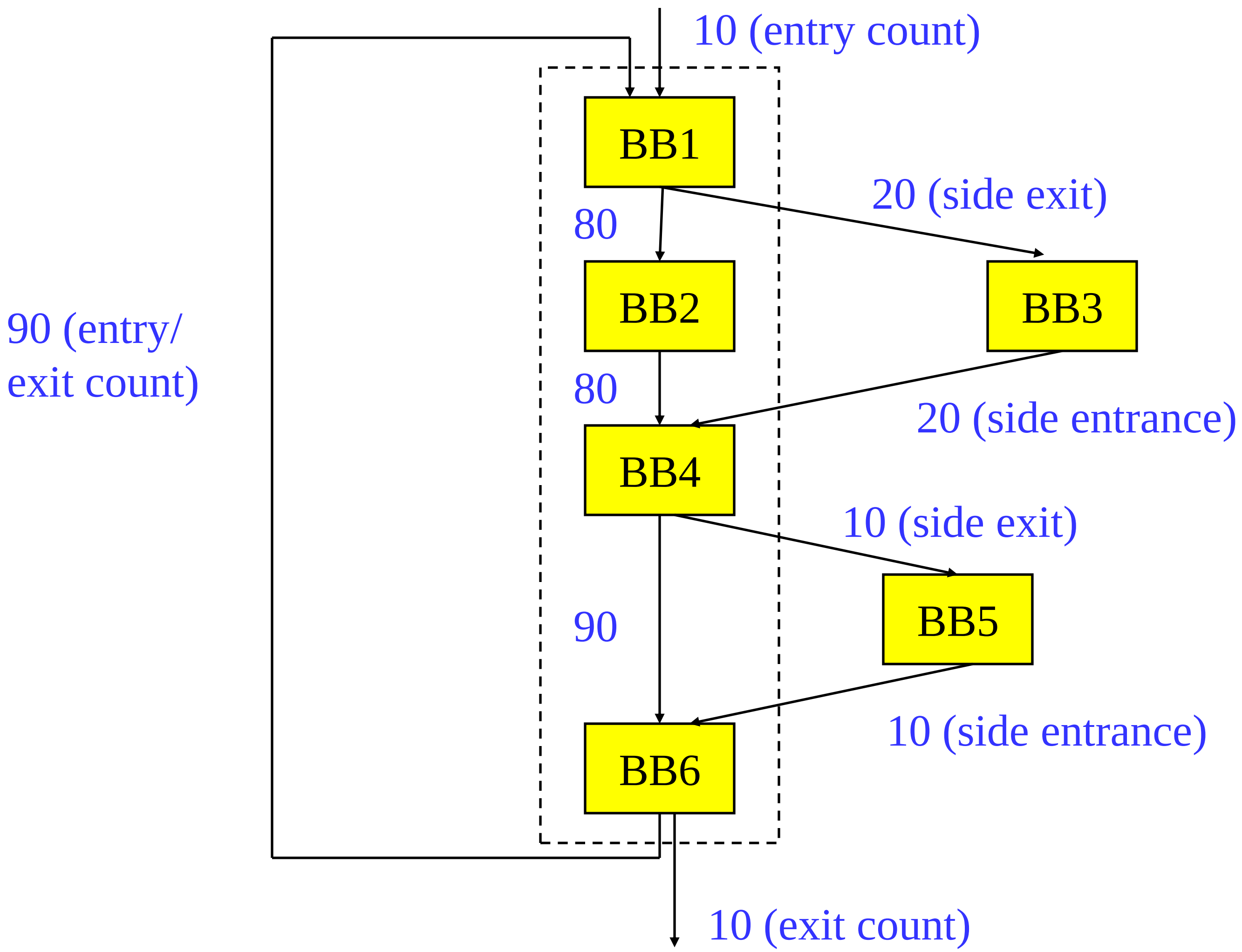
- Want
  - Intermediate sized regions with simple control flow
  - Bigger basic blocks would be ideal !!
  - Separate important code from less important
  - Optimize frequently executed code at the expense of the rest
- Solution
  - Define new region types that consist of multiple BBs
  - Profile information used in the identification
  - Sequential control flow (sorta)
  - Pretend the regions are basic blocks

Region Type 1 - Trace

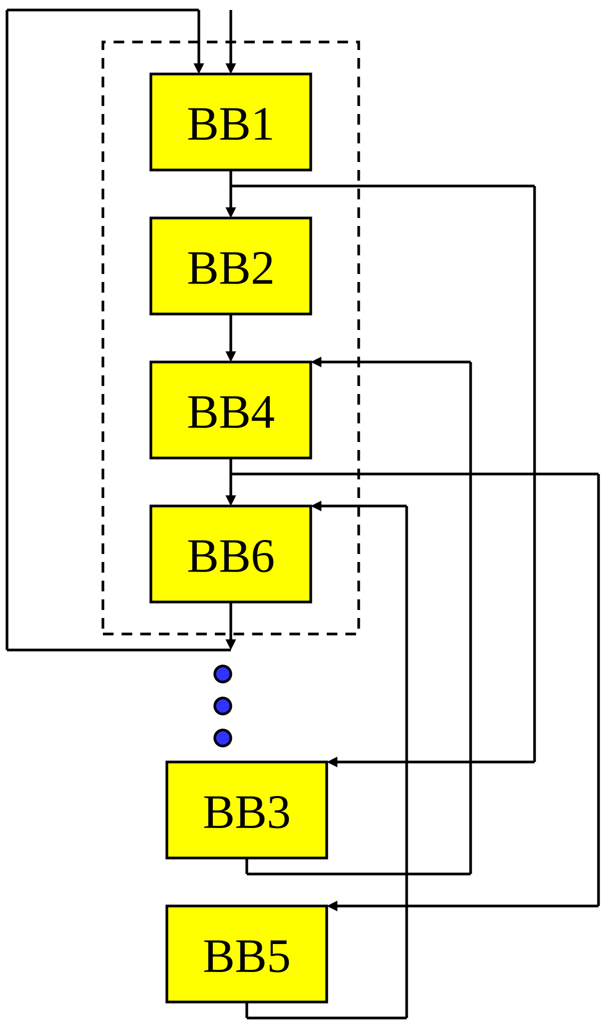
- Trace - Linear collection of basic blocks that tend to execute in sequence
  - “Likely control flow path”
  - Acyclic (outer backedge ok)
- Side entrance – branch into the middle of a trace
- Side exit – branch out of the middle of a trace
- Compilation strategy
  - Compile assuming path occurs 100% of the time
  - Patch up side entrances and exits afterwards
- Motivated by scheduling (i.e., trace scheduling)



Linearizing a Trace

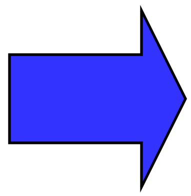


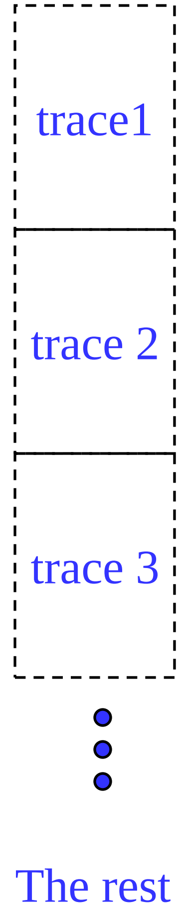
Intelligent Trace Layout for Icache Performance



Trace view

Intraprocedural code placement
Procedure positioning
Procedure splitting

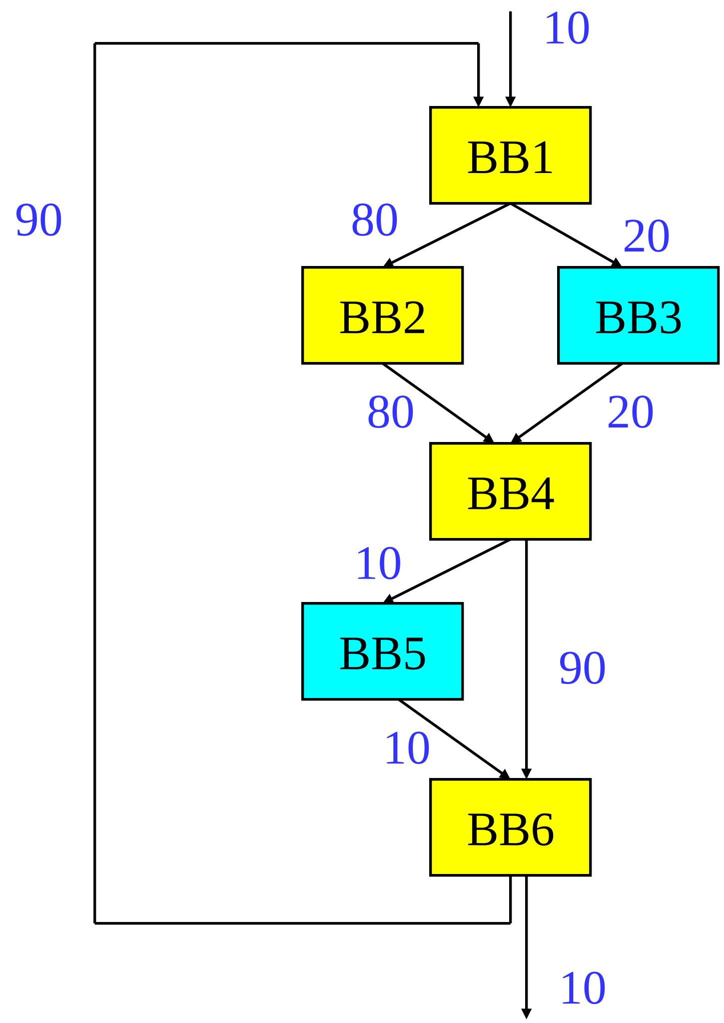




Procedure view

Issues With Selecting Traces

- Acyclic
  - Cannot go past a backedge
- Trace length
  - Longer = better ?
  - Not always !
- On-trace / off-trace transitions
  - Maximize on-trace
  - Minimize off-trace
  - Compile assuming on-trace is 100% (ie single BB)
  - Penalty for off-trace
- Tradeoff (heuristic)
  - Length
  - Likelihood remain within the trace



Trace Selection Algorithm

```
i = 0;
mark all BBs unvisited
while (there are unvisited nodes) do
    seed = unvisited BB with largest execution freq
    trace[i] += seed
    mark seed visited
    current = seed
    /* Grow trace forward */
    while (1) do
        next = best_successor_of(current)
        if (next == 0) then break
        trace[i] += next
        mark next visited
        current = next
    endwhile
    /* Grow trace backward analogously */
    i++
endwhile
```

Best Successor/Predecessor

- Node weight vs edge weight
  - edge more accurate
- THRESHOLD
  - controls off-trace probability
  - 60-70% found best
- Notes on this algorithm
  - BB only allowed in 1 trace
  - Cumulative probability ignored
  - Min weight for seed to be chose (ie executed 100 times)

```
best_successor_of(BB)
   e = control flow edge with highest
        probability leaving BB
   if (e is a backedge) then
      return 0
   endif
   if (probability(e) <= THRESHOLD) then
      return 0
   endif
   d = destination of e
   if (d is visited) then
      return 0
   endif
   return d
end procedure
```

Class Problems

Find the traces. Assume a threshold probability of 60%.