Control Flow Optimizations

Compilers for High Performance Architectures

Class Outline

- Control flow analysis
- Program profiling
- Control-flow optimizations
  - Branch to unconditional branch
  - Unconditional branch to branch
  - Branch to next basic block
  - Basic block merging
  - Branch to same target
  - Branch target expansion
  - Unreachable code elimination
  - Code hoisting
  - Partial predication
  - Full predication

Control Flow

- Control flow is only know at run-time
  - Branch speculation
  - Wrong path squash
- Compiler only knows the static control flow
  - The actual values are unknown
    - The actual control flow is unknown
  - Control flow analysis
    - Determine properties of the program control flow

Basic Block (BB)

- 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

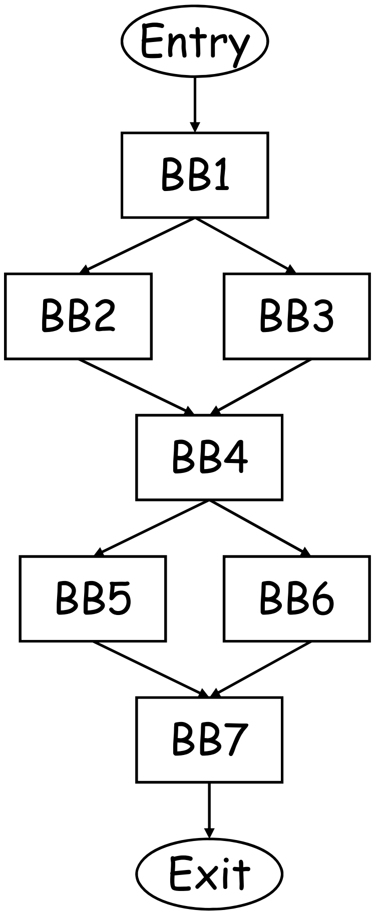
- Finding BB's
  - The first operation starts a BB
  - Any operation that is the target of a branch starts a BB
  - Any operation that immediately follows a branch starts a BB

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)

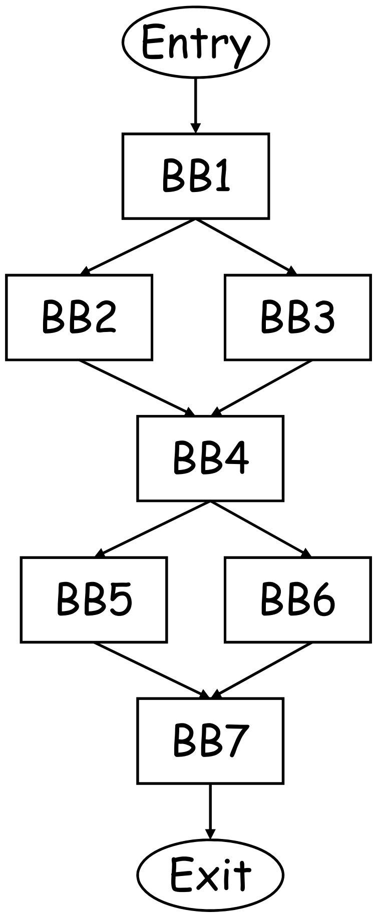
- Directed graph, G=(V,E) where each vertex V is a basic block, and there is an edge E=(v1,v2) if BB v1 can branch to BB v2
  - Standard representation used by many compilers
  - Often have 2 pseudo V's
    - entry node
    - exit node

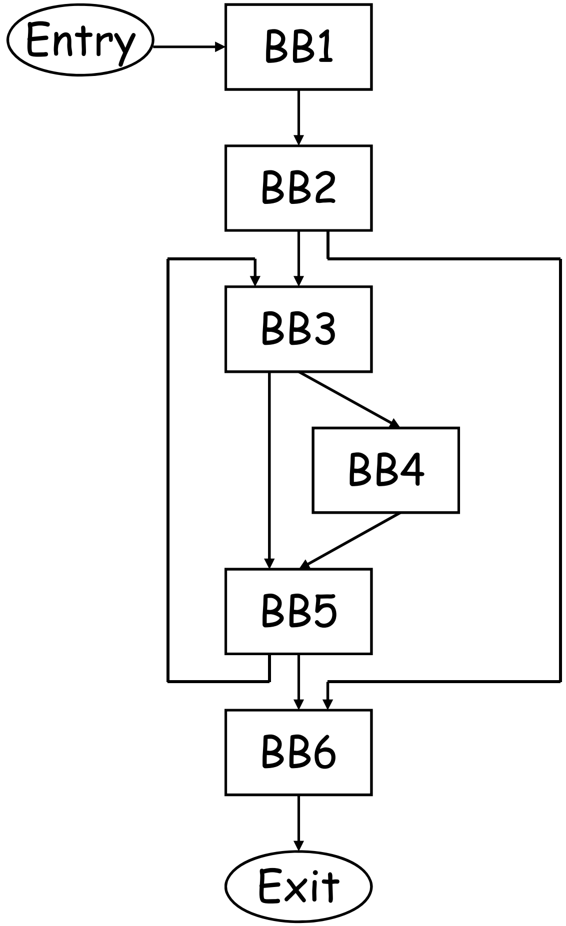


Dominator

- Dominator – Given a CFG, 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

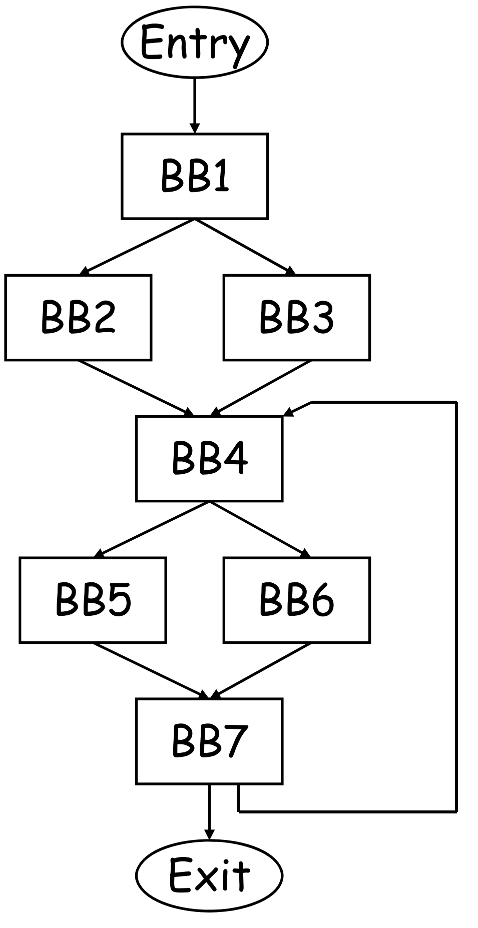
Dominator Examples





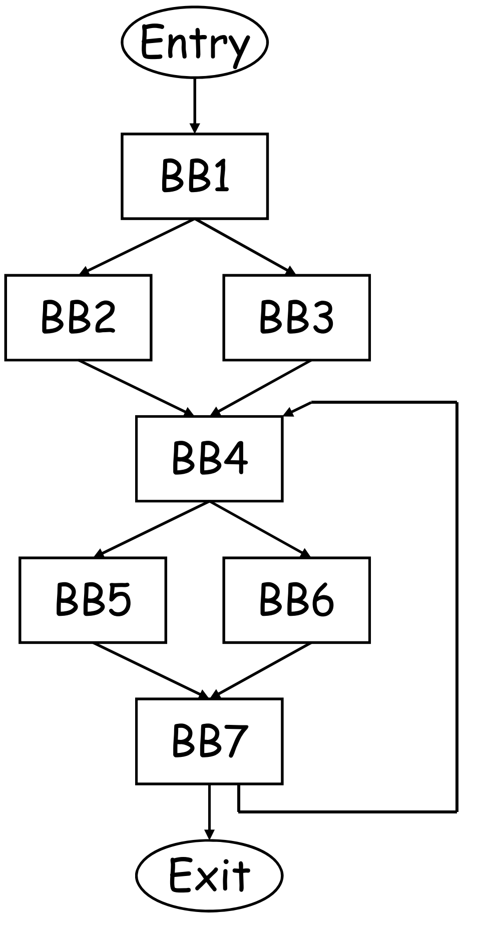
Dominator Analysis

- 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



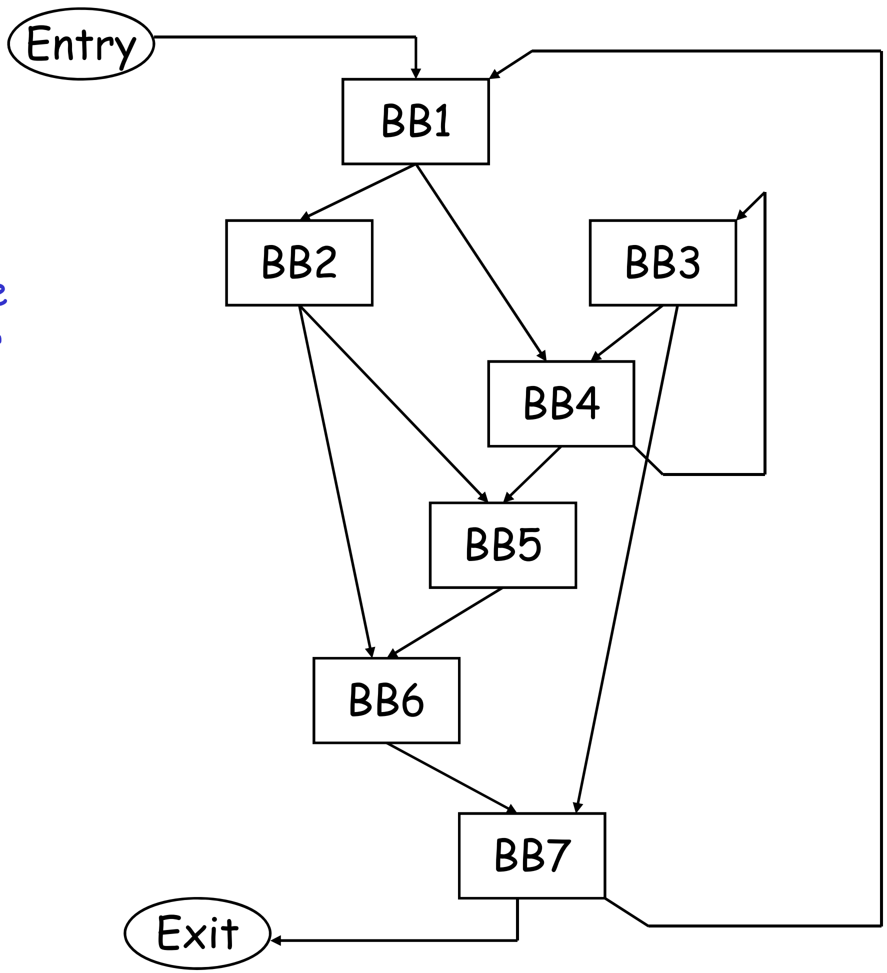
Immediate Dominator

- 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

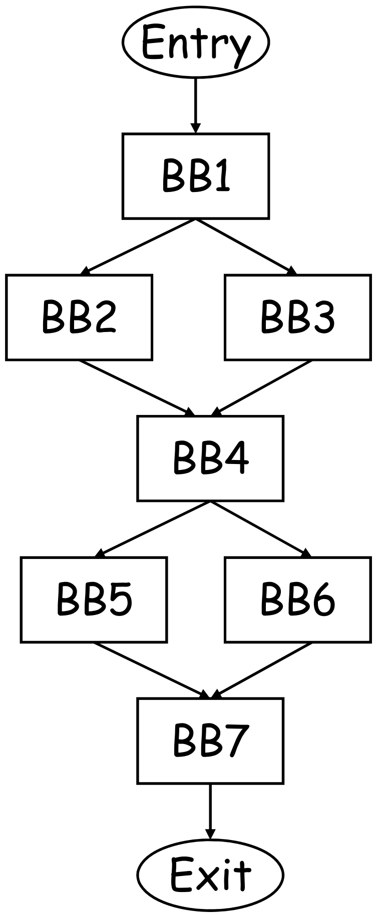
Calculate the DOM set for each BB

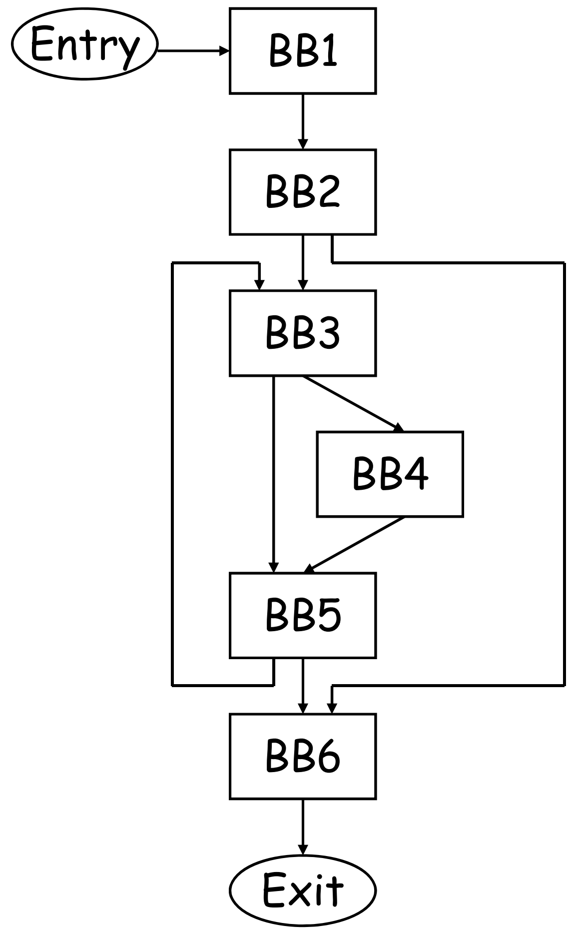


Post Dominator

- Reverse of 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
- Intuition
  - Given some BB, which blocks are guaranteed to have executed after executing the BB

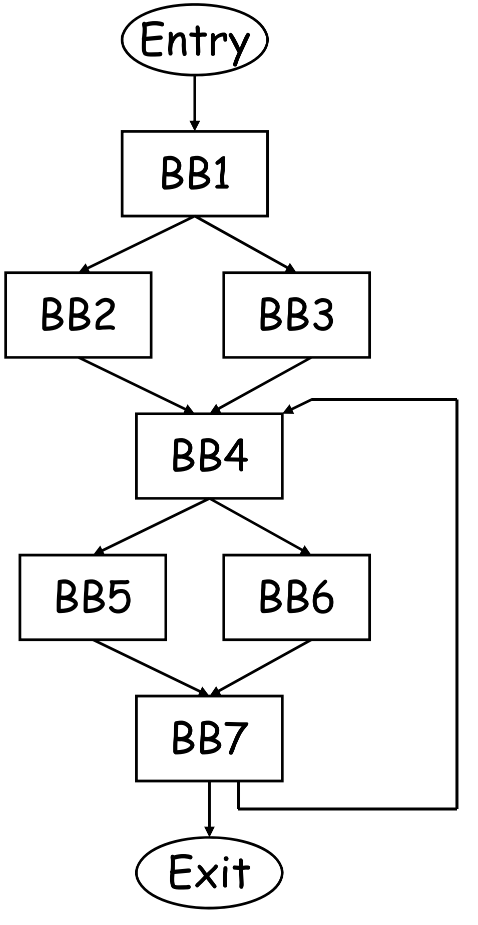
Post Dominator Examples





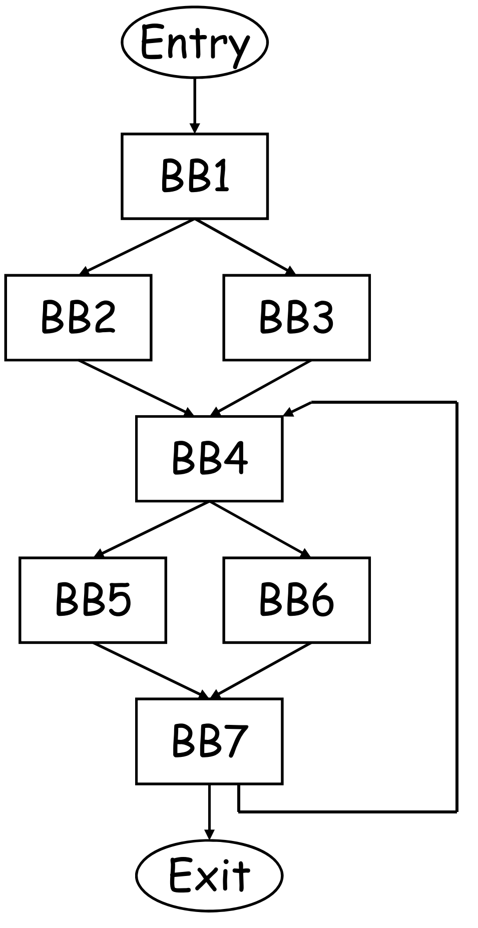
Post Dominator Analysis

- Compute 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



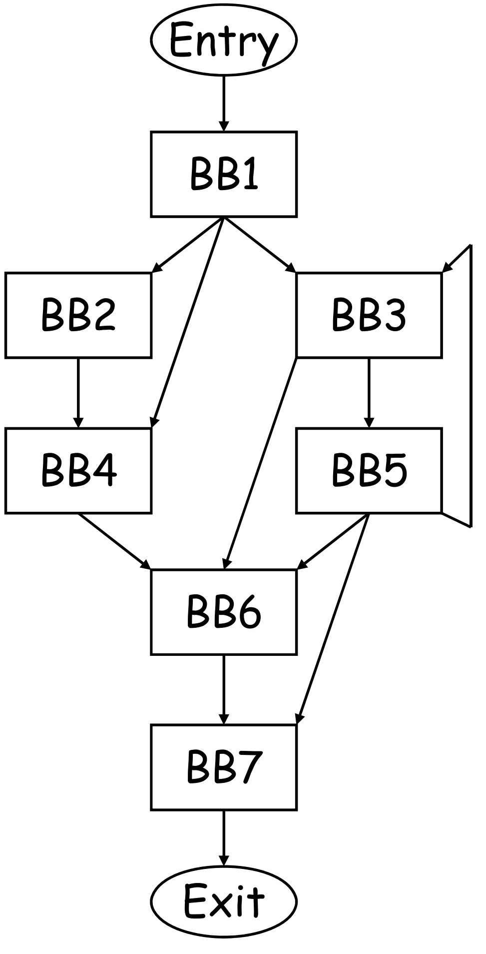
Immediate Post Dominator

- 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



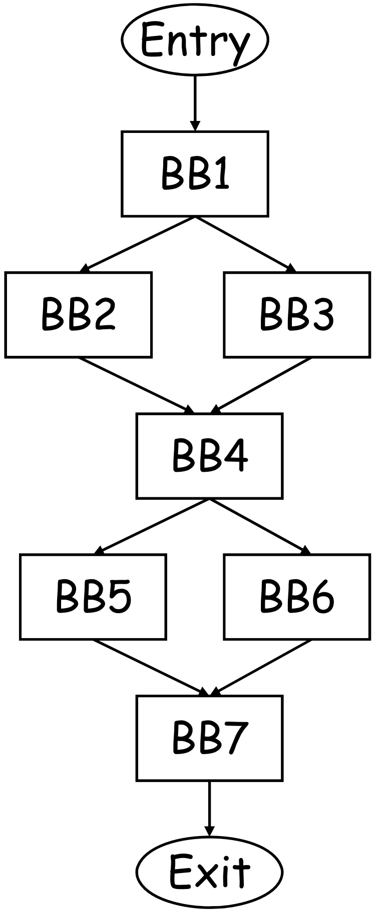
Class Problem 2

Calculate the PDOM set for each BB



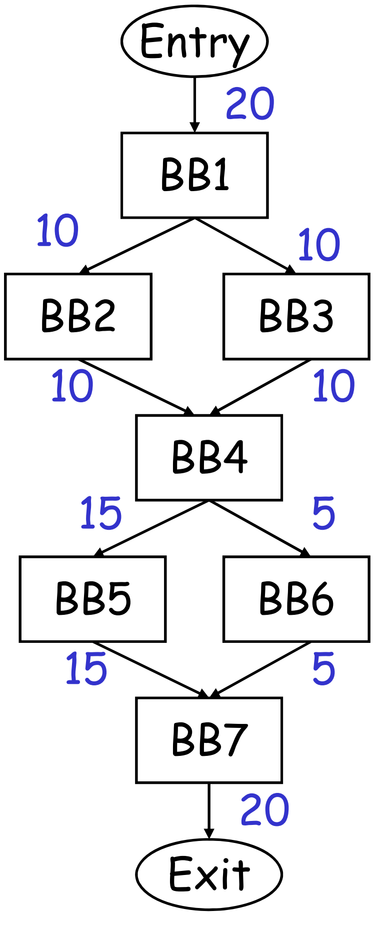
Why Do We Care About Dominators?

- 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



Weighted CFG

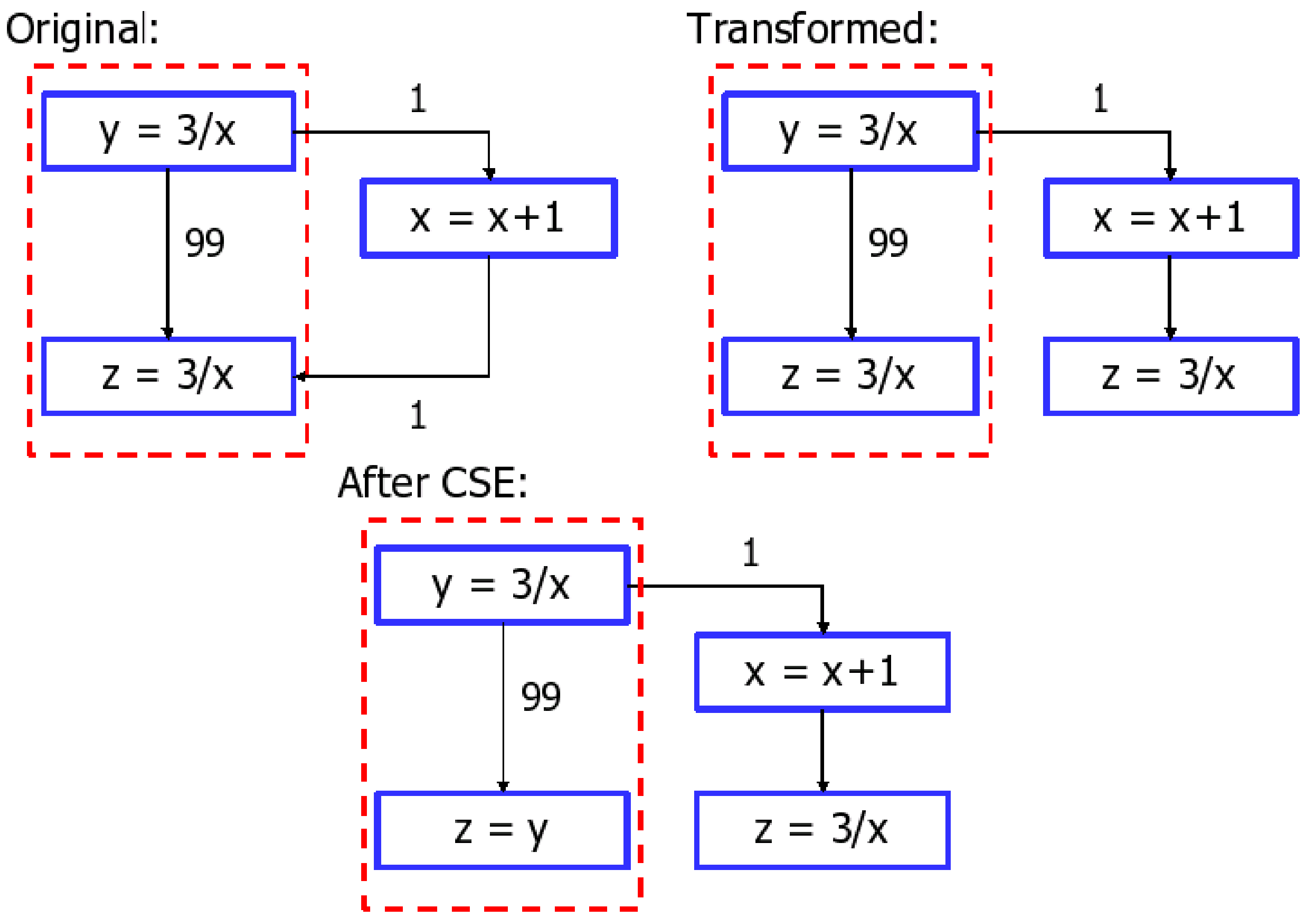
- Add profiling information to the CFG
  - Basic block execution counts
  - Edge execution counts



Program Profiling

- Many optimizations require run-time knowledge
  - Help classic optimizations
  - Apply optimizations to the common case
  - Make the right choice
    - Code predication
    - Data speculation
- Goals
  - High accuracy of data
    - Profile using a representative input
  - Low overhead
    - Do not change the actual runtime behavior: Race conditions, Spin-locks

Profiling usage example



Basic block profiling

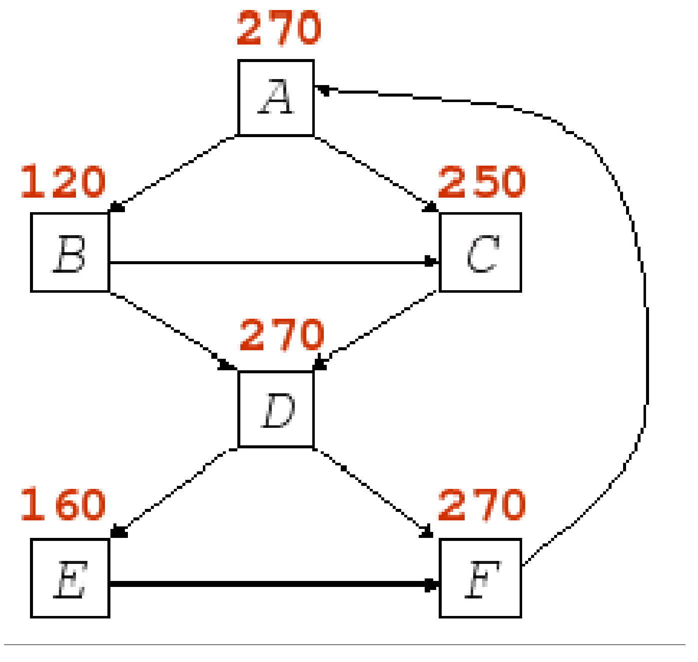
- Code instrumentation
  - Add code to the beginning of each basic block
    - Counts the number of times the BB executed
  - Add code to the exit point of the program to dump data
- Assign an identifier to each basic block
- Allocate a counters in memory
  - Consecutive array
    - Assign blocks a unique index
  - Group counters to optimize data cache misses
    - Assign blocks a unique address

Edge profiling

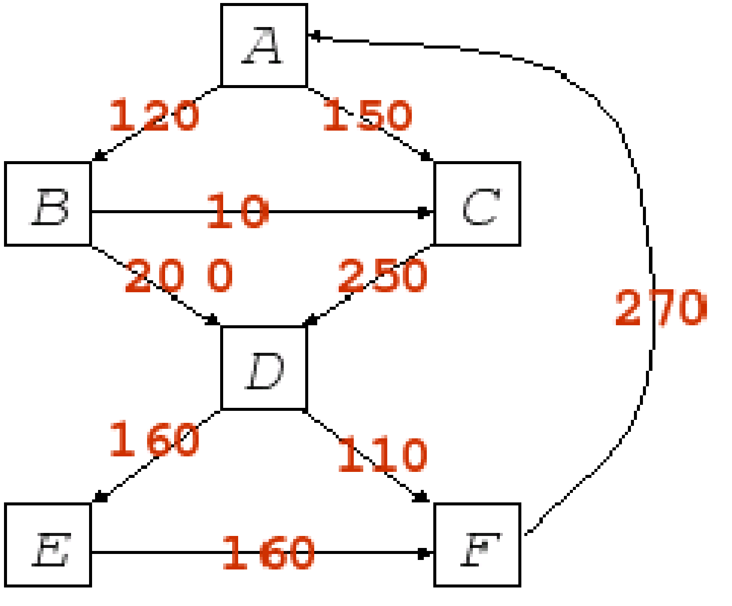
- Code instrumentation
  - Add code to the beginning of each basic block
    - Buid the edge identifier using previous and current BB identifier
    - Set current BB as ”previous BB”
  - Add code to the exit point of the program to dump data
- Assign an identifier to each basic block
- Allocate a counters in memory
  - Many more edges than blocks
    - Higher cache miss rate
    - Higher performance impact

BB and Edge profiling example

Basic bloc profiling



Edge profiling

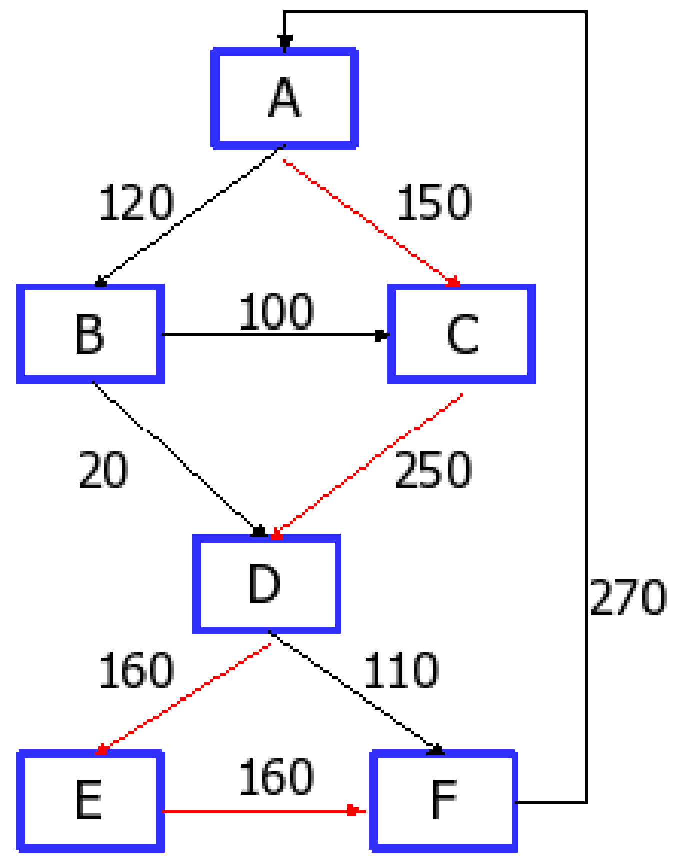


Path profiling

- Edge profiling not always correct
  - Common path should be ACDEF
  - But …

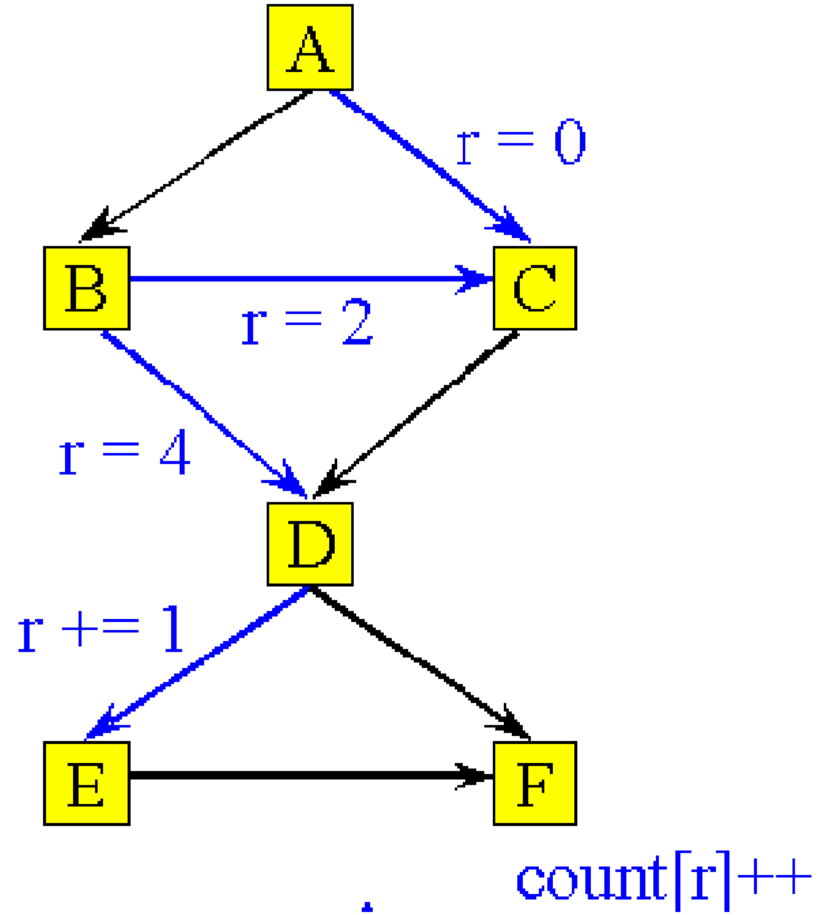
Path	prof1	prof2
ACDF	90	110
ACDEF	60	40
ABCDF	0	0
ABCDEF	100	100
ABDF	20	0
ABDEF	0	20

  - … also satify the edge profile



Efficient path profiling

- Trace the path through a region
  - Instrument all basic blocks
  - Maintain a list structure
    - Very expensive!
- Build a unique path ID as we progress
  - Instrument only selected edges
  - No need for a list structure
    - Still near 40% overhead vs. 15% for Edge profiling
- Statistic profiling
  - Turn profiling on and off to obtain data at random intervals
    - Aggregate data is still representative
    - Profiling overhead is reduced



Control Flow Optimizations for Acyclic Code

- Generally quite simplistic
- Goals
  - Reduce the number of dynamic branches
  - Make larger basic blocks
  - Reduce code size
- Classic control flow optimizations
  - Branch to unconditional branch
  - Unconditional branch to branch
  - Branch to next basic block
  - Basic block merging
  - Branch to same target
  - Branch target expansion
  - Unreachable code elimination

Control Flow Optimizations (1)

1. Branch to unconditional branch

L1: if (a < b) goto L2
. . .
L2: goto L3

⇒

L1: if (a < b) goto L3
. . .
L2: goto L3 → may be deleted

2. Unconditional branch to branch

L1: goto L2
. . .
L2: if (a < b) goto L3
L4:

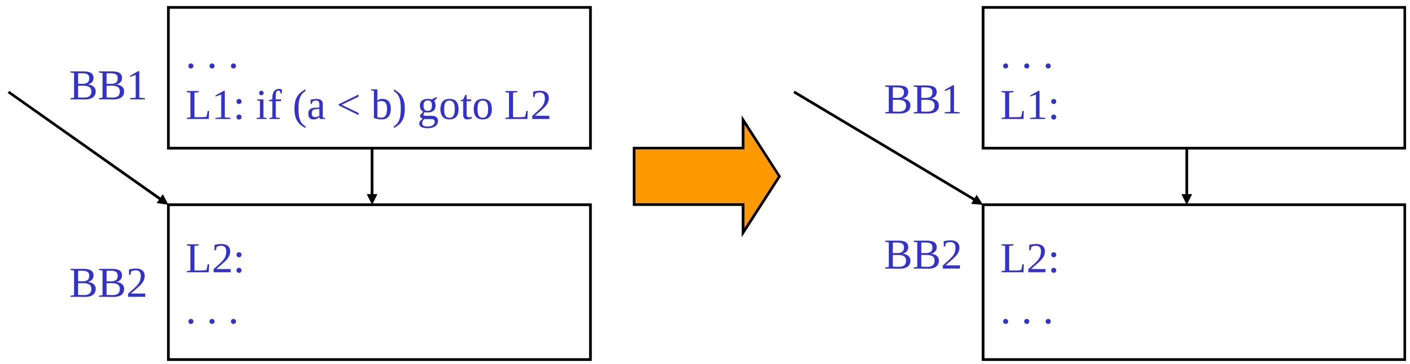
⇒

L1: if (a < b) goto L3
goto L4:
. . .
L2: if (a < b) goto L3 → may be deleted
L4:

Control Flow Optimizations (2)

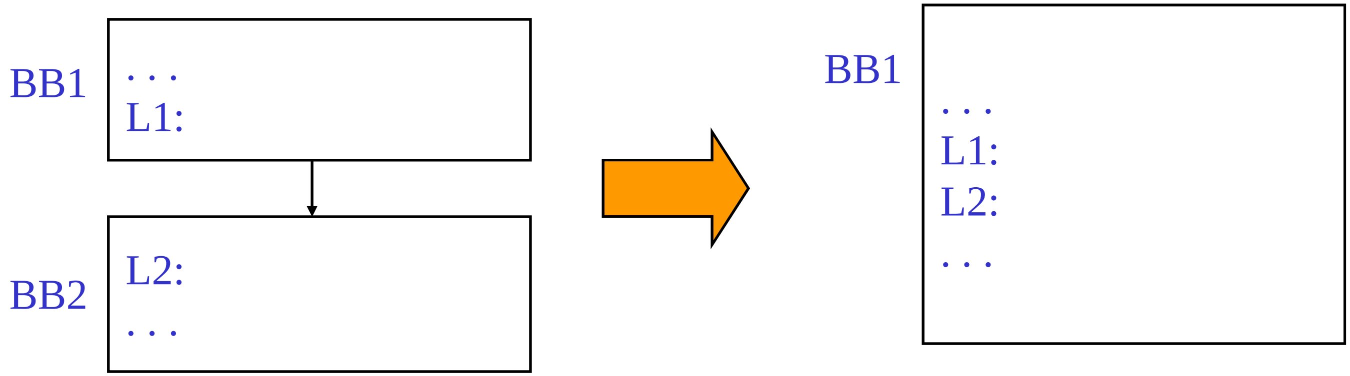
3. Branch to next basic block

Branch is unnecessary



4. Basic block merging

Merge BBs when single edge between



Control Flow Optimizations (3)

5. Branch to same target

```
. . .
L1: if (a < b) goto L2
goto L2
```

⇒

```
. . .
L1: goto L2
```

6. Branch target expansion

```
BB1   stuff1
      L1: goto L2

      • • •

BB2   L2: stuff2
      . . .
```

⇒

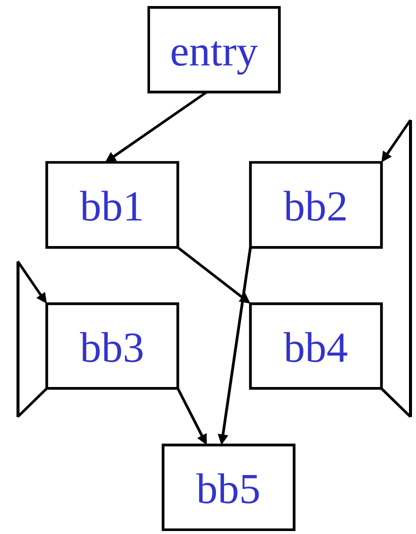
```
BB1   stuff1
      L1: stuff2
      . . .

      • • •

BB2   L2: stuff2
      . . .
```

What about expanding a conditional branch?

Unreachable Code Elimination

Algorithm

```
Mark procedure entry BB visited
to_visit = procedure entry BB
while (to_visit not empty) {
    current = to_visit.pop()
    for (each successor block of current) {
            Mark successor as visited;
            to_visit += successor
    }
}
Eliminate all unvisited blocks
```



Which BB(s) can be deleted?

Class Problem 3

Maximally optimize the control flow of this code

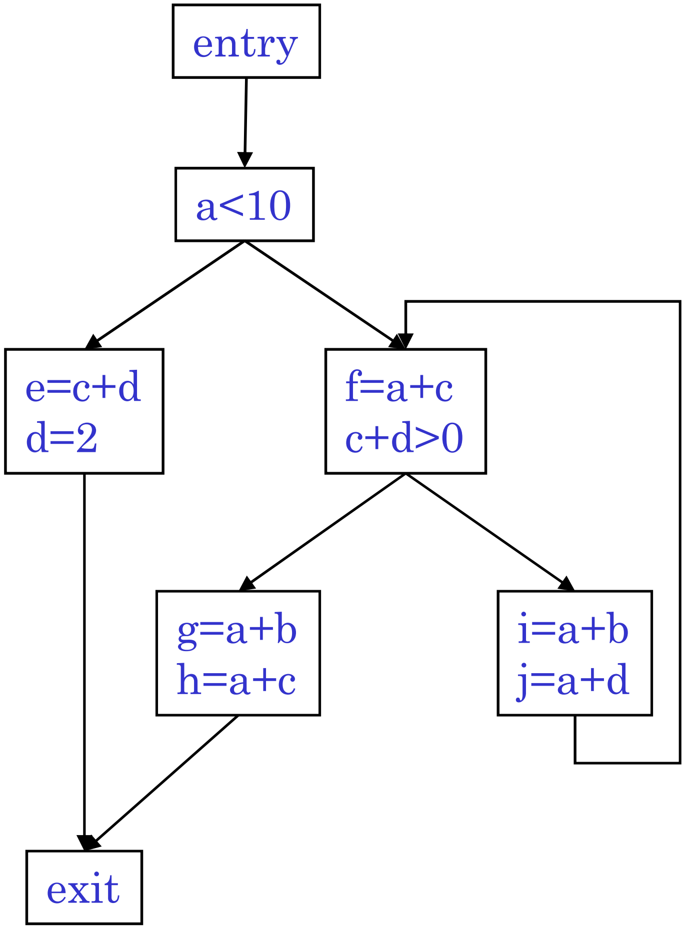
```
L1: if (a < b) goto L11
-----------------------
L2: goto L7
-----------------------
L3: goto L4
-----------------------
L4: stuff4
L5: if (c < d) goto L15
-----------------------
L6: goto L2
-----------------------
L7: if (c < d) goto L13
-----------------------
L8: goto L12
-----------------------
L9: stuff 9
L10: if (a < c) goto L3
-----------------------
L11:goto L9
-----------------------
L12: goto L2
-----------------------
L13: stuff 13
L14: if (e < f) goto L11
-----------------------
L15: stuff 15
L16: rts
```

Code hoisting

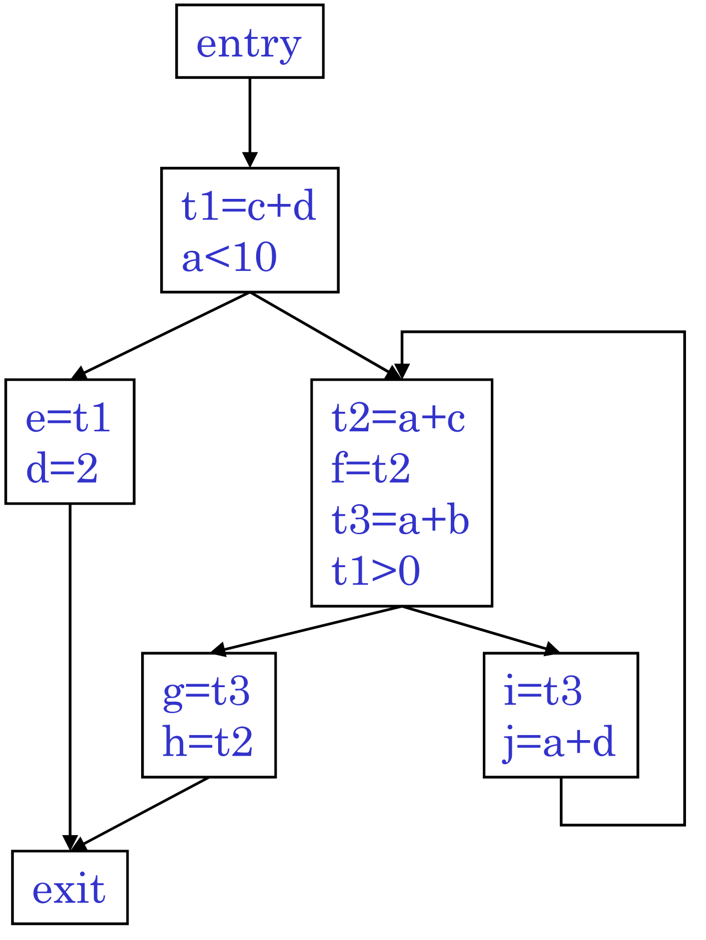
- Detect expressions which are always evaluated after a point in program execution
  - Regardless of control flow
- Move computation to the earliest point in the code where it can be executed
  - Add move / rename operations as needed
- In addition we can perform speculative code hoisting
  - Speculative loads
  - Speculative computation
  - speculative stores (dangerous)

Code hoisting example

- Before



- After



(partial) Predicated execution

- Conditional moves
  - Replace conditional code for code with no branches
  - Not all conditional codes can be converted
    - Limited to arithmetic operations
    - Can not store results to memory

Branching code:

```
        if (a > b) goto L1
        max = b
        goto L2
L1:     max = a
L2:     ...
```

Partially predicated code:

```
t1 = a > b
max = b
max = a    if t1
```

Predicated Execution

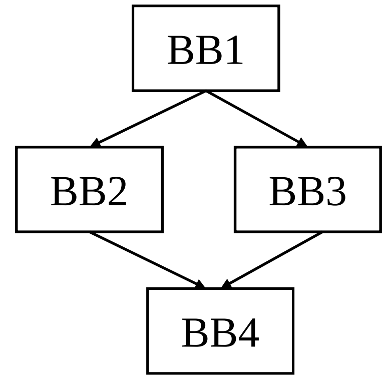
- Hardware mechanism that allows operations to be conditionally executed
- Add an additional boolean source operand (predicate)
  - ADD r1, r2, r3 if p1
    - if (p1 is True), r1 = r2 + r3
    - else if (p1 is False), do nothing (Add treated like a NOP)
    - p1 referred to as the <u>guarding predicate</u>
    - Predicated on True means always executed
    - Omitted predicated also means always executed
- Provides compiler with an alternative to using branches to selectively execute operations
  - If statements in the source
  - Realize with branches in the assembly code
  - Could also realize with conditional instructions
  - Or use a combination of both

Predicated Execution Example

Traditional branching code

```
a = b + c
if (a > 0)
    e = f + g
else
    e = f / g
h = i - j
```

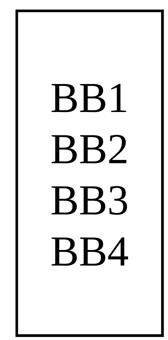
BB1		add a, b, c
BB1		bgt a, 0, L1
BB3		div e, f, g
BB3		jump L2
BB2	L1:	add e, f, g
BB4	L2:	sub h, i, j



Predicated code

p2 → BB2
p3 → BB3

BB1	add a, b, c if T
BB1	p2 = a > 0 if T
BB1	p3 = a <= 0 if T
BB3	div e, f, g if p3
BB2	add e, f, g if p2
BB4	sub h, i, j if T

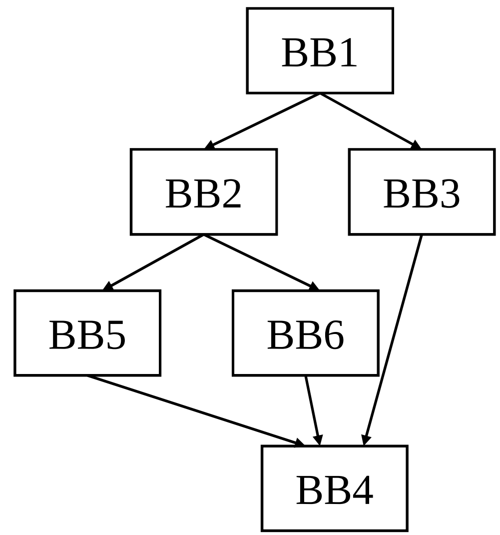


What about nested if-then-else's?

Nested if-then-else's

```
a = b + c
if (a > 0)
    if (a > 25)
        e = f + g
    else
        e = f * g
else
    e = f / g
h = i - j
```

BB1		add a, b, c
BB1		bgt a, 0, L1
BB3		div e, f, g
BB3		jump L2
BB2	L1:	bgt a, 25, L3
BB6		mpy e, f, g
BB6		jump L2
BB5	L3:	add e, f, g
BB4	L2:	sub h, i, j



Traditional branching code

Nested if-then-else's (2)

```
a = b + c
if (a > 0)
    if (a > 25)
        e = f + g
    else
        e = f * g
else
    e = f / g
h = i - j
```

BB1	add a, b, c if T
BB1	p2 = a > 0 if T
BB1	p3 = a <= 0 if T
BB3	div e, f, g if p3
BB2	p5 = a > 25 if p2
BB2	p6 = a <= 25 if p2
BB6	mpy e, f, g if p6
BB5	add e, f, g if p5
BB4	sub h, i, j if T

BB1
BB2
BB3
BB4
BB5
BB6

Predicated code

What do we assume to make this work ??

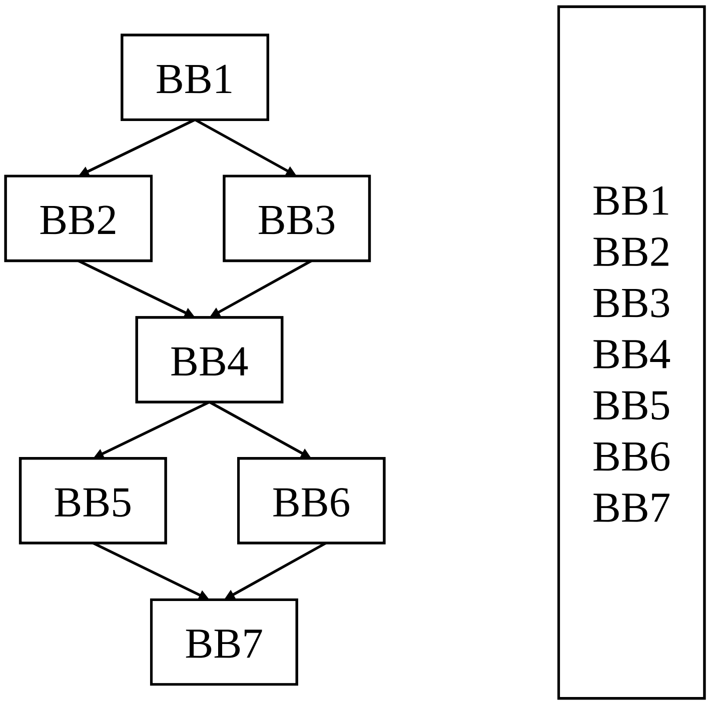
if p2 is False, both p5 and p6 are False

So, predicate setting instruction should set result to False if guarding predicate is false!!!

Benefits/Costs of Predicated Execution

- Benefits
  - Remove branches (both conditional and unconditional)
  - Remove branch mispredictions
  - Overlap execution of if-then-else statements
    - Branches tend to sequentialize operations
    - Predicates can be computed/used in parallel
- Costs
  - Useless instructions executed
  - Code size (extra operand, can’t fit into 32-bits)
  - Possibly longer schedule lengths
- The real story
  - Must be applied selectively or you get worse performance than not using it at all

Benefits/Costs of Predicated Execution (2)



Benefits:
- No branches, no mispredicts
- Can freely reorder independent operations in the predicated block
- Overlap BB2 with BB5 and BB6

Costs (execute all paths)
-worst case schedule length
-worst case resources required

Class Problem 4

```
if (a > 0) {
    r = t + s
    if (b > 0 || c > 0)
        u = v + 1
    else if (d > 0)
        x = y + 1
    else
        z = z + 1
}
```

a. Draw the CFG
b. Predicate the code removing all branches