EECS 583 – Class 3
Region Formation, Predicated Execution

University of Michigan

September 11, 2019

Announcements & Reading Material

- HW1 is out – Due Fri Sept 20
  - http://web.eecs.umich.edu/~mahlke/courses/583f19
- Today's class
  - "Trace Selection for Compiling Large C Applications to Microcode", Chang and Hwu, MICRO-21, 1988.
  - "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", Hwu et al., Journal of Supercomputing, 1993
- Material for Monday
  - "The Program Dependence Graph and Its Use in Optimization", J. Ferrante, K. Ottenstein, and J. Warren, ACM TOPLAS, 1987
    - This is a long paper – the part we care about is the control dependence stuff. The PDG is interesting and you should skim it over, but we will not talk about it now
  - "On Predicated Execution", Park and Schlansker, HPL Technical Report, 1991.

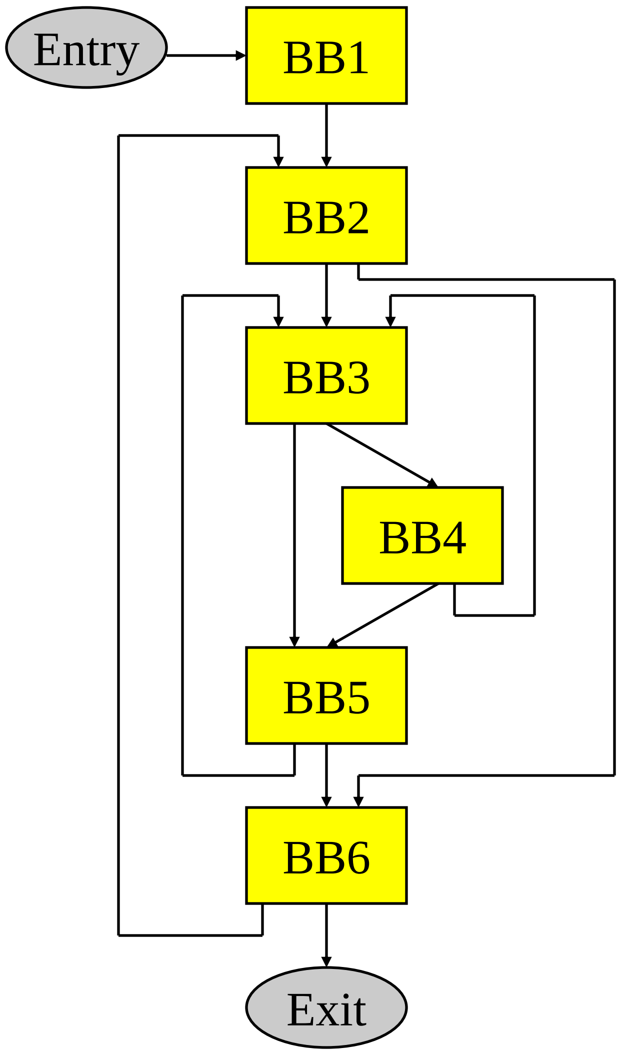
Announcements 2

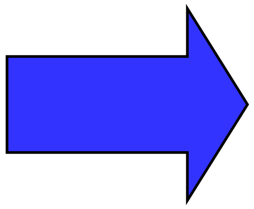
- Optional EECS 583 WhatsApp group
  - Zephaniah Hill organizing
  - Purpose: Arrange group study sessions and discussions.
  - To set up: Need WhatsApp and have your number attached to a WhatsApp account.
  - To join: Contact Zephaniah at 219-742-4398. Please identify that you are from EECS 583 and wish to be added to the WhatsApp group.

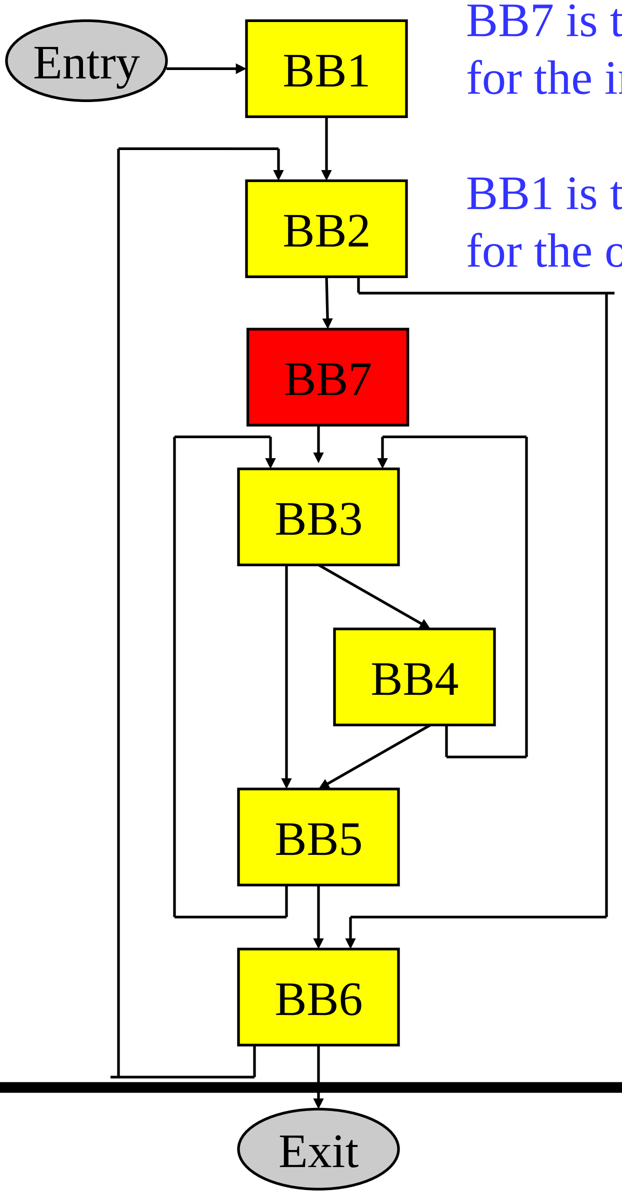
From Last Time: Loop Detection

- 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

From Last Time: Find the Preheaders for each Loop







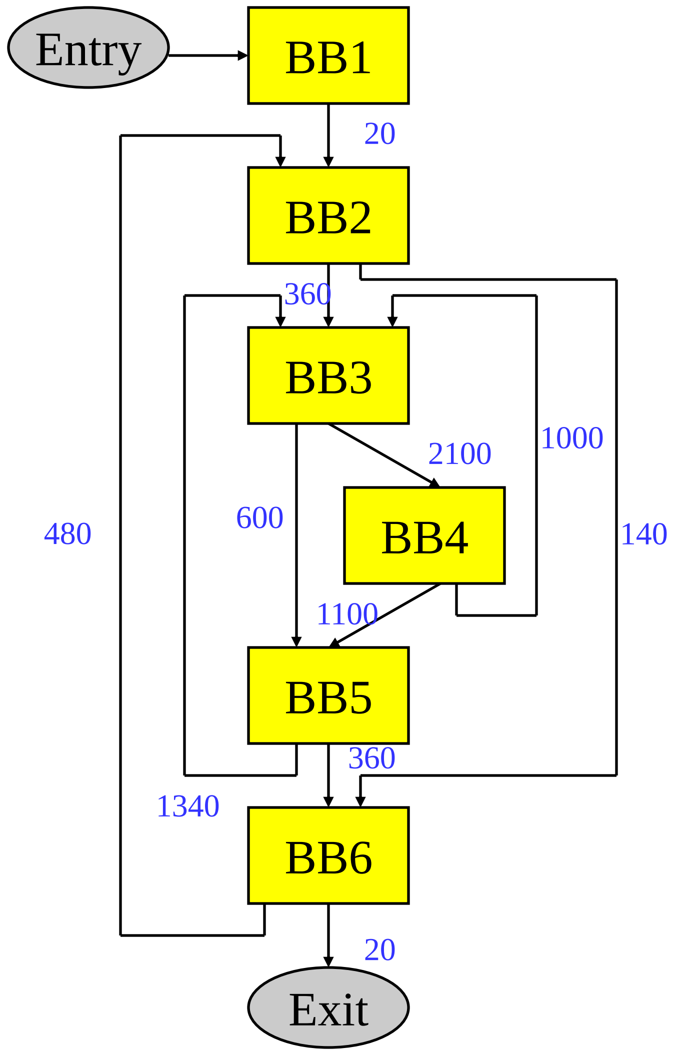
BB7 is the preheader for the inner loop

BB1 is the preheader for the outer 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)

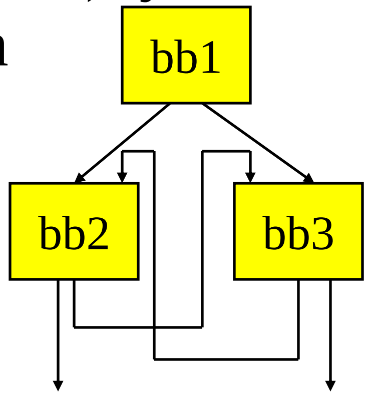
Trip Count Calculation Example



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→ y where y dominates x), you should have a connected, acyclic graph



Non-reducible!

Regions

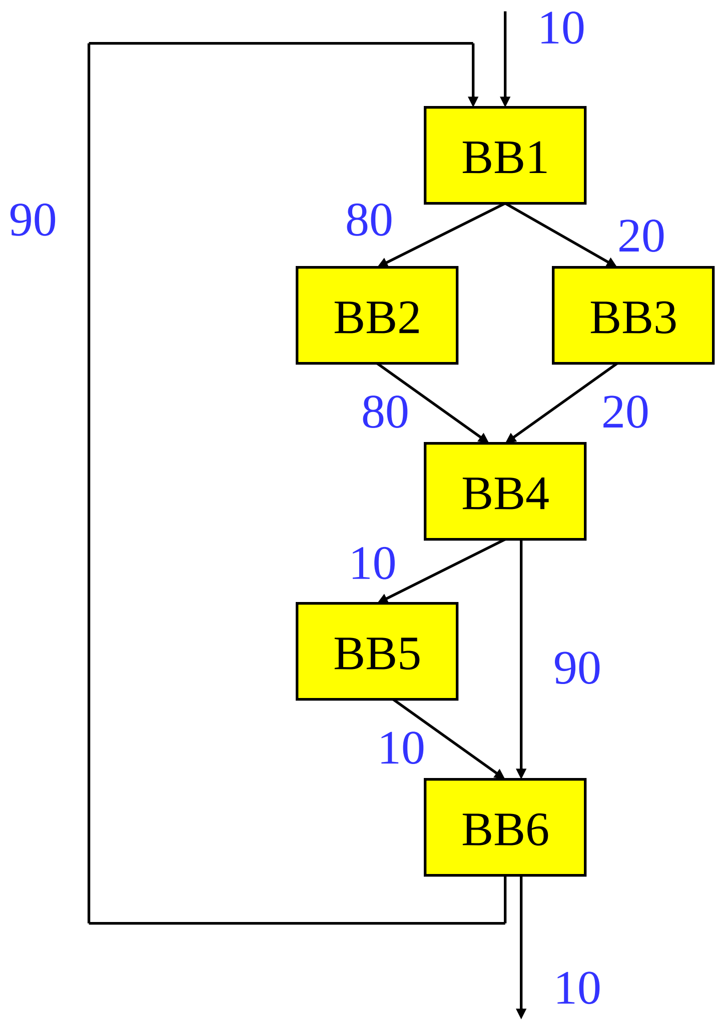
- Region: A collection of operations that are treated as a single unit by the compiler
  - » Examples
    - Basic block
    - Procedure
    - Body of a loop
  - » Properties
    - Connected subgraph of operations
    - Control flow is the key parameter that defines regions
    - Hierarchically organized
- 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)

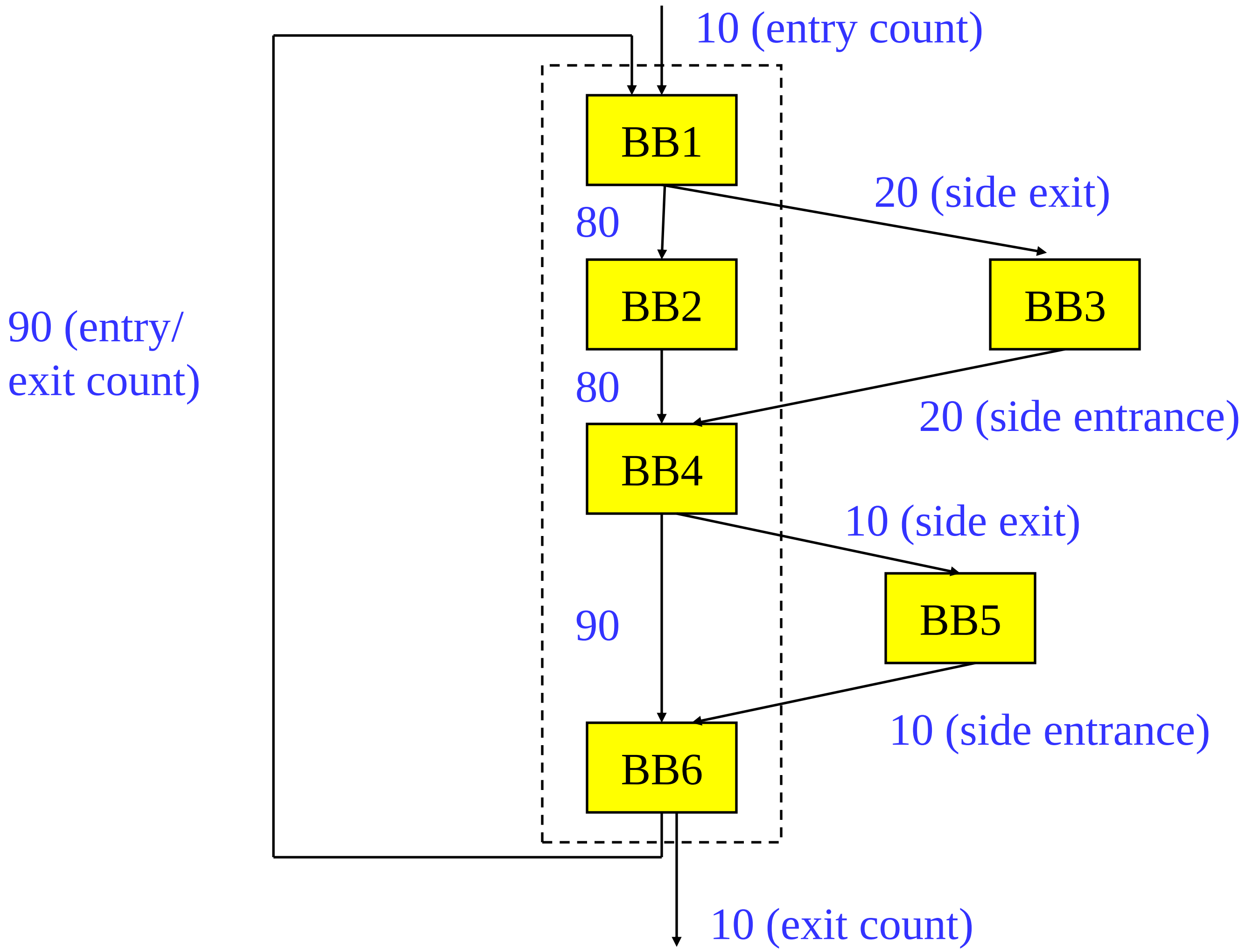
- Want
  - Intermediate sized regions with simple control flow
  - Bigger basic blocks would be ideal !!
  - Separate important code from less important
  - Optimize frequently executed code at the expense of the rest
- Solution
  - Define new region types that consist of multiple BBs
  - Profile information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

Region Type 1 - Trace

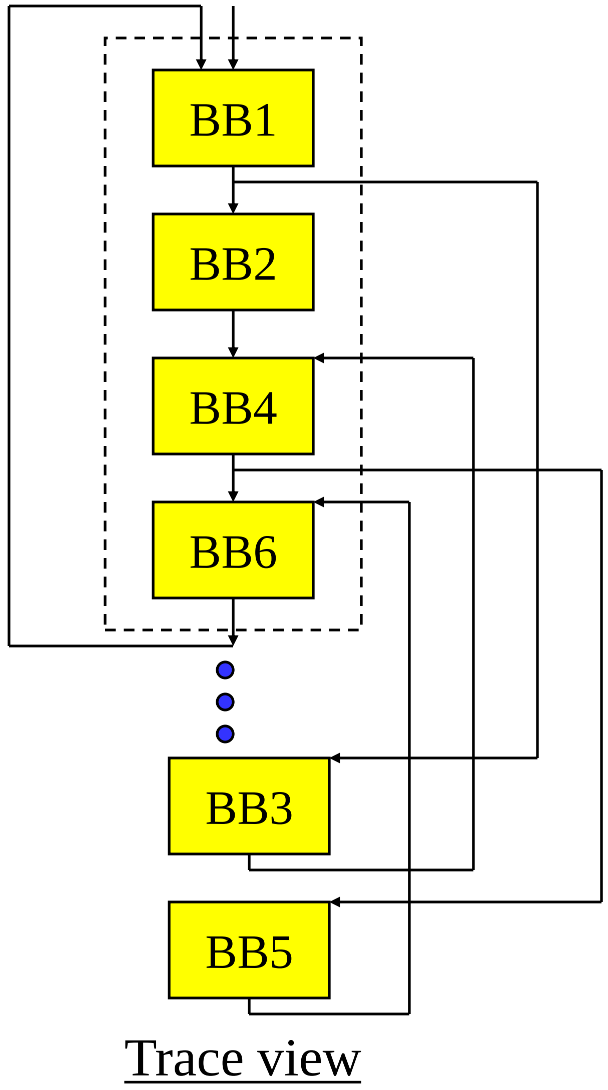
- Trace - Linear collection of basic blocks that tend to execute in sequence
  - "Likely control flow path"
  - Acyclic (outer backedge ok)
- Side entrance – branch into the middle of a trace
- Side exit – branch out of the middle of a trace
- Compilation strategy
  - Compile assuming path occurs 100%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



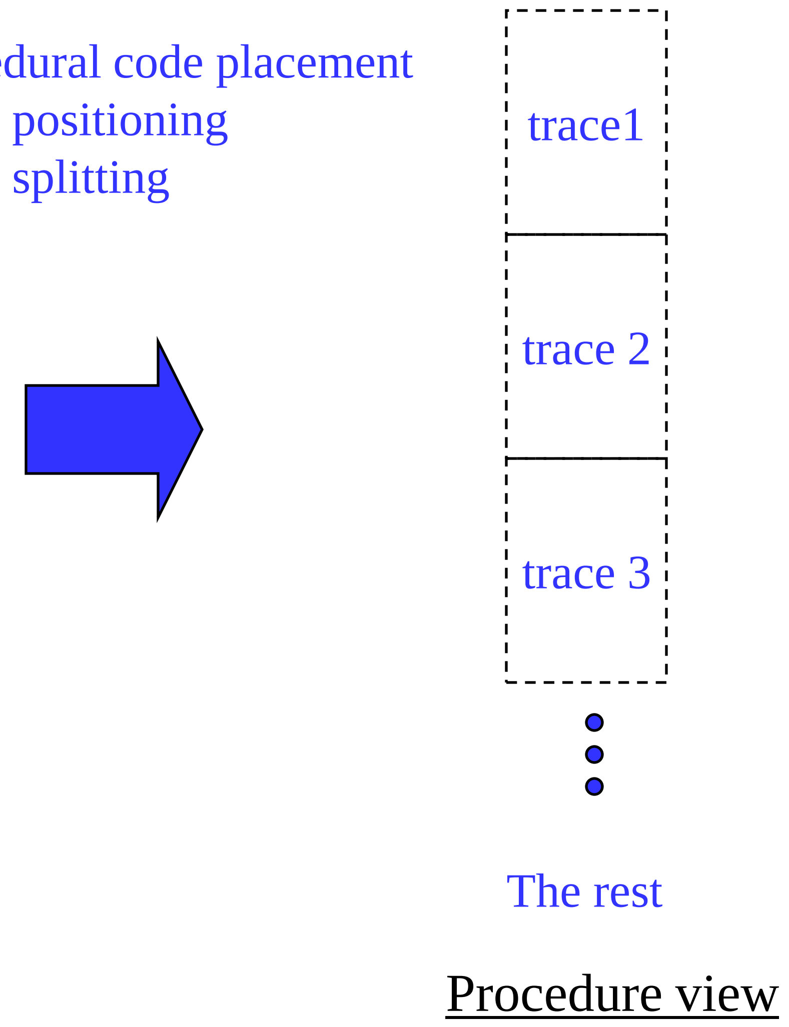
Linearizing a Trace



Intelligent Trace Layout for Icache Performance

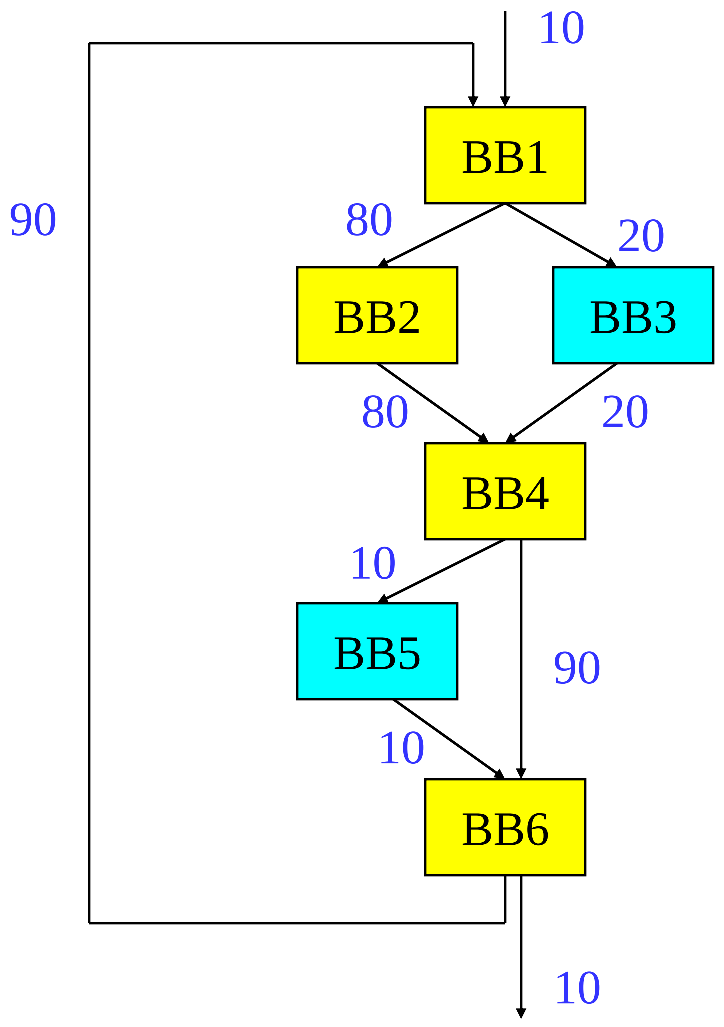


Intraprocedural code placement
Procedure positioning
Procedure splitting



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
```

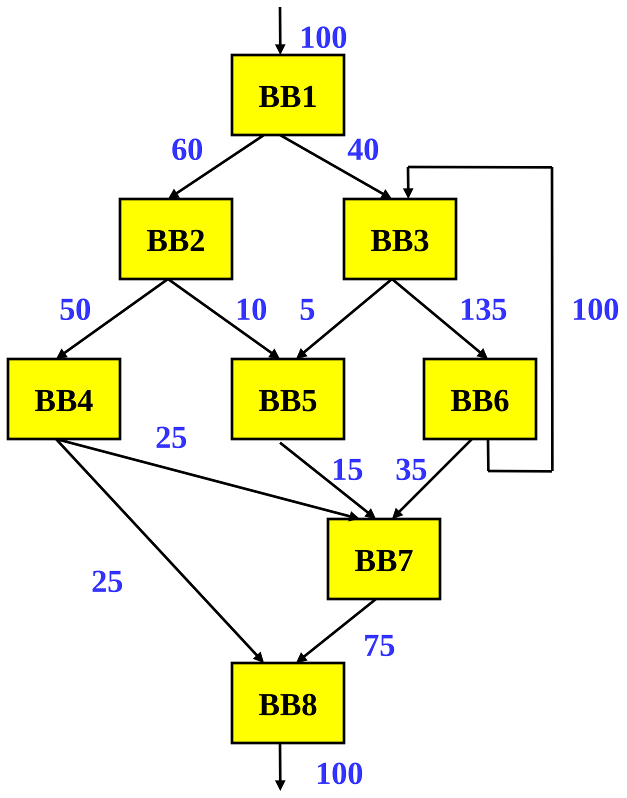
Best Successor/Predecessor

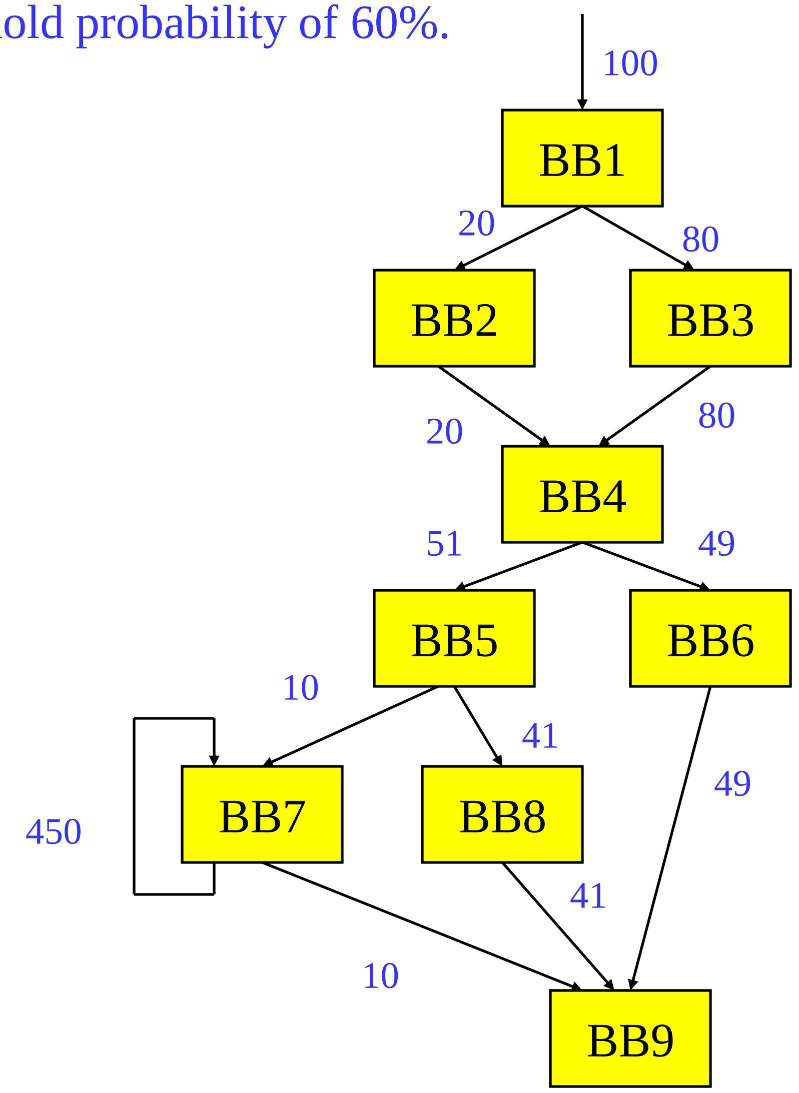
- Node weight vs edge weight
  - edge more accurate
- THRESHOLD
  - controls off-trace probability
  - 60-70% found best
- Notes on this algorithm
  - BB only allowed in 1 trace
  - Cumulative probability ignored
  - Min weight for seed to be chose (ie executed 100 times)

```
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

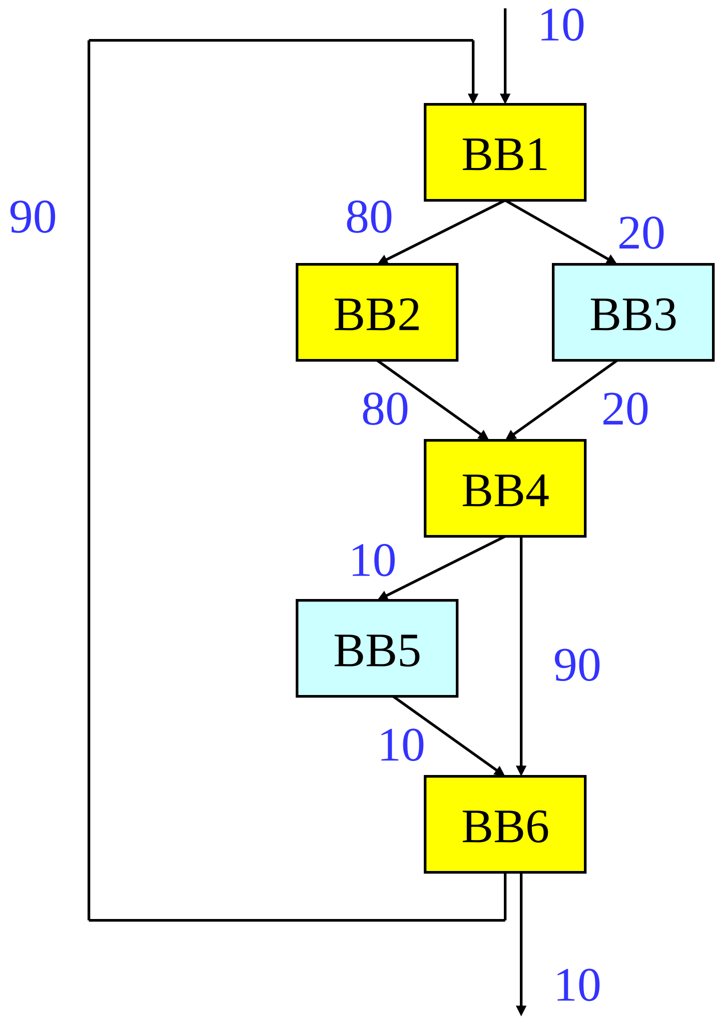
Find the traces. Assume a threshold probability of 60%.





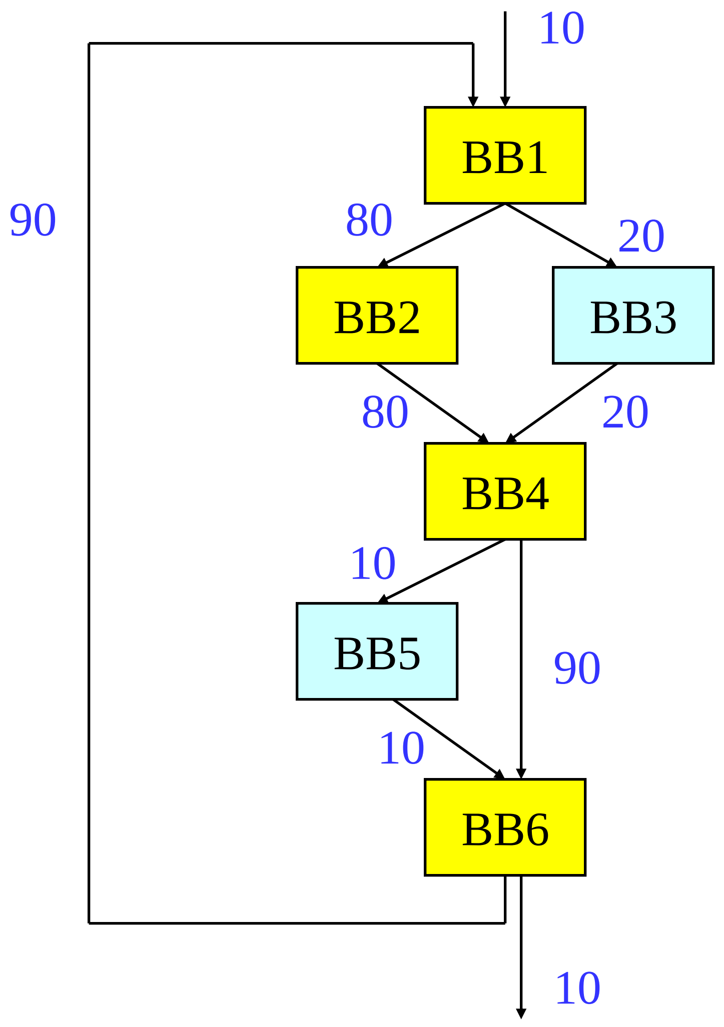
Traces are Nice, But …

- Treat trace as a big BB
  - » Transform trace ignoring side entrance/exits
  - » Insert fixup code
    - aka bookkeeping
  - » Side entrance fixup is more painful
  - » Sometimes not possible so transform not allowed
- Solution
  - » Eliminate side entrances
  - » The <u>superblock</u> is born



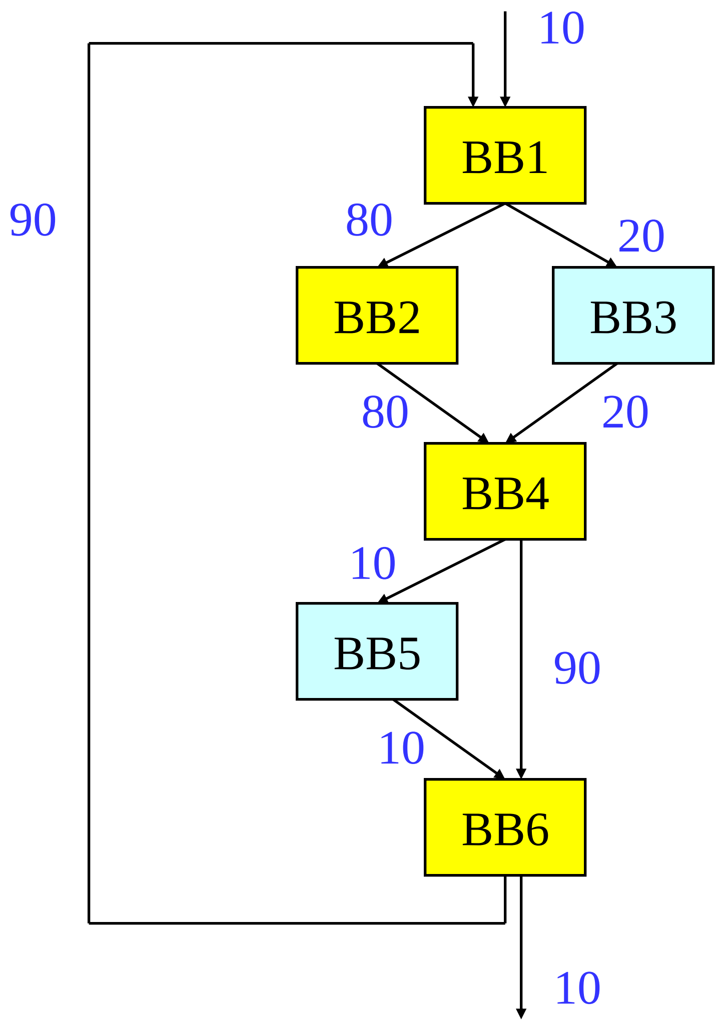
Region Type 2 - Superblock

- Superblock - Linear collection of basic blocks that tend to execute in sequence *in which control flow may only enter at the first BB*
  - “Likely control flow path”
  - Acyclic (outer backedge ok)
  - Trace with no side entrances
  - Side exits still exist
- Superblock formation
  - 1. Trace selection
  - 2. Eliminate side entrances

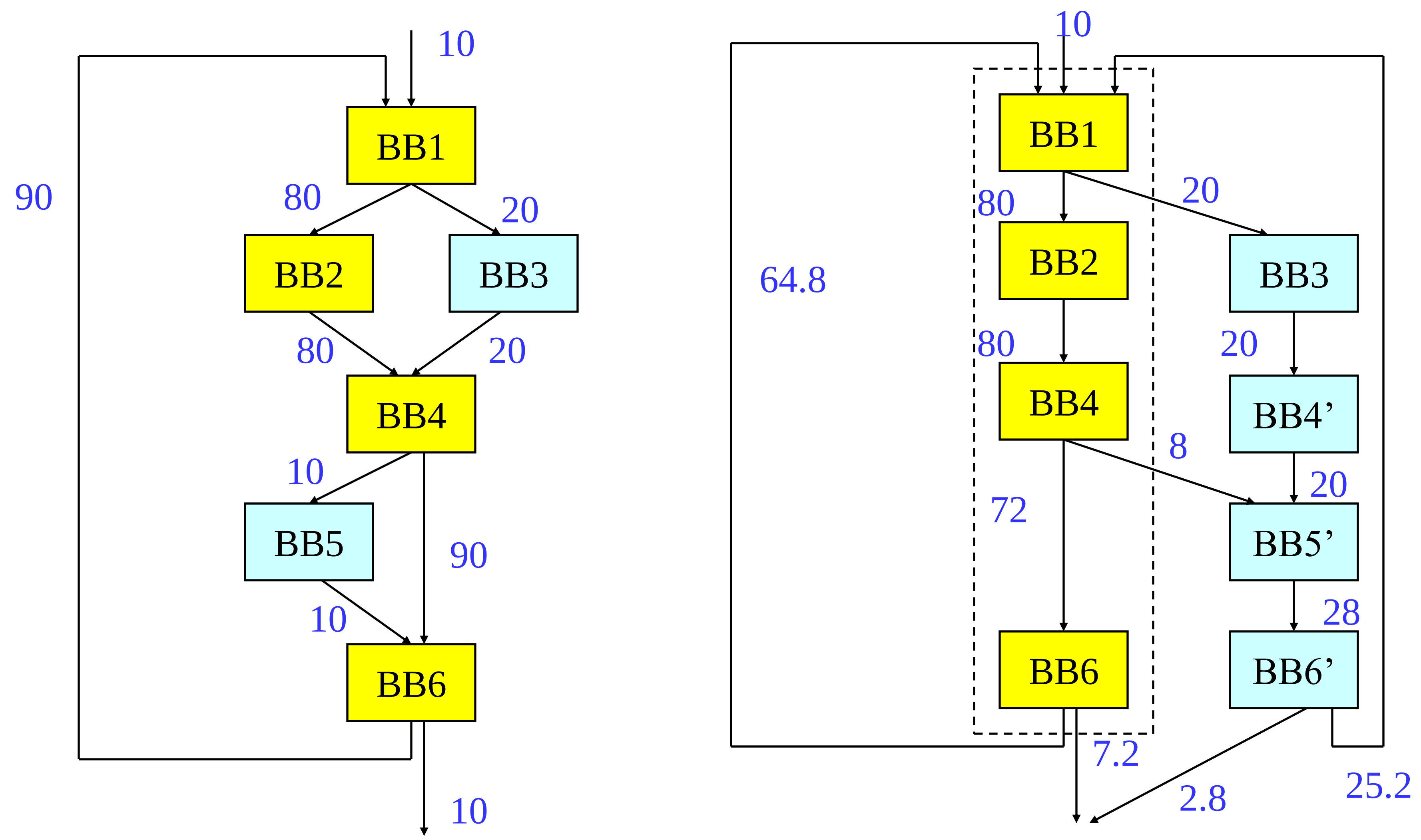


Tail Duplication

- To eliminate all side entrances replicate the "tail" portion of the trace
  - » Identify first side entrance
  - » Replicate all BB from the target to the bottom
  - » Redirect all side entrances to the duplicated BBs
  - » Copy each BB only once
  - » Max code expansion = 2x-1 where x is the number of BB in the trace
  - » Adjust profile information

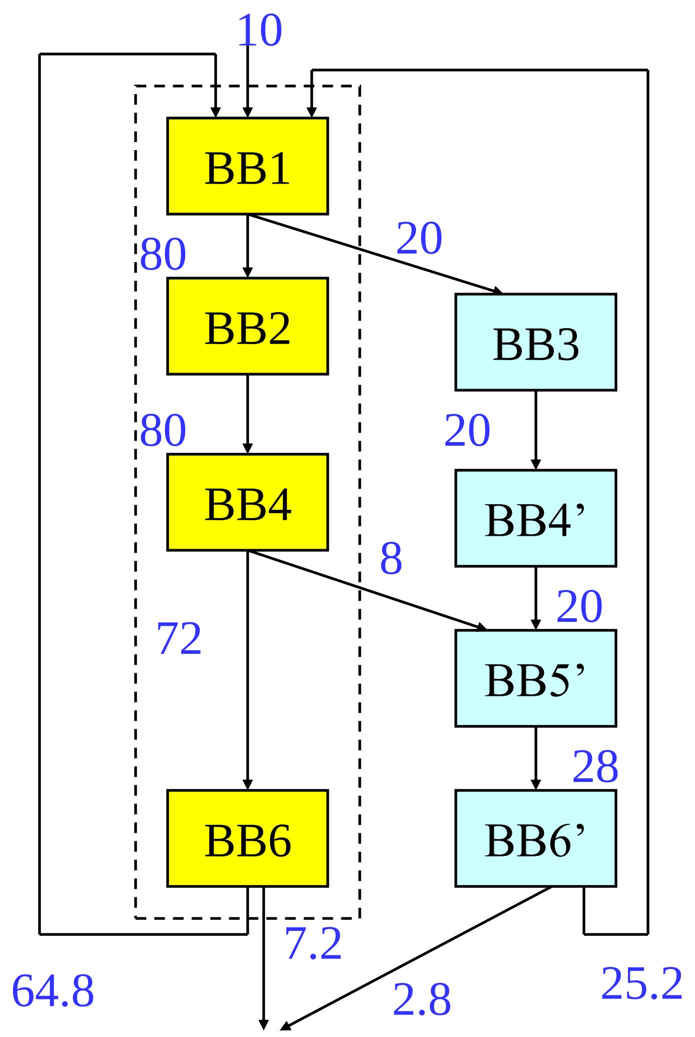


Superblock Formation

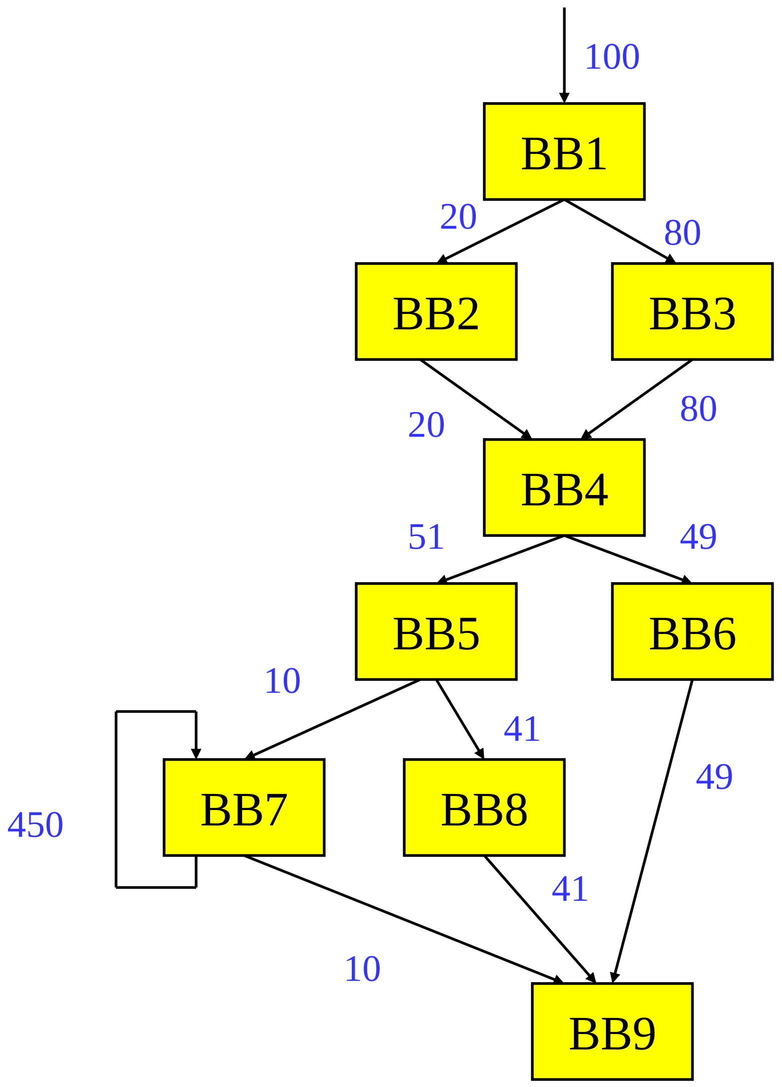


Issues with Superblocks

- Central tradeoff
  - Side entrance elimination
    - Compiler complexity
    - Compiler effectiveness
  - Code size increase
- Apply intelligently
  - Most frequently executed BBs are converted to SBs
  - Set upper limit on code expansion
  - 1.0 – 1.10x are typical code expansion ratios from SB formation



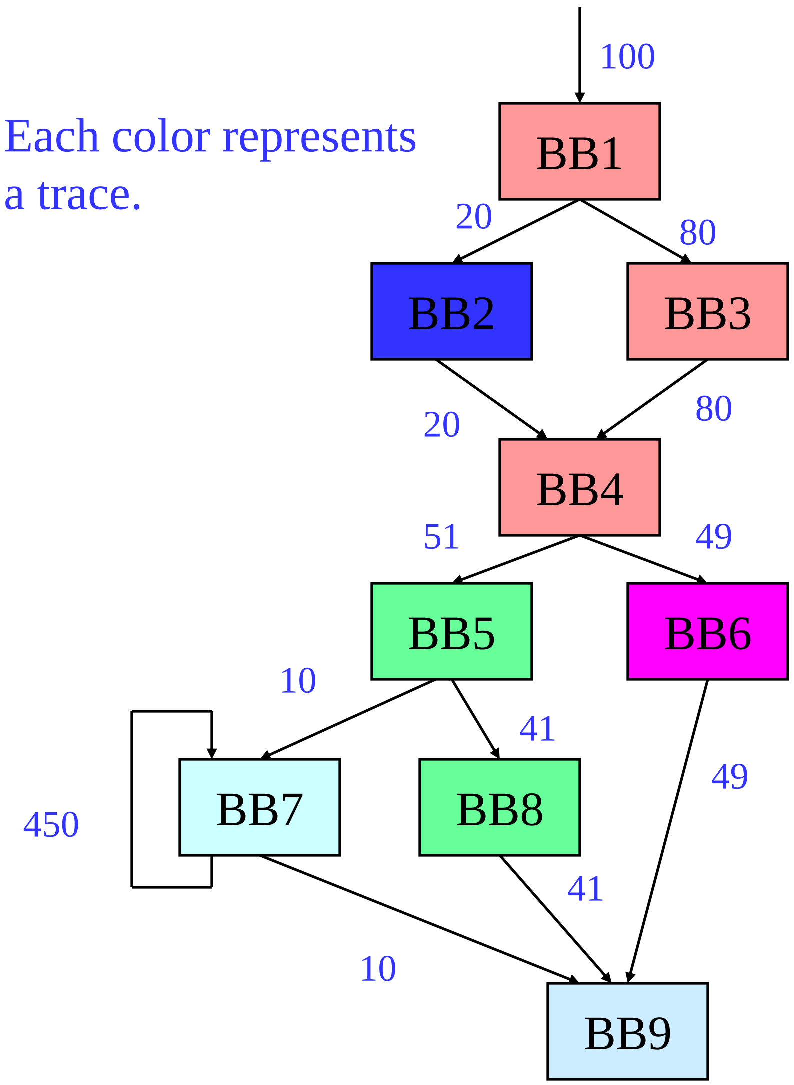
Class Problem



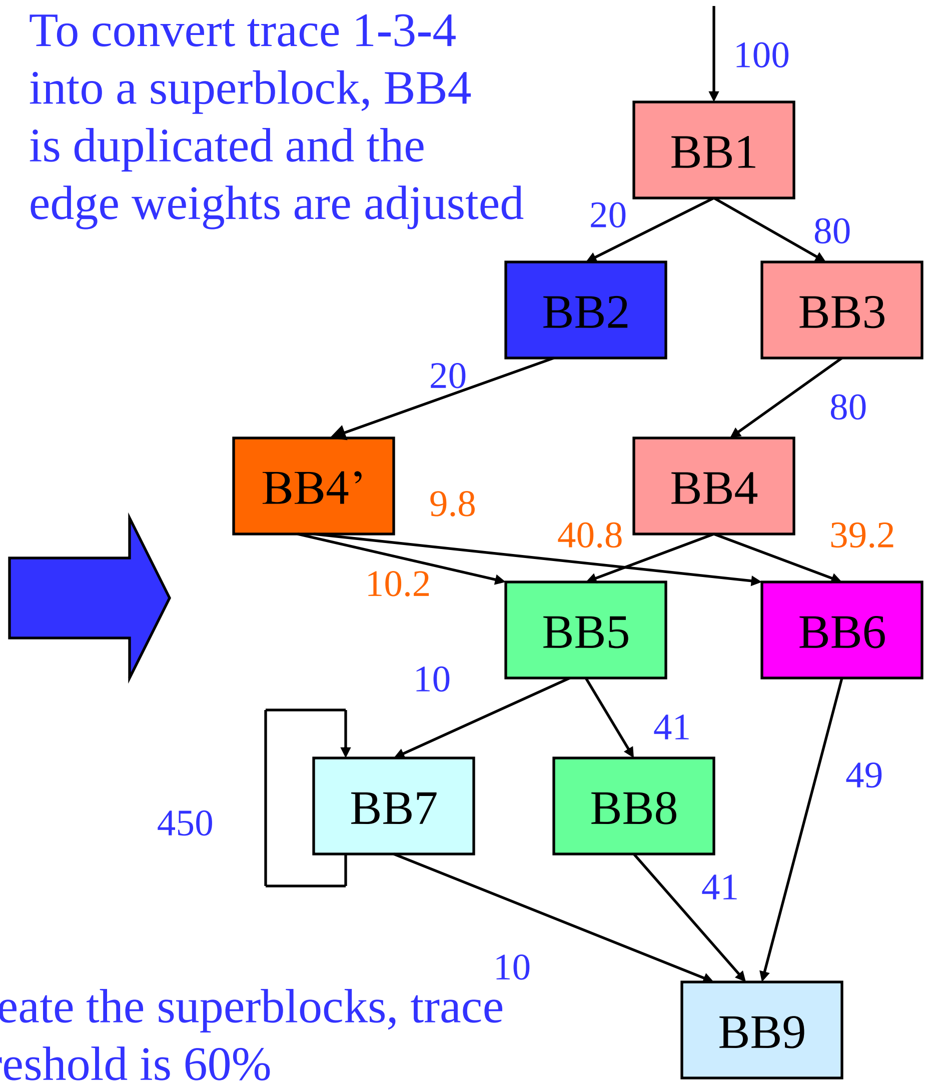
Create the superblocks, trace threshold is 60%

Class Problem Solution – Superblock Formation

Each color represents a trace.



To convert trace 1-3-4 into a superblock, BB4 is duplicated and the edge weights are adjusted



Create the superblocks, trace threshold is 60%

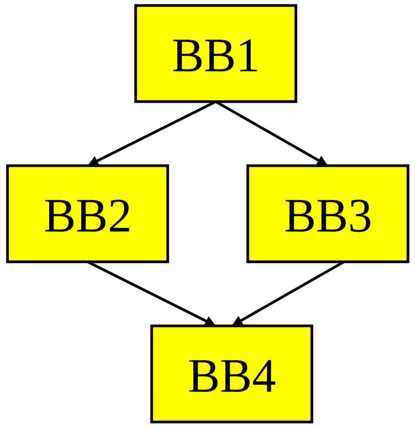
An Alternative to Branches: Predicated Execution

- 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 guarding predicate
    - 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

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

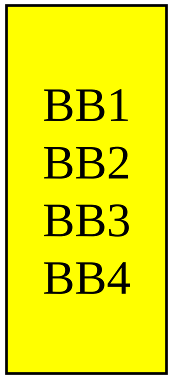
```
BB1    add a, b, c
BB1    bgt a, 0, L1
BB3    div e, f, g
BB3    jump L2
BB2    L1: add e, f, g
BB4    L2: sub h, i, j
```



Traditional branching 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
```

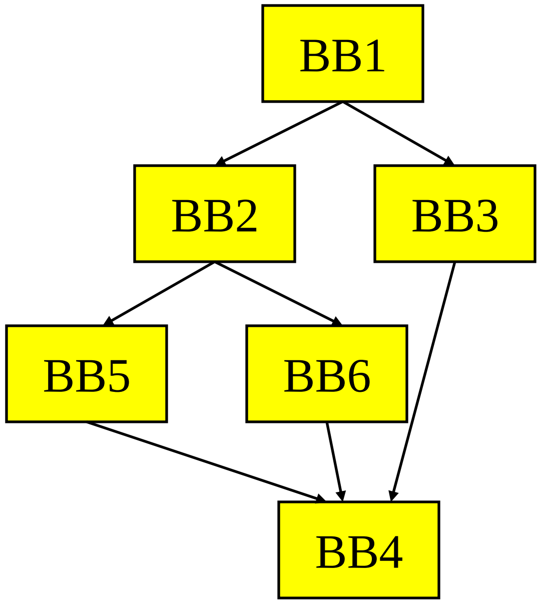


Predicated code

What About 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 – No Problem

```
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
BB3	p5 = a > 25 if p2
BB3	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

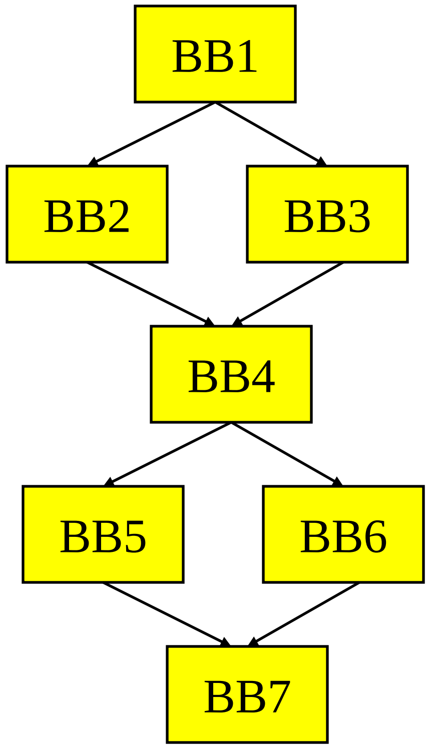
Predicated code

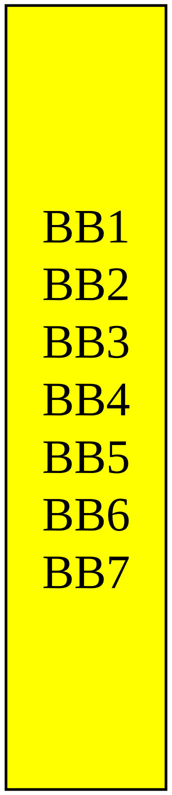
What do we assume to make this work ??

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


$$\begin{tikzcd}
 & BB1 \arrow[dl] \arrow[dr] & \\
BB2 \arrow[dr] & & BB3 \arrow[dl] \\
 & BB4 \arrow[dl] \arrow[dr] & \\
BB5 \arrow[dr] & & BB6 \arrow[dl] \\
 & BB7 &
\end{tikzcd}$$



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

HPL-PD Compare-to-Predicate Operations (CMPPs)

- How do we compute predicates
  - Compare registers/literals like a branch would do
  - Efficiency, code size, nested conditionals, etc
- 2 targets for computing taken/fall-through conditions with 1 operation

p1, p2 = CMPP.cond.D1a.D2a (r1, r2) if p3

p1 = first destination predicate
p2 = second destination predicate
cond = compare condition (ie EQ, LT, GE, …)
D1a = action specifier for first destination
D2a = action specifier for second destination
(r1,r2) = data inputs to be compared (ie r1 < r2)
p3 = guarding predicate

CMPP Action Specifiers

Guarding predicate	Compare Result	UN	UC	ON	OC	AN	AC
0	0	0	0	-	-	-	-
0	1	0	0	-	-	-	-
1	0	0	1	-	1	0	-
1	1	1	0	1	-	-	0

UN/UC = Unconditional normal/complement

This is what we used in the earlier examples

guard = 0, both outputs are 0

guard = 1, UN = Compare result, UC = opposite

ON/OC = OR-type normal/complement

AN/AC = AND-type normal/complement

OR-type, AND-type Predicates

```
p1 = 0
p1 = cmpp_ON (r1 < r2) if T
p1 = cmpp_OC (r3 < r4) if T
p1 = cmpp_ON (r5 < r6) if T
```

```
p1 = (r1 < r2) | (!(r3 < r4)) |
      (r5 < r5)
```

Wired-OR into p1

Generating predicated code for some source code requires OR-type predicates

```
p1 = 1
p1 = cmpp_AN (r1 < r2) if T
p1 = cmpp_AC (r3 < r4) if T
p1 = cmpp_AN (r5 < r6) if T
```

```
p1 = (r1 < r2) & (!(r3 < r4)) &
      (r5 < r5)
```

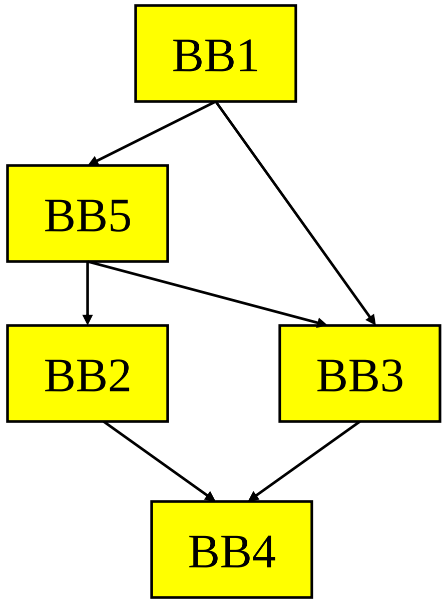
Wired-AND into p1

Talk about these later – used for control height reduction

Use of OR-type Predicates

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

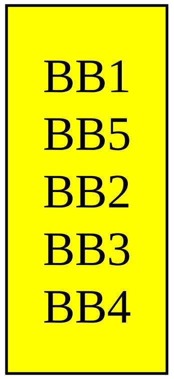
```
BB1    add a, b, c
BB1    ble a, 0, L1
BB5    ble b, 0, L1
BB2    add e, f, g
BB2    jump L2
BB3    L1: div e, f, g
BB4    L2: sub h, i, j
```


$$\begin{tikzcd}
 & BB1 \arrow[dl] \arrow[ddr] & \\
BB5 \arrow[d] \arrow[drr] & & \\
BB2 \arrow[dr] & & BB3 \arrow[dl] \\
 & BB4 &
\end{tikzcd}$$

Traditional branching code

p2 → BB2
p3 → BB3
p5 → BB5

```
BB1    add a, b, c if T
BB1    p3, p5 = cmpp.ON.UC a <= 0 if T
BB5    p3, p2 = cmpp.ON.UC b <= 0 if p5
BB3    div e, f, g if p3
BB2    add e, f, g if p2
BB4    sub h, i, j if T
```



Predicated code

Homework Problem – Answer Next Time

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

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