Lecture 4: Control Flow Optimization

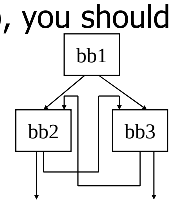
COS 598C – Advanced Compilers

Spyridon Triantafyllis

Prof. David August
Department of Computer Science
Princeton University

Reducible Flow Graphs

- A flow graph is <u>reducible</u> 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 <u>connected, acyclic</u> graph



Non-reducible!

Back to Loops – Assembly Generation Schema

```
for (i=x; i<y; i+=z) {
    body;
}
```

<u>while-do schema</u>

```
loop:   if (i >= y) goto done
        body;
        i += z;
        goto loop
done:
```

<u>do-while schema</u>

```
        if (i >= y) goto done
loop:   body;
        i += z;
        if (i < y) goto loop
done:
```

Question: which schema is better and why?

Loop Induction Variables

- Induction variables are variables such that every time they change value, they are incremented/decremented by some constant
- Basic induction variable – induction variable whose only assignments within a loop are of the form j = j+- C, where C is a constant
- Primary induction variable – basic induction variable that controls the loop execution (for I=0; I<100; I++), I (virtual register holding I) is the primary induction variable
- Derived induction variable – variable that is a linear function of a basic induction variable

Class Problem 1 (4 from last time)

Identify the basic, primary and derived inductions variables in this loop.

```
       r1 = 0
       r7 = &A
Loop:  r2 = r1 * 4
       r4 = r7 + 3
       r7 = r7 + 1
       r1 = load(r2)
       r3 = load(r4)
       r9 = r1 * r3
       r10 = r9 >> 4
       store (r10, r2)
       r1 = r1 + 4
       blt r1 100 Loop
```


Loop Unrolling

- Most renowned control flow opti
- Replicate the body of a loop N-1 times (giving N total copies)
  - Loop unrolled N times or Nx unrolled
  - Enable overlap of operations from different iterations
  - Increase potential for ILP (instruction level parallelism)
- 3 variants
  - Unroll multiple of known trip count
  - Unroll with remainder loop
  - While loop unroll

```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + 1
blt r2 100 Loop
```

Loop Unroll – Type 1

<u>Counted loop</u>
<u>All parms known</u>

r2 is the loop variable,
Increment is 1
Initial value is 0
Final value is 100
Trip count is 100

```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + 1
blt r2 100 Loop
```

```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + 1
--------------------
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + 1
blt r2 100 Loop
```

Remove branch from
first N-1 iterations

```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
--------------------
r1 = MEM[r2 + 1]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + 2
blt r2 100 Loop
```

Remove r2 increments
from first N-1 iterations
and update last increment

Loop Unroll – Type 2

<u>Counted loop</u>
<u>Some parms unknown</u>

r2 is the loop variable,
Increment is ?
Initial value is ?
Final value is ?
Trip count is ?

```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + X
blt r2 Y Loop
```

```
tc = final – initial
tc = tc / increment
rem = tc % N
fin = rem * increment
```

```
RemLoop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + X
blt r2 fin RemLoop
```

Remainder loop executes
the “leftover” iterations

```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
--------------------
r1 = MEM[r2 + X]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = r2 + (N*X)
blt r2  Y Loop
```

Unrolled loop same as Type 1,
and is guaranteed to execute
a multiple of N times

Loop Unroll – Type 3

<u>Non-counted loop</u>
<u>Some parms unknown</u>

pointer chasing, loop
var modified in a strange
way, etc.

```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = MEM[r2 + 0]
bne r2 0 Loop
```

```
Loop:
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = MEM[r2 + 0]
beq r2 0 Exit
--------------------
r1 = MEM[r2 + 0]
r4 = r1 * r5
r6 = r4 << 2
MEM[r3 + 0] = r6
r2 = MEM[r2 + 0]
bne r2 0 Loop
Exit:
```

Just duplicate the
body, none of the
loop branches can
be removed. Instead
they are converted into
conditional breaks

Can apply this
to any loop including
a superblock or
hyperblock loop !

Loop Unroll Summary

- Goal – Enable overlap of multiple iterations to increase ILP
- Type 1 is the most effective
  - All intermediate branches removed, least code expansion
  - Limited applicability
- Type 2 is almost as effective
  - All intermediate branches removed
  - Remainder loop is required since trip count not known at compile time
  - Need to make sure don't spend much time in rem loop
- Type 3 can be effective
  - No branches eliminated
  - But operation overlap still possible
  - Always applicable (most loops fall into this category!)
  - Use expected trip count to guide unroll amount

Class Problem 2

Unroll both the outer
loop and inner loop 2x

```
for (i=0; i<100; i++) {
    j = i;
    while (j < 100) {
            A[j]--;
            j += 5;
    }
    B[i] = 0;
}
```


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

BB1
. . .
L1: if (a < b) goto L2

BB2
L2:
. . .

Branch is unnecessary

BB1
. . .
L1:

BB2
L2:
. . .

4. Basic block merging

BB1
. . .
L1:

BB2
L2:
. . .

Merge BBs when single edge between

BB1
. . .
L1:
L2:
. . .

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
. . .

. . .

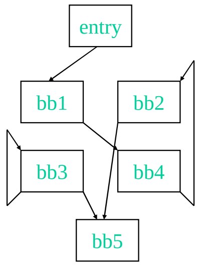
BB2
L2: stuff2
. . .

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

Regions

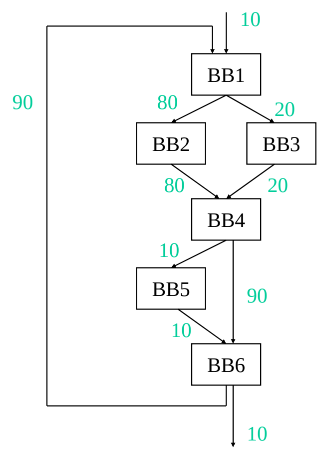
- Region: A collection of operations that are treated as a single unit by the compiler
  - Examples
    - Basic block
    - Procedure
    - Body of a loop
  - Properties
    - Connected subgraph of operations
    - Control flow is the key parameter that defines regions
    - Hierarchically organized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

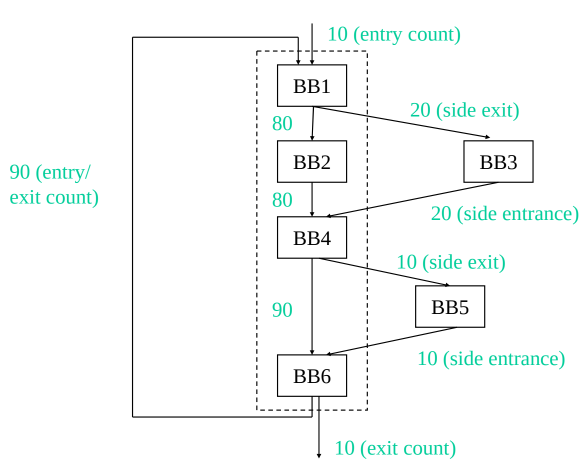
- Want
  - Intermediate sized regions with simple control flow
  - Bigger basic blocks would be ideal !!
  - Separate important code from less important
  - Optimize frequently executed code at the expense of the rest
- Solution
  - Define new region types that consist of multiple BBs
  - Profile information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

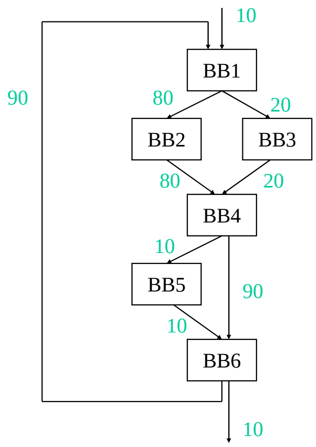


Linearizing a Trace



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
```

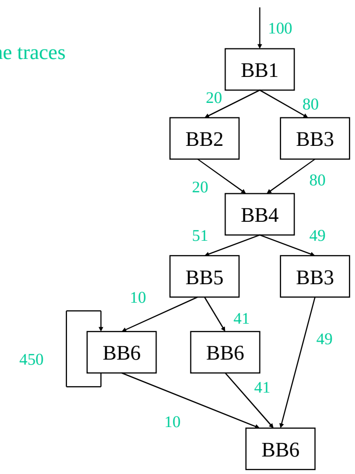

Best Successor/Predecessor

- 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)

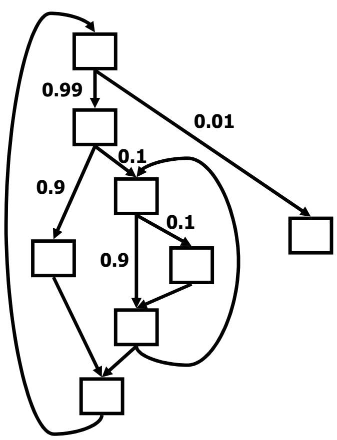
```
best_successor_of(BB)
   e = control flow edge with highest
        probability leaving BB
   if (e is a backedge) then
      return 0
   endif
   if (probability(e) <= THRESHOLD) then
      return 0
   endif
   d = destination of e
   if (d is visited) then
      return 0
   endif
   return d
endprocedure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

Class Problem 4

Find the traces



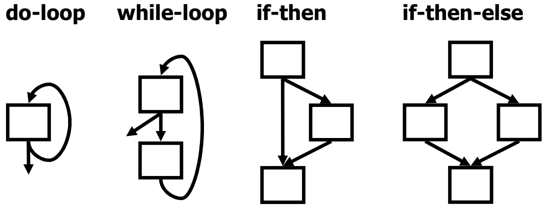
Free-form regions



- Choose a "related" control-flow subgraph
  - profile-guided selection
  - other criteria?
- Optimize as a unit
  - radically reduced compile time
  - minimal performance loss
  - … if regions are selected right!
- Found at:
  - IMPACT (earlier versions)
  - ORC (for scheduling only)
  - Open64
- Area still open for experimentation

Intervals & Structural Analysis

do-loop **while-loop** **if-then** **if-then-else** **etc.**



- Structural regions correspond to source-language structures
- Structural regions can be nested!
- Intervals: Structural regions with less detail (loops vs. non-loops)
- Useful for dataflow analysis
- Could they be used as compilation regions?

Intervals and Structural Analysis: Example