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

```
  b b 1
     Non-reducible!
    /   \
   b b 2/    \b b 3
```

Back to Loops – Assembly Generation Schema

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

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

```
do-while schema
loop:  if (i >= y) goto done
    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.

**Loop:**

\[
\begin{align*}
  r_1 &= 0 \\
  r_7 &= \& A \\
  r_2 &= r_1 * 4 \\
  r_4 &= r_7 + 3 \\
  r_7 &= r_7 + 1 \\
  r_1 &= \text{load}(r_2) \\
  r_3 &= \text{load}(r_4) \\
  r_9 &= r_1 + r_3 \\
  r_{10} &= r_9 >> 4 \\
  \text{store}(r_{10}, r_2) \\
  r_1 &= r_1 + 4 \\
  \text{blt} \ r_1 \ 100 \ \text{Loop}
\end{align*}
\]

**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:**

\[
\begin{align*}
  r_1 &= \text{MEM}(r_2 + 0) \\
  r_4 &= r_1 * r_5 \\
  r_6 &= r_4 << 2 \\
  \text{MEM}(r_3 + 0) &= r_6 \\
  r_2 &= r_2 + 1 \\
  \text{blt} \ r_2 \ 100 \ \text{Loop}
\end{align*}
\]
**Loop Unroll – Type 1**

**Loop Unroll – Type 2**

**Loop Unroll – Type 3**
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

```c
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
1. Branch to unconditional branch

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

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

3. Branch to next basic block

```
BB1
  ...  
L1: if (a < b) goto L2
BB2
  L2:
    ...
```

```
BB1
  ...  
L1:
BB2
  L2:
    ...
```

4. Basic block merging

```
BB1
  ...  
L1:
BB2
  L2:
    ...
```

```
BB1
  ...  
L1:
BB2
  L2:
    ...
```

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?

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)
• **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**

- **BB1**
  - 90 (entry count)
  - 10 (entry count)
- **BB2**
  - 80 (side exit)
  - 20 (side exit)
- **BB3**
  - 20 (side exit)
  - 20 (side enterance)
  - 10 (side enterance)
- **BB4**
  - 90 (entry/exit count)
  - 10 (exit count)
- **BB5**
  - 90 (entry/exit count)
  - 10 (exit count)
- **BB6**
  - 10 (exit count)
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**

```plaintext
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 chosen (ie executed 100 times)

```plaintext
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

- 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