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 instruction 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

Traditional branching code

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

p2 → bb2  
p3 → bb3

Predicated 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: bgt a, 25, l3  
bb6 mpy e, f, g  
bb6 jump l2  
bb5 l3: add e, f, g  
bb4 l2: sub h, i, j

What About Nested if-then-else’s?
Nested if-then-else’s – No Problem

\[
a = b + c \\
\text{if } (a > 0)\text{ then } e = f + g \text{ else } e = f * g \\
h = i - j
\]

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 (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

Compare-to-Predicate Operations (CMPPs)

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

\[
p_1, p_2 = \text{CMPP.cond.D1a.D2a (r1, r2) if } p_3 \\
p_1 = \text{first destination predicate} \\
p_2 = \text{second destination predicate} \\
\text{cond = compare condition (ie EQ, LT, GE, …) } \\
\text{D1a = action specifier for first destination} \\
\text{D2a = action specifier for second destination} \\
(r1,r2) = \text{data inputs to be compared (ie r1 < r2)} \\
p_3 = \text{guarding predicate}
\]
CMPP Action Specifiers

<table>
<thead>
<tr>
<th>Guarding predicate</th>
<th>Compare Result</th>
<th>UN</th>
<th>UC</th>
<th>ON</th>
<th>OC</th>
<th>AN</th>
<th>AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

UN/UC = Unconditional normal/complement
This is what we used in the earlier examples
guard = 0, both outputs are 0

ON/OC = OR-type normal/complement
UC = opposite

AN/AC = AND-type normal/complement

Use of OR-type Predicates

\[ a = b + c \]
\[ \text{if} \ (a > 0 \ \&\& \ b > 0) \]
\[ e = f + g \]
\[ \text{else} \]
\[ e = f / g \]
\[ h = i - j \]

Traditional branching code:

- BB1: \( \text{add } a, b, c \)
- BB1: \( \text{ble } a, 0, L1 \)
- BB5: \( \text{ble } b, 0, L1 \)
- BB2: \( \text{add } e, f, g \)
- BB2: \( \text{jump } L2 \)
- BB3: \( \text{L1: } \text{div } e, f, g \)
- BB4: \( \text{L2: } \text{sub } h, i, j \)

Predicated code:

- \( p2 \rightarrow BB2 \)
- \( p3 \rightarrow BB3 \)
- \( p5 \rightarrow BB5 \)
- BB1: \( \text{add } a, b, c \text{ if } T \)
- BB1: \( p3, p5 = \text{cmpp}_\text{ON}.\text{UC } a <= 0 \text{ if } T \)
- BB5: \( p3, p2 = \text{cmpp}_\text{ON}.\text{UC } b <= 0 \text{ if } p5 \)
- BB3: \( \text{div } e, f, g \text{ if } p3 \)
- BB2: \( \text{add } e, f, g \text{ if } p2 \)
- BB4: \( \text{sub } h, i, j \text{ if } T \)

OR-type, AND-type Predicates

- \( p1 = 0 \)
- \( p1 = \text{cmpp}_\text{ON} (r1 < r2) \text{ if } T \)
- \( p1 = \text{cmpp}_\text{OC} (r3 < r4) \text{ if } T \)
- \( p1 = \text{cmpp}_\text{ON} (r5 < r6) \text{ if } T \)

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

Wired-OR into \( p1 \)
Wired-AND into \( p1 \)

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

Talk about these later – used
for control height reduction

Class Problem

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

a. Draw the CFG
b. Predicate the code removing all branches
If-Conversion

- Algorithm for generating predicated code
  - Automate what we’ve been doing by hand
  - Handle arbitrary complex graphs
    - But, acyclic subgraph only!
    - Need a branch to get you back to the top of a loop
  - Efficient
- Roots are from Vector computer days
  - Vectorize a loop with an if-statement in the body
- 4 steps
  1. Loop backedge coalescing
  2. Control dependence analysis
  3. Control flow substitution
  4. CMPP compaction

Step 1: Backedge Coalescing

- Recall – Loop backedge is branch from inside the loop back to the loop header
- This step only applicable for a loop body
  - If not a loop body \(\rightarrow\) skip this step
- Process
  - Create a new basic block
    - New BB contains an unconditional branch to the loop header
  - Adjust all other backedges to go to new BB rather than header
- Why do this?
  - Heuristic step – Not essential for correctness
    - If-conversion cannot remove backedges (only forward edges)
    - But this allows the control logic to figure out which backedge you take to be eliminated
  - Generally this is a good thing to do

Running Example – Initial State

do {
  b = load(a)
  if (b < 0) {
    if ((c > 0) && (b > 13))
      b = b + 1
    else
      c = c + 1
      d = d + 1
  } else {
    e = e + 1
    if (c > 25) continue
  }
  a = a + 1
} while (e < 34)
Step 2: Control Dependence Analysis (CD)

- Control flow – Execution transfer from 1 BB to another via a taken branch or fall-through path
- Dependence – Ordering constraint between 2 operations
  - Must execute in proper order to achieve the correct result
    01: \( a = b + c \)
    02: \( d = a - e \)
- Control dependent on 01 (flow dependent)
- Control dependence – One operation controls the execution of another
  01: \( \text{blt } a, 0, \text{SKIP} \)
  02: \( b = c + d \)
  \( \text{SKIP} : \)
  - 02 control dependent on 01
- Control dependence analysis derives these dependences

Control Dependence Example

Control Dependences

- Recall
  - Post dominator – BBX is post dominated by BBY if every path from BBX to EXIT contains BBY
  - Immediate post dominator – First breadth first successor of a block that is a post dominator
- Control dependence – BBY is control dependent on BBX iff
  1. There exists a directed path P from BBX to BBY with all BBZ in P (excluding BBX and BBY) post dominated by BBY
  2. BBX is not post dominated by BBY
- In English,
  - A BB is control dependent on the closest BB(s) that determine(s) its execution
  - Its actually not a BB, it’s a control flow edge coming out of a BB

Running Example – CDs

First, nuke backedge(s)
Second, nuke exit edges
Then, Add pseudo entry/exit nodes
  - Entry \( \rightarrow \) nodes with no predecessors
  - Exit \( \rightarrow \) nodes with no successors

Control deps (left is taken)
BB1:
BB2:
BB3:
BB4:
BB5:
BB6:
BB7:

Notation
positive BB number = fall-thru direction
negative BB number = taken direction
Algorithm for Control Dependence Analysis

for each basic block x in region
  for each outgoing control flow edge e of x
    y = destination basic block of e
    if (y not in pdom(x)) then
      lub = ipdom(x)
      if (e corresponds to a taken branch) then
        x_id = -x.id
      else
        x_id = x.id
      endif
      t = y
      while (t != lub) do
        cd(t) += x_id;
        t = ipdom(t)
      endwhile
    endif
  endfor
endfor

Notes

Compute cd(x) which contains those BBs which x is control dependent on.

Iterate on per edge basis, adding edge to each cd set it is a member of
Running Example – CDs Via Algorithm (3)

Step 3: Control Flow Substitution

- Go from branching code → sequential predicated code
- 5 baby steps
  1. Create predicates
  2. CMPP insertion
  3. Guard operations
  4. Remove branches
  5. Initialize predicates

Predicate Creation

- R/K calculation – Mapping predicates to blocks
  - Paper more complicated than it really is
  - K = unique sets of control dependences
  - Create a new predicate for each element of K
  - R(bb) = predicate that represents CD set for bb, ie the bb’s assigned predicate (all ops in that bb guarded by R(bb))

CMPP Creation/Insertion

- For each control dependence set
  - For each edge in the control dependence set
    - Identify branch condition that causes edge to be traversed
    - Create CMPP to compute corresponding branch condition
      - OR-type = handles worst case
      - guard = True
      - destination = predicate assigned to that CD set
      - Insert at end of BB that is the source of the edge

K = \{\{-1\}, \{1\}, \{-2\}, \{-4\}, \{2, 4\}, \{-1, -3\}\}
predicates = \{p1, p2, p3, p4, p5, p6\}

bb = 1, 2, 3, 4, 5, 6, 7, 8, 9
CD(bb) = \{\{\text{none}\}, \{-1\}, \{1\}, \{-2\}, \{-4\}, \{2, 4\}, \{-1\}, \{-1, -3\}, \{\text{none}\}\}
R(bb) = T p1 p2 p3 p4 p5 p1 p6 T

p1 = cmpp.ON (b < 0) if T → BB1
Running Example – CMPP Creation

- Guard all operations in each bb by R(bb)
  - Including the newly inserted CMPPs
- Nuke all the branches
  - Except exit edges and backedges
- Initialize each predicate to 0 in first BB

Control Flow Substitution – The Rest

Step 4: CMPP Compaction

- Convert ON CMPPs to UN
  - All singly defined predicates don’t need to be OR-type
  - OR of 1 condition → Just compute it !!!
  - Remove initialization (Unconditional don’t require init)
- Reduce number of CMPPs
  - Utilize 2nd destination slot
  - Combine any 2 CMPPs with:
    - Same source operands
    - Same guarding operands
    - Same or opposite compare conditions
### Running Example - CMPP Compaction

Loop:
- p1 = p2 = p3 = p4 = p5 = p6 = 0
- b = load(a) if T
- p1 = cmpp.ON (b < 0) if T
- p2 = cmpp.ON (b >= 0) if T
- p6 = cmpp.ON (b < 0) if T
- p3 = cmpp.ON (c > 0) if p1
- p5 = cmpp.ON (c <= 0) if p1
- p4 = cmpp.ON (c > 0) if p3
- p5 = cmpp.ON (b <= 13) if p3
- b = b + 1 if p4
- c = c + 1 if p5
- d = c + 1 if p4
- e = c + 1 if p5
- z = z + 1
- a = a + 1 if p6
- bge e, 34, Done if p6
- jump Loop if T

Done:

### Class Problem

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. Compute CD
c. If-convert the code