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

```
add a, b, c      BB1
bgt a, 0, L1    BB1
div e, f, g     BB3
jump L2         BB3
```

**Traditional branching code**

```
p2 → BB2

add a, b, c if T BB1
bgt a, 0 if T    BB1
p2 = a > 0 if T  BB1
p3 = a <= 0 if T BB1
```

```
p3 → BB3

div e, f, g if p3 BB3
add e, f, g if p3 BB3
sub h, i, j if T BB4
```

**Predicated code**

```
p2 → BB2

add a, b, c if T BB1
bgt a, 0 if T    BB1
p2 = a > 0 if T  BB1
p3 = a <= 0 if T BB1
```

```
p3 → BB3

div e, f, g if p3 BB3
add e, f, g if p3 BB3
sub h, i, j if T BB4
```

```
p2 → BB2

add a, b, c if T BB1
bgt a, 0 if T    BB1
p2 = a > 0 if T  BB1
```

```
p3 → BB3

div e, f, g if p3 BB3
add e, f, g if p3 BB3
sub h, i, j if T BB4
```

```
What About Nested if-then-else's?

\[ a = b + c \]
if \((a > 0)\)
if \((a > 25)\)
\[ e = f + g \]
else
\[ e = f \times g \]
else
\[ e = f / g \]
\[ h = i - j \]

Traditional branching code

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

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

\[
p_1, p_2 = \text{CMPP}.\text{cond}.D_1a.D_2a (r_1, r_2) \text{ if } p_3
\]

\[
p_1 = \text{first destination predicate} \\
p_2 = \text{second destination predicate} \\
\text{cond} = \text{compare condition (ie EQ, LT, GE, ...)} \\
D_1a = \text{action specifier for first destination} \\
D_2a = \text{action specifier for second destination} \\
(r_1,r_2) = \text{data inputs to be compared (ie } r_1 < r_2) \\
p_3 = \text{guarding predicate}
\]

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>-</td>
<td>1</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
</tbody>
</table>

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
Use of OR-type Predicates

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

Traditional branching code

```
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
```

Predicated code

```
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
```

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. Predicate the code removing all branches
• 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

Running Example – Initial State

```plaintext
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 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 → 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
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
    - O1: \( a = b + c \)
    - O2: \( d = a - e \)
  - O2 dependent on O1 (flow dependent)
- Control dependence – One operation controls the execution of another
  - O1: \( \text{blt } a, 0, \text{SKIP} \)
  - O2: \( b = c + d \)
  - SKIP:
    - O2 control dependent on O1
- Control dependence analysis derives these dependences

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
  - If not a BB, it’s a control flow edge coming out of a BB
Control Dependence Example

Control dependences
BB1:
BB2:
BB3:
BB4:
BB5:
BB6:
BB7:

Notation
positive BB number = fall-thru direction
negative BB number = taken direction

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:
BB8:
BB9:

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 – Post Dominators

<table>
<thead>
<tr>
<th>pdom</th>
<th>ipdom</th>
</tr>
</thead>
<tbody>
<tr>
<td>BB1</td>
<td>1, 9, ex</td>
</tr>
<tr>
<td>BB2</td>
<td>2, 7, 8, 9, ex</td>
</tr>
<tr>
<td>BB3</td>
<td>3, 9, ex</td>
</tr>
<tr>
<td>BB4</td>
<td>4, 7, 8, 9, ex</td>
</tr>
<tr>
<td>BB5</td>
<td>5, 7, 8, 9, ex</td>
</tr>
<tr>
<td>BB6</td>
<td>6, 7, 8, 9, ex</td>
</tr>
<tr>
<td>BB7</td>
<td>7, 8, 9, ex</td>
</tr>
<tr>
<td>BB8</td>
<td>8, 9, ex</td>
</tr>
<tr>
<td>BB9</td>
<td>9, ex</td>
</tr>
</tbody>
</table>

### Running Example – CDs Via Algorithm

1. $2 \rightarrow 2$ edge (aka $-1$)

   - $x = 1$
   - $c =$ taken edge $1 \rightarrow 2$
   - $y = 2$
   - $y$ not in $pdom(x)$
   - $lub = 9$
   - $x_{id} = -1$
   - $c = 2$
     - $2 \Rightarrow 9$
     - $cd(2) =$-1
     - $t = 7$
   - $7 \Rightarrow 9$
     - $cd(7) =$-1
     - $t = 8$
   - $8 \Rightarrow 9$
     - $cd(8) =$-1
     - $t = 9$
     - $9 \Rightarrow 9$

### Running Example – CDs Via Algorithm (2)

3. $8 \rightarrow 3$ edge (aka $-3$)

   - $x = 3$
   - $c =$ taken edge $3 \rightarrow 8$
   - $y = 8$
   - $y$ not in $pdom(x)$
   - $lub = 9$
   - $x_{id} = -3$
   - $t = 8$
     - $8 \Rightarrow 9$
     - $cd(8) =$-3
     - $t = 9$
     - $9 \Rightarrow 9$

Class Problem: $1 \rightarrow 3$ edge (aka 1)
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
  - $R(bb) = \text{predicate that represents CD set for } bb, \text{ ie the bb's assigned predicate (all ops in that bb guarded by } R(bb))$

\[
K = \{\{-1\}, \{1\}, \{-2\}, \{-4\}, \{2,4\}, \{-1,-3\}\}
\]

\[
\text{predicates} = \{p_1, p_2, p_3, p_4, p_5, p_6\}
\]

\[
\begin{align*}
bb &= \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \\
\text{CD}(bb) &= \{\text{none}, \{-1\}, \{1\}, \{-2\}, \{-4\}, \{2,4\}, \{-1\}, \{-1,-3\}, \{\text{none}\}\} \\
R(bb) &= T \quad p_1 \quad p_2 \quad p_3 \quad p_4 \quad p_5 \quad p_6 \quad T
\end{align*}
\]
Running Example – CMPP Creation

Control Flow Substitution – The Rest
Running Example – Control Flow Substitution

Loop:
\[ \begin{align*}
p_1 &= p_2 = p_3 = p_4 = p_5 = p_6 = 0 \\
b &= \text{load(a)} \text{ if T} \\
p_1 &= \text{cmpp.ON (b < 0) if T} \\
p_2 &= \text{cmpp.ON (b >= 0) if T} \\
p_6 &= \text{cmpp.ON (b < 0) if T} \\
p_3 &= \text{cmpp.ON (c > 0) if p1} \\
p_5 &= \text{cmpp.ON (c <= 0) if p1} \\
p_4 &= \text{cmpp.ON (b > 13) if p3} \\
p_5 &= \text{cmpp.ON (b <= 13) if p3} \\
b &= b + 1 \text{ if p4} \\
c &= c + 1 \text{ if p5} \\
d &= d + 1 \text{ if p1} \\
p_6 &= \text{cmpp.ON (c <= 25) if p2} \\
e &= e + 1 \text{ if p2} \\
a &= a + 1 \text{ if p6} \\
bge e, 34, Done if p6 \\
jump Loop if T \\
Done: \end{align*} \]

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 predicate
    - Same or opposite compare conditions

Running Example - CMPP Compaction

Loop:
\[ \begin{align*}
p_5 &= p_6 = 0 \\
b &= \text{load(a)} \text{ if T} \\
p_1, p_2 &= \text{cmpp.UN.UC (b < 0) if T} \\
p_6 &= \text{cmpp.ON (b > 0) if T} \\
p_3, p_5 &= \text{cmpp.UN.UC (b > 13) if p1} \\
p_4, p_5 &= \text{cmpp.UN.UC (b <= 13) if p3} \\
b &= b + 1 \text{ if p4} \\
c &= c + 1 \text{ if p5} \\
d &= d + 1 \text{ if p1} \\
p_6 &= \text{cmpp.ON (c <= 25) if p2} \\
e &= e + 1 \text{ if p2} \\
a &= a + 1 \text{ if p6} \\
bge e, 34, Done if p6 \\
jump Loop if T \\
Done: \end{align*} \]
```c
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