Avoiding Conditional Branches
by Code Replication

by
Frank Mueller
Humboldt-Universität zu Berlin

David Whalley
Florida State University
Related Work

• Improving Performance via Code Replication
  — Inlining: Holler and Davidson
  — Loop Unrolling/Software Pipelining
  — Avoiding Pipeline Stalls: Golumbic and Rainish
  — Avoiding Unconditional Jumps: Mueller and Whalley

• Avoiding Conditional Branches via a Superoptimizer: Granlund and Kenner

• Branch Correlation: Krall, Young and Smith
Example of Avoiding Branches

- Conditional branches can often be avoided.

```
flag = 1;
while (cnd1 && flag) {
    A;
    if (cnd2) {
        B;
        flag = 0;
    }
    C;
}

flag = 1;
if (cnd1 && flag)
    do {
        A;
        if (cnd2) {
            B;
            flag = 0;
        }
        C;
    }
while (cnd1);
```
Overview

• Determine potentially avoidable branches.
• Restructure control flow.
• Compress restructured control flow.
• Replicate and position restructured code.
Determining Where Branches Are Affected

- Which registers and variables are affected in a basic block?
- Which registers and variables affect a comparison? Expand the comparison to find out.

```plaintext
r[1]=HI[_g];  /* sethi %hi(_g),%g1 */
r[8]=R[r[1]+LO[_g]];  /* ld [%g1+%lo(_g)],%o0 */
IC=r[8]?5;  /* cmp %o0,5 */
PC=IC<0,L20;  /* bl L20 */
=>
IC=R[HI[_g]+LO[_g]]?5;
```
Finding Avoidable Branches Algorithm

- The iterative algorithm below determines which branches can potentially be avoided.

DO
  FOR each block B in the loop DO
    B->in := NULL.
    FOR each immediate predecessor P of B DO
      B->in := B->in ∪ P->out.
      IF P contains a branch THEN
        B->in := B->in ∪ (any branches that the transition from P to B subsumes).
      END IF
    END FOR
    B->out := B->in - (the branches that B affects).
    B->out := B->out ∪ (the branches made known by the effects in B).
    IF B contains a branch THEN
      B->out := B->out ∪ B.
    END IF
  END FOR
END FOR
WHILE any changes
Example of Potentially Known Branches

Original Loop

- **1**
- **2**
  - F
  - J
  - 2 <= **6**
- **3**
  - F
  - J
- **4**
- **5**
  - F
  - 7 <= **7**
  - J
  - **8**

Potentially Known Branches

- 2 in: 2,3,7 2 out: 2,3,7
- 3 in: 2,3,7 3 out: 2,3,7
- 4 in: 2,3,7 4 out: 2,7
- 5 in: 2,3,7 5 out: 2,3,7
- 6 in: 2,3,7 6 out: 3,7
- 7 in: 2,3,7 7 out: 2,3,7
Conditional Branch States

• Each block is assigned a state for each avoidable branch.
  — unknown
  — fall-through
  — branch

• unknown state when
  — branch not yet encountered
  — branch unpredictably affected

• known state when
  — immediate predecessor block contains the branch
  — branch predictably affected
## Decidable Effects on Branches

- This table depicts three ways that some other block can cause a branch result to become known.

<table>
<thead>
<tr>
<th>Case</th>
<th>Decidable Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.</td>
<td>Avoidable Branch</td>
</tr>
<tr>
<td></td>
<td>Other Block</td>
</tr>
<tr>
<td>II.</td>
<td>Avoidable Branch</td>
</tr>
<tr>
<td></td>
<td>Other Block</td>
</tr>
<tr>
<td>III.</td>
<td>Avoidable Branch</td>
</tr>
<tr>
<td></td>
<td>Other Block</td>
</tr>
</tbody>
</table>
Subsumption Jump Requirements

- Cases when the result of one branch can cause another branch to be taken.

<table>
<thead>
<tr>
<th>known result</th>
<th>subsumable branch</th>
<th>jump requirement</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v = c_1 )</td>
<td>( v = c_2 )</td>
<td>( c_1 = c_2 )</td>
<td>( v = 10 \rightarrow v = 10 ) since ( 10 = 10 )</td>
</tr>
<tr>
<td></td>
<td>( v \neq c_2 )</td>
<td>( c_1 \neq c_2 )</td>
<td>( v = 10 \rightarrow v \neq 15 ) since ( 10 \neq 15 )</td>
</tr>
<tr>
<td></td>
<td>( v \text{ rel2 } c_2 )</td>
<td>( c_1 \text{ rel2 } c_2 )</td>
<td>( v = 10 \rightarrow v &lt; 20 ) since ( 10 &lt; 20 )</td>
</tr>
<tr>
<td>( v \neq c_1 )</td>
<td>( v = c_2 )</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>( v \neq c_2 )</td>
<td>( c_1 = c_2 )</td>
<td>( v \neq 10 \rightarrow v \neq 10 ) since ( 10 = 10 )</td>
</tr>
<tr>
<td>( v \text{ rel1 } c_1 )</td>
<td>( v \text{ rel2 } c_2 )</td>
<td>( \text{addeq(rel1)} = \text{addeq(rel2)} ) &amp;&amp; ( c_1* \text{ addeq(rel1)} c_2* )</td>
<td>( v \geq 11 \rightarrow v &gt; 10 ) since ( \geq ) ( \geq ) &amp;&amp; ( 11 \geq 10+1 )</td>
</tr>
<tr>
<td></td>
<td>( v = c_2 )</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>( v \neq c_2 )</td>
<td>( c_1 \text{ noeq(rel1)} c_2 )</td>
<td>( v \geq 20 \rightarrow v \neq 10 ) since ( 20 &gt; 10 )</td>
</tr>
</tbody>
</table>
Subsumption Fall-Through Requirements

- Cases when the result of one branch can cause another branch to not be taken.

<table>
<thead>
<tr>
<th>known result</th>
<th>subsumable branch</th>
<th>fall through requirement</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>v = c1</td>
<td>v = c2</td>
<td>c1 ≠ c2</td>
<td>v = 10 → ¬(v = 15) since 10 ≠ 15</td>
</tr>
<tr>
<td>v ≠ c2</td>
<td>c1 = c2</td>
<td></td>
<td>v = 10 → ¬(v ≠ 10) since 10 = 10</td>
</tr>
<tr>
<td>v rel2 c2</td>
<td>¬(c1 rel2 c2)</td>
<td></td>
<td>v = 10 → ¬(v &gt; 20) since ¬(10 &gt; 20)</td>
</tr>
<tr>
<td>v ≠ c1</td>
<td>v = c2</td>
<td>c1 = c2</td>
<td>v ≠ 10 → ¬(v = 10) since 10 = 10</td>
</tr>
<tr>
<td>v ≠ c2</td>
<td>N/A</td>
<td></td>
<td>N/A</td>
</tr>
<tr>
<td>v rel1 c1</td>
<td>v rel2 c2</td>
<td>opp(noeq(rel1), noeq(rel2)) &amp; &amp; ¬(c1* addeq(rel2) c2*)</td>
<td>v ≥ 10 → ¬(v &lt; 10) since opp(‘&gt;’, ’&lt;’) &amp; &amp; ¬(10 ≤ 10-1)</td>
</tr>
<tr>
<td>v = c2</td>
<td>c1 noeq(rel1) c2</td>
<td></td>
<td>v ≥ 20 → ¬(v = 10) since 20 &gt; 10</td>
</tr>
<tr>
<td>v ≠ c2</td>
<td>N/A</td>
<td></td>
<td>N/A</td>
</tr>
</tbody>
</table>
Restructuring Algorithm

- The iterative algorithm produces a dummy control-flow graph.

Set the initial dummy node to be the header of the original loop with a state of \textit{unknown} for all avoidable branches. Set the current dummy node to be this initial node. WHILE there are dummy nodes to process DO
  FOR each successor of the current dummy node DO
    Calculate the state of the successor.
    IF a node associated with the successor exists with the same exact state for all avoidable branches THEN
      Connect the current dummy node to that existing node.
    ELSE
      Create a new dummy node with this state, connect the current dummy node to it, and append it to list of dummy nodes.
    END IF
  END IF
END FOR
Advance to the next dummy node to be processed.
END WHILE
Example of Restructuring a Loop

Original Loop

Restructured Control Flow
Avoiding Branches Not in Innermost Loops

• The same algorithm was applied to levels of a function other than innermost loops.
  — Loops within a function are restructured in the order of most deeply nested first.
  — An inner loop is treated like a single basic block when restructuring an outer loop.
  — The outermost level of a function is treated as a loop with no backedges.
Compression Algorithm

• The iterative algorithm eliminates unnecessary nodes.

FOR each node in the dummy graph DO
    IF the node contains a branch and has a known state
        for that branch THEN
        Mark the node as reaching that branch.
    END IF
END FOR

DO
    FOR each node in the dummy graph DO
        IF the node has a known state for a branch and
            an immediate successor reaches that branch THEN
            Mark the node as reaching that branch.
        END IF
    END FOR
END DO

WHILE any changes
    FOR each node in the dummy graph DO
        Set the state of the node as unknown for any branch
            that is not marked as having been reached.
        IF another instance of the node exists
            with the same state THEN
            Delete the node and adjust the transitions in the graph.
        END IF
    END FOR
END FOR
Replication and Positioning of the Restructured Loop

- Heuristics to limit code expansion.

- New loop generated by replication, control flow is adjusted, and avoidable branches are eliminated.

- Blocks are positioned to reduce the number of unconditional jumps.

- Reapply other optimizations (dead code elimination).
Example of Positioning the Restructured Code
Positioning Algorithm

• The recursive algorithm positions the restructured code and attempts to reduce the frequency of unconditional jumps.

PROCEDURE order(List, B, S-List)
  IF B not marked as done AND
  none of the members of S-List dominate B THEN
    IF B is header of loop L AND there exists an
    unmarked successor of an exit block in L THEN
      B := unmarked successor of this exit block in L.
    END IF
  Mark B as done.
  S-list := successors of B ordered by loop frequency.
  WHILE S-list not empty DO
    S := head of S-list.
    S-list := tail of S-list.
    order(List, S, S-list).
  END WHILE
  Insert B at the head of List.
  END IF
END PROCEDURE
## Static Results

- Test programs and increase in code size.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Static Instructions Total</th>
<th>Static Instructions Restruct</th>
</tr>
</thead>
<tbody>
<tr>
<td>banner</td>
<td>banner generator</td>
<td>+18.24%</td>
<td>+170.59%</td>
</tr>
<tr>
<td>cacheall</td>
<td>cache simulator</td>
<td>+3.08%</td>
<td>+21.89%</td>
</tr>
<tr>
<td>cal</td>
<td>calendar generator</td>
<td>+21.51%</td>
<td>+120.97%</td>
</tr>
<tr>
<td>ctags</td>
<td>C tags generator</td>
<td>+10.53%</td>
<td>+35.49%</td>
</tr>
<tr>
<td>dhrystone</td>
<td>integer benchmark</td>
<td>+8.60%</td>
<td>+18.23%</td>
</tr>
<tr>
<td>join</td>
<td>relational join files</td>
<td>+6.65%</td>
<td>+17.09%</td>
</tr>
<tr>
<td>od</td>
<td>octal dump</td>
<td>+36.32%</td>
<td>+129.09%</td>
</tr>
<tr>
<td>sched</td>
<td>instruction scheduler</td>
<td>+34.25%</td>
<td>+87.02%</td>
</tr>
<tr>
<td>sdiff</td>
<td>side-by-side file diffs</td>
<td>+1.25%</td>
<td>+3.78%</td>
</tr>
<tr>
<td>wc</td>
<td>word counter</td>
<td>+39.22%</td>
<td>+172.73%</td>
</tr>
<tr>
<td>whetstone</td>
<td>FP benchmark</td>
<td>-0.89%</td>
<td>-8.74%</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td>+16.25%</td>
<td>+69.83%</td>
</tr>
</tbody>
</table>
Sources for Avoiding Branches

- **Not Affected**: A path exists from a branch back to the same branch without it being affected.

- **Constant Comparison**: An effect in another block causes the branch result to become known.

- **Same Direction**: An effect will not change the state of a branch given that the branch already has a specific result.

- **Subsumption**: The result of one branch may indicate the direction that a different branch will take.

![Pie Chart]

- 35.65% Not Affected
- 40.30% Constant Comparison
- 3.34% Same Direction
- 20.71% Subsumption
Future Work

• Restructuring across loop boundaries.
• Restructuring code containing indirect jumps.
• Use interprocedural and pointer analysis to avoid additional branches.
• Use profiling.
Future Work (Cont.)

- Investigate other methods for determining when the result of one branch subsumes another. Consider a machine that core dumps when a memory reference is performed at address zero. The second `if` statement below could be avoided when the first `if` statement is not entered.

<table>
<thead>
<tr>
<th>C Source Code Segment</th>
<th>Corresponding RTLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (p-&gt;value == value) {</td>
<td>( r[1] = R[r[2]] ; )</td>
</tr>
<tr>
<td>\hspace{1em} ...</td>
<td>( IC = r[1] ? r[3] ; )</td>
</tr>
<tr>
<td>\hspace{1em} p = p-&gt;next;</td>
<td>( PC = IC == 0, L10 ; )</td>
</tr>
<tr>
<td>}</td>
<td>( \ldots )</td>
</tr>
<tr>
<td>L10: IC=r[2]?0;</td>
<td>L10:</td>
</tr>
</tbody>
</table>
Conclusions

• Avoiding conditional branches by replicating code
  — relatively simple optimization to implement
  — can be frequently applied
  — significant performance improvements for the restructured code portions