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.

ORIGINAL

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

AFTER RESTRUCTURING

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

```
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;
```

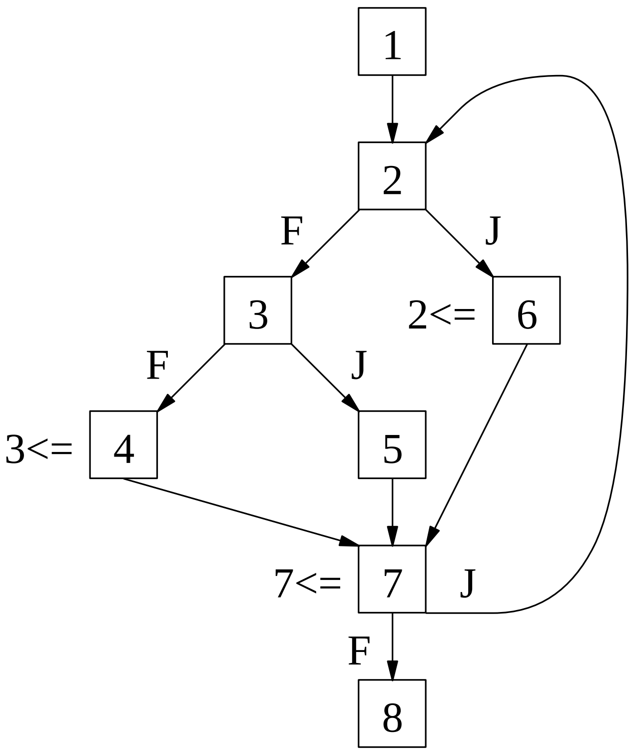
Finding Avoidable Branches Algorithm

- 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
WHILE any changes
```

Example of Potentially Known Branches

Original Loop



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.

Case	Decidable Effect	
I.	Avoidable Branch	`IC=r[8]?0;   /* cmp %o0,0 */` `PC=IC==0,L1; /* be L1 */`
	Other Block	`r[8]=-1;     /* move -1,%o0 */`
II.	Avoidable Branch	`IC=r[2]?50;  /* cmp %g2,50 */` `PC=IC>0,L2;  /* bg L2 */`
	Other Block	`r[2]=r[2]+1; /* add 1,%g2,%g2 */`
III.	Avoidable Branch	`IC=r[2]?76;  /* cmp %g2,76 */` `PC=IC>0,L4;  /* bg L4 */`
	Other Block	`IC=r[2]?83;  /* cmp %g2,83 */` `PC=IC<=0,L3; /* ble L3 */`

Subsumption Jump Requirements

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

known result	subsumable branch	jump requirement	example
v = c1	v = c2	c1 = c2	v = 10 → v = 10 since 10 = 10
	v ≠ c2	c1 ≠ c2	v = 10 → v ≠ 15 since 10 ≠ 15
	v rel2 c2	c1 rel2 c2	v = 10 → v < 20 since 10 < 20
v ≠ c1	v = c2	N/A	N/A
	v ≠ c2	c1 = c2	v ≠ 10 → v ≠ 10 since 10 = 10
v rel1 c1	v rel2 c2	addeq(rel1) = addeq(rel2) && c1* addeq(rel1) c2*	v ≥ 11 → v > 10 since ’≥’ = ’≥’ && 11 ≥ 10+1
	v = c2	N/A	N/A
	v ≠ c2	c1 noeq(rel1) c2	v ≥ 20 → v ≠ 10 since 20 > 10

Subsumption Fall-Through Requirements

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

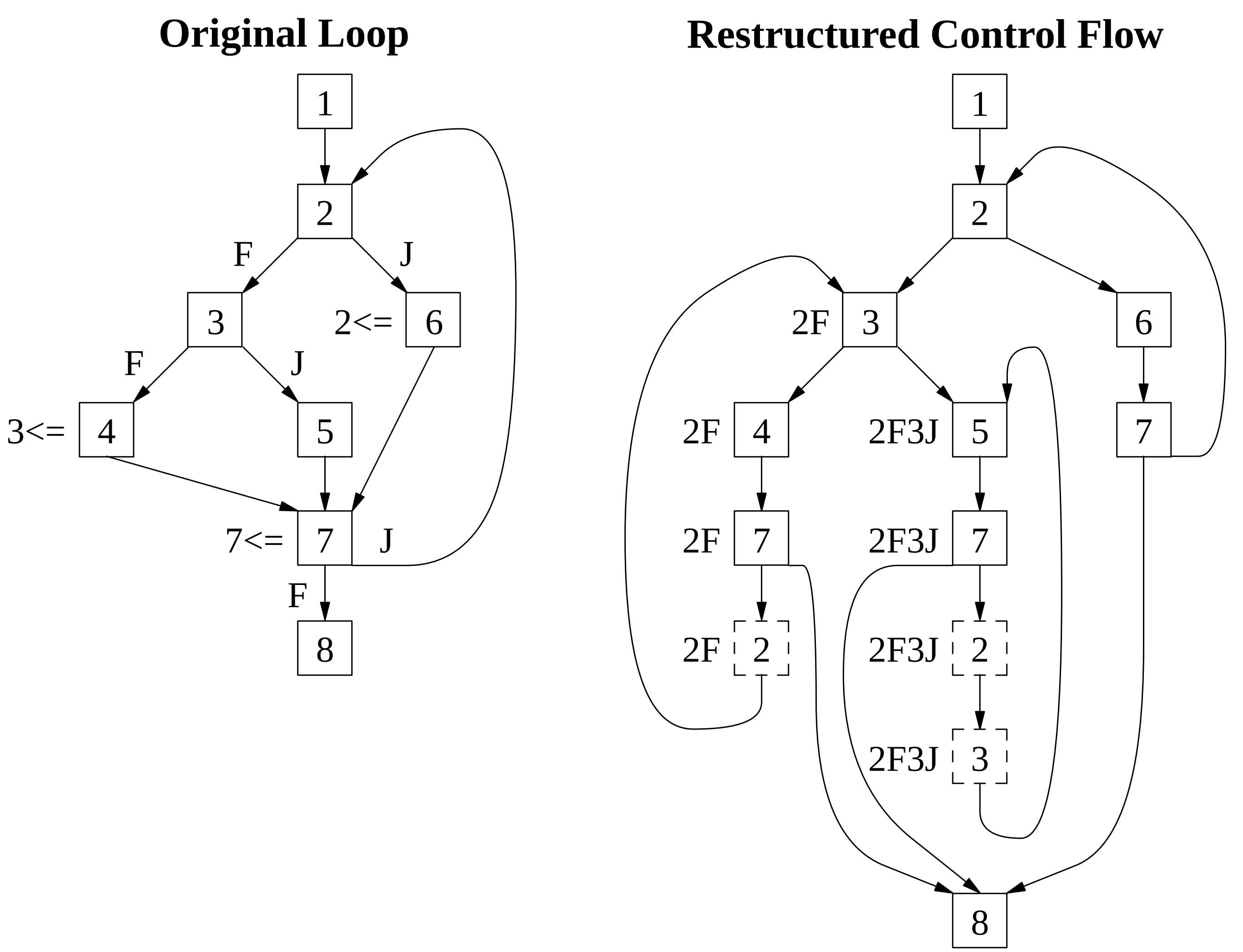
known result	subsumable branch	fall through requirement	example
v = c1	v = c2	c1 ≠ c2	v = 10 → ¬(v = 15) since 10 ≠ 15
	v ≠ c2	c1 = c2	v = 10 → ¬(v ≠ 10) since 10 = 10
	v rel2 c2	¬(c1 rel2 c2)	v = 10 → ¬(v > 20) since ¬(10 > 20)
v ≠ c1	v = c2	c1 = c2	v ≠ 10 → ¬(v = 10) since 10 = 10
	v ≠ c2	N/A	N/A
v rel1 c1	v rel2 c2	opp(noeq(rel1), noeq(rel2)) && ¬(c1* addeq(rel2) c2*)	v ≥ 10 → ¬(v < 10) since opp('>', '<') && ¬(10 ≤ 10-1)
	v = c2	c1 noeq(rel1) c2	v ≥ 20 → ¬(v = 10) since 20 > 10
	v ≠ c2	N/A	N/A

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 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 FOR
        Advance to the next dummy node to be processed.
END WHILE
```

Example of Restructuring a Loop



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.

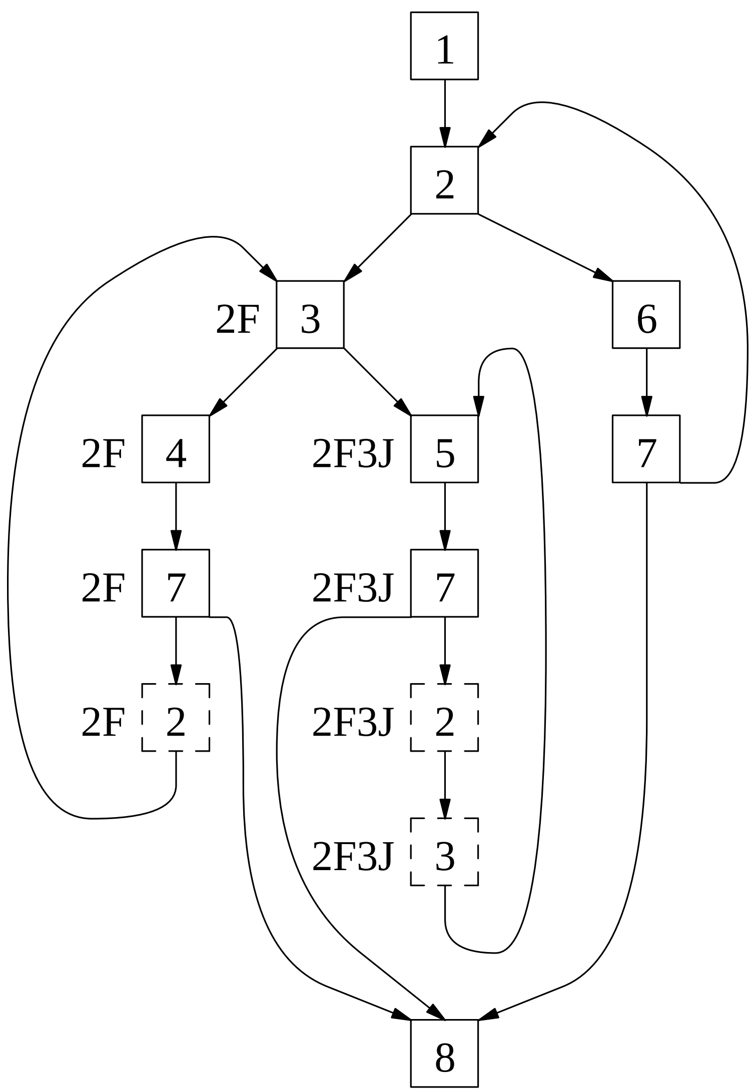
```
FOR each node in the dummy graph DO
        IF the node contains a branch and has a known state
              for that branch THEN
              Mark the node as reaching that branch.
        END IF
END FOR
DO
      FOR each node in the dummy graph DO
            IF the node has a known state for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

Replication and Positioning of the Restructured Loop

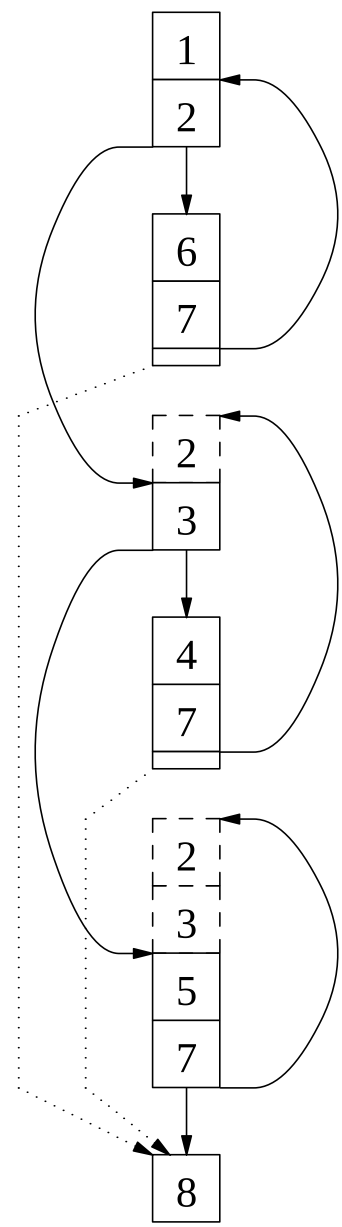
- Heuristics to limit code expansion.
- New loop generated by replication, control flow is adjusted, and avoidable branches are eliminated.
- Blocks are positioned to reduce the number of unconditional jumps.
- Reapply other optimizations (dead code elimination).

Example of Positioning the Restructured Code

Restructured Control Flow



Positioned 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
```

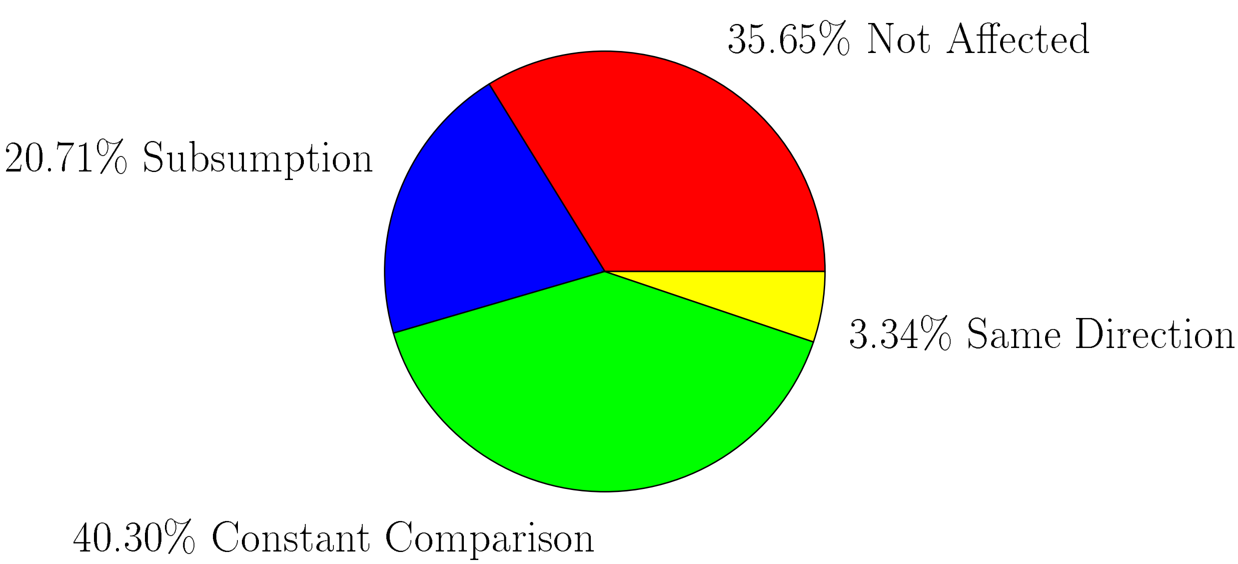
Static Results

- Test programs and increase in code size.

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

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.



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.

```
C Source Code Segment            Corresponding RTLs
----------------------          --------------------
if (p->value == value) {              r[1]=R[r[2]];
   ...                               IC=r[1]?r[3];
   p = p->next;                      PC=IC==0,L10;
   }                                 ...
if (p)                               r[2]=R[r[2]+4];
                                L10: IC=r[2]?0;
                                     PC=IC!=0,L20;
```

Conclusions

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