Avoiding Conditional Branches by Code Replication

FRANK MUELLER
Fachbereich Informatik
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany

DAVID B. WHALLEY
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019, U.S.A.
e-mail: whalley@cs.fsu.edu
phone: (904) 644-3506

SUMMARY

On-chip instruction caches are increasing in size. Compiler writers are exploiting this fact by applying a variety of optimizations that improve the execution performance of a program at the expense of increasing its code size. This paper describes a new optimization that can be used to avoid conditional branches by replicating code. The central idea is to determine if there are paths where the result of a conditional branch will be known and to replicate code to exploit it. Algorithms are described for detecting when branches are avoidable, for restructuring the control flow to avoid these branches, and for positioning the replicated blocks in the restructured code. The results indicate that the optimization can be frequently applied with reductions in both the number of instructions executed and total instruction cache work.

INTRODUCTION

This paper describes a new approach for avoiding conditional branches by using code replication. This approach is accomplished in three steps for each loop level of a function. First, analysis is performed to determine whether any conditional branches can be avoided. Second, an algorithm is used to determine how the control flow can be restructured to avoid conditional branches by replicating basic blocks. At this point heuristics are used to determine whether the transformation is worthwhile. Finally, new code is replicated and the physical position of the basic blocks is established.

RELATED WORK

There are several optimizations that have been developed in an attempt to improve performance despite the penalty of increasing code size. Inlining replaces a call by the body of the routine being invoked. Increased code size results when a routine is inlined from more than one call site. Execution performance benefits often occur since the call and return are avoided and more information is available for other optimizations [DaH88]. Loop unrolling replicates the body within a loop. This optimization reduces the number of compare and conditional branch instructions executed by the loop and can result in more effective scheduling of instructions within the loop body [HeP90]. Replicating portions of basic blocks has been performed to avoid pipeline stalls for superscalar machines [GoR90]. Software pipelining replicates code associated with loops to produce a revised kernel of the loop that has fewer pipeline stalls [Jai91].

Detecting the dependences between loop iterations has been described by Banerjee [Ban93]. This analysis has been used to reorder loop iterations to enhance performance for vectorization, parallelization, and caching.

Mueller and Whalley investigated avoiding unconditional jumps by code replication [MuW92]. Unconditional jumps were replaced with a replicated sequence of instructions that either reaches a return or falls into the block that positionally follows the original unconditional jump. The growth in code size was minimized by choosing the shortest path for the replacement.

A superoptimizer will generate an exhaustive set of bounded sequences of instructions with the goal of finding a sequence that will produce the same effect as a more expensive sequence of instructions. The more expensive sequence can then be recognized in a traditional optimizer and replaced with the less expensive sequence. This technique has been used to eliminate conditional branches over short instruction sequences in many instances on the IBM RS/6000 [GrK92].

The optimization described in this paper can be viewed as partial redundancy elimination (PRE) of conditional branches. PRE traditionally places copies of a computation at other points in the control flow to force the original copy to become fully redundant so it can be deleted [MoR79]. The transformation described in this paper differs since it involves restructuring the control flow by replicating entire sequences of basic blocks.

The research described in the current paper was inspired by the work of Krall [Kra94]. Similar work was later published by Young and Smith [YoS94]. Krall often found a high correlation between the past results of branches and the future results of the same or different branches in a loop. This correlation typically increased when code was replicated in the loop to distinguish between different loop paths. The authors of this paper suspected that the high correlation observed by Krall may indicate that many paths in the replicated code were not possible due to data dependencies. Thus, many branches in the replicated code could potentially be avoided.

MOTIVATION

While it has been shown that most unconditional jumps can be eliminated by code replication [MuW92], it is obvious that fewer conditional branches can be avoided. One may be inclined to believe that such transformations would only rarely be possible. However, there are many common instances in programs that can be improved. The examples in this section are given in C code to more concisely depict the transformations performed. The control flow of the restructured C code segments would be comparable to a restructured flow graph of basic blocks. Note that the actual implementation of this general optimization was performed in the back end of a compiler and many other types of transformation instances were applied.

I. Often, a flag is used by a programmer to exit a loop. For example, consider the following segments of code. The original loop will exit if either a general condition or a flag is false. The code to test the `flag` variable is not within the loop after restructuring since the replication of the code associated with statement "`C;`" allows this test to be avoided. The flag will be zero or nonzero depending upon the initial path taken and all subsequent tests of the variable `flag` are eliminated. Other optimizations that could be applied to further improve the code are depicted by italicizing the code portions that can be removed. The test of `cnd1` after `flag` is set to zero can be removed since the result of the conditional branch has no effect. The initial test of the `flag` variable would be avoided since the comparison is known to be true. In addition, the remaining assignments to `flag` can be removed, assuming that the value is not used after the loop. Finally, space for the local `flag` variable would not be needed given that this was the only place in the function in which it was used.

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);
```

II. Often, conditions may be tested that will result in a conditional branch always branching or always falling through once the branch has a certain result. For instance, consider the following code segments. The left code segment contains a simple loop that will exit when both conditions are false. Assume that the variable `i` is only incremented in the loop. If the condition `i < 100` is ever false, then it will always remain false for the remainder of the loop. The right code segment shows the testing of the first condition can be avoided after it becomes false.

ORIGINAL

```
while (i < 100 || somecnd) {
   A;
   i++;
   }
```

AFTER RESTRUCTURING

```
while (i < 100) {
   A;
   i++;
   }
while (somecnd) {
   A;
   i++;
   }
```

Similar behavior may be predicted for equality and inequality tests. Given that a variable is only incremented or only decremented for every iteration of a loop, an equality test of that variable with a loop invariant value will only be true at most once for the execution of the entire loop. Likewise, a test for inequality of that variable will only be false at most once.

III. A programmer may also often repeat conditions in `if` statements to improve readability. For instance, checking whether a pointer is not NULL may be done several times in different `if` statements in the same loop. Consider the following segments of code. Assume that the variable `p` in the original code segment is not affected by the code in `B`

ORIGINAL

```
while (somecnd) {
   A;
   if (p && *p == val1)
      B;
   else
      C;
   if (p && *p == val2)
      D;
   else
      E;
   F;
   }
```

AFTER RESTRUCTURING

```
while (somecnd) {
   A;
   if (p) {
      if (*p == val1)
         B;
      else
         C;
      if (*p == val2)
         D;
      else
         goto doE;
      }
   else {
      C;
doE:  E;
      }
   F;
   }
```

and C, but is affected by the code in A or F. The test of p in the second if statement can be avoided by replicating the code in C.

IV. An invariant condition may be tested inside of a loop. In fact, loop invariant conditions are often generated due to applying other optimizations. Consider the nested loops in the following left code segment. It appears there are no apparent invariant conditions. Yet, the inner loop will be transformed to avoid executing an unconditional jump at the end of the loop body on each iteration. As shown in the middle code segment, the test condition is also tested initially before the loop is entered. This condition is loop invariant in the outer loop and can be avoided by replicating code as shown in the right code segment.

```
  ORIGINAL      AFTER OPTIMIZATION     RESTRUCTURED
do {            do {                 A;
   A;              A;                i=0;
   for (i=0;       i=0;              if (0<N) {
        i<N;       if (0<N)             goto doB;
        i++)          do {              do {
      B;                 B;                A;
   C;                    i++;              i=0;
   }                     }                 do {
while (cnd);          while (i<N);       doB: B;
                  C;                          i++;
                  }                           }
               while (cnd);                 while (i<N);
                                            C;
                                            }
                                         while (cnd);
                                         }
                                      else {
                                         goto doC;
                                         do {
                                            A;
                                       doC: C;
                                            }
                                         while (cnd);
                                         }
```

A more traditional technique, called *unswitching*, would simply test the loop invariant condition initially and then enter one of two loops based on the result [LRS76]. Note that the total code replicated is greater in the general approach described in this paper than it would be for *unswitching*. This increase is due to only testing the invariant condition when it needs to be executed. Unlike *unswitching*, the general approach described in this paper will never increase the dynamic number of conditional branches executed.

DETERMINING WHETHER BRANCHES CAN BE AVOIDED

Analysis is performed to determine whether the conditional branches in a loop can be avoided by replicating code. The compiler first calculates the set of registers and variables upon which a conditional branch (and its associated compare instruction) depends. This set was calculated by expanding the effects of the compare instruction associated with the conditional branch. For instance, consider the following SPARC instructions represented as RTLs (Register Transfer Lists).

```
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                */
```

The effect of the comparison can be expanded to:

```
IC=R[HI[_g]+LO[_g]]?5;
```

In addition, for each basic block in the loop the compiler determines the set of registers and variables that are affected by instructions within the current block. Thus, the compiler can determine that a basic block updating the global variable g could affect the result of this conditional branch. Updates to the registers r[1] (%g1) or r[8] (%o0) would have no effect.

The compiler next attempts to determine if there exists a path through a loop from the point immediately after a conditional branch is encountered to the same branch without the comparison associated with the branch being affected. If a conditional branch is not affected in a path in which it is encountered, then that same path could be taken again and the result of executing the conditional branch would not change.

There are cases where a conditional branch could be affected in each path it is encountered, but still could be avoided. One instance is when a basic block can affect a conditional branch, but only if the result of the branch had not already been in a specific direction (as depicted in Example II in the motivation section). Detecting this situation requires remembering whether each branch was last taken or not.

The processing of a block may also make the result of a conditional branch known at that point. This situation may occur as the result of updating a variable or register (as depicted in Example I in the motivation section) or when the result of one conditional branch subsumes another (as shown in Example III in the motivation section). The compiler needs to detect if the conditional branch can be reached from this block without being affected again.

The algorithm for determining which branches can be potentially avoided is shown in Figure 1. Each block will have an *in* and an *out* state indicating the branches whose results are known at the beginning and end of the block.[1] A branch result can become known at a point in the flow graph due to the branch being executed, another branch being executed whose result subsumes the branch, or the effects of a block. A known branch can become

[1] The actual algorithm is a bit more complicated. Two *in* and *out* states were actually calculated for each block. Some effects will only make a branch result unknown if the branch result had been in a specific direction. In these situations, only the *out* state of the block associated with the specific direction affected will be the one updated.

unknown due to an effect within a block. A branch is potentially avoidable if it is in the block's *in* state and is not affected within the block (or is made known due to an effect within the same block).

```
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.
        B->out := B->out - (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
```

Figure 1: Finding Avoidable Branches Algorithm

An example is given in Figure 2 to illustrate the algorithm. Figure 2 (a) depicts the original loop. Blocks 2, 3, and 7 have a conditional branch. Block 4 affects the conditional branch in block 3 and block 6 affects the conditional branch in block 2. The conditional branch in block 7 is affected by other instructions in the same block. Figure 2 (b) shows the branches that are potentially known at the entry and exit point of each basic block. The branches for blocks 2 and 3 are avoidable since they are in the *in* states and not affected within their own blocks. While the branch in block 7 is in the *in* state for that block, the branch is not avoidable since it is affected within block 7. Figures 2 (c) and 2 (d) will be discussed later in the paper.

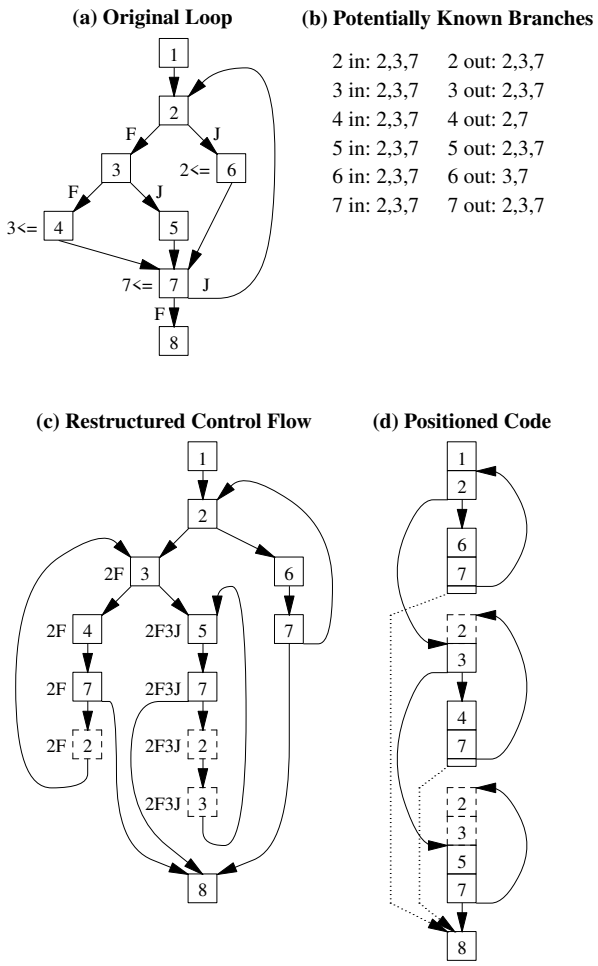

Figure 2: Restructuring a Loop to Avoid Branches

RESTRUCTURING THE CONTROL FLOW TO AVOID BRANCHES

Once it has been determined that one or more conditional branches in a loop can be avoided, the control flow within the loop can be restructured. An algorithm to accomplish the restructuring is based on keeping a state in each block for each avoidable conditional branch in the loop. A block can inherit its state from a predecessor block or change its state due to an effect within the block or due to being an immediate successor of an avoidable conditional branch. A state associated with a conditional branch can have one of three values: *unknown*, *fall-through*, or *branch*.

An *unknown* state indicates that it is not known whether or not the branch will be taken. A block will typically have this state for a conditional branch if the branch has not been encountered previously in the control flow. A block will also have an *unknown* state for a conditional branch if the block affects the conditional branch in a manner where the result of executing the branch cannot be predicted. The state for a conditional branch within a block will be set to *fall-through* or *branch* if the immediate predecessor was the block containing the conditional branch. The state to be set depends upon whether the successor is a fall through or target of the branch.

The state for a conditional branch can also be set to a *fall-through* or *branch* when an effect in another portion of the loop causes the result of the comparison to be known. Table 1 shows three such examples. Case I shows another block setting one of the operands of the expanded comparison to a constant. Thus, the result of the conditional branch can be determined to be *fall-through* at that point. Case II illustrates that the state of a conditional branch may not be changed even though a variable or register within the associated expanded comparison is updated. The other block

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 */`

Table 1: Decidable Effects on Branches

will have no effect on the result of the conditional branch when the state is already *branch*. Case III depicts a situation where one conditional branch may also subsume another conditional branch. In other words, the direction taken by one conditional branch may indicate the direction taken by another conditional branch in the same loop. Assume the instructions in the other block are executed and the branch is not taken. The avoidable conditional branch will be taken if `r[2]` is not affected between the execution of the two branches since the value in `r[2]` is guaranteed to be greater than 83. Note that the conditional branch cannot be avoided if the conditional branch in the other block is taken.

In general, a conditional branch can only be potentially subsumed by another conditional branch when one argument of each comparison is identical and the other argument of each comparison is a constant or the same invariant value. Table 2 depicts the different cases when the result of one conditional branch subsumes another branch.[2] Column 1 shows a known result from one conditional branch. This result is determined by not only the operands of the comparison and the branch relational operator, but also by whether or not the branch was taken. For instance, the known result after the conditional branch was not taken in the other block of Case III in Table 1 would be `r[2] > 83`. The second column in Table 2 depicts the condition associated with the branch to be subsumed. For instance, the subsumable condition associated with avoidable branch in Case III of Table 1 would be `r[2] > 76`. The third and fifth columns of Table 2 define the requirements for the avoidable branch to jump or fall through, respectively. A conditional branch with the condition `r[2] > 76` will jump when it is known that `r[2] > 83`, as indicated by the jump requirement in the third row from the bottom of Table 2.

The restructuring algorithm shown in Figure 3 will produce a dummy graph to efficiently represent the revised control flow of the loop. If it is later determined that the restructuring is worthwhile, then the dummy graph will be used to modify the actual control flow of the function. The central idea of the algorithm in Figure 3 is that a new node will be added when no current node for that block exists with the same set of states for the avoidable branches. Note that since each branch can have 3 states, the upper bound for the code size increase is $O(3^n)$, where n is the number of branches that can be avoided. In practice, such increases have not been observed. However, to avoid an excessive code increase, a heuristic was used to limit the value of n in a single loop. If more than n branches could be avoided in a loop, then n branches are chosen based on the likelihood of being reached from the loop header. A conditional branch in a node that has a known (*fall-through* or *branch*) state for that conditional branch will be eliminated in the restructured code. Note that the associated comparison and other dead instructions may be eliminated as well.

```
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
```

Figure 3: Restructuring Algorithm

Figure 2 (c) shows the restructured control flow (dummy graph). The state is given to the left of each block, which is depicted with the block number of the conditional branch and whether it fell through (F) or jumped (J) in the original loop. For instance, 2F3J indicates that the conditional branch in block 2 will fall through next time and the branch in block 3 will jump. A basic block represented with a dashed box indicates that the conditional branch (and typically its associated comparison) is unnecessary and will not be placed in the restructured code.

[2] Note determining that a branch can be avoided when it is encountered and not affected in a path (as shown in Figure 2) is really a case of the branch subsuming itself.

known result	subsumable branch	jump requirement	example	fall through requirement	example
v = c1	v = c2	c1 = c2	v = 10 → v = 10 since 10 = 10	c1 ≠ c2	v = 10 → ¬(v = 15) since 10 ≠ 15
	v ≠ c2	c1 ≠ c2	v = 10 → v ≠ 15 since 10 ≠ 15	c1 = c2	v = 10 → ¬(v ≠ 10) since 10 = 10
	v rel2 c2	c1 rel2 c2	v = 10 → v < 20 since 10 < 20	¬(c1 rel2 c2)	v = 10 → ¬(v > 20) since ¬(10 > 20)
v ≠ c1	v = c2	N/A	N/A	c1 = c2	v ≠ 10 → ¬(v = 10) since 10 = 10
	v ≠ c2	c1 = c2	v ≠ 10 → v ≠ 10 since 10 = 10	N/A	N/A
v rel1 c1	v rel2 c2	addeq(rel1) = addeq(rel2) && c1* addeq(rel1) c2*	v ≥ 11 → v > 10 since '≥' = '≥' && 11 ≥ 10+1	opp(noeq(rel1), noeq(rel2)) && ¬(c1* addeq(rel2) c2*)	v ≥ 10 → ¬(v < 10) since opp('>', '<') && ¬(10 ≤ 10-1)
	v = c2	N/A	N/A	c1 noeq(rel1) c2	v ≥ 20 → ¬(v = 10) since 20 > 10
	v ≠ c2	c1 noeq(rel1) c2	v ≥ 20 → v ≠ 10 since 20 > 10	N/A	N/A

where
(1) v is a variable
(2) c is a constant
(3) rel is '<', '≤', '>', or '≥'
(4) opp(rel1, rel2) returns true when (x rel1 y) && (x rel2 y) can never both be true (e.g. x > y && x < y)
(5) noeq(rel) returns the relational operator without any equality (e.g. noeq('≥') and noeq('>') both return '>')
(6) addeq(rel) returns the relational operator with an equality (e.g. addeq('≥') and addeq('>') both return '≥')
(7) c* is a constant that is adjusted by 1 in the appropriate direction if addeq(rel) != rel

Table 2: Subsumption Requirements

Note that if block 3 is reached, then the conditional branch in block 2 need never be executed for the remainder of the loop. Likewise, if block 5 is reached, then both the conditional branches in blocks 2 and 3 can be subsequently avoided.

AVOIDING BRANCHES NOT WITHIN INNERMOST LOOPS

The algorithm shown in Figure 3 was also extended to avoid branches that are not in the innermost loops of a program. The loops of a function are processed in decreasing order of their nesting level. Once an inner loop has been processed, the effects of all of its blocks are unioned before processing the next outer level loop. While processing the next outer loop, the inner loop is treated as if it were a single block. The outermost level of a function is treated as a loop with no backedges. Only branches at the current loop level are considered candidates for being avoided. Furthermore, the effects of an inner loop are not currently used to make the states of branches at outer loop levels known.

COMPRESSING THE RESTRUCTURED GRAPH

Occasionally, unnecessary nodes are introduced by the restructuring algorithm. For instance, consider the control flow of a function depicted in Figure 4 (a). Assume the conditional branch in block 2 subsumes the conditional branch in block 7. If the branch in block 2 jumps to block 7, then it is known that the branch in block 7 will transfer control to block 8. Likewise, if block 2 falls through to block 3, then the branch in block 7 will transfer control to block 9. However, the execution of block 6 affects the branch in block 7. Figure 4 (b) shows the restructured control flow using the algorithm in Figure 3. Duplicate nodes for blocks 3, 4, and 5 are generated since they have a different state for the branch in block 7. But these duplicate nodes are unnecessary since the state of the branch will be unknown upon transition to block 6.

Once the loop has been restructured, the dummy graph is then compressed to eliminate any unnecessary nodes. The algorithm for compressing the graph is shown in Figure 5. The central idea is that the state of a branch in a node will become unknown unless it can reach a node containing that branch with that state. Figure 4 (c) shows

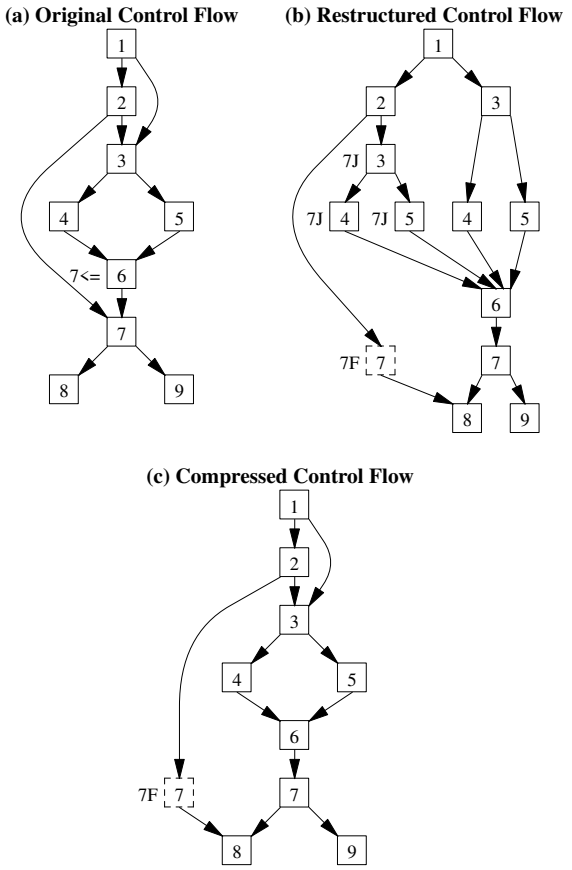

Figure 4: Eliminating Unnecessary Dummy Graph Nodes

the control flow of Figure 4 (b) after compression.[3]

REPLICATION AND POSITIONING FOR THE RESTRUCTURED CODE

Given the revised control flow represented in the dummy graph, the replication and positioning for the restructured code is accomplished in multiple stages. First, a set of heuristics are applied to determine if avoiding the conditional branches should be performed. Second, the blocks of the original loop are replicated. Third, the blocks are positionally ordered to reduce the number of unconditional jumps in the replicated code. Finally, a number of optimizations are reapplied to the function in order to exploit the simplified control flow in the restructured code.

[3] It may be possible to perform this analysis on the control flow associated with the original loop and adjust the states of the dummy nodes as they are produced in the algorithm shown in Figure 3. Thus, the last loop in Figure 5 would not be required if unnecessary nodes are never introduced.

```
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
```

Figure 5: Compression Algorithm

The first stage estimates the increase in size of the restructured code. The number of additional instructions for a loop was limited to avoid a significant increase in code size. When this limit was exceeded, the analysis was reinvoked with a decremented number of avoided branches to further reduce the amount of replicated code.

During the second stage, the restructured code is generated by replicating the corresponding blocks of the original loop and adjusting the control flow according to the dummy graph representing the revised control flow. This includes the elimination of avoidable branches and adjusting the control-flow transitions to enter and leave the restructured code instead of the original loop.

During the third stage, the loop information within the revised control flow is calculated. This information is subsequently used to adjust the positional order within the restructured code by calling the procedure *order*, which is shown in Figure 6. This recursive procedure is initially invoked with an empty list, the first block of the restructured code, and another empty list as parameters. The output of the algorithm is a list of blocks corresponding to a positional order such that unconditional jumps are avoided when possible. The algorithm attempts to reduce the number of unconditional jumps via code positioning. While the restructured control flow reduces the number of conditional branches, it also introduces replicated blocks within new loops. It is imperative to find a "good" positioning of these blocks or the benefit of avoided conditional branches may well be outweighed by the introduction of unconditional jumps to adjust the replicated control flow.

```
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
```

Figure 6: Positioning Algorithm

The algorithm works as follows. The recursive procedure *order* terminates when all blocks are marked as done. The dominator check forces the recursion to backtrack along the control flow when a block is encountered that is dominated by an unprocessed sibling block. The dominator check provides the means to position if-then-else statements (even nested ones) before any blocks following the if-then-else construct.

When a loop header is found, the algorithm follows the control flow backwards to an exit block of the loop. It then processes an unmarked successor of the exit block first. Thus, the algorithm attempts to process the exit block last, *i.e.* the exit block is positioned at the bottom of the loop. This avoids an unconditional jump at the bottom of the loop.

The successor list *S-list* of the current block is ordered in monotonically increasing loop frequency of the blocks. On a tie of frequency, a block outside the current loop (that includes block *B*) appears first in the list. This ordering ensures that the recursion is invoked on lower-frequency successor blocks first, thereby inserting these blocks in *List* before any higher frequency blocks. (Notice the post-recursion action to insert block *B* at the head of *List*.)

As an example, consider the restructured control-flow graph in Figure 2 (c). This graph contains a sequence of blocks, 5-7-2-3, in a separate loop. If block 3 was positionally the last block in the loop, it would contain an unconditional jump to block 5. The above algorithm will eventually result in a call to *order(list1, 5, list2)*. The procedure determines that 5 is a loop header. It then follows the control flow backwards inside the loop to find block 2, a successor of exit block 7. This results in a sequence of calls to *order* with blocks 3, 5, and 7, following the control flow forwards. On the post-recursion action, these blocks are collected to yield the *List = {2, 3, 5, 7}*. This positional order avoids any unconditional jumps inside the loop. The resulting positioned loop is shown in Figure 2 (d). The dashed boxes indicate basic blocks where conditional branches were eliminated. Notice that there are unconditional jumps following two instances of block 7 to block 8 as indicated by the dotted transitions. These jumps cannot be eliminated, but they have been moved outside of any loop within the restructured code. Thus, their execution frequency will be much less on average compared to any instruction inside a loop.

During the last stage, a number of standard optimizations are reapplied to the replicated code. This allows the compiler to take advantage of the simplified control flow due to the elimination of conditional branches. The absence of a branch and it's comparison operation often results in the elimination of a register assignment if the register was dead after the comparison. The more effective reapplied optimizations include: dead code elimination (to delete dead assignments), branch chaining (to minimize the overhead of branches from within the replicated code to it's surrounding code), global register allocation, common subexpression elimination, and code motion. The latter optimizations are applied to take advantage of the new loop structures within the replicated code. The effectiveness of avoiding conditional branches can only be fully exploited when these optimizations are reapplied.

At an earlier stage of this work, other basic block reordering algorithms were tested. It was found that the benefit of avoided conditional branches (and their corresponding compares) was sometimes outweighed by introducing unconditional jumps on frequently executed paths. Thus, an increase in the number of executed instructions occasionally occurred. The algorithm described in Figure 6, on the other hand, yielded the best results for programs with different replication patterns by introducing fewer unconditional jumps.

RESULTS

The optimization to avoid conditional branches was implemented in the compiler back-end VPO (Very Portable Optimizer) [BeD88]. The analysis and replication were performed after all other optimizations had been initially applied, except for filling delay slots, to maximize the benefit of the traditional optimizations first.[4] Measurements were collected on code generated by the compiler using EASE (Environment for Architectural Study and Experimentation) [DaW91] on the SPARC architecture for a

[4] Unstructured loops can be introduced in the restructured code. Thus, loop optimizations should be initially applied before the restructuring optimization.

Name	Description	Static Instructions		Dynamic Instructions			Cache Information		
		Total	Restruct	Total	Restruct	Branch	Hit Ratio	Change	Work
banner	banner generator	+18.24%	+170.59%	**-4.43%**	-5.43%	-10.51%	98.97%	-0.34%	-1.77%
cacheall	cache simulator	+3.08%	+21.89%	**-2.23%**	-18.58%	-23.48%	76.21%	+0.03%	-2.30%
cal	calendar generator	+21.51%	+120.97%	**-3.20%**	-12.48%	-40.13%	99.80%	-0.35%	-0.21%
ctags	C tags generator	+10.53%	+35.49%	**-1.67%**	-2.07%	-5.91%	98.92%	-0.40%	+1.54%
dhrystone	integer benchmark	+8.60%	+18.23%	**-1.06%**	-7.44%	-20.00%	84.62%	-0.70%	+1.56%
join	relational join files	+6.65%	+17.09%	**-7.91%**	-9.87%	-6.37%	98.15%	-0.79%	-2.28%
od	octal dump	+36.32%	+129.09%	**-9.57%**	-12.18%	-12.99%	95.56%	+1.60%	-18.89%
sched	instruction scheduler	+34.25%	+87.02%	**-5.55%**	-8.39%	-10.29%	96.00%	+1.22%	-13.15%
sdiff	side-by-side file diffs	+1.25%	+3.78%	**-4.15%**	-9.01%	+0.00%	97.45%	+0.09%	-4.78%
wc	word counter	+39.22%	+172.73%	**-11.11%**	-12.82%	-22.15%	99.89%	-0.07%	-10.52%
whetstone	FP benchmark	-0.89%	-8.74%	**-6.81%**	-59.18%	-75.00%	100.00%	-0.00%	-6.45%
average		+16.25%	+69.83%	**-5.24%**	-14.31%	-20.62%	95.05%	+0.03%	-5.21%

Table 3: Measurements

number of C programs, which included benchmarks, UNIX utilities, and user applications.

Table 3 shows the measurements for these programs. Each program was tested with and without avoiding conditional branches. The numbers in the table represent the percentage of change after applying the new optimization. Column 3 refers to the change in program size. Column 4 shows the increased percentage of static instructions only within the restructured code portions (loop or function level). Column 5 depicts the decrease in the total number of executed instructions. Column 6 illustrates the reduction of executed instructions only within the restructured code. Column 7 reveals the dynamic change of branch instructions. Columns 8 and 9 report the total hit ratio before the new optimization and the change in the hit ratio after applying the new optimization, respectively. Finally, column 10 refers to the effect on cache work for a direct-mapped 1kB instruction cache with a 16 byte line size. The cache work is calculated by the formula: *cache work = cache hits + cache misses * miss penalty*. The miss penalty was estimated at 10 cycles and a hit at 1 cycle [Smi82]. The cache work is a better measurement than the hit ratio for the evaluation of optimizations when the number of executed instructions changes [DaH92].

The static measurements show that replication results in an increase of code size of about 16%, depending on how many conditional branches could be avoided. The original code portions increased by about 70% on average when they were restructured.

The dynamic measurements indicate a savings of executed instructions of about 5% on average.[5] The restructured code resulted in about 14% fewer instructions executed compared to their original loops and 20% fewer executed conditional branches. The numbers indicate that the local savings of this new optimization can be substantial when the original code portion is compared with the restructured code. The overall savings for a program depend on the execution frequency of the restructured loop. It was surprising to find that even benchmark programs, such as dhrystone and whetstone, contained opportunities for the new optimization with respectable savings.

The hit ratio and it's change provides a general idea of the test programs' caching performance. However, for reasons mentioned previously, the cache work is a better indicator to evaluate the new optimization. The cache work indicates that the reduced number of executed instructions outweighs the increase in code size on average, even for a relatively small cache size of 1kB. These results improved with larger cache sizes. Due to changes in the layout of basic blocks, the cache measurements may vary from program to program. Thus, the average results seem more conclusive than the cache work of any single program.

Figure 7 shows the proportional benefit of the different techniques employed to avoid conditional branches. *Not affected* indicates branches that were encountered and not affected when reached on a subsequent loop iteration (as shown in Figure 2). These cases account for about 1/3

[5] The whetstone benchmark was reduced in code size due to a chain of unconditional jumps in whet1(), accompanied by sets and tests of the same local variable. These chains were greatly simplified by the new optimization. Furthermore, the code positioning algorithm eliminated unconditional branches (due to gotos) in the restructured code. This resulted in an overall reduction of code size even after replication. For sdiff, the number of compares and branches did not change. This was due to input data that never resulted in executing the restructured code portions where conditional branches were avoided. The dynamic savings were due to the execution of fewer unconditional jumps. It was observed that restructuring the code provided new opportunities to avoid unconditional jumps via code positioning.

of the avoided branches. *Subsumption* means avoiding branches whose direction can be inferred from the result of other branches (as depicted in Case III of Table 1) and accounts for over 1/5 of the savings. Branches avoided due to *constant comparisons* imply that an expanded comparison was known due to an effect along a control-flow path (see Case I of Table 1) and are responsible for over 40% of the savings. In a few cases, branches will follow the *same direction* since the result of the comparison can no longer be affected (as portrayed in Case II of Table 1).

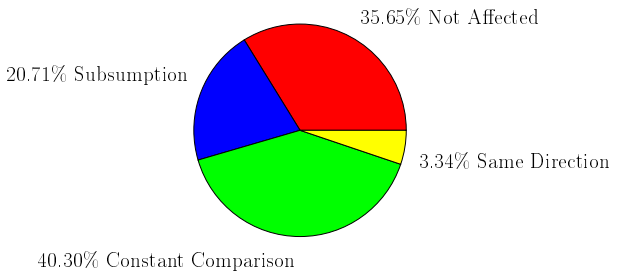

Figure 7: Sources for Avoiding Branches

A number of programs beyond the set in Table 3 were tested and it was found that some conditional branches could be avoided in every one of these programs. Yet, about 1/3 of the programs resulted in an execution benefit of 1% or less. It was also observed that the effectiveness of avoiding conditional branches is highly data dependent. If branches are avoided in loops with high execution frequencies, then the benefits can be quite high. This observation would suggest that avoiding conditional branches could be selectively applied where profiling data indicates that high benefits are more likely.

FUTURE WORK

There are several areas that could be explored to provide more opportunities for avoiding conditional branches. The effects at each exit of an inner loop are not currently used to avoid branches in outer loops. Yet, the results of inner loops are often tested in conditions in outer loops. The authors are considering applying the optimization to the control flow of an entire function all at once, rather than one loop at a time. Thus, the optimization could also be applied to functions containing unstructured loops. In addition, loops containing indirect jumps and associated jump tables are not currently restructured.

Opportunities for avoiding conditional branches would increase if more information was available. For instance, flags are often declared as global variables. A call to many functions, such as `printf`, would not affect a global flag. However, the current analysis, which does not perform interprocedural analysis, has to assume any global variable could be affected in an unknown manner whenever any function is invoked. In addition, it was assumed that any variable could be updated whenever a store through a pointer was encountered. Interprocedural and pointer analysis would provide additional opportunities for avoiding branches.

CONCLUSIONS

This paper described a general approach for avoiding conditional branches by replicating code. The restructured code often contains simplified control flow that allows other optimizations to be applied more effectively. Vectorizing and parallelizing compilers, in particular, may benefit from loops with fewer conditional branches. The optimization could often be applied and resulted in significant performance improvements for the code portions on which the transformations were applied. The benefits of this optimization will improve as instruction cache sizes continue to increase. There are also promising future improvements that could be made to allow a greater number of conditional branches to be avoided.

ACKNOWLEDGEMENTS

The authors thank Jack Davidson for allowing *vpo* to be used for this research. Ricky Benitez developed the ability in *vpo* to expand the effects of an RTL for use in other optimizations. This ability was used to expand the effects of compare instructions, which proved quite useful for determining the set of registers and variables upon which a conditional branch depends. Brad Calder, Emily Ratliff, Randy White, and the anonymous reviewers provided several helpful suggestions that improved the quality of the paper.

REFERENCES

[Ban93] U. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations,* Kluwer Academic Publishers, Norwell, MA (1993).

[BeD88] M. E. Benitez and J. W. Davidson, "A Portable Global Optimizer and Linker," *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pp. 329-338 (June 1988).

[DaH88] J. Davidson and A. Holler, "A Study of a C Function Inliner," *Software—Practice & Experience* **18**(8) pp. 775-790 (August 1988).

[DaH92] J. W. Davidson and A. M. Holler, "Subprogram Inlining: A Study of its Effects on Program Execution Time," *IEEE Transactions on Software Engineering* **18**(2) pp. 89-102 (February 1992).

[DaW91] J. W. Davidson and D. B. Whalley, "A Design Environment for Addressing Architecture and Compiler Interactions," *Microprocessors and Microsystems* **15**(9) pp. 459-472 (November 1991).

[GoR90] M. C. Golumbic and V. Rainish, "Instruction Scheduling beyond Basic Blocks," *IBM Journal of Research and Development* **34**(1) pp. 93-97 (January 1990).

[GrK92] T. Granlund and R. Kenner, "Eliminating Branches using a Superoptimizer and the GNU C Compiler," *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 341-352 (June 1992).

[HeP90] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann, San Mateo, CA (1990).

[Jai91] S. Jain, "Circular Scheduling: A New Technique to Perform Software Pipelining," *Proceedings of the SIGPLAN '91 Symposium on Programming Language Design and Implementation*, pp. 219-228 (June 1991).

[Kra94] A. Krall, "Improving Semi-static Branch Prediction by Code Replication," *Proceedings of the SIGPLAN '94 Symposium on Programming Language Design and Implementation*, pp. 97-106 (June 1994).

[LRS76] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns, *Compiler Design Theory,* Addison-Wesley, Reading, MA (1976).

[MoR79] E. Morel and C. Renvoise, "Global Optimizations by Suppression of Partial Redundancies," *Communications of the ACM* **22**(2) pp. 96-103 (February 1979).

[MuW92] F. Mueller and D. B. Whalley, "Avoiding Unconditional Jumps by Code Replication," *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 322-330 (June 1992).

[Smi82] A. J. Smith, "Cache Memories," *Computing Surveys* **14**(3) pp. 473-530 (September 1982).

[YoS94] C. Young and M. D. Smith, "Improving the Accuracy of Static Branch Prediction Using Branch Correlation," *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 232-241 (November 1994).