Improving Performance by Branch Reordering

MINGHUI YANG∗, GANG-RYUNG UH†, AND DAVID B. WHALLEY∗
∗Department of Computer Science, Florida State University, Tallahassee, FL 32306-4530
e-mail: {myang,whalley}@cs.fsu.edu; phone: (850) 644-3506
†Lucent Technologies, 1247 S. Cedar Crest Blvd., Allentown, PA 18103-6209
e-mail: uh@lucent.com; phone: (650) 712-2447

ABSTRACT

The conditional branch has long been considered an expensive operation. The relative cost of conditional branches has increased as recently designed machines are now relying on deeper pipelines and higher multiple issue. Reducing the number of conditional branches executed can often result in a substantial performance benefit. This paper describes a code-improving transformation to reorder sequences of conditional branches. First, sequences of branches that can be reordered are detected in the control flow. Second, profiling information is collected to predict the probability that each branch will transfer control out of the sequence. Third, the cost of performing each conditional branch is estimated. Fourth, the most beneficial ordering of the branches based on the estimated probability and cost is selected. The most beneficial ordering often included the insertion of additional conditional branches that did not previously exist in the sequence. Finally, the control flow is restructured to reflect the new ordering. The results of applying the transformation were significant reductions in the dynamic number of instructions and branches, as well as decreases in execution time.

1. INTRODUCTION

Sequences of conditional branches occur frequently in programs, particularly in nonnumerical applications. Sometimes these branches may be reordered to effectively reduce the dynamic number of branches encountered during program execution. One type of reorderable sequence consists of branches comparing the same variable or expression to constants. These sequences may occur when a *multiway* statement, such as a C `switch` statement, does not have enough *case*s to warrant the use of an indirect jump from a table. Also, control statements may often compare the same variable more than once.

Consider the following code segment in Figure 1(a). Assume that there is typically more than one blank read per line and EOF is only read once. Many astute programmers may realize that the order of the statements may be changed to improve performance. In fact, we found that the authors of most Unix utilities were quite performance conscious and would attempt to manually reorder such statements. A conventional manual reordering shown in Figure 1(b) would improve performance by performing the three comparisons in reverse order. However, the most commonly used characters (e.g. letters, digits, punctuation symbols) have an ASCII value that is greater than a blank (32), carriage return (10), or EOF (-1). Figure 1(c) shows an improved reordering of the statements that increases the static number of `if` statements and associated conditional branches, but normally reduces the dynamic number of conditional branches encountered during the execution.

```
while ((c=getchar()) != EOF)
   if (c == '\n')
      X;
   else if (c == ' ')
      Y;
   else
      Z;
```

(a) Original Code Segment

```
while (1) {
   c = getchar();
   if (c == ' ')
      Y;
   else if (c == '\n')
      X;
   else if (c == EOF)
      break;
   else
      Z;
   }
```

(b) Conventional Reordering

```
while (1) {
   c = getchar();
   if (c > ' ')
      goto def;
   else if (c == ' ')
      Y;
   else if (c == '\n')
      X;
   else if (c == EOF)
      break;
   else
def:  Z;
   }
```

(c) Improved Reordering

Figure 1: Example Sequence of Comparisons with the Same Variable

Manually reordering a sequence of comparisons of a common variable or inserting extra `if` statements to achieve performance benefits, as shown in Figures 1(b) and 1(c), can lead to obscure code. A general improving transformation to automatically reorder branches may help encourage the use of good software engineering principles by performance conscious programmers.

This paper describes a method for reordering code to reduce the number of branches executed. Figure 2 presents an overview of the compilation process for reordering branches. A first compilation pass is applied to a C source program. All conventional optimizations are applied except for filling delay slots. Sequences of reorderable branches comparing a common variable are detected in the control flow and an executable file is produced that is instrumented to collect profiling information about how often each branch in a sequence will transfer control out of the sequence. This profile data and an estimated cost for executing each branch is used during a second compilation pass to select the most beneficial branch sequence ordering. Delay slots are filled after branch reordering and the final executable is produced. The transformation was frequently applied with reductions in instructions executed and execution time.

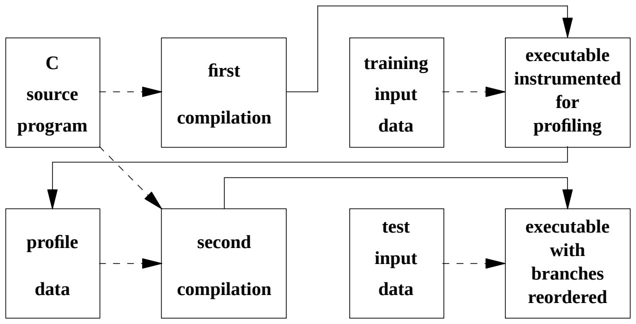


Figure 2: Overview of Compilation Process for Branch Reordering

2. RELATED WORK

There has been some research on other techniques for avoiding the execution of conditional branches. Loop unrolling has been used to avoid executions of the conditional branch associated with a loop termination condition [DaJ96]. Loop unswitching moves a conditional branch with a loop-invariant condition before the loop and duplicates the loop in each of the two destinations of the branch [AlC71]. Different search methods based on static heuristics for the cases associated with a *multiway* statement have been studied [Spu94]. Conditional branches have also been avoided by code duplication [MuW95]. This method determines if there are paths where the result of a conditional branch will be known and duplicates code to avoid execution of the branch. The method of avoiding conditional branches using code duplication has been extended using interprocedural analysis [BGS97]. Finally, conditional branches have been coalesced together into an indirect jump from a jump table [UhW97].

There have also been studies about reordering or aligning basic blocks to minimize pipeline penalties associated with conditional branches [CaG94, YJK97]. However, this reordering or alignment of basic blocks does not change the order or number of conditional branches executed. Instead, it only changes whether the branches will fall through or be taken.

3. DETECTING A REORDERABLE SEQUENCE

The approach used for finding a sequence of reorderable branches that compare a common variable required associating branch targets with ranges of values.

Definition 1. *A **range** is a set of contiguous integer values.*

Definition 2. *A **range condition** is a branch or a pair of consecutive branches that tests if an integer variable is within a range.*

Definition 3. *A **consecutive sequence of range conditions** $[R_1,...,R_n]$ is a path, where each node is a range condition testing the same variable and each edge is a control-flow transition to the next range condition in the sequence.*

Definition 4. *A **reorderable sequence of range conditions** is a consecutive sequence of range conditions, where the range conditions may be interchanged in any permutation with no effect on the semantics of the program.*

The possible types of ranges and the corresponding range conditions are shown in Table 1, where **v** stands for the branch variable and **c** represents a constant. When a range is a single value or a range is unbounded in one direction, a single conditional branch can be used to test if the variable is within the range. Two conditional branches are needed when a range is bounded and spans more than a single value, as depicted in Form 4 in Table 1.

Form	Range	Range Condition
1	c..c	v == c
2	MIN..c	v <= c
3	c..MAX	v >= c
4	c1..c2	c1 <= v && v <= c2

Table 1: Ranges and Corresponding Range Conditions

Figure 3(a) depicts a sequence of two range conditions. **R1** and **R2** are range conditions that can consist of one or two branches that check to see if a variable is in a range. **T1** and **T2** are target blocks of the range conditions and the corresponding range of values for the range condition is given to the right of these blocks. **T3** is the default target block when neither range condition is satisfied. Figure 3(b) shows how the sequence can be reordered.

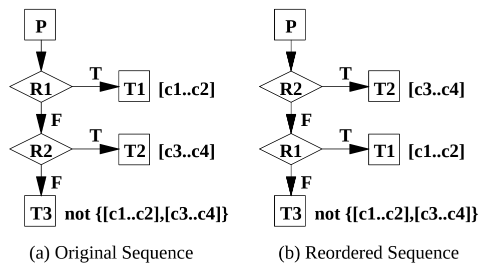


Figure 3: Example of Reordering Range Conditions with No Intervening Side Effects

Definition 5. *Two ranges are* ***nonoverlapping*** *if they do not have any common values.*

Definition 6. *A* ***side effect*** *in a range condition is an instruction that updates a variable or a register and the updated value can reach a use of that variable or register outside of the range condition.*

Theorem 1. *A sequence of two consecutive nonoverlapping range conditions can be reordered with no semantic effect on the program if the sequence can only be entered through the first range condition, the two range conditions contain only pairs of comparisons and conditional branches, and the sequence has no side effects.*

Proof: Consider the original and reordered sequences of range conditions in Figure 3. The two sequences are semantically equivalent given that (1) the state of the program is equivalent in both sequences when blocks **T1**, **T2**, and **T3** are reached, (2) blocks **T1**, **T2**, and **T3** are always reached in both sequences under the same conditions, and (3) no new error exceptions are raised.

Condition 1 is satisfied since the range conditions **R1** and **R2** have no side effects. Condition 2 is satisfied since the ranges associated with **T1**, **T2**, and **T3** are nonoverlapping, there are no assignments in either range condition that can affect the other, and the only predecessor of the second range condition is the first range condition. Condition 3 can be satisfied by considering the following two facts. First, no new error exceptions can be introduced after exiting the reordered sequence due to conditions 1 and 2. Second, no new error exceptions can be introduced in **R1** or **R2** since comparison and conditional branch instructions cannot raise error exceptions on the target architecture. □

Given Theorem 1, it can be shown through induction that a sequence of nonoverlapping range conditions testing the same variable can be reordered with no semantic effect on the program if the sequence can only be entered through the first range condition, the sequence contains only pairs of comparisons and conditional branches, and the sequence has no side effects.[1]

[1] The inductive proofs are not given in this paper due to lack of space and that these proofs were straightforward.

The detection of a sequence of reorderable range conditions was accomplished using the algorithm in Figure 4. Instead of storing a sequence of branches, we instead store a sequence of ranges. The algorithm first finds two nonoverlapping range conditions comparing the same variable. Afterwards, it repeatedly detects an additional nonoverlapping range condition until no more range conditions with nonoverlapping ranges can be found.

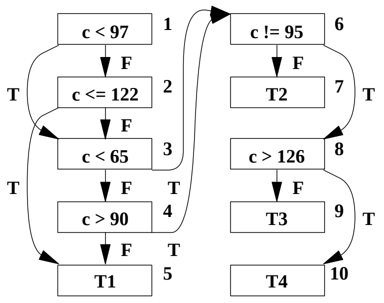
```
FOR each block B DO
    IF (B is not marked AND
         B has a branch that compares
           a variable V to a constant) THEN
        IF (Find_First_Two_Conds(B, V, R1, R2, N)) THEN
            Ranges = {R1, R2};
            C = N;
            mark blocks associated with R1 and R2;
            WHILE Find_Range_Cond(Ranges, V, C, R, N) DO
                Ranges += R;
                C = N;
                mark block(s) associated with R;
    Store info about Ranges for profiling;

BOOL FUNCTIONFind_First_Two_Conds(B, V, R 1, R2, N)
{
    IF (Find_Range_Cond({}, V, B, R1, N1) AND
         Find_Range_Cond(R1, V, N1, R2, N2)) THEN
        N = N2;
        RETURN TRUE;
    ELSE
        Rt = R1;
        IF (Find_Range_Cond(Rt, V, B, R1, N1) AND
             Find_Range_Cond(R1, V, N1, R2, N2)) THEN
            N = N2;
            RETURN TRUE;
    RETURN FALSE;
}

BOOL FUNCTION Find_Range_Cond(Ranges, V, B , R, N)
{
    IF (B has a branch that compares V to a constant C) THEN
        IF branch operator is "==" THEN
            R = C..C;
            N = B's fall-through succ;
            RETURN Nonoverlapping(R, Ranges);
        ELSE IF branch operator is "!=" THEN
            R = C..C;
            N = B's taken succ;
            RETURN Nonoverlapping(R, Ranges);
        ELSE IF (B's b ranch and the branch of a succ S of B
                    form a bounded range R AND
                  B and S have a common succ AND
                  Nonoverlapping(R, Ranges)) THEN
            N = the succ of S not associated with R;
            RETURN TRUE;
        ELSE
            SWITCH (branch operator)
                CASE  "<": R = MIN..C-1;   I = C..MAX;
                CASE "<=": R = MIN..C;     I = C+1..MAX;
                CASE ">=": R = C..MAX;     I = MIN..C-1;
                CASE  ">": R = C+1..MAX;  I = MIN..C;
            IF (Nonoverlapping(R, Ranges)) THEN
                N = B's fall-through succ;
                RETURN TRUE;
            ELSE
                N = B's taken succ;
                RETURN Nonoverlapping(R = I, Ranges);
    RETURN FALSE;
}
```

Figure 4: Detecting a Reorderable Sequence

Figure 5 shows an example of detecting a sequence of range conditions. Figures 5(a) and 5(b) show a C code segment and the corresponding control flow produced by the compiler. Figure 5(c) shows the sequence of reorderable range conditions that are detected using the algorithm in Figure 4. Note that all of the ranges are nonoverlapping.

```
if (c >= ’a’ && c <= ’z’ ||
    c >= ’A’ && c <= ’Z’)
   T1;
else if (c == ’_’)
   T2;
else if (c <= ’˜’)
   T3;
else
   T4;
```

(a) C Code Segment



(b) Control Flow

Blocks	Range	Target
1,2	[97..122]	T1
3,4	[65..90]	T1
6	[95..95]	T2
8	[127..MAX]	T4

(c) Reorderable Range Conditions

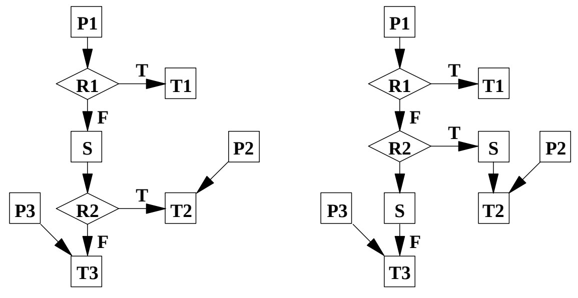
Figure 5: Example of Detecting Range Conditions

A more complete set of branches that compare a common variable to constants may be detected by propagating value ranges through both successors of each branch (i.e. detecting a DAG of branches instead of a path of range conditions) [UhW97]. There were two reasons why reordering was limited to sequences of range conditions. First, there were very few cases that we examined where a sequence of range conditions did not capture the entire set of branches comparing a common variable to constants. Second, we show in this paper that it is possible to start with a sequence and guarantee an improved reordered sequence with respect to profile and cost estimates.

4. MAKING SEQUENCES REORDERABLE

It may appear that the restrictions in Theorem 1 would result in few reorderable sequences of range conditions being detected. In fact, most of the sequences could be altered to meet these restrictions. For instance, the range conditions can be duplicated to ensure that the sequence is always entered at the head. Likewise, if a basic block containing **R1** did have a preceding side effect, then it could be split apart into the portion with the side effect and the portion without one. Only the latter portion containing the range condition would be reordered. Finally, there will typically be no assignments of registers or variables associated with a range condition. The branch variable may be loaded into a register preceding the first range condition. Any subsequent loads of the branch variable in the sequence would be redundant and are usually eliminated by the compiler. Thus, each range condition could usually be accomplished with just comparison and branch instructions since the value of the branch variable was typically available in a register and the constants tested in the range condition could be represented in the comparison instructions for most ranges of values. Theorem 1 can be extended to allow other instructions to produce the values being compared, as long as these instructions have no side effects and do not affect previous range conditions in the original sequence.

Sometimes intervening side effects do exist between range conditions. Rather than attempting to reorder such sequences directly, we instead determine if we can move the side effects out of the sequence by duplicating code. Figure 6(a) shows a sequence of two range conditions with an intervening side effect **S**, which is actually in a block containing **R2**. Figure 6(b) portrays how the side effect can be moved after **R2** by duplicating **S** on both transitions from the range condition. Note that the transitions from **P2** and **P3** require that the side effect **S** be placed in separate basic blocks. The resulting sequence of range conditions now has no intervening side effects and can be reordered.



(a) Original Sequence (b) Transformed Sequence

Figure 6: Moving Side Effects from
a Sequence of Two Range Conditions

Theorem 2. *A side effect between two consecutive range conditions can be duplicated to follow the second range condition with no semantic effect on the program if the side effect does not affect the branch variable of the second range condition and the sequence can only be entered through the first range condition.*

Proof: Consider the original and transformed sequences of range conditions in Figure 6. The two sequences are semantically equivalent given that (1) state of the program is equivalent in both sequences when blocks **T2** and **T3** are reached, (2) blocks **T2** and **T3** are always reached in both sequences under the same conditions, and (3) no new error exceptions are raised.

Condition 1 is satisfied since the range condition **R2** in the transformed sequence has no side effects, **S** is executed in both sequences when **T2** or **T3** is reached after executing **R2**, and **S** is not executed if **T2** or **T3** is reached without executing **R2**. Condition 2 is satisfied since **S** does not affect the branch variable of **R2**. Condition 3 can be satisfied by noting that no new side effects are introduced in the transformed sequence. □

Given Theorem 2, it can be shown through induction that a sequence of range conditions with intervening side effects can be transformed to have no intervening side effects and still have the same semantic effect on the program if the side effects do not affect the branch variable of the range conditions and the sequence is only entered through the first range condition.

5. PRODUCING THE PROFILE INFORMATION

The profiling code for reordering range conditions checks if the common variable is within ranges of values. The compiler needs to know how often each range condition in the sequence would have a transition out of the sequence given it was executed when the head of the sequence is encountered. The instrumentation code for obtaining profile information about the sequence was entirely inserted at the head of the sequence to check every range condition in the sequence. However, additional ranges have to be determined from the ones calculated by the algorithm in Figure 4.

Definition 7. *An **explicit range** is a range that is checked by a range condition.*

Definition 8. *A **default range** is a range that is not checked by a range condition.*

Consider the original sequence of range conditions in Figure 7(a). There are additional ranges associated in the default target **TD** since these ranges will span any remaining values not covered by the other ranges. It is assumed in this figure that **MIN < c1**, **c2+1 < c3**, and **c4 < MAX**. Figure 7(b) shows an equivalent sequence with these default ranges explicitly checked. Figure 7(c) shows a reordered sequence of range conditions, where the range condition for the last default range in 7(b) was placed first in the sequence. Once a point is reached in the sequence where there is only a single target possible, then all remaining range conditions need not be explicitly tested as shown in Figure 7(d). The compiler calculated these remaining ranges by sorting the explicit ranges and adding the minimum number of ranges to cover the remaining values.

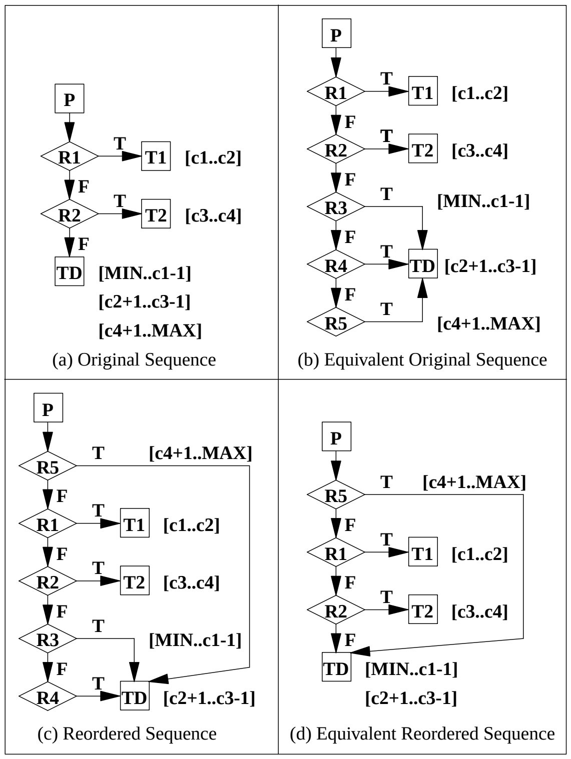


Figure 7: Example of Reordering Default Range Conditions

6. SELECTING THE SEQUENCE ORDERING

The ordering for a reorderable sequence of range conditions was chosen by using the factors specified in the following two definitions.

Definition 9. ***p_i*** *is the probability that range condition R_i will exit the sequence of range conditions.*

Each ***p_i*** was calculated using the profile information indicating how often the corresponding range condition ***R_i*** would exit the sequence if it was executed. The accuracy of this probability depends on the correlation of the branch results between using the training data set and the test data set. It has been found that conditional branch results can often be accurately predicted using profile data [FiF92].

Definition 10. ***c_i*** *is the cost of testing range condition R_i.*

Each ***c_i*** was estimated by determining the number of instructions required for the corresponding range condition. This cost includes the conditional branch(es), associated comparison(s), and any instructions that produce the values being compared. (A more accurate cost estimate could be obtained by estimating the latency and pipeline stalls associated with these instructions.) Some factors of the cost can vary depending upon the ordering of range conditions selected. In these cases, a conservative estimation of the cost was used.

Definition 11. *The **Cost([R1,...,Rn])** is the estimated cost of executing a sequence of range conditions.*

The explicit cost of a sequence of range conditions is calculated as a sum of products. One factor is the probability that a range condition will be reached and will exit the sequence, which is equal to the probability that the range condition will be satisfied since the range conditions are associated with nonoverlapping ranges. The other factor is the cost of performing the instructions in that range condition and all preceding range conditions in the sequence. Equation 1 represents the explicit cost of executing a sequence of *n* range conditions, where every range associated with the sequence is explicitly checked.

$$\begin{aligned} &Explicit\_Cost([R_1, \ldots, R_n]) \\ &\quad = p_1c_1 + p_2(c_1 + c_2) + \cdots + p_n(c_1 + c_2 + \cdots + c_n) \end{aligned} \tag{1}$$

Theorem 3. *A sequence of two consecutive nonoverlapping range conditions can be optimally ordered with respect to the probability and cost estimates as [Ri,Rj] when pi/ci ≥ pj/cj.*

Proof: An optimal ordering of two consecutive nonoverlapping range conditions can be achieved when the explicit cost of the selected ordering is less than or equal to the explicit cost of the other ordering.

$$\begin{aligned} Explicit\_Cost([R_i, R_j]) &\le Explicit\_Cost([R_j, R_i]) \\ p_ic_i + p_j(c_i + c_j) &\le p_jc_j + p_i(c_j + c_i) \\ p_ic_i + p_jc_i + p_jc_j &\le p_jc_j + p_ic_j + p_ic_i \\ p_jc_i &\le p_ic_j \\ p_j/c_j &\le p_i/c_i \\ p_i/c_i &\ge p_j/c_j \end{aligned}$$

□

Intuitively, this means that it is desirable to first execute the range conditions that have a high probability of exiting the sequence along with a low cost. Given Theorem 3, it can be shown through induction that an entire sequence of explicitly specified nonoverlapping range conditions can be optimally ordered.

However, there is also a default cost, which occurs when no range condition is satisfied and the control transfers to the default target. Equation 2 shows the complete cost of a sequence, where only the first *n* ranges are explicit.

$$\begin{aligned} Cost([R_1, \ldots, R_n]) = Explicit\_Cost([R_1, \ldots, R_n]) + \\ (1 - (p_1 + \cdots + p_n))(c_1 + \cdots + c_n) \end{aligned} \tag{2}$$

Once only a single target remains, then the range conditions associated with that target need not be tested. Consider again the example in Figure 7(a). The three targets of the range conditions are **T1**, **T2**, and **TD**. Each of these targets could be potentially used as the default target and its associated range conditions would not have to be tested. The **TD** target has three associated ranges. If any of these ranges are explicitly checked, then Theorem 3 should be used to establish its best position relative to the other explicitly checked range conditions to achieve the lowest cost for the sequence. If **TD** will be used as the default target, then at least one of the three range conditions should not be explicitly checked.

Definition 12. ***mindefault(Ti)*** *is the minimum cost of any ordering of a range condition sequence, where **Ti** is used as the default target.*

For each potential default target having *m* associated ranges, there are 2^m-1 possible combinations of these range conditions that do not have to be explicitly checked. The compiler used the ordering ***p1/c1*** ≥ ... ≥ ***pm/cm*** between the *m* ranges of a target to consider only *m* possible combinations of default range conditions, {{***Rm***}, {***Rm-1***,***Rm***}, ..., {***R1***,...,***Rm***}}. The compiler selected the lowest cost combination of default ranges by calculating the minimum cost of the sequence excluding the range conditions associated with each of these sets. Assume that *t* is the number of unique targets out of the sequence. The compiler then calculates the minimum of {**mindefault(T1)**, **mindefault(T2)**, ..., **mindefault(Tt)**}. Note that only the cost of *n* sequences have to be considered, where *n* is the total number of ranges for all of the targets.

Our approach is not guaranteed to be optimal. However, we also implemented an exhaustive approach to find the lowest cost sequence. Our approach always selected the optimal sequence for every reorderable sequence in every test program for the training data sets.

Equation 3 represents the cost of executing a sequence of *n*-1 explicitly checked range conditions, where only range condition *i* is a default range.

$$\begin{aligned} &Cost([R_1, \ldots, R_{i-1,} R_{i+1,} \ldots, R_n]) \\ &\quad = p_1c_1 + \cdots + p_{i-1}(c_1 + \cdots + c_{i-1}) \\ &\qquad + p_{i+1}(c_1 + \cdots + c_{i-1} + c_{i+1}) + \cdots \\ &\qquad + p_n(c_1 + \cdots + c_{i-1} + c_{i+1} + \cdots + c_n) \\ &\qquad + p_i(c_1 + \cdots + c_{i-1} + c_{i+1} + \cdots + c_n) \end{aligned} \tag{3}$$

However, Equation 3 can be rewritten as Equation 4, where the cost of a sequence of range conditions with a default range can be calculated by subtracting the difference from Equation 1.

$$\begin{aligned} &Cost([R_1, \ldots, R_{i-1,} R_{i+1,} \ldots, R_n]) \\ &\quad = Cost([R_1, \ldots, R_n]) + p_i(c_{i+1} + \cdots + c_n) \\ &\qquad\qquad - c_i(p_i + \cdots + p_n) \end{aligned} \tag{4}$$

The ordering of a sequence of range conditions was selected using the algorithm in Figure 8. The algorithm first uses Equation 1 to calculate the cost of the optimal sequence when all of the range conditions are explicitly checked. It then uses Equation 4 to avoid calculating the complete cost of the *n* different sequences. The complexity of the algorithm is *O(n)*, where *n* is the number of ranges in the sequence.

```
/* Assume the range conditions are sorted in descending order of Pi/Ci.
   Calculate the cost with all range conditions explicitly checked.  */
Explicit_Cost = 0.0;
cost = 0;
FOR i = 1 to n DO
   cost += C[i];
   Explicit_Cost += P[i]*cost;

/* tcost[i] = Ci+1 + ... + Cn and tprob[i] = Pi + Pi+1 + ... + Pn.  */
tcost[n] = 0;
tprob[n] = P[n];
FOR i = n-1 downto 1 DO
   tcost[i] = C[i+1] + tcost[i+1];
   tprob[i] = P[i] + tprob[i+1];

/* Now find the sequence with the lowest cost.  */
Lowest_Cost = Explicit_Cost;
FOR each unique target T DO
   Cost = Explicit_Cost;
   Elim_Cost = 0;
   FOR each range Ri in T from lowest
       to highest P[i]/C[i] DO
      Cost += P[i]*(tcost[i] - Elim_Cost)
              - C[i]*tprob[i];
      IF Cost < Lowest_Cost THEN
         Lowest_Cost = Cost;
         Best_Sequence = current sequence;
      Elim_Cost += C[i];
```

Figure 8: Sequence Ordering Selection Algorithm

7. IMPROVING THE SELECTED SEQUENCE

Other improvements were obtained after the ordering decision was made. The compiler can determine the best ordering of the two branches within a single range condition that is of type Form 4 shown in Table 1. The compiler assumed that both branches would be executed in estimating the cost for selecting the range condition ordering. If the result of the first branch indicates that the range condition is not satisfied, then the second branch need not be executed. Assume that such a range condition, ***R_i***, is the *i*th range condition in the sequence and is associated with the range **[c1..c2]**. The probability that the value of the common variable is below or above the range **[c1..c2]** at the point that the range condition is performed can be determined as follows. We know that the range conditions associated with the sequence **[R_1,R_2,...,R_{i-1}]** have already been tested and the value of the common variable cannot be in these ranges if ***R_i*** is reached. Given that there are *n* total range conditions, the compiler examined the probability for each of the remaining ranges, **[R_{i+1},R_{i+2},...,R_n]**, to determine the probability that **v < c1** versus that **v > c2**. Based on these probabilities, the branch is placed first that is most likely to determine if the range condition is not satisfied.

Another improvement that was performed after the range conditions have been ordered is to eliminate redundant comparisons. For instance, consider Figure 9(a). There are two consecutive range conditions that test if the common variable is in the ranges **[c+1..max]** and **[c..c]**. Figure 9(b) shows a semantically equivalent comparison and branch for the first range condition. The comparison instruction within the second range condition becomes redundant and the compiler eliminates it.

first comparison:	`IC=v?c+1;`	`IC=v?c;`
first branch:	`PC=IC>=0->L1;`	`PC=IC>0->L1;`
second comparison:	`IC=v?c;`	
second branch:	`PC=IC==0->L2;`	`PC=IC==0->L2;`
	(a) Before	(b) After

Figure 9: Eliminating Redundant Comparisons Example

8. APPLYING THE TRANSFORMATION

Once a branch ordering has been selected, the compiler will apply the reordering transformation. Figure 10(a) shows a control-flow segment containing a sequence of three explicit range conditions (**R1**, **R2**, and **R3**) and two intervening side effects (**S1** and **S2**). Figure 10(b) shows the control flow with the replicated range conditions (**R1'**, **R2'**, and **R3'**) inserted. The predecessors of the first original range condition now have transitions to the first replicated range condition. Note that the target **TD** in Figure 10(a) has a fall-through predecessor. Code starting at the target block **TD** was duplicated until an unconditional jump, return, or indirect jump was found. This approach avoided increasing the number of unconditional jumps executed from the reordered sequence and also simplified the estimation of the cost of a reordered sequence. A similar approach has been used when transforming code to improve branch prediction [YoS94]. Figure 10(c) shows the control flow with the two intervening side effects duplicated to allow the sequence of range conditions to be reordered. **T2** is also duplicated to avoid an extra unconditional jump. Figure 10(d) shows the control flow after reordering the range conditions. **R4** was one of the original default range conditions and is now explicit and first in the replicated sequence. **R1'** and **R2'** have also been reversed. Figure 10(e) shows the code after applying dead code elimination. The original range conditions **R1** and **R2** were deleted, while range condition **R3** remains since it was still reachable from another path. Other optimizations, such as code repositioning and branch chaining to minimize unconditional jumps, were also reinvoked to improve the code.

9. RESULTS

Table 2 shows the three different sets of heuristics used when translating **`switch`** statements. The front end used Heuristic Set I, which are the same heuristics used in the *pcc* front end [Joh79], when compiling for a SPARC IPC and a SPARC 20. The authors used the dual loop method [CDV86] and found that indirect jumps on the SPARC Ultra I were about four times more expensive than indirect jumps on the SPARC IPC or SPARC 20 [Uh97]. Therefore, Heuristic Set II used for the Ultra only generated an indirect jump when $n \geq 16$. Finally, Heuristic Set III always generated a linear search, which achieved the maximum benefit from reordering.

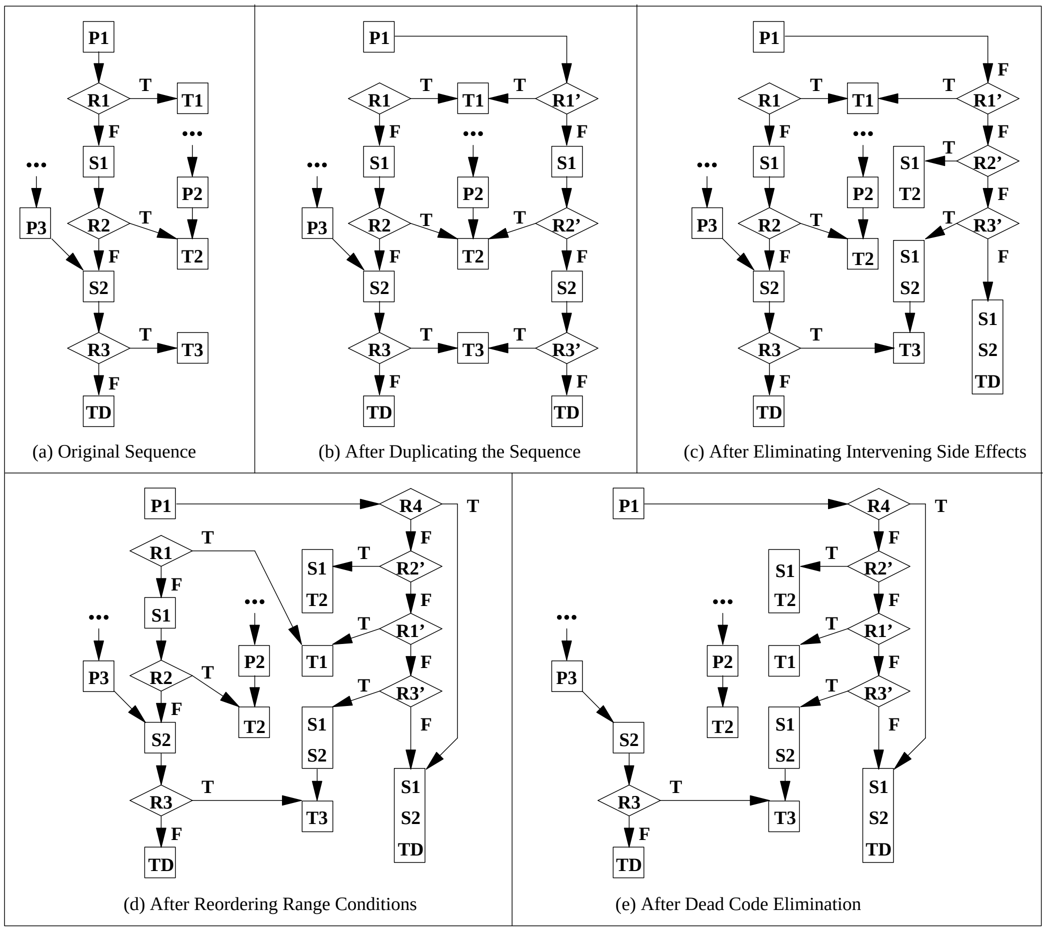

P1
R1
T1
S1
R2
T2
P2
P3
S2
R3
T3
TD
T
F
R1'
R2'
R3'
R4
(a) Original Sequence
(b) After Duplicating the Sequence
(c) After Eliminating Intervening Side Effects
(d) After Reordering Range Conditions
(e) After Dead Code Elimination

Figure 10: Applying the Reordering Transformation

Term	Definition		
n	Number of cases in a **`switch`** statement.		
m	Number of possible values between the first and last case.		
Heuristic Set	Indirect Jump	Binary Search	Linear Search
I	$n \geq 4$ && $m \leq 3n$	!indirect_jump && $n \geq 8$	!indirect_jump && !binary_search
II	$n \geq 16$ && $m \leq 3n$	!indirect_jump && $n \geq 8$	!indirect_jump && !binary_search
III	never	never	always

Table 2: Heuristics Used for Translating *switch* Statements

Program	Description
awk	Pattern Scanning and Processing Language
cb	A Simple C Program Beautifier
cpp	C Compiler Preprocessor
ctags	Generates Tag File for *vi*
deroff	Removes *nroff* Constructs
grep	Searches a File for a String or Regular Expression
hyphen	Lists Hyphenated Words in a File
join	Relational Database Operator
lex	Lexical Analysis Program Generator
nroff	Text Formatter
pr	Prepares File(s) for Printing
ptx	Generates a Permuted Index
sdiff	Displays Files Side-by-Side
sed	Stream Editor
sort	Sorts and Collates Lines
wc	Displays Count of Lines, Words, and Characters
yacc	Parsing Program Generator

Table 3: Test Programs

Measurements were collected on the code generated for the SPARC architecture by the *vpo* compiler [BeD88] using the *ease* environment [DaW91]. Table 3 shows the test programs used for this study. We chose these non-numerical applications since they tend to have complex control flow and a higher density of conditional branches. Table 4 shows the dynamic frequency measurements that were obtained. The *Original Insts* column contains the number of instructions executed with all of *vpo*'s conventional optimizations applied. We present in the rest of the table the percentage change in the number of instructions and branches executed after reordering sequences of range conditions. The reordering transformation had significant benefits both in reducing the total number of instructions and conditional branches. The original default target in a sequence was almost always selected as the default target for the reordered sequence. However, the profile data also

Switch Translation Heuristics	Program	Original	Reordered	
		Insts	Insts	Branches
Set I	awk	13,611,150	-2.02%	-4.19%
	cb	17,100,927	-7.65%	-15.46%
	cpp	18,883,104	-0.13%	-0.19%
	ctags	71,889,513	-9.10%	-14.72%
	deroff	15,460,307	-1.53%	-2.63%
	grep	9,256,749	-3.60%	-8.31%
	hyphen	18,059,010	+3.42%	+3.40%
	join	3,552,801	-1.68%	-2.12%
	lex	10,005,018	-4.56%	-10.39%
	nroff	25,307,809	-2.48%	-6.35%
	pr	73,051,342	-16.25%	-29.96%
	ptx	20,059,901	-9.18%	-13.28%
	sdiff	14,558,535	-16.09%	-37.03%
	sed	14,229,310	-1.16%	-2.03%
	sort	23,146,400	-47.20%	-57.38%
	wc	25,818,199	-15.05%	-26.26%
	yacc	25,127,817	-0.25%	-0.44%
	average	23,477,465	-7.91%	-13.37%
Set II	awk	13,552,831	-2.97%	-6.15%
	cb	17,100,927	-7.65%	-15.46%
	cpp	18,880,116	-0.13%	-0.19%
	ctags	71,824,093	-9.02%	-14.64%
	deroff	15,451,383	-1.39%	-2.38%
	grep	9,938,414	-10.53%	-22.04%
	hyphen	18,059,010	+3.42%	+3.40%
	join	3,552,801	-1.68%	-2.12%
	lex	10,003,391	-4.57%	-10.40%
	nroff	25,313,527	-2.50%	-6.39%
	pr	73,051,352	-16.25%	-29.96%
	ptx	20,059,901	-9.18%	-13.28%
	sdiff	14,558,530	-16.09%	-37.03%
	sed	14,243,263	-1.28%	-2.32%
	sort	23,146,400	-47.20%	-57.38%
	wc	25,818,199	-15.05%	-26.26%
	yacc	25,127,817	-0.25%	-0.44%
	average	23,510,571	-8.37%	-14.30%
Set III	awk	13,651,335	-3.63%	-7.44%
	cb	19,662,207	-21.79%	-37.41%
	cpp	30,477,974	-28.37%	-41.85%
	ctags	72,222,399	-9.13%	-14.73%
	deroff	15,491,185	-1.40%	-2.39%
	grep	11,810,072	-32.04%	-51.42%
	hyphen	18,059,010	+3.42%	+3.40%
	join	3,552,801	-1.68%	-2.12%
	lex	10,028,151	-4.77%	-10.73%
	nroff	25,339,678	-2.53%	-6.45%
	pr	73,051,352	-16.25%	-29.96%
	ptx	20,059,901	-9.18%	-13.28%
	sdiff	14,558,530	-16.09%	-37.03%
	sed	15,368,724	-10.07%	-17.01%
	sort	23,146,434	-47.20%	-57.38%
	wc	25,818,199	-15.05%	-26.26%
	yacc	25,168,370	-0.47%	-0.76%
	average	24,556,842	-12.72%	-20.75%

Table 4: Dynamic Frequency Measurements

indicated that one of the original default ranges was frequently satisfied and was explicitly checked in the reordered sequence. Also, comparison instructions became redundant and were eliminated much more often when an original default range became an explicit range in the reordered sequence. One may notice that the transformation had a slight negative impact on *hyphen*, which occurred for a couple of reasons. First, different test input data was used as compared to the training input data for the results presented in the table. When we used the same test input data as the training input data, the number of branches never increased. Second, the reordering transformation was applied after all optimizations except for filling delay slots. Sometimes delay slots would be filled from the other successor and would not execute a useful instruction. One should note that inconsistent filling of delay slots also sometimes resulted in increased performance benefits. The transformation may also have very significant benefits when a program executes most of its instructions in a reorderable sequence, such as in *sort*. The differences between using the different sets of heuristics indicates that the effectiveness of branch reordering increases as indirect jumps become more expensive. It is also interesting to note that the total number of instructions executed after reordering often decreased as fewer indirect jumps were generated. This shows that profile information should be used to decide if an indirect jump should be generated or branch reordering should instead be applied.

Branch prediction measurements shown in Table 5 were obtained for the SPARC Ultra I, which supports branch prediction with a (0,2) predictor with 2048 entries. The authors anticipated that the number of branch mispredictions would decrease since the number of total branches executed was substantially reduced. Fewer mispredictions had been observed when branches were coalesced into indirect jumps [Uh97]. However, the misprediction results for branch reordering were mixed. Nine of the test programs had fewer mispredictions after reordering and the remaining eight had increases. Overall, the average number of mispredictions increased. We suspect that adding more branches to a sequence caused additional mispredictions to

Program	Original Mispreds	Reordered Mispreds	Inst Ratio
awk	243,027	-0.46%	N/A
cb	440,712	+5.77%	51.41
cpp	389,566	-1.75%	N/A
ctags	569,753	+225.50%	5.04
deroff	62,819	-2.87%	N/A
grep	115,007	-4.30%	N/A
hyphen	266,177	+84.12%	-2.76
join	50,440	-5.62%	N/A
lex	66,534	+1.93%	355.47
nroff	141,167	-0.93%	N/A
pr	750,570	+0.33%	4,793.65
ptx	215,218	+37.58%	22.78
sdiff	156,440	-5.35%	N/A
sed	83,579	-1.84%	N/A
sort	171,619	-10.41%	N/A
wc	481,767	+0.18%	4,519.65
yacc	373,825	+0.55%	30.28
average	269,307	+18.97%	1,221.94

Table 5: Branch Prediction Measurements Using a (0,2) Predictor with 2048 Entries

occur. But the average ratio of decreased instructions executed to the increased number of branch mispredictions was 1221.94 to 1 for these eight programs. Thus, the increase in mispredictions was far outweighed by the benefit of reducing the number of instructions executed. Comparable results were obtained using other branch predictors as shown in Table 6.

Entries	(0,1) Predictor		(0,2) Predictor		(2,2) Predictor	
	Reordered Mispreds	Inst Ratio	Reordered Mispreds	Inst Ratio	Reordered Mispreds	Inst Ratio
32	+16.65%	681.20	+17.37%	1313.47	+17.05%	805.78
64	+21.96%	720.73	+21.15%	1082.02	+20.77%	640.08
128	+21.91%	8583.19	+20.60%	1091.28	+19.40%	661.92
256	+21.91%	972.87	+20.21%	953.70	+19.03%	569.88
512	+19.67%	5852.38	+18.09%	1200.25	+17.34%	681.98
1024	+20.45%	13331.71	+18.88%	1217.61	+18.44%	664.03
2048	+20.59%	13311.73	+18.97%	1221.94	+37.65%	653.02
average	+21.43%	6207.69	+19.32%	1154.32	+21.38%	668.10

Table 6: Branch Prediction Measurements

The execution time measurements shown in Table 7 were obtained from the average reported *user* times of ten executions of each program using the C run-time library function *times()*. One should note that in Table 4 the measurements from the code compiled by our compiler did not include the C run-time library code. However, the library code did contribute to the execution times. Also, the benefits for the Ultra I were probably not as significant due to the increase in the number of mispredictions.

Machine	Heuristic Set	Average Execution Time
SPARC IPC	I	-4.94%
SPARC 20	I	-5.57%
SPARC Ultra I	II	-2.88%

Table 7: Execution Times

Table 8 shows static measurements for the same set of programs. There was only about a 5% increase in the number of instructions generated. The *Total Seqs* column represents the total number of reorderable sequences detected in each program. The *Seqs* column indicates the percentage of these sequences that were actually reordered. The single most common factor that prevented a sequence from being reordered was that profile data indicated that the sequence was never executed. Using multiple sets of profile data to provide better test coverage would increase this percentage. The *Avg Seq Len* shows the average number of branches in each reordered sequence before and after reordering. The length of each reordered sequence typically increased since often one or more default ranges became explicit after reordering. Heuristic Set III resulted in fewer sequences since no binary searches were generated when translating **switch** statements. Each binary search generated for Heuristic Sets I and II resulted in several reorderable sequences being detected.

Switch Translation Heuristics	Program	Insts	Total Seqs	Reordered		
				Seqs	Avg Seq Len	
					Orig	After
Set I	awk	+1.91%	48	16.67%	2.88	3.75
	cb	+8.32%	12	83.33%	2.50	2.80
	cpp	+1.57%	15	33.33%	2.20	3.20
	ctags	+9.48%	28	39.29%	2.64	3.36
	deroff	+1.58%	38	23.68%	2.67	2.89
	grep	+3.51%	7	28.57%	3.50	4.50
	hyphen	+8.70%	3	100.00%	2.67	3.33
	join	+7.61%	8	37.50%	3.33	3.67
	lex	+8.55%	95	58.95%	2.55	2.95
	nroff	+1.62%	87	21.84%	2.95	3.53
	pr	+2.40%	10	50.00%	3.00	4.20
	ptx	+1.47%	4	75.00%	3.00	4.33
	sdiff	+3.48%	8	37.50%	2.67	3.33
	sed	+4.22%	34	47.06%	2.88	3.50
	sort	+3.68%	16	56.25%	2.33	2.78
	wc	+10.20%	3	33.33%	5.00	5.00
	yacc	+6.42%	35	77.14%	3.70	4.48
	avg	+4.98%	26	48.20%	2.97	3.62
Set II	awk	+2.05%	56	19.64%	3.91	4.55
	cb	+8.32%	12	83.33%	2.50	2.80
	cpp	+1.57%	16	31.25%	2.20	3.20
	ctags	+9.47%	29	37.93%	2.64	3.36
	deroff	+1.76%	41	24.39%	3.00	3.20
	grep	+4.11%	19	36.84%	2.57	2.86
	hyphen	+8.70%	3	100.00%	2.67	3.33
	join	+7.61%	8	37.50%	3.33	3.67
	lex	+8.98%	103	58.25%	2.68	3.07
	nroff	+1.73%	93	25.81%	2.83	3.33
	pr	+2.62%	11	54.55%	3.67	4.67
	ptx	+1.47%	5	60.00%	3.00	4.33
	sdiff	+3.49%	10	40.00%	3.00	3.50
	sed	+4.32%	41	51.22%	2.81	3.29
	sort	+3.68%	16	56.25%	2.33	2.78
	wc	+10.20%	3	33.33%	5.00	5.00
	yacc	+6.42%	35	77.14%	3.70	4.48
	avg	+5.09%	29	48.67%	3.05	3.61
Set III	awk	+1.97%	42	30.95%	18.15	18.69
	cb	+11.17%	6	66.67%	5.50	7.75
	cpp	+2.47%	16	37.50%	14.33	16.50
	ctags	+6.50%	21	38.10%	3.50	4.50
	deroff	+1.23%	34	20.59%	5.29	5.57
	grep	+3.29%	9	44.44%	8.00	8.50
	hyphen	+8.70%	3	100.00%	2.67	3.33
	join	+7.61%	8	37.50%	3.33	3.67
	lex	+6.25%	54	59.26%	6.16	7.00
	nroff	+1.71%	46	32.61%	6.00	6.87
	pr	+2.62%	11	54.55%	3.67	4.67
	ptx	+1.47%	5	60.00%	3.00	4.33
	sdiff	+3.49%	10	40.00%	3.00	3.50
	sed	+5.32%	25	48.00%	7.75	8.58
	sort	+3.76%	11	63.64%	3.57	4.29
	wc	+10.20%	3	33.33%	5.00	5.00
	yacc	+6.64%	29	79.31%	4.52	5.65
	avg	+4.96%	19	49.79%	6.08	6.96

Table 8: Static Measurements

Figures 11, 12, and 13 show the distribution of the number of branches in reordered sequences for each of the three heuristic sets. Note that most of the original sequences contained only two or three branches. This shows that much of the benefit for reordering comes from short sequences of branches that would never be translated into indirect jumps.

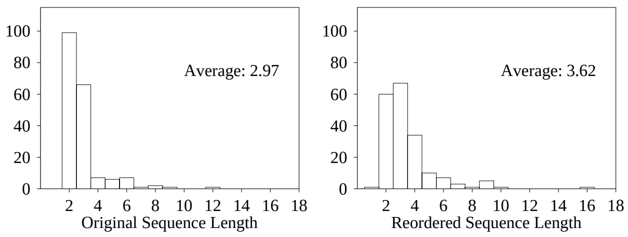


Figure 11: Sequence Length for Heuristic Set I

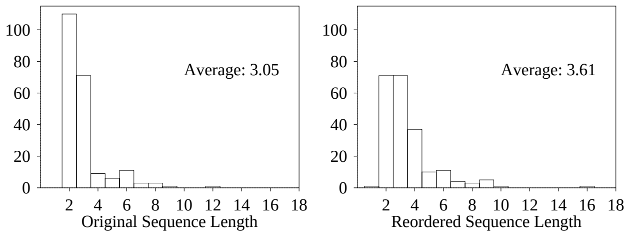


Figure 12: Sequence Length for Heuristic Set II

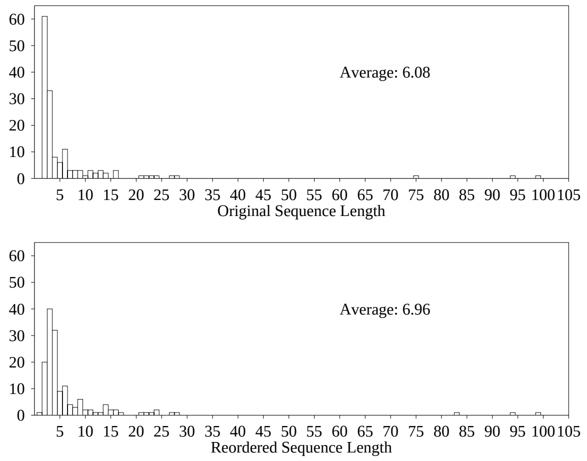


Figure 13: Sequence Length for Heuristic Set III

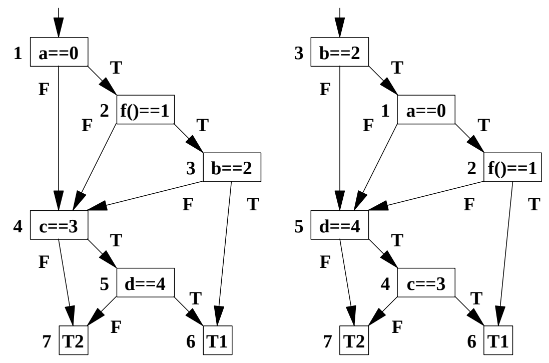
10. FUTURE WORK

There are several areas in which reordering branches could be extended. A sequence of range conditions is one of several approaches that could be used to determine a target associated with the value of an expression. Essentially, a sequence of range conditions is a linear search. Some of these other approaches include performing a binary search, using a jump table, and hashing [Spu94]. Profile data could be used to more effectively apply these other approaches as a semi-static search method and to decide when each method or a combination of methods is most beneficial.

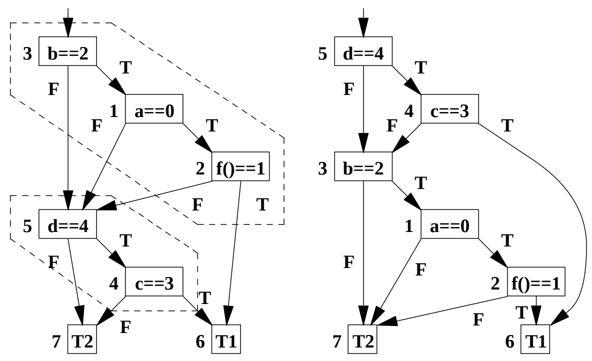
A different type of sequence of branches that can be reordered using profile data would consist of consecutive branches with a common successor. Figure 14(a) shows a C source code segment containing relational and logical expressions and Figure 14(b) shows the corresponding control-flow graph. The sequence of branches in blocks 1, 2, and 3 have block 4 as a common successor. Likewise, the sequence of branches in blocks 4 and 5 have block 7 as a common successor. Figure 14(c) shows these two sequences of branches after reordering. Note that a reorderable sequence of branches with common successors cannot contain intervening side effects. While side effects could be moved out of such a sequence, the resulting sequence would not contain a common successor block. Interprocedural analysis could be used to determine if invoked functions do not cause a side effect. Avoiding the execution of a function call, such as depicted in block 2, could have significant performance benefits. Figure 14(d) depicts that a sequence of branches with a common successor can be viewed as a single block containing a branch since such a sequence would have only two possible successors. The first sequence (blocks 3, 1, and 2) has two successors (blocks 5 and 6). Likewise, the second sequence (blocks 5 and 4) also has two successors (blocks 6 and 7). Figure 14(e) shows that these sequences can be reordered when there are no side effects between the sequences.

```
if (a == 0 && f() == 1 && b == 2 || c == 3 && d == 4)
   T1;
else
   T2;
```

(a) C Source Code Segment



(b) Before Reordering Branches (c) After Reordering Branches



(d) Before Reordering Sequences (e) After Reordering Sequences

Figure 14: Reordering Branches with Common Successors

Obtaining profile data for a sequence of branches with a common successor will differ from obtaining profile data for a sequence of nonoverlapping range conditions testing a common variable. While at most one range condition will be satisfied for a given execution of a sequence of nonoverlapping range conditions testing the same branch variable, more than one branch in a sequence of branches with a common successor could branch to the common successor. Thus, all combinations of branch results would have to be obtained using an array of profile counters. This approach may be reasonable for a small sequence length (e.g. $n \leq 7$), which seem to handle most branch sequences with a common successor [Yan98].

11. CONCLUSIONS

This paper described an approach for using profile information to decrease the number of conditional branches executed by reordering branch sequences. An algorithm for detecting a reorderable sequence of branches testing a common variable was presented. Profiling was performed to estimate the probability that each branch will transfer control out of the sequence. The most beneficial orderings for these sequences with respect to profiling and cost estimates were obtained. The results showed significant reductions in the number of branches and instructions executed, as well as decreases in execution time.

ACKNOWLEDGEMENTS

The authors thank Jack Davidson for allowing *vpo* to be used for this research. Michael Sjödin, Chris Healy, and the anonymous reviewers provided several helpful suggestions that improved the quality of the paper. We also thank Mooly Sagiv for reviewing the final version of this paper.

REFERENCES

[AlC71] F. Allen and J. Cocke, “A Catalogue of Optimizing Transformations,” pp. 1-30 in *Design and Optimization of Compilers*, ed. R. Rustin, Prentice-Hall, Englewood Cliffs, NJ (1971).

[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).

[BGS97] R. Bodik, R. Gupta, and M. Soffa, “Interprocedural Conditional Branch Elimination,” *Proceedings of the SIGPLAN ’97 Conference on Programming Language Design and Implementation*, pp. 146-158 (June 1997).

[CaG94] B. Calder and D. Grunwald, “Reducing Branch Costs via Branch Alignment,” *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 242-251 (October 1994).

[CDV86] R. M. Clapp, L. Duchesneau, R. A. Volz, T. N. Mudge, and T. Schultze, “Toward Real-Time Performance Benchmarks for Ada,” *Communications of the ACM* **19**(8) pp. 760-778 (August 1986).

[DaJ96] J. W. Davidson and S. Jinturkar, “Aggressive Loop Unrolling in a Retargetable, Optimizing Compiler,” *Proceedings of Compiler Construction Conference*, pp. 59-73 (April 1996).

[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).

[FiF92] J. A. Fisher and S. M. Freudenberger, “Predicting Conditional Branch Directions from Previous Runs of a Program,” *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85-95 (October 1992).

[Joh79] S. C. Johnson, “A Tour Through the Portable C Compiler,” *Unix Programmer’s Manual, 7th Edition* **2B** p. Section 33 (January 1979).

[MuW95] F. Mueller and D. B. Whalley, “Avoiding Conditional Branches by Code Replication,” *Proceedings of the SIGPLAN ’95 Conference on Programming Language Design and Implementation*, pp. 56-66 (June 1995).

[Spu94] D. A. Spuler, “Compiler Code Generation for Multiway Branch Statements as a Static Search Problem,” Technical Report 94/03, James Cook University, Townsville, Australia (January 1994).

[Uh97] G. Uh, *Effectively Exploiting Indirect Jumps*, PhD Dissertation, Florida State University, Tallahassee, FL (December 1997).

[UhW97] G. R. Uh and D. B. Whalley, “Coalescing Conditional Branches into Efficient Indirect Jumps,” *Proceedings of the International Static Analysis Symposium*, pp. 315-329 (September 1997).

[Yan98] Minghui Yang, *Improving Performance by Branch Reordering*, Masters Thesis, Florida State University, Tallahassee, FL (1998).

[YJK97] C. Young, D. S. Johnson, D. R. Karger, and M. D. Smith, “Near-optimal Intraprocedural Branch Alignment,” *Proceedings of the SIGPLAN ’97 Conference on Programming Language Design and Implementation*, pp. 183-193 (June 1997).

[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 (October 1994).