Efficient and Effective Branch Reordering Using Profile Data

MINGHUI YANG
Oracle Corporation
and
GANG-RYUNG UH
Boise State University
and
DAVID B. WHALLEY
Florida State University

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 often results in a substantial performance benefit. This paper describes a code-improving transformation to reorder sequences of conditional branches that compare a common variable to constants. The goal is to obtain an ordering where the fewest average number of branches in the sequence will be executed. 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 includes 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 are on average reductions of about 8% fewer instructions executed and 13% branches performed, as well as about a 4% decrease in execution time.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors – compilers; optimization

General Terms: Algorithms, Languages

Additional Key Words and Phrases: conditional branches, profiling, branch reordering

---

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

---

A preliminary version of this research was described in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* under the title "Improving Performance by Branch Reordering."
Authors' addresses: M. Yang, 400 Oracle Parkway, Redwood Shores, CA 94065; e-mail: Minghui.Yang@oracle.com; phone: (650) 633-6958; G. Uh, Computer Science Department, College of Engineering, Boise State University, 1910 University Avenue, Boise, ID 83725; e-mail: uh@cs.boisestate.edu; phone: (208) 426-3505; D. Whalley, Computer Science Department, Florida State University, Tallahassee, FL 32306-4530, U.S.A.; e-mail: whalley@cs.fsu.edu; phone: (850) 644-3506

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. We find that such sequences of branches occur quite frequently in non-numerical applications.

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 find that the authors of most Unix utilities are quite performance conscious and attempt to manually reorder such statements. A conventional manual reordering shown in Figure 1(b) improves 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 single variable to constants 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 obtain performance improvements and still 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. The compiler used was the *vpo* compiler [1], which is part of the *zephyr* system used in the National Compiler Infrastructure project. All conventional optimizations are applied except for filling delay slots. Sequences of reorderable branches comparing a single variable to constants are detected in the control flow. 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 are 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 is frequently applied with reductions in instructions executed and execution time.

The remainder of this paper has the following organization. Section 2 mentions work related to avoiding the execution or reducing the cost of conditional branches. Section 3 presents the method used for detecting reorderable sequences of branches. Section 4 describes the techniques for transforming sequences of branches that are not reorderable into reorderable sequences. Section 5 characterizes the type of profile data that is produced. Section 6 shows how sequences of branches are reordered with respect to

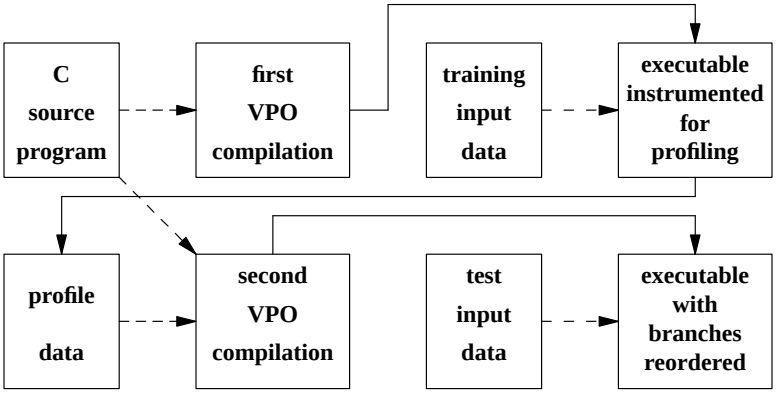


Figure 2: Overview of Compilation Process for Branch Reordering

the profile information and cost. Section 7 depicts other techniques to improve performance after a specific ordering for the sequence of branches is selected. Section 8 illustrates how the reordering transformation is applied. Section 9 analyzes the results of applying this code-improving transformation on a number of non-numerical applications. Section 10 discusses other approaches that can be used to reduce the number of executed branches. Finally, we give the conclusions of the paper in Section 11.

2. RELATED WORK

There has been some research on other techniques for avoiding the execution of conditional branches. Loop unrolling duplicates the body of the loop when the number of loop iterations is known at the point the loop is entered. This has the effect of avoiding executions of the conditional branch associated with a loop termination condition [2], [3]. 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. This transformation has the effect of moving conditional branches outside of a loop [4]. Conditional branches have also been avoided by code duplication [5]. 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 [6]. Conditional branches comparing the same variable to constants have also been coalesced together into an indirect jump from a jump table [7]. The effectiveness of this approach depends on the relative cost of an indirect jump versus the average number of branches executed that the indirect jump replaces. Finally, sequences of conditional branches representing a likely-taken or critical path have been avoided by forming a single bypass branch that checks if the sequence of conditions associated with these branches violate any of the conditions for the critical path [8]. This approach required predicate registers, which are often used to support predicated execution on multiple issue machines. None of these approaches attempted to reduce the cost of a sequence of branches by changing the order in which the branches are executed.

Different search methods based on static heuristics for the cases associated with a *multiway* selection statement, such as a C `switch` or Pascal `case` statement, have been studied [9]. These methods include using a linear search, a binary search, hashing, and an indirect jump from a table. These methods all assume that each case of a multiway selection are equally likely. This study is the most similar to the approach we describe in this paper. However, our approach is performed at a low level and can improve sequences of branches that are produced from source statements other than multiway selection statements. In addition, we were able to advantageously use profile data rather than relying on static heuristics.

There have also been studies about reordering or aligning basic blocks to minimize pipeline penalties associated with conditional branches [10], [11]. 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. These approaches use profile information to minimize the number of taken branches and unconditional jumps executed. Branch alignment can be valuable for architectures where taken branches causes delays.

3. DETECTING A REORDERABLE SEQUENCE

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

Definition 1. *A **branch variable** is a scalar variable or a register containing an integer value being tested in a conditional branch.*

Definition 2. *A **range** is a set of contiguous integer values.*

Definition 3. *A **range condition** is a branch or a pair of consecutive branches that tests if a branch variable is within a range.*

Definition 4. *A **consecutive sequence of range conditions [R_1,...,R_n]** is a path, where each node is a range condition and one outgoing edge is a control-flow transition to the next range condition in the sequence.*

Definition 5. *A **common variable** is a single branch variable whose value is tested in multiple range conditions.*

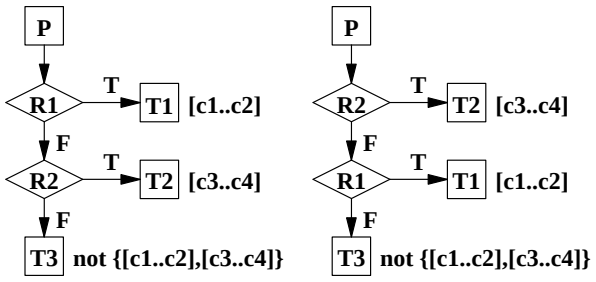
Definition 6. *A **reorderable sequence of range conditions** is a consecutive sequence of range conditions testing a common variable, where the range conditions may be interchanged in any permutation with no effect on the semantics of the program.*

The possible types of ranges and the corresponding range conditions are shown in Table 1, where **v** stands for the branch variable, **c**, **c1**, and **c2** represent constants, and **MIN** and **MAX** stands for the minimum and maximum integer values that can be represented on the machine. 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. ***P*** is a predecessor basic block of the range condition. ***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.



(a) Original Sequence (b) Reordered Sequence

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

Definition 7. *Two ranges are **nonoverlapping** if they do not have any common values.*

Definition 8. *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 after the range condition.*

Theorem 1. *A sequence of two consecutive range conditions testing a common variable for nonoverlapping ranges can be reordered with no semantic effect on the program if (1) the sequence can only be entered through the first range condition, (2) the two range conditions each contain only instructions that cannot cause exceptions,*[1] *and (3) each range condition in the sequence has no side effects.*[2,3]

Corollary 1. *A consecutive sequence of range conditions testing a common variable for nonoverlapping ranges can be reordered with no semantic effect on the program if (1) the sequence can only be entered through the first range condition, (2) the sequence contains only instructions that cannot cause exceptions, and (3) each range condition in the sequence has no side effects.*

The detection of a sequence of reorderable range conditions is accomplished using the algorithm in Figure 4. Instead of storing a sequence of branches, we store a sequence of ranges. The algorithm first finds two range conditions testing nonoverlapping ranges for the same variable. Afterwards, it repeatedly detects an additional range condition until no more range conditions with nonoverlapping ranges can be found.[4] Note that the algorithm does not address dealing with the elimination of side effects and that the sequence is entered only through the first range condition. Transformations to make sequences reorderable are described later in the paper.

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 associated with this code segment. 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.

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) [7]. 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

[1] Range conditions were implemented in this study on the SPARC by using comparison and conditional branch instructions, which cannot cause exceptions.

[2] Note that condition codes are a special-purpose register on many architectures. A comparison instruction causes a side effect when the condition codes value it sets is live after the branch and used in subsequent branches.

[3] All proofs are given in the Appendix.

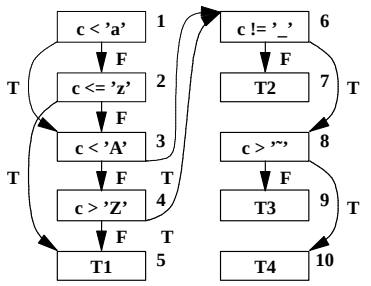
[4] If multiple range conditions with nonoverlapping ranges can follow the current range condition, then we choose a bounded range condition before an unbounded range condition. If there are still multiple options, then we choose a fall through successor before a taken successor in the sequence. We felt this would in general result in the longest sequence of branches to reorder.

```
# B is a basic block in the function.
# V is a variable in a branch being compared.
# R1 and R2 are ranges of values.
# N is the next basic block in the sequence.
# C is the current basic block in the sequence.
# Ranges is the set of ranges associated with the sequence.

# Find reorderable sequences of range conditions.
FOR each basic block B DO

    # Find the first two range conditions in the sequence.
    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

            # Find the remaining range conditions in the sequence.
            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;

# Finds the first two range conditions in a reorderable sequence.
BOOL FUNCTIONFind_First_Two_Conds
(B, V, out R1, out R2, out N)
    # B is the block containing the first range condition.
    # V is the variable compared in each range condition.
    # R1 is the range associated with the first range condition.
    # R2 is the range associated with the second range condition.
    # N is the next block in the sequence.
{
    # N1 is the block in the sequence after the first range condition
    # N2 is the block in the sequence after the second range condition

    # Check if two range conditions can be found starting at block B.
    IF (Find_Range_Cond({}, V, B, R1, N1) AND
        Find_Range_Cond({R1}, V, N1, R2, N2)) THEN
        N = N2;
        RETURN TRUE;
    ELSE

        # Check if two range conditions can be found with ranges
        # that do not overlap with the previous value of R1.
        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;
}

# Determine if there is a range condition for a range that does
#       not overlap with the existing set of ranges.
BOOL FUNCTION Find_Range_Cond(Ranges, V, B, out R, out N)
    # Ranges is the set of ranges already found.
    # V is the variable compared in each range condition.
    # B is the basic block being tested to see if it contains a range condition.
    # R is the range tested by the range condition.
    # N is the next block in the sequence after the range condition.
{
    # C is the constant being compared to the variable V in the branch.
    # I is the range when the branch exits the sequence by falling through.

    # Check if an appropriate branch and any side effects can be removed.
    IF (B has a branch that compares V to a constant C AND
        V is not affected in the block AND
        the result of the comparison is not used in a later branch) 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 branch 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;
}

# Returns true if R is a distinct range from the ones in Ranges.
BOOL FUNCTION Nonoverlapping(R, Ranges)
    # R is a range of values.
    # Ranges is a set of ranges.
{
    IF (R does not overlap with any of the ranges in Ranges) THEN
        RETURN TRUE;
    ELSE
        RETURN FALSE;
}
```

Figure 4: Detecting a Reorderable Sequence of Range Conditions

```
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	Char Range	Int Range	Target
1,2	['a'..'z']	[97..122]	T1
3,4	['A'..'Z']	[65..90]	T1
6	['_'..'_']	[95..95]	T2
8	['~'+1..MAX]	[127..MAX]	T4

(c) Reorderable Range Conditions

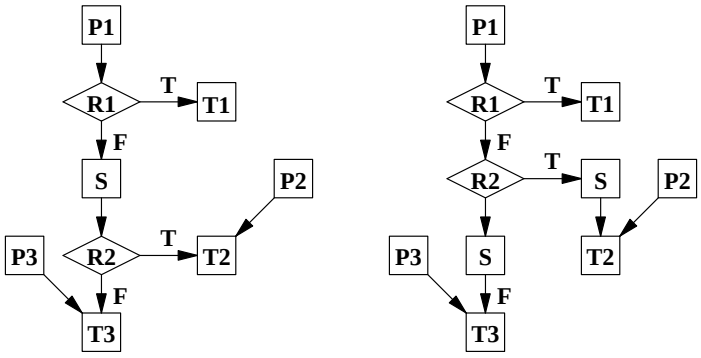
Figure 5: Example of Detecting Range Conditions

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 obtain an improved reordered sequence with respect to profile and cost estimates. The future work section discusses using profile information to improve other search methods.

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, we always duplicate the sequence of range conditions to ensure that the sequence is always entered at the head, which will be described in Section 8. Likewise, if a basic block containing the first range condition did have a preceding side effect, then it could be split 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 are typically 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 a compiler. Thus, each range condition can usually be accomplished with just comparison and branch instructions since the value of the branch variable is typically available in a register and the constants tested in the range condition are represented in the comparison instructions for most ranges of values.

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***. ***T1***, ***T2***, and ***T3*** are target basic blocks of the range conditions. ***P1***, ***P2***, and ***P3*** are predecessor basic blocks of the initial range condition and the last two targets. 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 and the sequence can only be entered through the first range condition.*

Corollary 2. *A consecutive sequence of range conditions 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.*

Figure 7 shows a code segment from the unix utility *wc*. The different **`if`** statements that compare **`c`** to constants are contiguous in the control flow except for first **`if`** statement due to the side effect updating **`charct`**.

```
for(;;) {
        c = getc(fp);
        if(c == EOF)
                break;
        charct++;
        if(' '<c && c<0177) {
                if(!token) {
                        wordct++;
                        token++;
                }
                continue;
        }
        if(c=='\n') {
                linect++;
        }
        else if(c!=' ' && c!='\t')
                continue;
        token = 0;
}
```

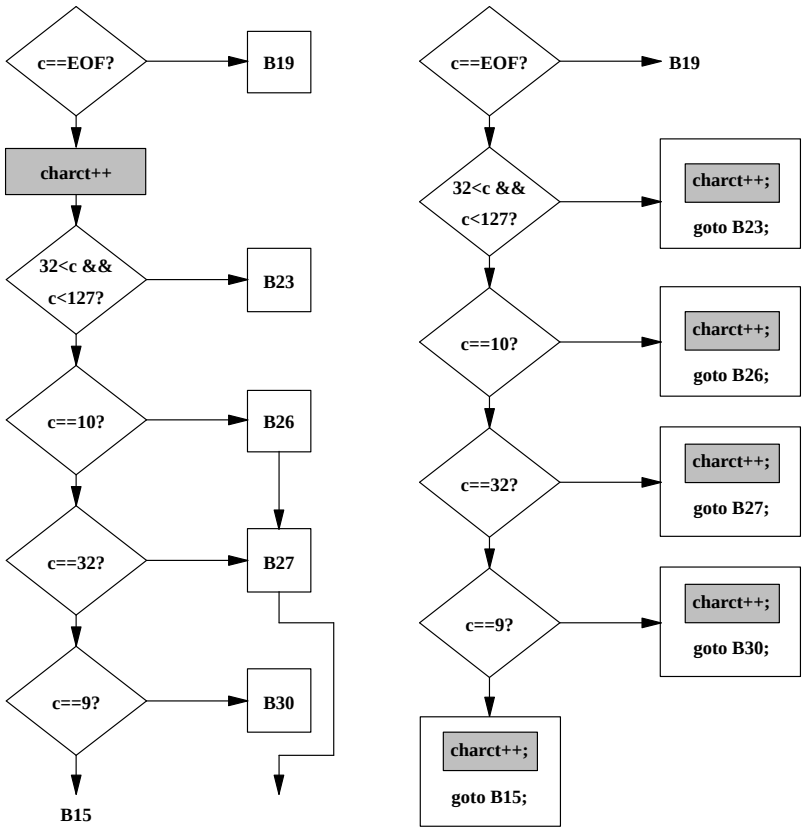
Figure 7: Source Code Segment from *wc*

Figure 8(a) shows the corresponding control-flow graph for the code segment in Figure 7. Since the side effect **`charct++`** must be executed whenever **`c != EOF`**, the compiler duplicates **`charct++`** at each target except for block 19, as shown in Figure 8(b). In general, for a given target the transformation technique has to duplicate all side effects along the path from the head of the sequence to the target.

An unconditional jump instruction could be added to the end of each one of the copies containing a duplicated side effect as shown in Figure 8(b). But that could possibly increase the dynamic number of instructions and also would make the cost estimation more complex. In order to avoid these two problems, a simple code duplication algorithm is used. The transformation technique duplicates basic blocks starting at the target by following the fall-through transitions until we reach a block containing an unconditional jump, return, or indirect jump instruction. Thus, an unconditional jump instruction need not be inserted since we duplicated code until another unconditional transfer of control is encountered that would have been executed anyway in that path. A similar approach is used when transforming code to improve branch prediction [12].

5. PRODUCING THE PROFILE INFORMATION

The profiling code for reordering range conditions checks if the common variable is within ranges of values. This profile information had to be collected in a different manner from conventional profiling. One may believe that instrumentation code could simply be inserted at the basic block containing a branch in a reorderable sequence or inserted either on the fall-through or taken transition. However, this approach will not be sufficient since each branch in a sequence of range conditions may not be encountered every time the sequence is executed. Likewise, a range condition in an original sequence may be executed without first executing the head of the sequence when there are multiple entry points to the sequence. A compiler needs to know how often each range condition in the sequence would have a transition out of the sequence given it is executed when the head of the sequence is encountered. The instrumentation code for obtaining profile information about the sequence is entirely inserted at the head of the sequence to check every range condition in the sequence. This involves duplicating the instructions associated with each branch as instrumentation code. However, additional ranges have to be determined from the ones calculated by the algorithm in Figure 4.



(a) Original CFG (b) CFG after Duplicating Side Effects

Figure 8: Control Flow Graph for Code Segment in Figure 7

Definition 9. *An **explicit range** is a range that is checked by a range condition.*

Definition 10. *A **default range** is a range that is not checked by a range condition.*

Consider the original sequence of range conditions in Figure 9(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 9(b) shows an equivalent sequence with these default ranges explicitly checked. Figure 9(c) shows a reordered sequence of range conditions, where the range condition for the last default range in Figure 9(b) is 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 9(d). We calculated these remaining ranges by sorting the explicit ranges and adding the minimum number of ranges to cover the remaining values.

All of the ranges, both explicit and default, are checked by inserting instrumentation code at the head of the sequence of branches. Thus, the code associated with Figure 9(a) would require 5 ranges to be checked and only 1 of 5 counters associated with this sequence would be incremented each time the head of the sequence of branches is encountered. The instrumentation code that would be inserted in the assembly file for the sequence in Figure 9(a) is illustrated at the source code level in Figure 10.

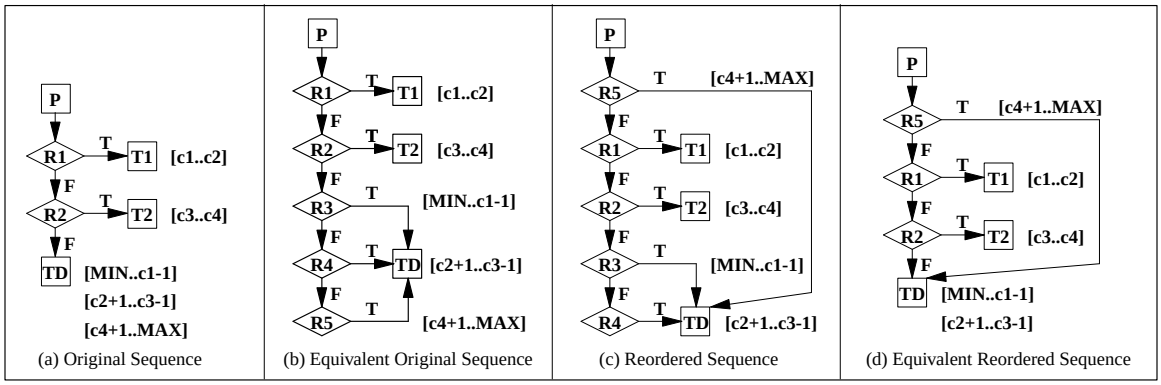


Figure 9: Example of Reordering Default Range Conditions

```
/* counters for sequence in Figure 9(a) */
unsigned int count[5];
...
/* start of instrumentation code testing branch variable v */
if (v < c1)
   count[0]++;   /* [MIN..c1-1] */
else if (c1 <= v && v <= c2)
   count[1]++;   /* [c1..c2] */
else if (c2+1 <= v && v <= c3-1)
   count[2]++;   /* [c2+1..c3-1] */
else if (c3 <= v && v <= c4)
   count[3]++;   /* [c3..c4] */
else
   count[4]++;   /* [c4..MAX] */
/* first branch in R1 of Figure 9(a) */
...
```

Figure 10: Instrumentation Code for Obtaining Profile Information
for the Sequence of Branches in Figure 9(a)

6. SELECTING THE SEQUENCE ORDERING

The ordering for a reorderable sequence of range conditions is chosen by using the items specified in the following definitions.

Definition 11. ***pi*** *is the probability that range condition* ***Ri*** *will exit the sequence of range conditions.*

Each ***pi*** is calculated using the profile information indicating how often the corresponding range condition ***Ri*** would exit the sequence if it is executed. Note that each ***pi*** is independent since the ranges are nonoverlapping. 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 [13].

Definition 12. ***ci*** *is the cost of testing range condition* ***Ri.***

Each ***ci*** is 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 13. *The **Explicit_Cost([R1,...,Rn])** is the estimated cost of executing a sequence of n range conditions when one of the n range conditions will be satisfied.*

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.

$$Explicit\_Cost([R_1, \ldots, R_n]) = p_1c_1 + p_2(c_1 + c_2) + \cdots + p_n(c_1 + c_2 + \cdots + c_n) \quad (1)$$

This explicit cost can be alternately expressed using summations.

$$Explicit\_Cost([R_1, \ldots, R_n]) = \sum_{i=1}^{n}(p_i \sum_{j=1}^{i} c_i)$$

Theorem 3. *A reorderable sequence of two consecutive explicit range conditions can be optimally ordered with respect to the probability and cost estimates as [**R1**,**R2**] when **p1/c1** ≥ **p2/c2**.*

Corollary 3. *A reorderable sequence of explicit range conditions can be optimally reordered as [**R1**, **R2**, ..., **Rn**], when **p1/c1** ≥ **p2/c2** ≥ ... **pn/cn** with respect to the probability and cost estimates.*

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.

However, there is also a default cost, which occurs when no range condition is satisfied and the control transfers to the default target. The default cost is shown in Equation 2 and Equation 3 shows the complete cost of a sequence, where only the first *n* ranges are explicit.

$$Default\_Cost([R_1, \ldots, R_n]) = (1 - (p_1 + \cdots + p_n))(c_1 + \cdots + c_n) \quad (2)$$

$$Cost([R_1, \ldots, R_n]) = Explicit\_Cost([R_1, \ldots, R_n]) + Default\_Cost([R_1, \ldots, R_n]) \quad (3)$$

Once only a single target remains, then the range conditions associated with that target need not be tested. Consider again the example in Figure 9(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*** is used as the default target, then at least one of the three range conditions should not be explicitly checked.

Definition 14. ***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 possible combinations of these range conditions that do not have to be explicitly checked. We used the ordering **p1/c1** ≥ ... ≥ **pm/cm** between the *m* ranges of a target to consider only *m*+1 possible combinations of default range conditions, {{}, {**Rm**}, {**Rm-1**,**Rm**}, ..., {**R1**,...,**Rm**}}. We 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. We then calculate the minimum of {**mindefault(T1)**, **mindefault(T2)**, ..., **mindefault(Tt)**}. Note that only the cost of *n*+1 sequences have to be considered, where *n* is the total number of ranges in the sequence.

Our approach is not guaranteed to be optimal. However, we also implemented an exhaustive approach to find the lowest cost sequence. We discovered that our approach always selected the optimal sequence for every reorderable sequence in every test program for the training data sets. Thus, selecting among the *m* combinations of default range conditions serves as an excellent heuristic.

Equation 4 represents the cost of executing a sequence of *n*-1 explicitly checked range conditions, where only range condition *i* is a default range.

$$\begin{aligned} Cost([R_1, \ldots, R_{i-1,} R_{i+1,} \ldots, R_n]) = {} & p_1 c_1 + \cdots + p_{i-1}(c_1 + \cdots + c_{i-1}) \\ & + p_{i+1}(c_1 + \cdots + c_{i-1} + c_{i+1}) + \cdots \\ & + p_n(c_1 + \cdots + c_{i-1} + c_{i+1} + \cdots + c_n) \\ & + p_i(c_1 + \cdots + c_{i-1} + c_{i+1} + \cdots + c_n) \end{aligned} \quad (4)$$

However, Equation 4 can be rewritten as Equation 5, where the cost of a sequence of range conditions with a default range can be calculated by subtracting the difference from Equation 1.

$$Cost([R_1, \ldots, R_{i-1,} R_{i+1,} \ldots, R_n]) = Explict\_Cost([R_1, \ldots, R_n]) + p_i(c_{i+1} + \cdots + c_n) - c_i(p_i + \cdots + p_n) \quad (5)$$

The ordering of a sequence of range conditions is selected using the algorithm in Figure 11. 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 5 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 condition 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 11: Sequence Ordering Selection Algorithm

7. IMPROVING THE SELECTED SEQUENCE

Other improvements are obtained after the ordering decision was made. A compiler can determine the best ordering of the two branches within a single range condition ***Ri*** that is of type Form 4 [c1..c2] shown in Table 1. The transformation technique 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, ***Ri***, 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 **[*R1*,*R2*,...,*Ri-1*]** have already been tested and the value of the common variable cannot be in these ranges if ***Ri*** is reached. Given that there are *n* total range conditions, we examined the probability for each of the remaining ranges, **[*Ri+1*,*Ri+2*,...,*Rn*]**, to determine the probability that **v < c1** versus that **v > c2**. Remember that these probabilities are obtained from the data obtained during the profile run, as stated in the description of Definition 11. Based on these probabilities, the branch is placed first that is most likely to determine if the range condition is not satisfied. In effect, we attempt to short circuit the second branch in a bounded range condition.

Another improvement we perform after the range conditions have been ordered is to eliminate redundant comparisons. For instance, consider Figure 12(a). There are two consecutive range conditions that test if the common variable is in the ranges **[const+1..max]** and **[const..const]**. Figure 12(b) shows a semantically equivalent comparison and branch for the first range condition. The comparison instruction within the second range condition becomes redundant and it is eliminated.

first comparison:	`IC = reg ? const+1;`	`IC = reg ? const;`
first branch:	`PC = IC>=0 then label1;`	`PC = IC>0 then label1;`
second comparison:	`IC = reg ? const;`	
second branch:	`PC = IC==0 then label2;`	`PC = IC==0 then label2;`
	(a) Before	(b) After

Figure 12: Eliminating Redundant Comparisons Example

8. APPLYING THE TRANSFORMATION

Once a branch ordering has been selected, the reordering transformation is applied. Figure 13(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 13(b) shows the control flow with the duplicated range conditions (***R1'***, ***R2'***, and ***R3'***) inserted. The predecessors of the first original range condition now have transitions to the first duplicated range condition. We always duplicated the sequence of range conditions before reordering since the duplicated sequence has only a single entry point at the first range condition through which the sequence can be entered. Note that the target ***TD*** in Figure 13(a) has a fall-through predecessor. Code starting at the target block ***TD*** is duplicated until an unconditional jump, return, or indirect jump was found. This approach avoids 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 is used when transforming code to improve branch prediction [12]. Figure 13(c) shows the control flow with the two side effects duplicated to allow the sequence of range conditions to be reordered. ***T2*** is also duplicated to avoid an extra unconditional jump. Figure 13(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 duplicated sequence. ***R1'*** and ***R2'*** have also been reversed. Figure 13(e) shows the code after applying dead code elimination. The original range conditions ***R1*** and ***R2*** are deleted, while range condition ***R3*** remains since it is still reachable from another path. Other optimizations, such as code repositioning and branch chaining to minimize unconditional jumps, are also reinvoked to improve the code.

Figure 14 shows the point at which reordering of branches is performed in the compilation by the *vpo* compiler. We decided to perform branch reordering late in the compilation process after most optimizations are performed. The same point is used during the first compilation pass to obtain the profile data. Performing this optimization late in the compilation process gives the compiler more opportunities to exploit the transformation and better estimates on the cost for each branch. The other optimizations that are reinvoked after branch reordering are also depicted in italics in the figure.

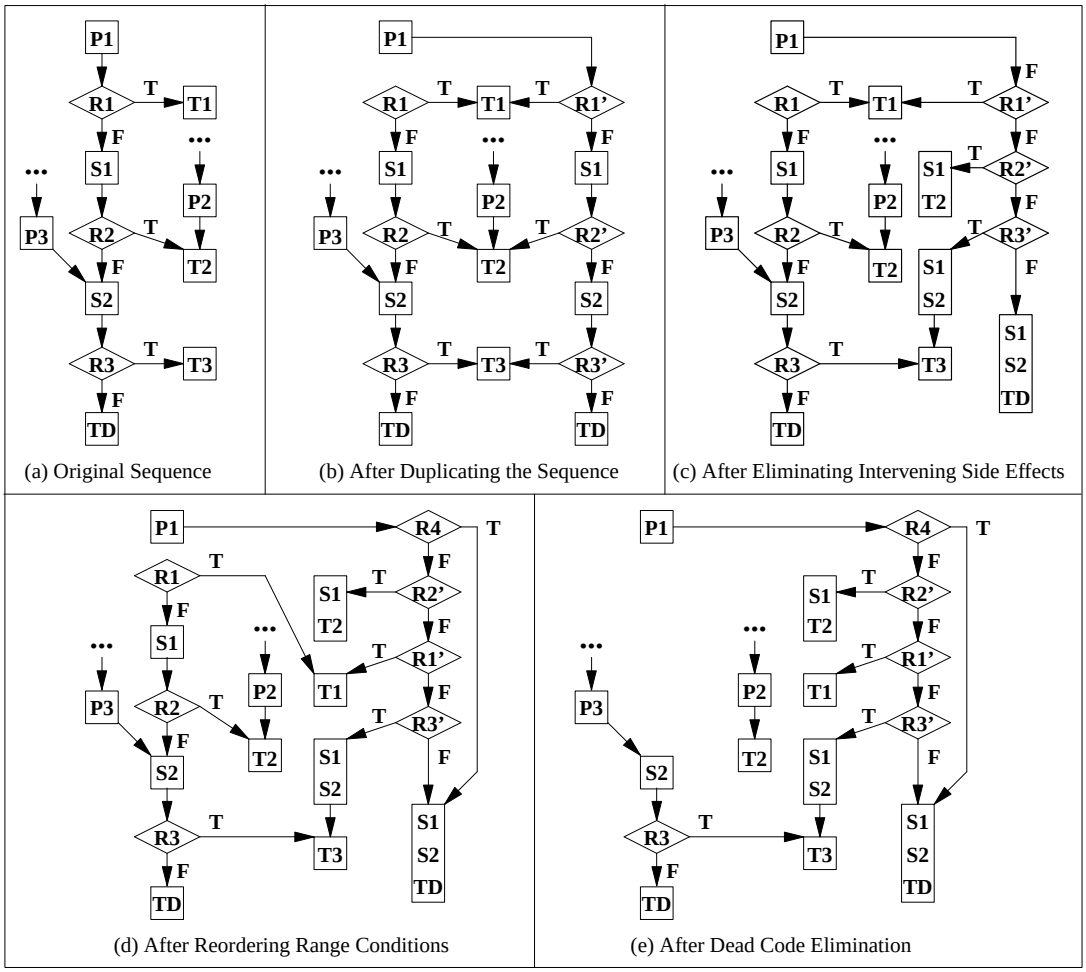


Figure 13: Applying the Reordering Transformation

9. EXPERIMENTAL RESULTS

Table 2 shows the three different sets of heuristics used when translating **`switch`** statements. The front end uses Heuristic Set I, which are the same heuristics used in the *pcc* front end [14], when compiling for a SPARC IPC and a SPARC 20. The authors used the dual loop method [15] and found that indirect jumps on the SPARC Ultra II were about four times more expensive than indirect jumps on the SPARC IPC or SPARC 20.[5] This was due to the SPARC Ultra II using branch prediction to reduce the cost of branches, while the SPARC IPC and SPARC 20 did not use branch prediction. All three machines provided no support for indirect jump prediction. Therefore, Heuristic Set II used for the Ultra only generates an indirect jump when $n \geq 16$. Finally, Heuristic Set III always generates a linear search, which achieves the maximum benefit from reordering.

[5] The dual loop method is used to estimate the time required for a short sequence of instructions, which is difficult to obtain directly using system calls. This method involves executing a loop for a large number of iterations. For our experiments three different loops were produced. The first loop only has the instructions to execute the loop. The second loop includes a linear sequence of branches comparing a common variable. The third loop has an indirect jump within the loop. One may determine the loop overhead by timing the first loop. This loop overhead is then subtracted from the time required to execute the second and third loops. These times are then divided by the number of loop iterations to estimate the time for the sequence of branches and the indirect jump. Thus, we were able to determine the relative cost of executing branches versus indirect jumps on each of these three machines.

Branch Chaining
Useless Jump Elimination
Dead Code Elimination
Eliminating Unconditional Jumps by Code Positioning
Instruction Selection
Evaluation Order Determination
Global Instruction Selection
Register Assignment
Jump Minimization by Reversing Branches
Instruction Selection
DO {
    Register Allocation
    Instruction Selection
    Common Subexpression Elimination
    Dead Variable Elimination
    Loop Optimizations Performed Innermost First
        Code Motion
        Recurrence Elimination
        Loop Strength Reduction
        Induction Variable Elimination
    Useless Jump Elimination
    Strength Reduction
    Instruction Selection
} WHILE (change)
Branch Reordering
Dead Code Elimination
Eliminating Unconditional Jumps by Code Positioning
Branch Chaining
Useless Jump Elimination
Setup Entry and Exit
Filling Delay Slots

Figure 14: Ordering of Optimizations

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

Measurements were collected on the code generated for the SPARC architecture by the *vpo* compiler [1] using the *ease* environment [17]. 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. For each program we used realistic training and test data that were often similar to the examples found in the man pages describing these applications. In each case the training data was smaller than the test data, resulting in fewer instructions executed in the training run than in the test runs reported in this section. Table 4 shows the dynamic frequency measurements that were obtained. The *Original Insts*

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

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 has significant benefits both in reducing the total number of instructions and conditional branches. One may notice that the transformation has a slight negative impact on *hyphen*, which occurred for a couple of reasons. First, different test input data is used as compared to the training input data for the results presented in the table. When we use the same test input data as the training input data, the number of branches never increased. Second, the reordering transformation is applied after all optimizations except for filling delay slots.

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

Table 4: Dynamic Frequency Measurements

Sometimes delay slots are filled from the other successor and do not execute a useful instruction. One should note that inconsistent filling of delay slots 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*. Thus, the benefits depend on how often sequences of branches could be profitably reordered and what percent of the total instructions executed did such sequences comprise.

We found that the original default target in a sequence is almost always selected as the default target for the reordered sequence. However, the profile data also indicates 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.

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 decreases as fewer indirect jumps were generated. In fact, the average number of instructions for the test programs that were executed after reordering is actually the smallest for *Set III*. 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 were simulated by modifying the *ease* environment [17] associated with the *vpo* compiler [1]. Table 5 shows the branch prediction measurements that were obtained for the SPARC Ultra II. The SPARC Ultra II uses branch prediction (for branches and not indirect jumps) and the SPARC IPC and SPARC 20 do not. A branch predictor is often described using the notation (*m*, *n*), where the predictor will use the behavior of the last *m* branches encountered to choose from 2^m predictors and each predictor will use *n* bits [18]. The SPARC Ultra II 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 [16]. However, the misprediction results for branch

Program	Original Number of Mispredictions	Mispredictions after Reordering	Ratio of Decreased Instructions to Increased Mispredictions
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

reordering were mixed. Nine of the test programs had fewer mispredictions after reordering and the remaining eight had more. Overall, the average number of mispredictions increased.

After examining a few of the sequences that have been reordered, we realized that our branch reordering transformation avoids the execution of the predictably executed branches. Consider the source code in Figures 1(b) and 1(c), which would be translated into a loop containing branches comparing `c` to a blank, newline, and an end-of-file character. Assume that most of the characters read in are letters, digits, or punctuation symbols and there are rarely consecutive blanks or newlines. When `c > ' '`, all three branches in Figure 1(b) and the branch checking if `c > ' '` in Figure 1(c) are all likely to be predicted correctly. When `c <= ' '`, the branches in Figure 1(b) will have zero to three branches mispredicted. However, the branches in Figure 1(c) will have one to four mispredictions. Thus, it appears that branch reordering can increase the number of branch mispredictions.

One may question the value of branch reordering since it sometimes increases the number of branch mispredictions. In Table 5 we show that the average ratio of decreased instructions executed to the increased number of branch mispredictions was 1221.94 to 1 for the eight programs that exhibited an increased number of mispredictions. Thus, the increase in mispredictions was on average 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. Again the benefit of reducing the number of instructions executed was far more significant than the increase in the number of branch mispredictions.

Entries	(0,1) Predictor		(0,2) Predictor		(2,2) Predictor	
	Mispredictions after Branch Reordering	Decreased Instructions to Increased Mispredictions	Mispredictions after Branch Reordering	Decreased Instructions to Increased Mispredictions	Mispredictions after Branch Reordering	Decreased Instructions to Increased Mispredictions
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%		+19.32%		+21.38%	

Table 6: Branch Prediction Measurements for a Variety of Predictor Configurations

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()* on the SPARC IPC and SPARC 20 and the *ptime* utility on the SPARC Ultra II. 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, which did contribute to the execution times. Also, the benefits for the Ultra II were probably not as significant due to the impact of issuing multiple instructions simultaneously and some additional branch mispredictions. The SPARC Ultra II had a superscalar implementation and the SPARC IPC and SPARC 20 did not.

Machine	Heuristic Set	Average Execution Time
SPARC IPC	I	-4.94%
SPARC 20	I	-5.57%
SPARC Ultra II	II	-3.65%

Table 7: Average Effect from Reordering on Execution Times

Table 8 shows static measurements for the same set of programs. There is 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 *Percent Reordered* column indicates the percentage of these

Switch Translation Heuristics	Program	Instructions	Total Seqs	Reordered Sequences		
				Percent Reordered	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

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 coverage of the branches in a program 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.

We also found that sometimes sequences did have intervening side effects. We found that 1.554% of the sequences that were reordered initially had intervening side effects, such as the one shown for *wc* in Figures 7 and 8. Note that this does not include side effects in the first range condition. In this case we just split the initial basic block of the first range condition into two blocks, one containing the side effects and one containing the comparison and branch instructions.

Figures 15, 16, and 17 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.

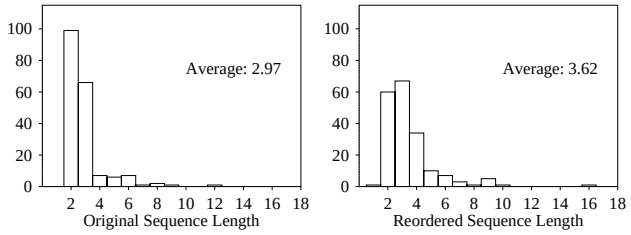


Figure 15: Sequence Length for Heuristic Set I

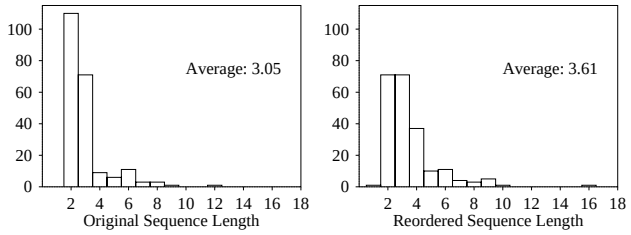


Figure 16: Sequence Length for Heuristic Set II

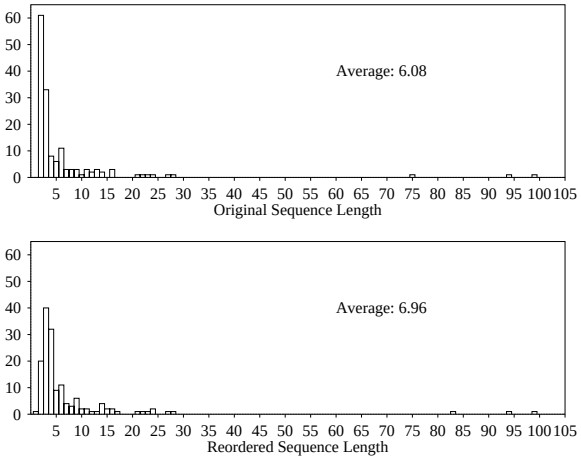


Figure 17: Sequence Length for Heuristic Set III

Table 9 shows that the overhead of performing this optimization is relatively low. We obtained the compilation times for each of the test programs with and without applying branch reordering. The compilation time when performing the reordering code-improving transformation is only 1.044 times slower on average than the normal compilation time. This includes not only the time for performing the code-improving transformation, but also the time required to reinvoke a number of additional transformations, as shown in Figure 14. Note that this overhead does not include the first compilation pass. The execution time when obtaining profile information is 2.545 times slower on average than normal execution time. The profiling overhead can vary depending upon the frequency in which reorderable sequences of branches are encountered relative to the total number of instructions executed. However, similar profile information could be used to support a variety of other code-improving transformations.

Program	Compilation Time	Profiling Time
awk	1.035	1.514
cb	1.076	1.909
cpp	1.017	1.118
ctags	1.061	3.286
deroff	1.024	2.200
grep	1.030	1.800
hyphen	1.073	2.750
join	1.062	4.000
lex	1.072	2.429
nroff	1.052	2.105
pr	1.035	2.467
ptx	1.013	1.300
sdiff	1.026	2.231
sed	1.041	1.300
sort	1.037	7.714
wc	1.067	3.833
yacc	1.033	1.316
average	1.044	2.545

Table 9: Compilation and Profiling Time Overhead

10. CONCLUSIONS AND FUTURE WORK

This paper describes 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. We also described techniques for transforming a sequence of branches to make it reorderable. 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.

There are several advantages of using the approach described in this paper for reordering branches. First, the approach is simple to understand and implement. Second, we have shown that opportunities for reordering sequences of branches to reduce the number of executed branches occur frequently in applications. Third, branches are expensive and the relative cost of executing branches compared to executing other instructions is likely to increase as the complexity of pipelining and multiple issue architectures grows. Finally, our approach may be a good fit for run-time optimization systems since the analysis required for our transformation is relatively inexpensive.

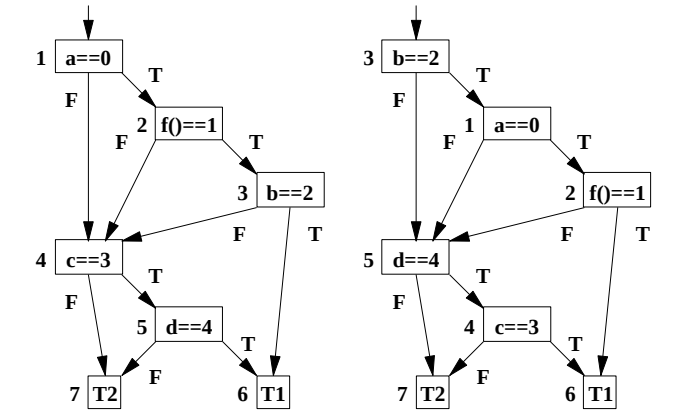
There are several areas in which reordering branches could be extended. A sequence of range conditions is one of several approaches that could be used to determine a target associated with the value of an expression. Essentially, a sequence of range conditions is a linear search. Some of these other approaches include performing a binary search, using a jump table, and hashing [9]. 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. For instance, a sequence of 100 branches comparing a common variable to a compact number of cases may not guarantee that an indirect jump from a table is the best approach. If a few of these cases dominate and the cost of an indirect jump is more expensive than a conditional branch, then the compiler could instead generate a sequence of branches for these cases and coalesce the remaining cases into a jump table.

A different type of sequence of branches that can be reordered using profile data would consist of consecutive branches with a common successor. Figure 18(a) shows a C source code segment containing relational and logical expressions and Figure 18(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 18(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

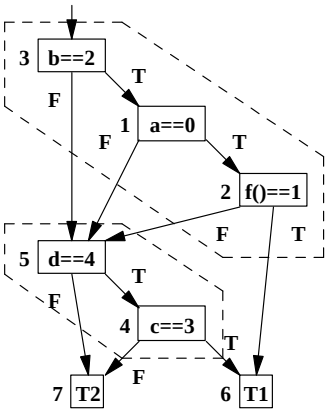
```
if (a == 0 && f() == 1 && b == 2 || c == 3 && d == 4)
   T1;
else
   T2;
```

(a) C Source Code Segment

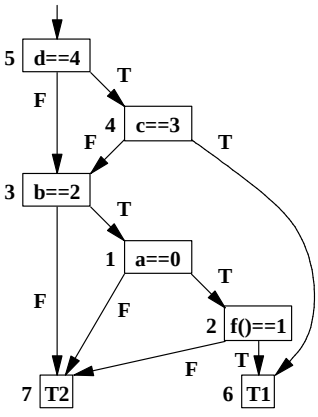


(b) Before Reordering Branches

(c) After Reordering Branches



(d) Before Reordering Sequences



(e) After Reordering Sequences

Figure 18: Reordering Branches with Common Successors

depicted in block 2, could have significant performance benefits. Figure 18(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 18(e) shows that these sequences can be reordered when there are no side effects between the sequences.

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 [19].

APPENDIX

Proof of Theorem 1: 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 T_1, T_2, and T_3 are reached, (2) blocks T_1, T_2, and T_3 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 R_1 and R_2 have no side effects. Condition 2 is satisfied since the ranges associated with T_1, T_2, and T_3 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 R_1 or R_2 since the instructions in a range condition cannot raise error exceptions. □

Proof of Corollary 1: Suppose for sequences with length of 2, 3, ..., n, Corollary 1 is true. Now we need to prove for a sequence with length of n+1, [R_1, R_2, ..., R_{n+1}], is semantically equivalent to any sequence that is an arbitrary permutation of these range conditions.

(i) Suppose the first range condition of the permutation is R_1. Then the rest of the permutation is a permutation of [R_2, R_3, ..., R_n, R_{n+1}], which is a sequence of length n and by induction hypothesis it is equivalent to [R_2, R_3, ..., R_n, R_{n+1}]. In this case, the whole permutation is equivalent to [R_1, R_2, R_3, ..., R_n, R_{n+1}]. We know this sequence is valid since this is the original order of the sequence.

(ii) Suppose the first range condition of the permutation is R_i ($i \neq 1$). Then the rest of the permutation is a permutation of [R_1, R_2, ..., R_{i-1}, R_{i+1}, ..., R_n, R_{n+1}] (except R_i), which is a sequence of length n and it is equivalent to [R_1, R_2, ..., R_{i-1}, R_{i+1}, ..., R_n, R_{n+1}] (a sequence of length n without R_i). So the whole permutation is equivalent to [R_i, R_1, R_2, ..., R_{i-1}, R_{i+1}, ..., R_n, R_{n+1}]. Since the sequence [R_i, R_1] has a length of 2 and thus is equivalent to [R_1, R_i], so the whole sequence is equivalent to [R_1, R_i, R_2, ..., R_{i-1}, R_{i+1}, ..., R_n, R_{n+1}]

Now R_1 is the first range condition, by (i) we know that the whole permutation is equivalent to [R_1, R_2, R_3, ..., R_n, R_{n+1}]. □

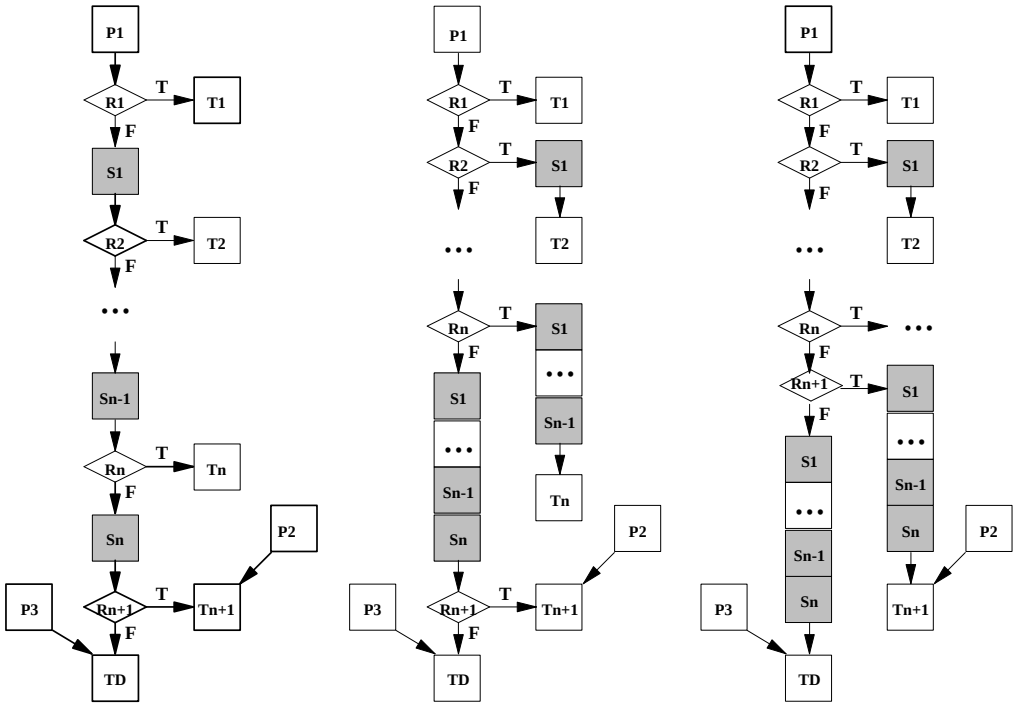
Proof of Theorem 2: Consider the original and transformed sequences of range conditions in Figure 6. The two sequences are semantically equivalent given that (1) state of the program is equivalent in both sequences when blocks T_2 and T_3 are reached, (2) blocks T_2 and T_3 are always reached in both sequences under the same conditions, and (3) no new error exceptions are raised.

Condition 1 is satisfied since the range condition R_2 in the transformed sequence has no side effects, S is executed in both sequences when T_2 or T_3 is reached after executing R_2, and S is not executed if T_2 or T_3 is reached without executing R_2. Condition 2 is satisfied since S does not affect the branch variable of R_2.

Condition 3 can be satisfied by noting that no new side effects are introduced in the transformed sequence. □

Proof of Corollary 2: Suppose for sequences with length *n*, Corollary 2 is true. Now we need to prove for a sequence with length of *n*+1, [***R1***, ***R2***, ..., ***Rn+1***], as shown in Figure 19(a), it can be transformed to have no intervening side effects and still have the same semantic effect on the program.

Consider the sequence [***R1***, ***R2***, ..., ***Rn***], which is of length *n*. By induction hypothesis we know that this sequence can be transformed to have no intervening side effects and still have the same semantic effect on the program, as shown in Figure 19(b). Since the only predecessor of range condition ***Rn+1*** is range condition ***Rn***, so the duplicated side effects ***S1***, ***S2***, ..., ***Sn-1*** can be inserted directly into ***Rn+1*** rather than creating a new block containing ***S1***, ***S2***, ..., ***Sn-1***. Now consider the sequence [***Rn***, ***Rn+1***]. It satisfies the condition of theorem 2 and as a result the side effect ***S1***, ***S2***, ..., ***Sn*** can be moved out of the sequence as shown in Figure 19(c). Combining the above transformations together, we have proved the corollary. □



(a) Original Sequence (b) Intermediate Sequence (c) Final Transformed Sequence

Figure 19: Moving Side Effects from a Sequence of *n*+1 Range Conditions

Proof of Theorem 3: 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.

$$\begin{aligned} \textit{Explicit\_Cost}([R_1, R_2]) &\leq \textit{Explicit\_Cost}([R_2, R_1]) \\ p_1c_1 + p_2(c_1 + c_2) &\leq p_2c_2 + p_1(c_2 + c_1) \\ p_1c_1 + p_2c_1 + p_2c_2 &\leq p_2c_2 + p_1c_2 + p_1c_1 \\ p_2c_1 &\leq p_1c_2 \\ p_2/c_2 &\leq p_1/c_1 \\ p_1/c_1 &\geq p_2/c_2 \end{aligned}$$

□

Proof of Corollary 3: Suppose for a sequence of length *n*, Corollary 3 is true. In order to prove [***R1***, ***R2***, ..., ***Rn+1***] is the optimal order for a *n*+1 length sequence, we need to prove that an arbitrary permutation [***Ri1***, ***Ri2***, ..., ***Rin+1***] will have an explicit cost that is at least as great as *Explicit_Cost*([***R1***, ***R2***, ..., ***Rn+1***]).

(i) Assume the first condition is $\boldsymbol{R_1}$. If we only consider a subsequence formed by [$\boldsymbol{R_2}$, $\boldsymbol{R_3}$, ..., $\boldsymbol{R_{n+1}}$], then it is a sequence of length n. By induction, [$\boldsymbol{R_2}$, $\boldsymbol{R_3}$, ..., $\boldsymbol{R_{n+1}}$] should have the lowest explicit cost. The *Explicit_Cost*([$\boldsymbol{R_{i_2}}$, $\boldsymbol{R_{i_3}}$, ..., $\boldsymbol{R_{i_{n+1}}}$]) is greater than or equal to the *Explicit_Cost*([$\boldsymbol{R_2}$, $\boldsymbol{R_3}$, ..., $\boldsymbol{R_{n+1}}$]). This proves that the sequence [$\boldsymbol{R_1}$, $\boldsymbol{R_2}$, ..., $\boldsymbol{R_{n+1}}$] has a cost that is less than or equal to the cost of the sequence [$\boldsymbol{R_1}$, $\boldsymbol{R_{i_2}}$, ..., $\boldsymbol{R_{i_{n+1}}}$].

(ii) Assume the first condition is $\boldsymbol{R_i}$, where $i \neq 1$. By applying the induction hypothesis and the result given in (i), we have:

$$
\begin{aligned}
& \mathit{Explicit\_Cost}([R_i, R_{i_2}, \ldots, R_{i_{n+1}}]) & \\
& \geq \mathit{Explicit\_Cost}([R_i, R_1, \ldots, R_{n+1}]) & \text{(sort } \boldsymbol{R_{i_2}}, \ldots, \boldsymbol{R_{i_{n+1}}} \text{ by } p/c) \\
& \geq \mathit{Explicit\_Cost}([R_1, R_i, \ldots, R_{n+1}]) & \text{(swap } \boldsymbol{R_1} \text{ and } \boldsymbol{R_i}) \\
& \geq \mathit{Explicit\_Cost}([R_1, R_2, \ldots, R_{n+1}]) & \text{(sort } \boldsymbol{R_i}, \ldots, \boldsymbol{R_{n+1}} \text{ by } p/c)
\end{aligned}
$$

□

ACKNOWLEDGEMENTS

The authors thank Jack Davidson for allowing *vpo* to be used for this research. Michael Sjödin, Chris Healy, Richard Whaley, and the anonymous reviewers provided several helpful suggestions that improved the quality of the paper. This research was supported in part by the National Science Foundation grants EIA-9806525, CCR-9904943, and EIA-0072043.

REFERENCES

1. 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).
2. J. Dongarra and A. Hinds, "Unrolling Loops in FORTRAN," *Software Practice & Experience* **9** pp. 219-226 (1979).
3. J. W. Davidson and S. Jinturkar, "Aggressive Loop Unrolling in a Retargetable, Optimizing Compiler," *Proceedings of Compiler Construction Conference*, pp. 59-73 (April 1996).
4. 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).
5. 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).
6. 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).
7. 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).
8. M. Schlansker, S. Mahlke, and R. Johnson, "Control CPR: A Branch Height Reduction Optimization for EPIC Architectures," *Proceedings of the SIGPLAN '99 Symposium on Programming Language Design and Implementation*, pp. 155-168 (May 1999).
9. 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).
10. 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).
11. 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).
12. 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).

13. 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).
14. S. C. Johnson, "A Tour Through the Portable C Compiler," *Unix Programmer's Manual, 7th Edition* **2B** p. Section 33 (January 1979).
15. 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).
16. G. Uh, *Effectively Exploiting Indirect Jumps*, PhD Dissertation, Florida State University, Tallahassee, FL (December 1997).
17. 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).
18. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition*, Morgan Kaufmann, San Francisco, CA (1996).
19. M. Yang, *Improving Performance by Branch Reordering*, Masters Thesis, Florida State University, Tallahassee, FL (1998).