THE FLORIDA STATE UNIVERSITY
COLLEGE OF ARTS AND SCIENCES

IMPROVING PERFORMANCE BY BRANCH REORDERING

By

MINGHUI YANG

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Summer Semester, 1998

The members of the Committee approve the thesis of Minghui Yang defended on March 31, 1998.

David B. Whalley
Professor Directing Thesis

Theodore P. Baker
Committee Member

Kyle A. Gallivan
Committee Member

Approved:

R.C. Lacher, Chair, Department of Computer Science

ACKNOWLEDGEMENTS

I am deeply indebted in gratitude to my advisor, Dr. David Whalley, for his able guidance and encouragement. It is to his excellent method of teaching and mentoring that I owe my kindling of interest in the area of optimizing compilers. I thank my committee members Dr. Baker and Dr. Gallivan for reviewing this thesis and subsequent valuable suggestions. Finally, I give thanks to Gang-Ryung Uh, Mikael Sjodin and Chris Healy for their assistance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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 thesis 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.

CHAPTER 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 having a common successor. For instance, a logical expression may consist of several relational expressions connected by logical operators (e.g. `||` and `&&` in C). Each relational expression will typically be translated into a conditional branch. Applying a logical operator between two relational expressions will result in a common successor for these associated two branches (i.e. the *True* or *False* target of the logical expression). For instance, Figure 1.1 shows two different logical expressions with the common successors identified. Nested control statements, such as `if` and `while` statements may also result in similar reorderable sequences.

In Figure 1.1(a), for example, control flow will reach code segment `x` if either `a == 0` or `b == 1` is True. With the semantics of the C language, the expression `b == 1` will not be evaluated if the evaluation of expression `a == 0` turns out to be True. In programming language parlance, we say in this case expression `a == 0` short-circuits expression `b == 1`. If we know the fact that expression `b == 1` is more likely to short-circuit expression `a == 0`, we would

```
if (a == 0 || b == 1)
   x;
y;
```

(a) x Is the Common Successor
If Either Expression is True

```
if (a == 0 && b == 1)
   x;
y;
```

(b) y Is the Common Successor
If Either Expression Is False

Figure 1.1: Example Sequences of Branches with a Common Successor.

like to restructure Figure 1.1(a) in the way as shown in Figure 1.2(a).

```
if (b == 1 || a == 0)
   x;
y;
```

(a) Equivalent Structure of Code
Segment of Figure 1.1(a)

```
if (b == 1 && a == 0)
   x;
y;
```

(b) y Is the Common Successor
Segment of Figure 1.1(b)

Figure 1.2: Reordering of Branches in Figure 1.1.

Another 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 cases to warrant the use of an indirect jump from a jump table. Also, control statements may often compare the same variable more than once. Consider the following original code segment in Figure 1.3(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

```
while (c=getchar())
       != EOF)
   if (c == '\n')
      x;
   else if (c == ' ')
      y;
   else
      z;
```

(a) Original Code

```
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 1.3: Example Sequence of Comparisons with the Same Variable.

were quite performance conscious and would attempt to manually reorder such statements. A conventional manual reordering shown in Figure 1.3(b) would improve performance by performing the three comparisons in reverse order. In fact, 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 1.3(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. Manually reordering a sequence of comparisons of a common variable or inserting extra `if` statements to achieve performance benefits, as shown in Figure 1.3(b) and (c), can lead to obscure code. A general improving transformation to reorder branches automatically may help encourage the use of good software engineering principles by performance conscious programmers.

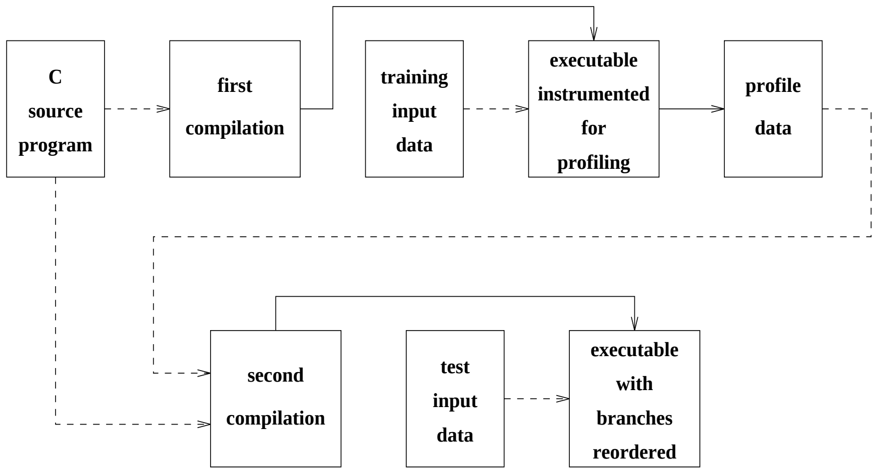


Figure 1.4: Overview of Compilation Process for Branch Reordering.

This thesis presents methods for reordering code to reduce the number of branches executed. Figure 1.4 shows 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.[1] Sequences of reorderable branches 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 the sequence will transfer control out of the sequence. This profile data and estimated cost for executing each branch is used during a second compilation pass to estimate the most beneficial branch reordering. Delay slots are filled after branch reordering and the final executable

[1]We do not want to allow filling of delay slots first. This would complicate reordering branches since before filling delay slots a branch would have to be the last RTL within a block. Eventually, we may reinvoke some other transformations after branch reordering if we believe there would be additional opportunities as a result of the branch reordering transformation.

is produced. The transformation was frequently applied with reductions in instructions executed and execution time.

The thesis is organized as follows. Chapter 2 briefly describes some related work. Chapters 3 to 6 are presented in the same sequential order by which the transformation is performed. Chapter 3 shows how to detect a sequence of reorderable branches. Chapter 4 presents how to perform profiling and gives real examples of profile information. Chapter 5 discusses how to select the ordering of branches. Chapter 6 explains how to apply the reordering transformation. Chapter 7 presents results including the reductions for the total number of instructions, total number of branch instructions, execution time, etc. Chapter 8 discusses future work in this topic and Chapter 9 gives the conclusions for the thesis.

CHAPTER 2
RELATED WORK

There has been some research on 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 [1]. Figure 2.1 gives an example of loop unrolling.

```
for (i = 0; i < 102; i++)
   f(i);
```

(a) Before Loop Unrolliing

```
for (i = 0; i < 100; i += 4) {
   f(i);
   f(i+1);
   f(i+2);
   f(i+3);
}
for ( ; i < 102; i++)
   f(i);
```

(b) After Loop Unrolling

Figure 2.1: Loop Unrolling

To perform loop unrolling, in general we need to replicate the loop body one or more times, change the increment of the loop induction variable, and add new code to execute the excess iterations of the loop as shown in Figure 2.1.

Loop unswitching moves a conditional branch with a loop-invariant condition before the loop and replicates the loop in each of the two destinations of the branch [2]. Figure 2.2 is an example of loop unswitching. The relational expression `a == 0` is loop-invariant, hence it can be moved out of the loop. After unswitching, the expression `a == 0` is only evaluated once, therefore reducing the execution time.

```
for (i = 0; i < 100; i++)
   if (a == 0)
      x[i] = 0;
   else
      y[i] = 0;
```

(a) Before Loop Unswitching

```
if (a == 0)
   for (i = 0; i < 100; i++)
      x[i] = 0;
else
   for (i = 0; i < 100; i++)
      y[i] = 0;
```

(b) After Loop Unswitching

Figure 2.2: Loop Unswitching

Different search methods based on static heuristics for the cases associated with a multiway statement have been studied [3]. These methods include linear search, binary search, hashing, and indirect jump from a table. These studies all assume that each case of a multiway statement are equally likely.

Conditional branches have also been avoided by code replication [4]. This method determines if there are paths where the result of a conditional branch will be known and replicates code to avoid execution of the branch. The method of avoiding conditional branches using code replication has been extended using interprocedural analysis [5].

Finally, conditional branches have been coalesced together into an indirect jump from a jump table [6]. This method extends the use of an indirect jump table far beyond the translation of a multiway statement and allows many other coalescing opportunities to be exploited.

There have also been studies about reordering or aligning basic blocks to minimize pipeline penalties associated with conditional branches [7, 8]. 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. Usually these approaches use profile information to minimize the number of taken branches on architectures where taken branches cause delays.

CHAPTER 3

DETECTING A SEQUENCE OF REORDERABLE BRANCHES

Reorderable sequences of branches detected in this thesis consist of two types, those branches having a common successor block and those branches that compare the same variable or expression to constants. When a sequence of branches with a common successor overlapped with a sequence of branches comparing a common variable, the latter type of sequence was used since this type of reordering was found to be more effective as described later in the thesis.

3.1 Detecting a Sequence of Reorderable Branches with a Common Successor

Having two branches with a common successor means that both of the blocks containing the branches have a transition to the same target block. Figure 3.1 depicts an example of two branches with a common successor T.

Definition 1. *A* ***consecutive sequence of branches*** $[\boldsymbol{B}_1, \ldots, \boldsymbol{B}_n]$ *is a path in the control flow graph, where each node is a basic block that contains a branch and each edge is a control-flow transition to the next basic block in the sequence.*

Definition 2. *A* ***reorderable sequence of branches*** *is a consecutive sequence where the branches may be interchanged in any permutation with no effect on the semantics of the program.*

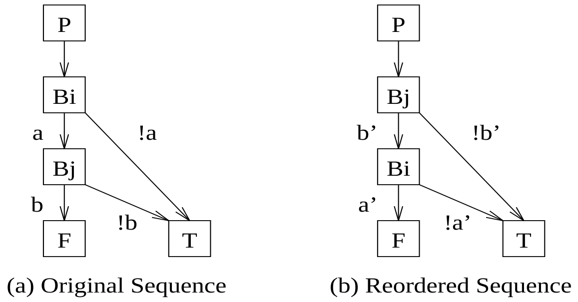


Figure 3.1: Reordering Branches with Common Successors.

Definition 3. *A* ***side effect*** *in a basic block is an instruction that updates a variable and the updated value can reach a use of that variable or register outside of the basic block.*

Function calls were also considered side effects in this thesis since interprocedural analysis was not performed at this time.

Definition 4. ***Sets(B)*** *is the set of variables and registers that are set in basic block B.*

Definition 5. ***Uses(B)*** *is the set of variables and registers that are used in basic block B.*

Definition 6. ***Ins(B)*** *is the set of variables and registers that are live when basic block B is entered.*

Theorem 1. *A consecutive sequence of two branches* $[B_i, B_j]$ *with a common successor can be reordered with no semantic effect on the program if*

1. *both* B_i *and* B_j *do not have side effects,*

2. *Sets(B_j) $\cap$ Uses(B_i) $\cap$ Ins(B_i) $= \emptyset$, and*

3. *the only predecessor of B_j is B_i.*

Proof: Consider the original and reordered sequences of blocks in Figure 3.1. B_i and B_j are blocks that contain branches that can be reordered. a, b, a', b' are conditions associated with the branches.

The two sequences are semantically equivalent given that

1. state of the program is equivalent in both sequences when blocks F and T are reached, and

2. blocks F and T are always reached in both sequences under the same conditions.

3. no new error exceptions are introduced.

Condition 1 can be satisfied by noting that blocks B_i and B_j have no side effects. Note that if B_i did have a side effect, then it could be split apart into the portion with a side effect and the portion without one. Only the latter portion would be considered by the theorem.

To satisfy condition 2, we can evaluate under what conditions blocks F and T will be reached in both sequences, note that the only predecessor of B_j is B_i.

- In the original sequence F will be reached iff a `&&` b.

- In the original sequence T will be reached iff $!a$ `||` a `&&` $!b$, which simplifies to $!a$ `||` $!b$.

- In the reordered sequence F will be reached iff a' `&&` b'.

- In the reordered sequence T will be reached iff b' `&&` $!a'$ `||` $!b'$, which simplifies to $!a'$ `||` $!b'$.

Condition 2 can be satisfied if $a \equiv a'$ and $b \equiv b'$. These conditions are equivalent since both B_i and B_j have no side effects and $Sets(B_j)$ does not affect $Uses(B_i) \cap Ins(B_i)$.

Condition 3 can be satisfied if both B_i and B_j cannot raise exceptions. Note no new error exceptions will be introduced after leaving the sequence due to condition 1 and 2. On one hand, B_j must not raise an exception if moved before B_i. In this situation, the branch in B_i serves as a guard preventing an exception in B_j. For instance, this could occur if there is a memory reference in B_j that has an invalid address. [1] On the other hand, B_i must not raise an error exception since otherwise if B_j is moved before B_i, B_j may short circuit B_i and hence the exception raised by B_i may get lost. [2]

$\square$

The compiler was updated to conservatively check if each memory reference in B_j was to an identifiable scalar reference. Pointer analysis could be performed to sometimes ease this restriction. No other instructions in our target architecture can cause an error exception. [3]

Corollary 1. *A consecutive sequence of branches with a single common successor can be reordered in any arbitrary permutation given that there are no*

[1]The compiler identified the following memory references as safe: reference a global scalar, reference a local scalar variable, reference a parameter from caller's activation record.

[2]There are different opinions about whether B_i should have to raise an error exception or not. Some people think it is OK to get rid of an exception through code transformation.

[3]The compiler should also check for potential integer divisions by zero in B_j, but in our target architecture, integer division is implemented as a function call and hence is considered as side effect which violates condition 1 of theorem 1.

intervening side effects, the sets of a given block in the sequence do not affect the uses that are live entering any preceding block in the original sequence, and the sequence is only entered through the first branch.

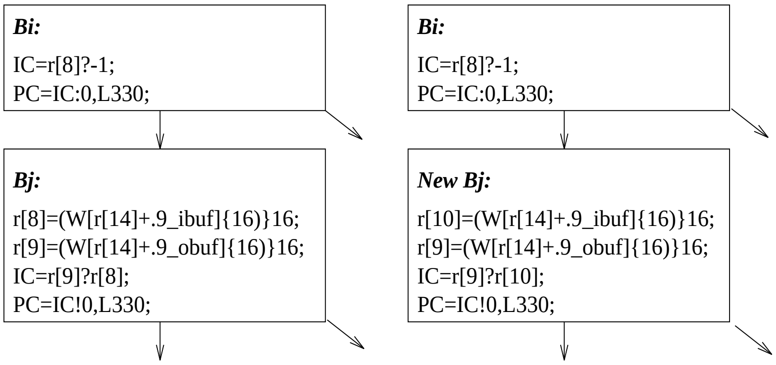
Proof: Suppose for sequences with length of 2, 3, ..., n, the above corollary is true. Now we need to prove for a sequence with length of $n+1$, [B_1, B_2, ..., B_{n+1}], is semantically equivalent to any sequence that is an arbitrary permutation of these blocks.

(i) Suppose the first block of the permutation is B_1, then the rest of the permutation is a permutation of [B_2, B_3, ..., B_n, B_{n+1}], which is a sequence of length n and by induction hypothesis it is equivalent to [B_2, B_3, ...,B_n, B_{n+1}]. In this case, the whole permutation is equivalent to [B_1, B_2, B_3, ..., B_n, B_{n+1}]. We know this sequence is valid since this is the original order of the sequence.

(ii) Suppose the first block of the permutation is B_i ($i \neq 1$), then the rest of the permutation is a permutation of [B_1, B_2, ..., B_{i-1}, B_{i+1}, ..., B_n, B_{n+1}] (except B_i), which is a sequence of length n and it is equivalent to [B_1, B_2, ..., B_{i-1}, B_{i+1}, ..., B_n, B_{n+1}] (a sequence of length n without B_i). So the whole permutation is equivalent to [B_i, B_1, B_2, ..., B_{i-1}, B_{i+1}, ..., B_n, B_{n+1}]. Since the sequence [B_i, B_1] has a length of 2 and thus is equivalent to [B_1, B_i], so the whole sequence is equivalent to [B_1, B_i, B_2, ..., B_{i-1}, B_{i+1}, ..., B_n, B_{n+1}].

Now B_1 is the first block, by (i) we know that the whole permutation is equivalent to B_1, B_2, B_3, ..., B_n, B_{n+1}.

□



(a) Before Register Renaming (b) After Register Renaming

Figure 3.2: Register Renaming.

The detection of a reorderable sequence of more than two branches with a common successor is not described in this thesis. This study limited the detection of such sequences to two branches since we could not guarantee an optimal ordering for longer sequences. The reason for this limitations is discussed in Chapter 5. Also, in our experiments the $Sets(B_j) \cap Uses(B_i) \cap Ins(B_i) = \emptyset$ restriction never prevented branch reordering since the compiler was able to find an available register and successfully apply register renaming every time a set in B_j affected a use that was live entering B_i. An example of register renaming is shown in Figure 3.2. The register $r[8]$ is live entering B_i, but B_j sets $r[8]$. After renaming $r[8]$ with $r[10]$ in B_j, the condition $Sets(B_j) \cap Uses(B_i) \cap Ins(B_i)$ $= \emptyset$ is satisfied so this becomes a reorderable sequence. [4]

[4]Theoretically, it is possible that all the registers are busy at that point and hence register renaming cannot be performed.

3.2 Detecting a Sequence of Reorderable Range Conditions Comparing a Common Variable to Constants

The approach used for finding a sequence of reorderable branches that compare a common variable or expression was quite different from finding a sequence of reorderable branches having a common successor since it required associating branch targets with ranges of values.

Definition 7. *A* ***range*** *is a contiguous range of integer values.*

Definition 8. *A* ***range condition*** *is a branch or a pair of consecutive branches that tests if an integer variable is within a range.*

Definition 9. *A* ***consecutive sequence of range conditions*** $[\boldsymbol{R}_1, \ldots, \boldsymbol{R}_n]$ *is a path in the control flow graph, where each node is a range condition testing the same variable and each edge is control-flow transition to the next range condition in the sequence.*

Definition 10. *A* ***reorderable sequence of range conditions*** *is a consecutive sequence 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 3.1, 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 3.1.

Table 3.1: Range Conditions and Corresponding Range of Values.

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

Figure 3.3(a) depicts a sequence of two range conditions. R_1 and R_2 are range conditions that can consist of one or two branches that check to see if a variable is in a range. T_1 and T_2 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. T_3 is the default target block when neither range condition is satisfied. Figure 3.3(b) shows how the sequence can be reordered. Note that T_1 and T_2 can be the same target.

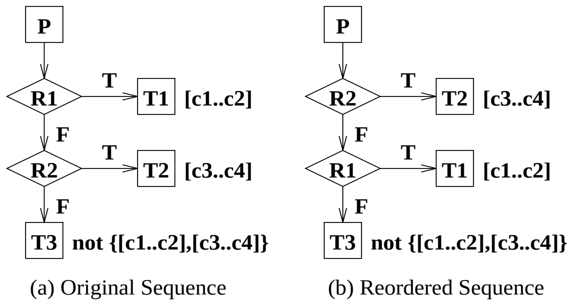


Figure 3.3: Reordering Range Conditions with No Intervening Side Effects.

Definition 11. *Two ranges are* ***nonoverlapping*** *if they do not have any common values.*

Theorem 2. *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 3.3. The two sequences are semantically equivalent given that

1. state of the program is equivalent in both sequences when blocks T_1, T_2, and T_3 are reached, and

2. blocks T_1, T_2, and T_3 are always reached in both sequences under the same conditions.

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 comparison and conditional branch instructions cannot raise error exceptions on the target architecture. Note that there will be no assignments of registers or variables associated with a range condition since each range condition could be accomplished with just comparison and branch instructions.

□

Corollary 2. *A sequence of range conditions associated with 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.* [5]

The detection of a sequence of reorderable range conditions was accomplished using the algorithm in Figure 3.4. 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.

Figure 3.5 shows an example of detecting a sequence of range conditions. Figures 3.5(a) and 3.5(b) show a C code segment and the corresponding control flow produced by the compiler. Figure 3.5(c) shows the sequence of reorderable range conditions that are detected using the algorithm in Figure 3.5. Note that all of the ranges detected are nonoverlapping.

A more complete set of branches that compare a common variable or expression 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) [6].

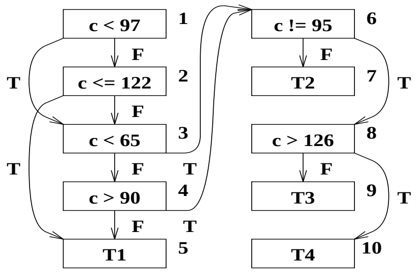
[5]The proof is the same as Corollary 1.

```
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 FUNCTION Find_First_Two_Conds(B,V,R1,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 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(I,Ranges);
   RETURN FALSE;
}
```

Figure 3.4: 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	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 3.5: Example of Detecting Range Conditions.

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. Note that it is possible that the reordering may be improved if a binary search was used instead of a linear search (i.e. a sequence of range conditions). However, we will show later in this thesis that a near-optimal ordering of a sequence can be found using an algorithm requiring only linear complexity. Our initial investigations have shown that an algorithm to select an optimal binary search would require both exponential time and space complexity. We will describe our plans to investigate the use of a binary search and contrast the benefits of a binary search versus a linear search in

Chapter 10.

CHAPTER 4

HANDLING SIDE EFFECTS IN A COMMON VARIABLE SEQUENCE

We can convert a sequence of nonoverlapping range conditions with intervening side effects into a sequence of nonoverlapping range conditions without side effects through code replication, given that the side effects do not affect the common variable being compared. [1] Thus, we can further exploit some opportunities where there are intervening side effects among range conditions.

Theorem 3. *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 4.1. [2]

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

[1] We allow side effects between range conditions, but not within properly bounded range conditions. A properly bound range condition is treated as atomic and indivisible.

[2] The side effect S is actually in a basic block containing R_2.

3. no new error exceptions are raised.

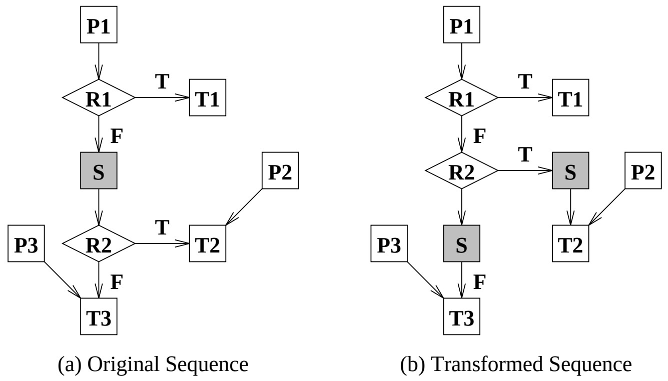


Figure 4.1: Moving Side Effects from a Sequence of Two Range Conditions

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. Note that the transitions from P_2 and P_3 require that the replicated side effects S be placed in separate basic blocks.

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.

□

Corollary 3. *A sequence of range conditions attached with 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.*

Proof: Suppose for sequences with length of n, the above corollary is true. Now we need to prove for a sequence with length of $n+1$, $[R_1, R_2, \ldots, R_{n+1}]$, as shown in Figure 4.2 (a), it can be transformed to have no intervening side effects and still have the same semantic effect on the program.

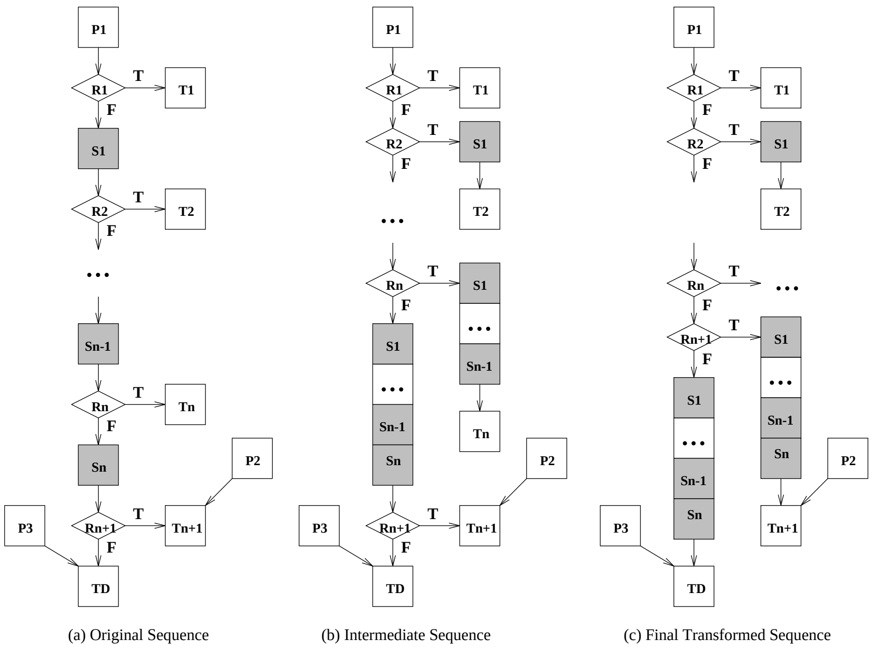


Figure 4.2: Moving Side Effects from a Sequence of $n+1$ Range Conditions

Consider the sequence $[R_1, R_2, \ldots, R_n]$, it is a sequence 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 4.2 (b). Since the only predecessor of range condition R_{n+1}

is block R_n, so the replicated side effects $S_1S_2, \ldots, S_{n-1}$ can be inserted directly into R_{n+1} rather than creating a new block containing $S_1S_2, \ldots, S_{n-1}$. Now consider the sequence $[R_n,\ R_{n+1}]$, it satisfies the condition of theorem 3 and as a result the side effect $S_1, S_2 \ldots, S_n$ can be moved out of the sequence as shown in Figure 4.2 (c). Combine the above transformations together, we have proved the corollary.

□

```
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 4.3: Source Code Segment from wc.c .

Figure 4.3 shows a real code segment from unix utility program *wc.c*. Figure 4.4 is the corresponding control flow graph for code segment in Figure 4.3. We use B_n to represent block n.

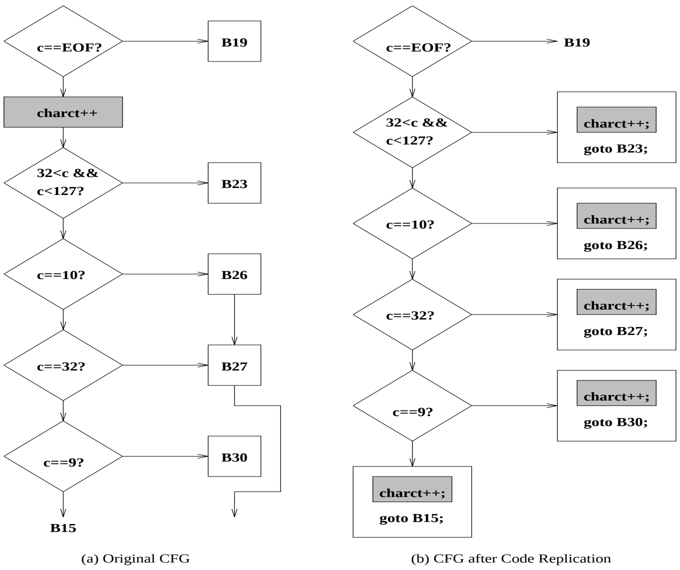


Figure 4.4: Handling Side Effect in a Common Variable Sequence.

Since the side effect portion `charct++` must be executed whenever `c != EOF`, so we want to replicate `charct++` at each target except block 19. In general, for a given target we have to replicate all those side effect portions along the path from the head of the sequence to the target.

An unconditional branch instruction could be added to the end of each one of the copies as shown in Figure 4.4. But that could possibly increase the dynamic number of instructions and also would make cost estimation more complex. In order to avoid these two problems, a simple code replication algorithm was used. We just replicate block(s) by following the fall-through link of the target until we reach a block containing an unconditional jump, return, or indirect jump instruction. A similar approach has been used when transforming code to improve branch prediction [9].

For example block 26 in Figure 4.4 is not a terminal block since it ends with a fall-through path. Block 27 is a terminal block since it ends with an unconditional branch. Hence instead of generating the unconditional branch `goto block 26` after `charct++`, we are going to replicate block 27 following block 26, as shown in Figure 4.5.

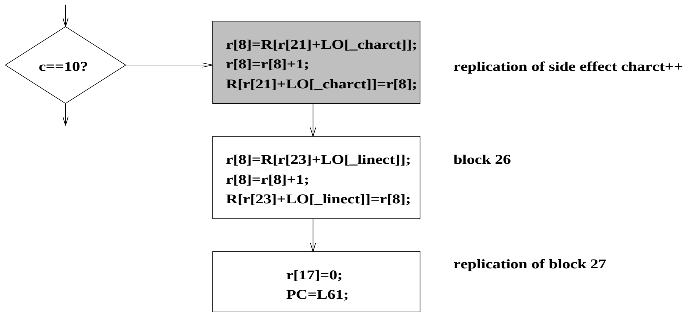


Figure 4.5: Avoid Extra Unconditional Branch by Block Duplication.

Note that when we produce profile information, we don't have to replicate any code. All we need is to detect a reorderable sequence of range conditions. Code replication occurs when we really perform branch reordering if the reordering is deemed to be beneficial.

CHAPTER 5
PERFORMING PROFILING

The profile information required for branch reordering 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 and either on the fall-through or taken transition. However, this approach will not be sufficient since each branch or range condition in the sequence may not be encountered every time the sequence is executed. The compiler needs to know how often each branch or 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 branch or range condition in the sequence.

5.1 Producing Profile Information for Common Successor Sequence

Profile information for common successor sequence is produced by inserting instrumentation code at the head of a sequence to check each branch in the sequence. First, we declare a set of counters for each sequence, including a counter indicating the number of times that the head of the sequence is entered, and a counter for each branch indicating the number of times that it is not taken. Since the length of a reorderable sequence of branches with a common successor is limited to 2, we will declare 3 counters for each sequence. Second, we insert instrumentation code to increment those counters when the head of a sequence is entered or corresponding branches are not taken. This involved replicating the instructions associated with each branch as instrumentation code. Finally we insert a call to a function (`__ease_end_g`) that dumps the profile information (the values of these counters) into a file (`profile.dat`) at every possible exit point in the program.

Figure 5.1 gives an example of inserting instrumentation code. Figure 5.1 (a) shows a common successor sequence of length 2 with the target label L233 as the common successor. The head of the sequence (i.e. block 6) is split into two separate portions, one with side effect (the first RTL) and the other without side effect (the last three RTLs). The instrumentation instructions is inserted right after the side effect portion as shown in Figure 5.1 (b). The lines ending with '!' are instrumentation instructions inserted to increment those counters. The lines starting with a '-' are SPARC assembly instructions and will be copied directly into the target SPARC assembly code. Note that RTLs associated with each branch are replicated before the first original branch in the sequence.

Figure 5.2 shows the complete code of the function `__ease_end_g`. `__ease_main_head_1` is the counter indicating the number of times that the first sequence of function `main` is entered. `__ease_main_nt_1[0]` is the counter for the first branch of the sequence indicating how many times it is not taken.

The function `__ease_end_g` was called at every possible exit point in the instrumented executable code. It is generated automatically and is linked with the instrumented executable. In order to generate this function automatically,

```
! block 7
r[8]=HI[_nfile];
r[8]=R[r[8]+LO[_nfile]];
IC=r[8]?1;
PC=IC'0,L290;
! block 8
r[8]=HI[_hflag];
r[8]=R[r[8]+LO[_hflag]];
IC=r[8]?0;
PC=IC:0,L290;
```

(a) Original Common Successor Sequence

```
! block 7
-       .seg    "data"                                   !
-       .align  4                                        !
-       .global ___ease_succeed_head_1                   !
-___ease_succeed_head_1:                                 !
-.word 0                                                 !
-       .global ___ease_succeed_nt_1                     !
-___ease_succeed_nt_1:                                   !
-.word 0                                                 !
-.word 0                                                 !
-       .seg    "text"                                   !
-       std     %o4,[%sp-8]                              !
-       sethi   %hi(___ease_succeed_head_1),%o5          !
-       ld      [%o5+%lo(___ease_succeed_head_1)],%o4    !
-       add     %o4,1,%o4                                !
-       st      %o4,[%o5+%lo(___ease_succeed_head_1)]    !
-       ldd     [%sp-8],%o4                              !
r[8]=HI[_nfile];
r[8]=R[r[8]+LO[_nfile]];
IC=r[8]?1;
PC=IC'0,L0022;
-       nop                                              !
-       std     %o4,[%sp-8]                              !
-       sethi   %hi(___ease_succeed_nt_1+0),%o5          !
-       ld      [%o5+%lo(___ease_succeed_nt_1+0)],%o4    !
-       add     %o4,1,%o4                                !
-       st      %o4,[%o5+%lo(___ease_succeed_nt_1+0)]    !
-       ldd     [%sp-8],%o4                              !
-L0022:                                                  !
r[8]=HI[_hflag];
r[8]=R[r[8]+LO[_hflag]];
IC=r[8]?0;
PC=IC:0,L0023;
-       nop
-       std     %o4,[%sp-8]                              !
-       sethi   %hi(___ease_succeed_nt_1+4),%o5          !
-       ld      [%o5+%lo(___ease_succeed_nt_1+4)],%o4    !
-       add     %o4,1,%o4                                !
-       st      %o4,[%o5+%lo(___ease_succeed_nt_1+4)]    !
-       ldd     [%sp-8],%o4
-L0023:
r[8]=HI[_nfile];
r[8]=R[r[8]+LO[_nfile]];
IC=r[8]?1;
PC=IC'0,L290;
! block 8
r[8]=HI[_hflag];
r[8]=R[r[8]+LO[_hflag]];
IC=r[8]?0;
PC=IC:0,L290;
```

(b) Instrumented Common Successor Sequence

Figure 5.1: Instrumentation Code.

```
void __ease_end_g()  {
    FILE* fp;
    extern int __ease_main_head_1;
    extern int __ease_main_nt_1[];
    extern int __ease_main_head_2;
    extern int __ease_main_nt_2[];
    extern int __ease_main_head_3;
    extern int __ease_main_nt_3[];
    extern int __ease_number_head_1;
    extern int __ease_number_nt_1[];
    extern int __ease_cal_head_1;
    extern int __ease_cal_nt_1[];
    int i;

    fp = fopen("profile.dat", "w");
    fprintf(fp, "main\n\n");
    fprintf(fp, "%d\n", __ease_main_head_1);
    for (i = 0; i < 2; i++)
        fprintf(fp, "%d ", __ease_main_nt_1[i]);
    fprintf(fp, "\n\n");

    fprintf(fp, "%d\n", __ease_main_head_2);
    for (i = 0; i < 2; i++)
        fprintf(fp, "%d ", __ease_main_nt_2[i]);
    fprintf(fp, "\n\n");

    fprintf(fp, "%d\n", __ease_main_head_3);
    for (i = 0; i < 2; i++)
        fprintf(fp, "%d ", __ease_main_nt_3[i]);
    fprintf(fp, "\n\n");

    fprintf(fp, "number\n\n");
    fprintf(fp, "%d\n", __ease_number_head_1);
    for (i = 0; i < 2; i++)
        fprintf(fp, "%d ", __ease_number_nt_1[i]);
    fprintf(fp, "\n\n");

    fprintf(fp, "cal\n\n");
    fprintf(fp, "%d\n", __ease_cal_head_1);
    for (i = 0; i < 2; i++)
        fprintf(fp, "%d ", __ease_cal_nt_1[i]);
    fprintf(fp, "\n\n");
    fclose(fp);
}
```

Figure 5.2: Function `__ease_end_g` .

we need to know how many sequences there are in a function. We use a file (`profile.inf`) to capture this information, as shown in Figure 5.3.

```
main
2
getchar
1
```

Figure 5.3: Example of `profile.inf` File for Common Successor Sequence.

Before performing any optimization the compiler will open the file `profile.inf`, or create the file `profile.inf` if it does not already exist. This file contains the name of a function followed by the number of reorderable sequences for that function. The compiler will read the information from `profile.inf` into an array if the file already exists. Then the compiler is going to update the information in the array when performing profiling operation. Before the compiler terminates normally, it is going to write the information in the array back to the file `profile.inf`.

The information in `profile.inf` is solely for automating the process of producing the function `__ease_end_g`. Even when we compile a program with multiple modules, the scheme of using `profile.inf` still works since function names are unique even across different modules. It is up to the programmer to remove this file before compiling a program for the first time.

Figure 5.4 shows an example portion of profile information for common successor sequences stored in `profile.dat`. In this example, there are two reorderable common successor sequences detected in function `main`. The first sequence in `main` was never entered, so all the three counters are 0. The second sequence in `main` was entered 230 times. Its first branch was not taken 196 times, or it was taken 230 - 196 = 34 times. The second branch was not taken 111 times, or was taken 230 - 111 = 119 times. Since the second branch would have transferred the control out of the sequence more often than the first branch, if both branches have the same cost, this sequence will be identified to be reordered to improve performance.

```
main
0
0 0
230
196 111

getch
1368
1232 1245
```

Figure 5.4: Example of `profile.dat` File for Common Successor Sequence.

5.2 Producing Profile Information for Common Variable Sequence

The profiling code for reordering range conditions checks if the common variable is within ranges of values. However, additional ranges have to be determined from the ones calculated by the algorithm in Figure 3.4.

Definition 12. *An* ***explicit range*** *is a range that is checked by a range condition.*

Definition 13. *A* ***default range*** *is a range that is not checked by a range condition.*

Consider the original sequence of range conditions in Figure 5.5(a). There are additional ranges associated in the default target T_D since these ranges will span any remaining values not covered by the other ranges. It is assumed in this figure that $MIN < c_1$, $c_2+1 < c_3$, and $c_4 < MAX$. Figure 5.5(b) shows an equivalent sequence with these default ranges explicitly checked. Figure 5.5(c) shows a reordered sequence of range conditions, where the range condition for the last default range in 5.5(b) was placed first in the sequence. Once a point is reached in the sequence where there is only a single target possible, all remaining range conditions need not be explicitly tested as shown in Figure 5.5(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 5.6 (a) is an example showing a portion of the profile information for common variable sequences stored in `profile.dat`. There are 2 reorderable sequences in the function *cmp*, and both are never entered. Function *skip* contains three reorderable sequences. The first sequence of *skip* was entered 1052 times. With all ranges (both explicit ranges and default ranges) being checked, the length of the sequence is 5. The first four ranges were never satisfied, while the last range, which was a default range was always satisfied. We can greatly improve performance if we explicitly check this default range first.

The function `__ease_end_g` in Figure 5.2 was extended to print out profile information for common variable sequences. This was accomplished by storing additional information, or counters for common variable sequences. The instru-

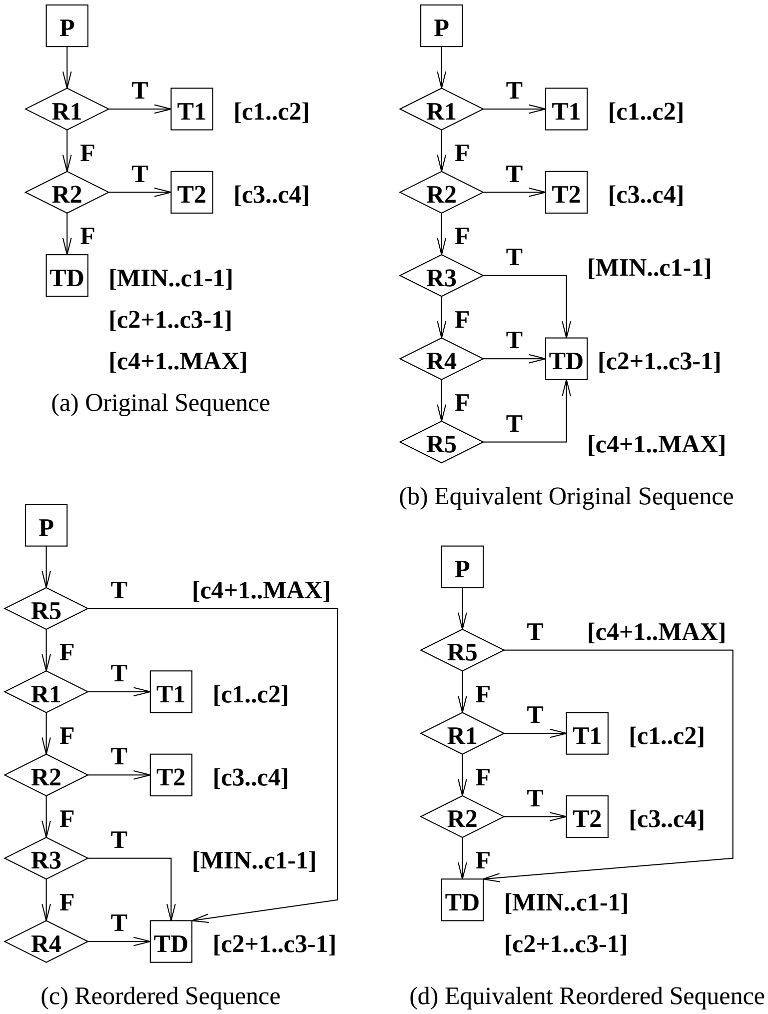


Figure 5.5: Example of Reordering Default Range Conditions.

```
cmp
5
0
0 0 0 0 0
5
0
0 0 0 0 0

skip
5
1052
0 0 0 0 1052
6
36518
0 0 1052 0 0 35466
5
0
0 0 0 0 0
```

(a) Portion of File profile.dat

```
cmp
2
5 5
skip
3
5 6 5
```

(b) Portion of File profile.inf

Figure 5.6: Example of Profile Information For Common Variable Sequence.

mentation code was inserted to test each range (not each branch) including the default ranges. Figure 5.6 (b) gives an example of `profile.inf` for common variable sequences. Unlike common successor sequences, which always have length of 2, common variable sequences may have various lengths. Function *cmp* has two common variable sequences, each with length of 5. Function *skip* has three common variable sequences, with lengths of 5, 6 and 5, respectively.

CHAPTER 6
SELECTING THE ORDERING OF BRANCHES

The ordering for a reorderable sequence of branches or range conditions was chosen by using two factors defined as follows.

Definition 14. $\boldsymbol{p_i}$ *is the probability that* N_i *(branch* B_i *or range condition* R_i*) will exit the sequence.*

Each p_i was calculated using the profile information indicating how often the corresponding branch B_i (in a common successor sequence) or range condition R_i (in a common variable sequence) 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 [10].

Definition 15. $\boldsymbol{c_i}$ *is the cost of performing* N_i *(branch* B_i *or range condition* R_i*).*

Each c_i was estimated by determining the number of instructions required to perform the corresponding branch or range condition. This cost includes the conditional branch, associated comparison, 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 selected. In these cases, a conservative estimation of the cost was used.

Definition 16. $\boldsymbol{Cost}([\boldsymbol{N}_1, \ldots, \boldsymbol{N}_n)]$ *is the estimated cost of executing a sequence of nodes (branches or range conditions.)*

The cost of a sequence of branches is calculated as a sum of products. One factor is the probability that a branch will be reached and will exit the sequence. The other factor is the cost of performing that branch and all preceding branches in the sequence. Note that the last branch in the sequence will always exit the sequence when reached.

6.1 Selecting the Order of a Sequence of Branches with Common Successors

Equation 1 represents the cost of executing a sequence of two reorderable branches with common successors $[B_i, B_j]$.

$$Cost([B_i, B_j]) = p_i c_i + (1 - p_i)(c_i + c_j) \leq p_j c_j + (1 - p_j)(c_j + c_i) \quad (1)$$

Theorem 4. *A sequence of two reorderable branches with common successors can be optimally ordered as $[B_i, B_j]$ when $p_i/c_i \geq p_j/c_j$ with respect to the probability and cost estimates.*

Proof: The optimal ordering of a sequence of two branches with a common successor can be obtained when the cost of one ordering is less than the cost of the other ordering.

$$Cost([B_i, B_j]) \leq Cost([B_j, B_i])$$

$$\Leftrightarrow \; p_i c_i + (1 - p_i)(c_i + c_j) \leq p_j c_j + (1 - p_j)(c_j + c_i)$$

$$\Leftrightarrow \; p_i c_i + c_i + c_j - p_j c_i - p_i c_j \leq p_j c_j + c_j + c_i - p_j c_j - p_j c_i$$

$$\Leftrightarrow \; c_i + c_j - p_i c_j \leq c_j + c_i - p_j c_i$$

$$\Leftrightarrow \; -p_i c_j \leq -p_j c_i$$

$$\Leftrightarrow \; p_i c_j \geq p_j c_i$$

$$\Leftrightarrow \; p_i / c_i \geq p_j / c_j$$

$\square$

Intuitively, this means that it is desirable to first execute the branch that has a high probability of exiting the sequence combined with a low cost. Note that we cannot determine an optimal ordering based on the profile data that was obtained in chapter 4 when the number of branches with common successors is greater than two. Consider three branches B_i, B_j and B_k, where p_i=0.8, p_j=0.6, p_k=0.4, and $c_i = c_j = c_k = 2$. One can determine an ordering $[B_i, B_j, B_k]$ where $pi/ci \geq pj/cj \geq pk/ck$ (i.e. $0.4 \geq 0.3 \geq 0.2$). However, it is possible that B_j will only exit the sequence when B_i has already exited the sequence. Likewise, B_k may only exit the sequence when B_i does not. The optimal ordering in this case would be $[B_i, B_k, B_j]$. Determining an optimal ordering for a sequence of such branches would require obtaining profile data about all combinations of branch results in the sequence. Using such profile data would be expensive in both time and space. The compiler limited the length of these sequences to two branches in this study to ensure that the reordered sequence always resulted in an improvement with respect to the estimated probabilities and costs. The most common length of sequences of branches with common successors we encountered was two.

6.2 Selecting the Order of a Sequence of Range Conditions Comparing a Common Variable

A sequence of explicit range conditions can be optimally ordered with respect to the probability and cost estimates. First, consider the cost of an entire sequence of all the range conditions (i.e. the default range conditions also being specified.)

Equation 2 represents the cost of executing a sequence of n range conditions, where every range is explicitly checked.

$$Explicit\_Cost([R_1, \ldots, R_n])$$
$$= p_1c_1 + p_2(c_1 + c_2) + \ldots + p_n(c_1 + c_2 + \ldots + c_n) \quad (2)$$

Theorem 5. *A sequence of two reorderable range conditions can be optimally ordered as* $[R_i, R_j]$ *when* $p_i/c_i \geq p_j/c_j$ *with respect to the probability and cost estimates.*

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.

$$Explicit\_Cost([R_i, R_j]) \leq Explicit\_Cost([R_j, R_i])$$

$$\Leftrightarrow \; p_ic_i + p_j(c_i + c_j) \leq p_jc_j + p_j(c_j + c_i)$$

$$\Leftrightarrow \; p_ic_i + p_jc_i + p_jc_j \leq p_jc_j + p_ic_j + p_ic_i$$

$\Leftrightarrow\ p_j c_i \leq p_i c_j$

$\Leftrightarrow\ p_j / c_j \leq p_i / c_i$

$\Leftrightarrow\ p_i / c_i \geq p_j / c_j$

□

Corollary 4. *A reorderable sequence of range conditions can be optimally reordered as* $[R_1, R_2, \ldots, R_n]$ *when* $p_1/c_1 \geq p_2/c_2 \geq \ldots \geq p_n/c_n$ *with respect to the probability and cost estimates.*

Proof: Suppose for sequence with length of 2, 3, …, n, the above corollary is true. In order to prove for a sequence with length of $n + 1$, $[R_1, R_2, \ldots, R_{n+1}]$ is the optimal order, we need to prove that an arbitrary permutation $[R_{i_1}, R_{i_2}, \ldots, R_{i_{n+1}}]$ will have an explicit cost that is at least as great as Explicit_Cost($[R_1, R_2, \ldots, R_{n+1}]$).

(i) If the first condition of this permutation is R_1 and the permutation is $[R_1, R_{i_2}, \ldots, R_{i_{n+1}}]$, then:

Explicit_Cost($[R_1, R_{i_2}, \ldots, R_{i_{n+1}}]$)

$= p_1 \times c1 + p_{i_2} \times (c_1 + c_{i_2}) + \ldots + p_{i_{n+1}} \times (c_1 + c_{i_2} + \ldots + c_{i_{n+1}})$

$= p_{i_2} \times c_{i_2} + \ldots + p_{i_{n+1}} \times (c_{i_2} + \ldots + c_{i_{n+1}}) + c_1 \times (p_1 + p_2 + \ldots + p_n)$

Explicit_Cost($[R_1, R_2, \ldots, R_{n+1})]$

$= p_1 \times c_1 + p_2 \times (c_1 + c_2) + \ldots + p_{n+1} \times (c_1 + c_2 + \ldots + c_{n+1})$

$= p_2 \times c_2 + \ldots + p_{n+1} \times (c_2 + \ldots + c_{n+1}) + c_1 \times (p_1 + p_2 + \ldots + p_n)$

If we only consider a sequence formed by $[R_2, R_3, \ldots, R_{n+1}]$, then it is a sequence of length n, by induction, $[R_2, R_3, \ldots, R_{n+1}]$ should have the lowest explicit cost which is:

$p_2 \times c_2 + \ldots + p_{n+1} \times (c_2 + \ldots + c_{n+1})$

The Explicit_Cost($[R_{i_2}, R_{i_3}, \ldots, R_{i_{n+1}}]$ is

$p_{i_2} \times c_{i_2} + \ldots + p_{i_{n+1}} \times (c_{i_2} + \ldots + c_{i_{n+1}})$

which is at least as great as the Explicit_Cost($[R_{i_2}, R_{i_3}, \ldots, R_{i_{n+1}}]$.

This proves that sequence $[R_1, R_2, \ldots, R_{n+1}]$ has a lower cost than sequence $[R_1, R_{i_2}, \ldots, R_{i_{n+1}}]$.

(ii) If the first condition is R_i, $i \neq 1$. Then by applying induction hypotheses and the result of (i), we have:

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

□

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

$$Cost([R_1, \ldots, R_n]) =$$
$$Explicit\_Cost([R_1, \ldots, R_n]) + (1 - (p_1 + \ldots + p_n))(c_1 + \ldots + c_n) \tag{3}$$

Once only a single target remains, the range conditions associated with that target need not be tested. Consider again the example in Figure 5.5. The three targets of the range conditions are T_1, T_2, and T_D. Each of these targets could be potentially used as the default target and its associated range conditions would not have to be tested. The T_D target has three associated ranges. If any of these ranges are explicitly checked, then Corollary 4 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 T_D is used as the default target, then at least one of the three range conditions should not be explicitly checked.

Definition 17. ***mindefault*($\boldsymbol{T}_i$)** *is the minimum cost of any ordering of a range condition sequence, where* $\boldsymbol{T}_i$ *is used as the default target.*

For each potential target having m associated ranges, there are $2^m - 1$ possible combinations of these range conditions that could not be explicitly checked. The compiler used the ordering $p_1/c_1 \geq \ldots \geq p_m/c_m$ between the m ranges of a target to consider only m possible combinations of default range conditions, $\{\{R_m\}, \{R_{m-1}, R_m\}, \{R_1, \ldots, R_{m-1}, R_m\}\}$. The compiler selected the lowest cost combination of default ranges by calculating the minimum cost of the sequences 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 $\{mindefault(T_1),\ mindefault(T_2),\ \ldots,\ mindefault(T_t)\}$. Note that only the cost of n sequences have to be calculated, 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. We discovered that our approach selected the optimal sequence for every reorderable sequence in every test program for the training and test data sets.

Equation 4 represents the cost of executing a sequence of n explicitly checked range conditions, where only range condition i is a default range. Note that all explicitly specified range conditions must be checked before a target associated with a default range can be reached.

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

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.

$$\begin{aligned}
&Cost([R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_n]) \\
&\quad = Cost([R_1, \ldots, R_n]) + p_i(c_{i+1} + \ldots + c_n) \\
&\qquad\qquad\qquad\qquad - c_i(p_i + \ldots + p_n)
\end{aligned} \tag{5}$$

The ordering of a sequence of range conditions was selected using the algorithm in Figure 6.1. 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 algorithm requires a complexity of $O(n)$, where n is the number of ranges associated with the targets of 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;

/* Calculate 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];

/* 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 6.1: Selecting the Ordering of a Sequence of Range Conditions

CHAPTER 7

IMPROVING THE SELECTED SEQUENCE OF RANGE CONDITIONS

Other improvements were obtained after the ordering decision was made for a sequence of range conditions. 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 3.1. 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 ith range condition in the sequence and is associated with the range $[c_1..c_2]$. The probability that the value of the common variable is below or above the range at the point that the range condition is performed can be determined as follows. We know that the range conditions associated with the sequence $[R_1, R_2, \ldots R_{i-1}]$ 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, $[R_{i+1}, R_{i+2}, \ldots, R_n]$, to determine the probability that $v < c_1$ versus the probability that $v > c_2$. Based on these probabilities, the branch is placed first that is most likely to determine if the range condition is not satisfied.

For example, suppose we have this situation:

$[R_i, R_{i+1}, \ldots, R_n]$

where R_i is [65..97], R_{i+1} is [48..55], R_{i+2} is [33, 33], and R_{i+3} is [104..MAX].

Suppose the number of times when the value of the common variable falls into R_{i+1}, R_{i+2} and R_n are 200, 100, and 5,000 respectively. R_i is a properly bounded range and hence two branches need to check if the value of the common variable falls into R_i. We can have two different orders for determining if R_i, is satisfied, as shown in Figure 7.1.

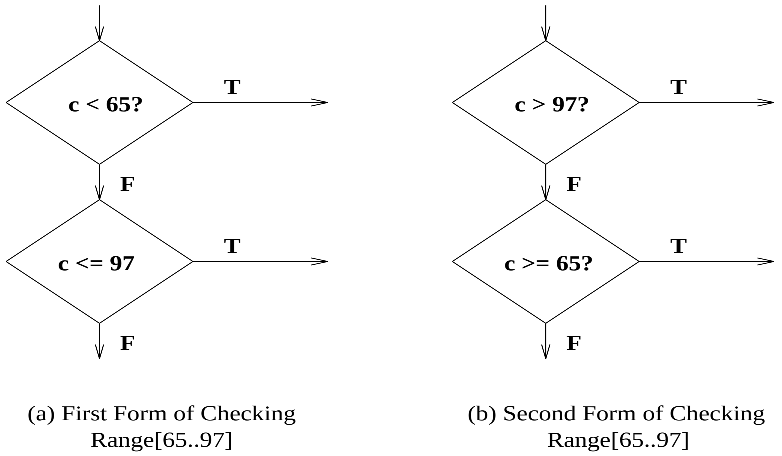


Figure 7.1: Branches for Deciding a Properly Bounded Range.

We use the counts of the satisfied ranges below and above R_i. We have 300 samples which are less than 65, and we have 5,000 samples which are greater than 97. If we choose the first form as shown in Figure 7.1 (a), then the total cost will be $300 \times 2 + 5,000 \times 4 = 20,600$ since each condition requires both a comparison and branch instruction. If we choose the second form, then the total cost will be $5,000 \times 2 + 300 \times 4 = 11,200$. Obviously the second form has a lower cost than the first form.

Suppose the total number of samples falling into ranges less than R_i is n_1, and the total number of samples falling into ranges above R_i is n_2. n_1 and n_2 can be easily obtained by sorting ranges $R_i, R_{i+1}, R_{i+2}, \ldots, R_n$. The cost for (a) and (b) will be $n_1 \times 1 + n_2 \times 2$ versus $n_2 \times 1 + n_1 \times 2$. It is easy to see that we should choose (a) when $n_1 > n_2$ and (b) when $n_2 > n_1$.

Another improvement that was performed after the range conditions have been ordered is to eliminate redundant comparisons. For instance, consider Figure 7.2(a). There are two consecutive range conditions that test if the common variable is in the ranges [c+1..max] and [c..c]. Figure 7.2(b) shows an equivalent comparison and branch for the first range condition. The comparison instruction within the second range condition is redundant and the compiler eliminates it.

```
IC=v?c+1;     # first comparison
PC=IC>=0->L1;# first branch
IC=v?c;       # second comparison
PC=IC==0->L2;# second branch
```

(a) Before

```
IC=v?c;       # first comparison
PC=IC>0->L1; # first branch

PC=IC==0->L2;#second branch
```

(b) After

Figure 7.2: Eliminating Redundant Comparisons.

CHAPTER 8

APPLYING THE REORDERING TRANSFORMATION

Once a branch ordering has been selected, the compiler will apply the reordering transformation. Figure 8.1 (a) shows a control-flow segment containing a common successor sequence. Figure 8.1 (b) shows the control flow with the reordered branch blocks. The predecessors of the first original branch block now have transitions to the first reordered branch block, which in this case is a replication of the second original branch block. Also we do not generate an extra unconditional branch instruction by replicating the default target T_D, assuming T_D is a terminal block. This will guarantee the semantic equivalence whenever the head of the sequence is entered. Figure 8.1 (c) shows the code after applying dead code elimination. The original branch block B_1 was deleted, while branch block B_2 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.

Figure 8.2 (a) through (e) shows the reordering transformation for a common variable sequence. In Figure 8.2 (b), the range conditions are replicated in order to get rid of the side transition into the sequence from block P3, again T_D is replicated to avoid an unconditional jump. We can skip this step if there is no side transition into the sequence. Note that although we draw the side effects S_1 and S_2 as separate blocks, they are really contained in the basic blocks

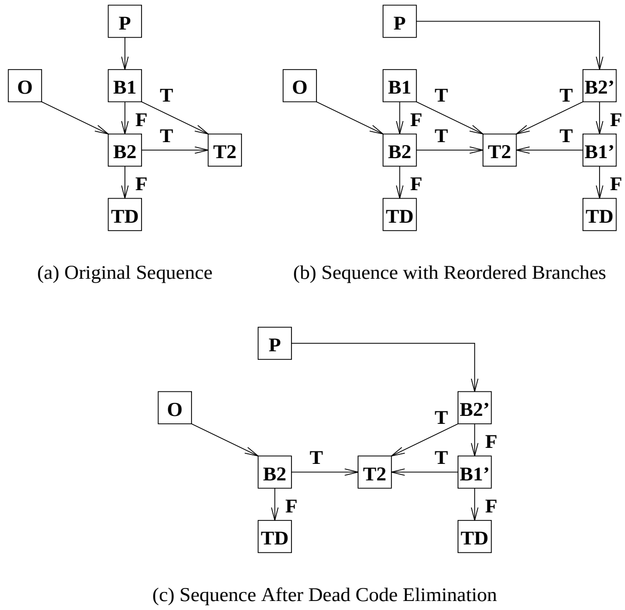
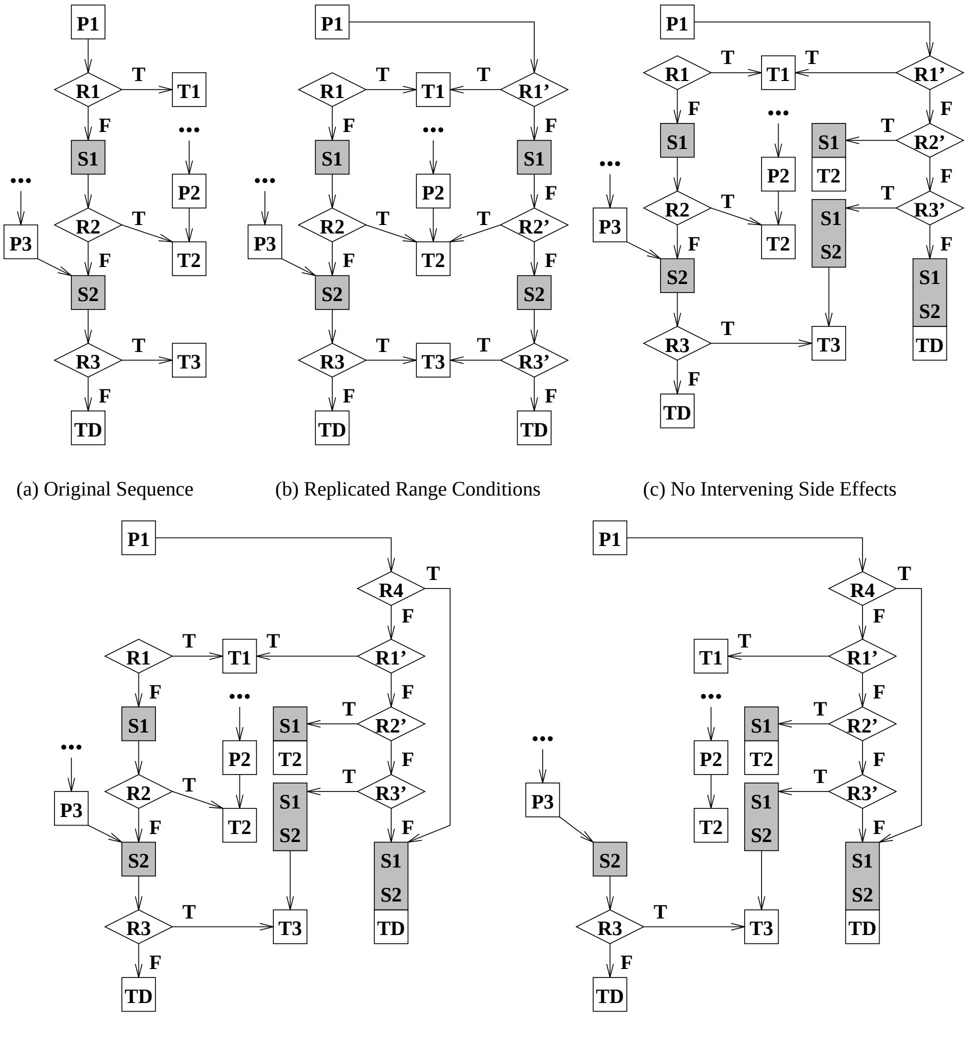


Figure 8.1: Reordering Transformation for Common Successor Sequence

containing R_1 and R_2, respectively.

The next step is to transform the sequence into a reorderable sequence by replicating the side effects, as shown in Figure 8.2 (c). Note T_2 is also replicated to avoid the introduction of an unconditional jump. In Figure 8.2 (d), the range conditions are reordered, with an additional range condition R_4 being moved out of the original default range conditions to the head of the sequence. Finally, we invoke dead code elimination and the result is Figure 8.2 (e).

When a sequence of branches with a common successor overlapped with a sequence of branches comparing a common variable, we need to make a choice as to which type should be used. Our experiments indicated that the common variable type is more effective (see the results in Chapter 7.) Hence the com-



(a) Original Sequence

(b) Replicated Range Conditions

(c) No Intervening Side Effects

(d) Reordered Range Conditions

(e) Dead Code Elimination

Figure 8.2: Example of Applying the Reordering Transformation

piler gave the common variable sequence precedence over the common successor sequence.

Figure 8.3(a) shows an example in which $[N_1, N_2]$ is a common variable sequence and $[N_2, N_3]$ is a common successor sequence. Assume that $[N_1, N_2]$ and $[N_2, N_3]$ should be reordered based on profile information and there are no

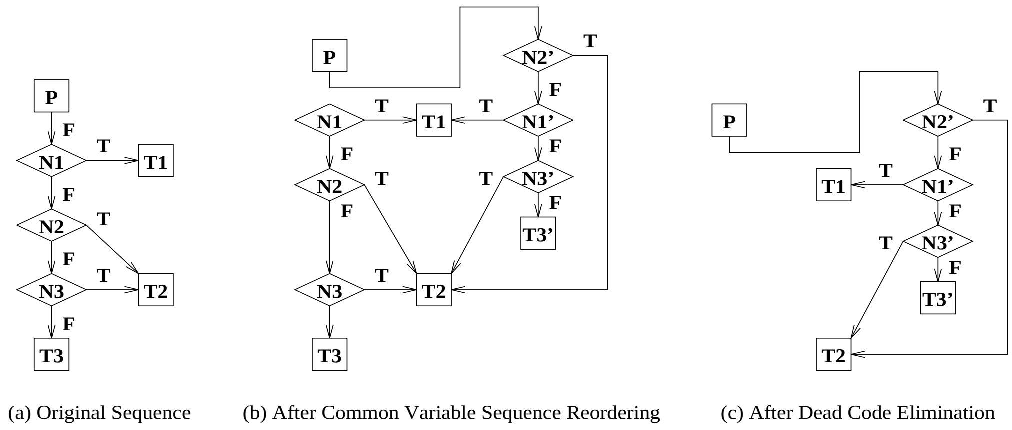


Figure 8.3: Resolving Sequence Overlapping.

side transitions for both sequences. The compiler is going to reorder sequence $[N_1, N_2]$ before reordering $[N_2, N_3]$. Figure 8.3(b) shows that four new basic blocks are created to apply the reordering of $[N_1, N_2]$, among which N_1' and N_2' correspond to range conditions N_1 and N_2 and N_3' and T_3' are replicated code of blocks N_3 and T_3. Here we assume block T_3 is a terminal block. Since we need to replicate code until we reach a terminal block, we cannot stop replication at N_3' and hence we continue to replicate T_3'. After we finish reordering all the common variable sequences, we are going to reorder common successor sequences. When we come to the common successor sequence $[N_2, N_3]$, we are not going to apply the reordering since now block N_2 does not have any predecessors. As we can see in Figure 8.3 (c), the basic blocks N_1, N_2, N_3 and T_3 are deleted.

The compiler also checks for a special case in which the reordering of a common successor sequence should not be applied at all. Figure 8.4(a) gives an example in which $[N_1, N_2]$ is a common successor sequence and $[N_2, N_3]$ is a common variable sequence. Figure 8.4 (b) shows the situation after reordering $[N_2, N_3]$. Note that in Figure 8.4 (b) the basic block N_2 has lost all of its predecessors, including N_1. In this case the original relationship between N_1 and N_2 is lost, i.e, they do not form a common successor sequence any more. So the compiler will not apply the reordering of $[N_1, N_2]$.

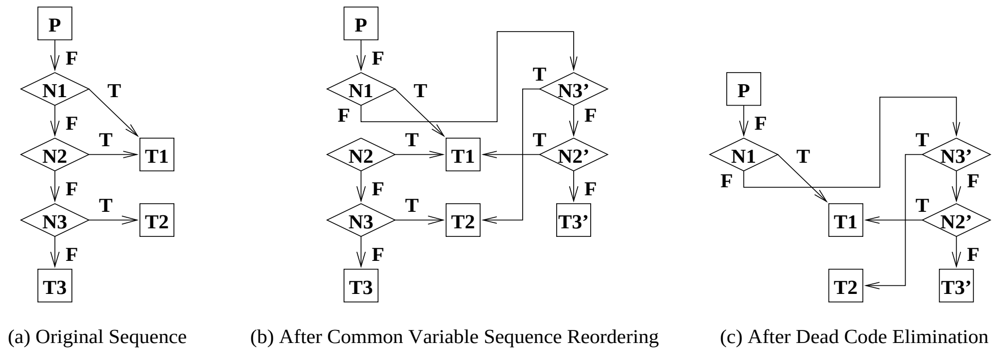


Figure 8.4: Special Case of Sequence Overlapping.

CHAPTER 9
RESULTS

Measurements were collected on the code generated for the SPARC architecture by the *vpo* compiler using the *ease* environment [11]. We chose non-numerical applications since they tend to have complex control flow and a higher density of conditional branches. Table 9.1 shows the dynamic frequency measurements from several common Unix utilities. [1]

The Original 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 branches with a common successor, reordering sequences of range conditions comparing a common variable, and reordering both types of sequences. The approach for reordering branches with common successors had very little impact. The compiler could only find very few sequences that were deemed worthwhile to reorder. The approach for reordering range conditions comparing a common variable had significant benefits both in reducing the total number of instructions and conditional branches. 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 reordering transformations had a slight negative impact on a few programs, such as *hyphen*, which occurred for a couple of reasons.

[1] The heuristic for Table 9.1 is Heuristics Set I as defined later.

Table 9.1: Dynamic Frequency Measurements for Heuristic Set I.

Program	Original	Common Successor		Common Variable		Both	
		Insts	Branches	Insts	Branches	Insts	Branches
awk	13,611,150	-3.232%	-3.828%	-2.02%	-4.19%	-3.88%	-7.36%
cb	17,100,927	-0.253%	-0.300%	-7.65%	-15.46%	-7.90%	-15.76%
cpp	18,883,104	+0.160%	+0.322%	-0.13%	-0.19%	-0.06%	+0.13%
ctags	71,824,053	-0.004%	-0.004%	-9.10%	-14.72%	-9.00%	-14.62%
deroff	15,458,812	-0.032%	-0.003%	-1.52%	-2.63%	-1.55%	-2.62%
grep	9,256,749	-0.028%	-0.053%	-3.60%	-8.31%	-3.62%	-8.37%
hyphen	18,059,010	0.000%	0.000%	+3.42%	+3.40%	+3.41%	+3.34%
join	3,552,801	0.000%	0.000%	-1.68%	-2.12%	-1.68%	-2.12%
lex	10,005,018	-0.546%	-0.919%	-4.56%	-10.39%	-4,81%	-10.52%
nroff	25,307,809	-1.118%	-1.869%	-2.48%	-6.35%	-3.21%	-7.32%
pr	73,051,342	0.000%	0.000%	-16.25%	-29.96%	-16.25%	-29.96%
ptx	20,059,901	-0.860%	-1.4174%	-9.18%	-13.28%	-10.04%	-14.69%
sdiff	14,558,535	-0.043%	-0.037%	-16.09%	-37.03%	-16.13%	-37.07%
sed	14,229,310	-0.166%	-0.310%	-1.16%	-2.03%	-1.33%	-2.34%
sort	23,146,400	0.000%	0.000%	-47.20%	-57.38%	-47.20%	-57.38%
wc	25,818,199	0.000%	0.000%	-15.05%	-26.26%	-15.05%	-26.25%
yacc	25,127,817	-0.022%	-0.034%	-0.25%	-0.44%	-0.27%	-0.47%
average	23,473,585	-0.361%	-0.497%	-7.91%	-13.37%	-6.745%	-13.73%

First, different test input data was used as compared to the training input data. 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 often 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 significant benefits when a program executes most of its instructions in a reorderable sequence, such as in *sort*.

Since there is little benefit from reordering common successor sequences, the following tables only consider reordering common variable sequences. Three different sets of heuristics were used when translating `switch` statements. Assume there are n cases in a switch statement and there are m possible values between the first and last case. When compiling for a SPARC IPC and a SPARC 20, the front end used Heuristic Set I that would only generate an indirect jump from a table if $n \geq 4$ and $m \leq 3n$. If an indirect jump was not generated, then a binary search was used when $n > 8$. These are the same heuristics used in the *pcc* front end [12]. The authors used the dual loop method [13] 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 [14]. 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 when translating a `switch` statement, which resulted in the maximum benefit from reordering. 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. Tables 9.2 and 9.3 show the dynamic frequency measurements for Heuristic Set II and Heuristic Set III, respectively.

Tables 9.4, 9.5, and 9.6 show static measurements for the same set of programs for Heuristic Set I, II, and III, respectively. 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

Table 9.2: Dynamic Frequency Measurements for Heuristic Set II.

Program	Original		Reordered	
	Insts	Branches	Insts	Branches
awk	13,552,831	2,195,748	-2.97%	-6.15%
cb	17,100,927	2,882,466	-7.65%	-15.46%
cpp	18,880,116	2,642,973	-0.13%	-0.19%
ctags	71,824,093	18,948,699	-9.02%	-14.64%
deroff	15,449,146	2,714,435	-1.38%	-2.36%
grep	9,938,414	2,115,757	-10.53%	-22.04%
hyphen	18,059,010	2,831,171	+3.42%	+3.40%
join	3,552,801	983,936	-1.68%	-2.12%
lex	10,003,391	1,764,417	-4.57%	-10.40%
nroff	25,313,527	3,690,741	-2.50%	-6.39%
pr	73,051,352	12,078,585	-16.25%	-29.96%
ptx	20,059,901	3,310,268	-9.18%	-13.28%
sdiff	14,558,530	2,765,574	-16.09%	-37.03%
sed	14,243,263	2,549,635	-1.28%	-2.32%
sort	23,146,400	6,277,167	-47.20%	-57.38%
wc	25,818,199	5,227,974	-15.05%	-26.26%
yacc	25,127,817	4,851,335	-0.25%	-0.44%
average	23,510,571	4,578,287	-8.37%	-14.30%

Table 9.3: Dynamic Frequency Measurements for Heuristic Set III.

Program	Original		Reordered	
	Insts	Branches	Insts	Branches
awk	13,651,335	2,230,559	-3.63%	-7.44%
cb	19,662,207	3,538,146	-21.79%	-37.41%
cpp	30,477,974	6,730,186	-28.37%	-41.85%
ctags	72,222,399	19,042,284	-9.13%	-14.73%
deroff	15,491,185	2,722,474	-1.40%	-2.39%
grep	11,810,072	2,526,865	-32.04%	-51.42%
hyphen	18,059,010	2,831,171	+3.42%	+3.40%
join	3,552,801	983,936	-1.68%	-2.12%
lex	10,028,151	1,771,795	-4.77%	-10.73%
nroff	25,339,678	3,697,534	-2.53%	-6.45%
pr	73,051,352	12,078,585	-16.25%	-29.96%
ptx	20,059,901	3,310,268	-9.18%	-13.28%
sdiff	14,558,530	2,765,574	-16.09%	-37.03%
sed	15,368,724	3,014,722	-10.07%	-17.01%
sort	23,146,434	6,277,177	-47.20%	-57.38%
wc	25,818,199	5,227,974	-15.05%	-26.26%
yacc	25,168,370	4,864,310	-0.47%	-0.76%
average	24,556,842	4,918,444	-12.72%	-20.75%

Table 9.4: Static Measurements for Heuristic Set I.

Program	Insts	Total Seqs	Reordered		
			Seqs	Avg Seq Len	
				Orig	After
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%	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
average	+4.98%	26	48.20%	2.97	3.62

Table 9.5: Static Measurements for Heuristic Set II.

Program	Insts	Total Seqs	Reordered		
			Seqs	Avg Seq Len	
				Orig	After
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%	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
average	+5.09%	29	48.67%	3.05	3.61

Table 9.6: Static Measurements for Heuristic Set III.

Program	Insts	Total Seqs	Reordered		
			Seqs	Avg Seq Len	
				Orig	After
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%	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%	3.00	4.33
sdiff	+3.49%	10	40%	3.00	3.50
sed	+5.32%	25	48%	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
average	+4.96%	19	49.79%	6.08	6.96

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. Note that most of the sequences contained only two or three branches. The length of each reordered sequence typically increased since often one or more default ranges became explicit after reordering.

Branch prediction measurements were obtained for the SPARC Ultra I, which supports branch prediction with a (0,2) predictor with 2048 entries. Table 9.7 shows branch prediction measurements for the same set of programs. Column *Branches* is the total number of branch instructions executed. Column *Miss#* is the number of mispredictions and column *Miss%* is the percentage of mispredictions. Column *Miss# Change%* is the change in percentage of number of mispredictions between *Original Miss#* and *Reordered Miss#*. It was 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 [14]. 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. 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. Table 9.8 shows that comparable results were obtained when simulations were performed using other

Table 9.7: Branch Prediction Measurements.

Program	Original		Reordered		Miss #	Inst
	Miss #	Miss %	Miss #	Miss %	Change %	Ratio %
awk	243,027	11.15%	241,916	11.83%	-0.46%	N/A
cb	440,712	15.29%	466,158	19.13%	+5.77%	51.41
cpp	389,566	14.22%	382,761	14.00%	-1.75%	N/A
ctags	569,753	3.01%	1,854,523	11.26%	+225.50%	5.04
deroff	62,819	2.32%	61,016	2.31%	-2.87%	N/A
grep	115,007	5.44%	110,064	6.67%	-4.30%	N/A
hyphen	266,177	9.40%	490,095	16.74%	+84.12%	-2.76
join	50,440	5.13%	47,605	4.94%	-5.62%	N/A
lex	66,534	3.77%	67,820	4.29%	+1.93%	355.47
nroff	141,167	3.82%	139,849	4.05%	-0.93%	N/A
pr	750,570	6.21%	753,046	8.90%	+0.33%	4,793.65
ptx	215,218	6.50%	296,103	10.31%	+37.58%	22.78
sdiff	156,440	5.66%	148,078	8.50%	-5.35%	N/A
sed	83,579	3.23%	82,037	3.25%	-1.84%	N/A
sort	171,619	2.73%	153,745	5.75%	-10.41%	N/A
wc	481,767	9.22%	482,627	12.52%	+0.18%	4,519.65
yacc	373,825	7.71%	375,899	7.78%	+0.55%	30.28
average	269,307	6.75%	361,961	8.96%	+18.97%	1,221.94

branch predictors.

The execution time measurements shown in Table 9.9 were obtained from the average reported *user* times of ten executions of each program using the C run-time library function *times*(). The execution time decrease was not as significant as the reduction in instructions executed on these machines. One should note that in Table 9.9 the frequency 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.

Table 9.8: Branch Prediction Measurements for Different Configurations

Entries	(0,1) Predictor		(0,2) Predictor		(2,2) Predictor	
	Miss # Change %	Inst Ratio	Miss # Change %	Inst Ratio	Miss # Change %	Inst Ratio
32	+16.65%	681.20	+17.37%	1,313.47	+17.05%	805.78
64	+21.96%	720.73	+21.15%	1,082.02	+20.77%	640.08
128	+21.91%	8,583.19	+20.60%	1,091.28	+19.40%	661.92
256	+21.91%	972.87	+20.21%	953.70	+19.03%	569.88
512	+19.67%	5,852.38	+18.09%	1,200.25	+17.34%	681.98
1024	+20.45%	13,331.71	+18.88%	1,217.61	+18.44%	664.03
2048	+20.59%	13,311.73	+18.97%	1,221.94	+37.65%	653.02

Table 9.9: Execution Times.

Machine	Average Execution Time
SPARC IPC	-4.942%
SPARC 20	-5.565%
Ultra SPARC	-2.878%

CHAPTER 10
FUTURE WORK

There are several areas in which reordering branches could be extended. Additional sequences of branches with common successors could be optimally reordered. Interprocedural analysis could be used to determine if invoked functions do not cause a side effect. Avoiding the execution of a function call could have significant performance benefits. Sequences having more than two branches could be optimally reordered by obtaining all combinations of branch results using an array of profile counters. This approach may be reasonable for a small sequence length (e.g. $n <= 7$), which may handle most branch sequences with a common successor. In addition, a sequence of branches with a common successor can be viewed as a single block with a branch since it has two successors (the common successor and the other successor of the last branch in the sequence). Thus, an entire sequence may be reordered with another branch or sequence if they have a common successor. This situation may occur from the translation of complex logical expressions applying different logical operators (e.g. both `||` and `&&` operators in C.)

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 [3]. 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. We plan to investigate the use of a binary search approach. Instead of finding a near-optimal sequence of range conditions, we can select an optimal or near-optimal binary search tree. Our initial investigations have shown that an algorithm using dynamic programming to select an optimal binary search would require both exponential time and space complexity. We can reduce this complexity by first using heuristics to partition ranges and then applying dynamic programming to find the optimal binary search tree for smaller sets of ranges. We plan to study how much more benefit will be obtained using the binary search approach instead of the linear search approach at the expense of increasing the complexity.

CHAPTER 11
CONCLUSIONS

This thesis described an approach of using profile information to reduce the number of conditional branches executed by reordering sequences of branches. Algorithms for detecting sequences of reorderable branches having a common successor or comparing a common variable were 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 based on profiling and cost estimates can often be obtained. The results showed reductions in the number of branches and instructions executed and execution time.

REFERENCES

[1] J. W. Davidson and S. Jinturkar, "Aggressive loop unrolling in a retargetable, optimizing compiler", *Proceedings of Compiler Construction Conference*, pp. 59–73, April 1996.

[2] F. Allen and J. Cocke, *A Catalogue of Optimizing Transformations*, ed. R. Rustin, Prentice-Hall, Englewood Cliffs, NJ, 1971.

[3] D. A. Spuler, *Compiler Code Generation for Multiway Branch Statements as a Static Search Problem*, Addison-Wesley, U.S.A, June 1987.

[4] 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.

[5] R. Gupta R. Bodik and M. Soffa, "Interprocedural conditional branch elimination", *Proceedings of the SIGPLAN'97 Conference on Programming Language Design and Implementation*, pp. 146–158, June 1997.

[6] 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.

[7] B. Calder and D.Grunwald, "Recuding 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.

[8] D. R. Karger C. Young, D. S. Johnson 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.

[9] 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.

[10] 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 Suport for Programming Languages and Operating Systems*, pp. 85–95, October 1992.

[11] 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.

[12] S. C. Johnson, *A Tour Through the Portable C Compiler*, ed. R. Rustin, Prentice-Hall, Englewood Cliffs, NJ, 1979.

[13] R. A. Volz T. N. Mudge T. Schultze R. M. Clapp, L. Duchesneau, "Toward real-time performance benchmarks for ada", *Communications of the ACM bf 19(8)*, pp. 760–778, August 1986.

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