THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

EFFECTIVELY EXPLOITING INDIRECT JUMPS

By

Gang-Ryung Uh

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Fall Semester, 1997

The members of the Committee approve the dissertation of Gang-Ryung Uh defended on October 23, 1997.

David B. Whalley
Professor Directing Dissertation

Steven F. Bellenot
Outside Committee Member

R. C. Lacher
Committee Member

Theodore P. Baker
Committee Member

Susan I. Hruska
Committee Member

Approved:

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

To my wife and parents

Acknowledgements

I would like to thank everyone who helps me get through this long journey.

Contents

List of Tables

List of Figures

Abstract

This dissertation describes a general code-improving transformation that can coalesce conditional branches into an indirect jump from a table. Applying this transformation allows an optimizer to exploit indirect jumps for many other coalescing opportunities besides the translation of multiway branch statements. First, dataflow analysis is performed to detect a set of coalescent conditional branches, which are often separated by blocks of intervening instructions. Second, several techniques are applied to reduce the cost of performing an indirect jump operation, often requiring the execution of only two instructions on a SPARC. Finally, the control flow is restructured using code duplication to replace the set of branches with an indirect jump. Thus, the transformation essentially provides early resolution of conditional branches that may originally have been some distance from the point where the indirect jump is inserted. The transformation can be frequently applied with often significant reductions in the number of instructions executed, total cache work, and execution time. In fact, over twice the benefit was achieved from exploiting indirect jumps as a general code-improving transformation instead of using the traditional approach of producing indirect jumps as an intermediate code generation decision. In addition, the author show that with comparable branch target buffer support, indirect jumps improve branch prediction since they cause fewer mispredictions than the set of branches they replaced.

Chapter 1
INTRODUCTION

Most high-level languages provide *multiway branch* statements to allow programmers to write more readable code. The characteristic feature of a multiway statement is the ability to select an action based on the value of a control expression. Without performing any optimization, a compiler would translate each case label of the multiway statement into a conditional branch. Because of the widespread usage of multiway statements, instruction sets commonly support an *indirect jump* from a table in order to reduce the cost of such sequences of conditional branches. As a result, compiler front ends typically generate an indirect jump from a table as one translation alternative[1] for multiway statements [25, 28].

This traditional approach for using indirect jumps poses two problems. First, it is difficult to determine when the indirect jump can be effectively used in a machine-independent fashion since an accurate cost-benefit estimate can only be made after generating machine code. Second, many code-improving opportunities suitable for using an indirect jump may be missed when only considering this operation for the translation of a multiway statement.

This dissertation describes a general code-improving transformation that exploits indirect jumps after code generation. As the instruction issue rate and pipeline depth of processors increase, efficient handling of branches becomes

[1]The other popular alternatives include *linear search*, *binary search*, and *hashing*.

more vital. This improving transformation reduces the number of branches and mispredictions by coalescing several conditional branches into an indirect jump. First, dataflow analysis is performed to detect a set of possibly non-contiguous conditional branches that can be potentially coalesced into a single indirect jump. Second, control-flow analysis is used to determine how the control flow should be restructured to perform the coalescing. Third, analysis is accomplished to determine how to most efficiently generate the indirect jump operation. The cost of the original branches is also estimated and the indirect jump transformation is applied when deemed worthwhile. Finally, the original control flow is modified by duplicating basic blocks when necessary.

1.1 Motivation

Exploiting indirect jumps after code generation can be quite beneficial since additional branches from other control statements besides multiway statements can be coalesced into a single indirect jump. The examples in this section are given in C to more concisely depict branches that can be coalesced into indirect jumps. The control flow of the restructured C code segments would be comparable to a restructured flow graph of basic blocks with an indirect jump from a table.

1.1.1 Indirect Jumps with Branches

Consider the `Original` code segment shown in Figure 1.1. A typical C compiler would translate the `switch` statement into an indirect jump from a table and would generate a conditional branch for the `for` statement. Yet, the conditional branch comparing `*sp` with zero would immediately precede the indirect jump.

An optimizer could recognize this sequence of branches and be able to coalesce the extra conditional branch that compares the variable with zero into the indirect jump. Note that one can view this branch as another `case` for the `switch` statement as shown in the Restructured code segment.

Original

```
for (sp = line; *sp; sp++) {
   switch (*sp) {
   case ’p’:
            ...
   case ’k’:
            ...
   ...
  }
}
```

Restructured

```
for (sp = line;  ; sp++) {
   switch (*sp) {
   case ’\0’:
             goto out;
   case ’p’:
             ...
   case ’k’:
            ...
}
out:
```

Figure 1.1: Example from *ctags* (C tag generator)

1.1.2 Sequence of Contiguous Branches

Other common instances may occur due to programming style. The Original code segment, depicted in Figure 1.2, shows a series of `if` statements comparing the same variable to different constants. A typical C compiler would translate these `if` statements as a sequence of conditional branches. However, the code could have been equivalently written as a single `switch` statement as shown in the Restructured code segment. An optimizer could detect the original sequence of conditional branches and could coalesce such contiguous branches into a single indirect jump. Use of multiple macros may also result in several consecutive comparisons being performed. Thus, branch coalescing is appealing since coalescing is less affected by program style (whether or not multiway branches are used).

Original

```
if ((c = *sp++) == 0)
    goto cerror;
if (c == '<') { ... }
if (c == '>') { ... }
if (c == '(') { ... }
if (c == ')') { ... }
if (c >= '1' && c <= '9') { ... }
...
```

Restructured

```
c = *sp++;
switch (c) {
case 0:      goto cerror;
case '<':      ...
case '>':      ...
case '(':      ...
case ')':      ...
case '1': case '2': case '3':
case '4': case '5': case '6':
case '7': case '8': case '9':
               ...
default:       ...
}
```

Figure 1.2: Example from *grep* (pattern search utility)

1.1.3 Set of Contiguous and Noncontiguous Branches

Often there are paths in which intervening instructions exist between branches that compare the same variable to constants and these intervening instructions do not update this variable. Consider the following `Original` code segment shown in Figure 1.3. A typical C compiler would translate each `if` statement into conditional branch(es). At first, it may appear that only the sequence of conditional branches shown in the shaded boxes can be coalesced into an indirect jump. However, the statement ***charct++;*** does not affect the branch variable `c`. An optimizer could determine the existence of path(s) between branches comparing the same variable to constants where the variable is unaffected. The optimizer could modify the original control flow by duplicating code to allow the branch for the `EOF` check to also be coalescent. As shown in `the Duplicated` code segment, all of the branches in the shaded boxes can be effectively considered as being contiguous and coalescent for a single indirect jump. The `Restructured` code segment shows equivalent code written with a

Original

```
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;
}
```

Duplicated

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

Restructured

```
for (; ;) {
  c = getc(fp);
  switch (c) {
  case EOF:     goto out;
  case 041:
  ...
  case 0176:    charct++;
                if (!token){
                    wordct++;
                    token++;
                }
                continue;
  case '\n':    charct++;
                linect++;
                goto end;
  default:      charct++;
                continue;
  case ' ':
  case '\t':    charct++;
  }
  end:  token = 0;
}
out:
```

Figure 1.3: Example from *wc* (word count utility)

`switch` statement.

1.2 Organization of Dissertation

The dissertation is organized in the following manner. Chapter 2 gives a description of the compiler that has been used and modified to exploit indirect jumps after code generation. Chapter 3 briefly describes related compiler optimizations to reduce the cost of conditional branches. In order to detect and replace more branches into a single indirect jump than would be done in the traditional way, several detection and restructuring algorithms are introduced in Chapters 4 and 5 that can allow a compiler to better exploit indirect jumps as a code-improving transformation. Chapter 4 explains the algorithms to detect a contiguous sequence of coalescent conditional branches and to transform the control flow for coalescing the detected branches into a single indirect jump. Chapter 5 depicts the more general algorithms to detect a set of potentially noncontiguous coalescent conditional branches, which are often separated by blocks of intervening instructions, and to restructure the control flow by code duplication when necessary. These algorithms allow the compiler to detect and coalesce more branches per indirect jump than the algorithms described in Chapter 4. Thus, the performance benefits from coalescing noncontiguous branches can be contrasted with the simpler analysis required for only coalescing contiguous branches.

Chapter 6 presents several techniques that reduce the cost of performing an indirect jump operation, often requiring the execution of only two instructions on a SPARC. The task of filling delay slots for indirect jumps is also dealt with in this chapter. Chapter 7 shows execution time results from performing dual loop tests [10, 2] on SPARCstations to estimate the impact on pipeline stalls when the branch coalescing transformation was applied as another code improving transformation. Furthermore, the benefits of target buffer support for indirect jumps are discussed in this chapter. Various performance measurements are given in Chapter 8 that justify the validity of applying the code-improving transformation that is described in this dissertation. Chapter 9 suggests topics for future research. Finally, Chapter 10 concludes the dissertation.

Chapter 2
MODIFICATIONS TO THE COMPILER

2.1 Overview of the Compiler

Figure 2.1 shows the overall structure of the *vpo* (Very Portable Optimizer [4]) compiler system. The front-end of the compiler, *cfe* [12], produces intermediate code from a given C preprocessed file. The *code expander* translates the intermediate code into unoptimized lists of machine-dependent effects, called *RTL*s (Register Transfer Lists). RTLs have the form of conventional expressions and assignments over the hardware's storage cells. For example, the RTLs

```
IC=r[8]?10;
PC=IC:0,L001;
```

represent two machine instructions.[1] The first RTL compares a register to constant `10` and the second RTL transfers the control to the address `L001` when the two values are equal. While any particular RTL is machine specific, the general form of the RTL is machine-independent. This allows general machine-independent algorithms to be written that implement code improving transformations on machine-dependent code.

All phases of the back-end of the compiler, *vpo* (Very Portable Optimizer), manipulate RTLs. The RTLs are stored in a data structure that also contains

[1]These instructions are generated by *cfe* when translating high level control statements, such as `if` or `if-then-else` statements.

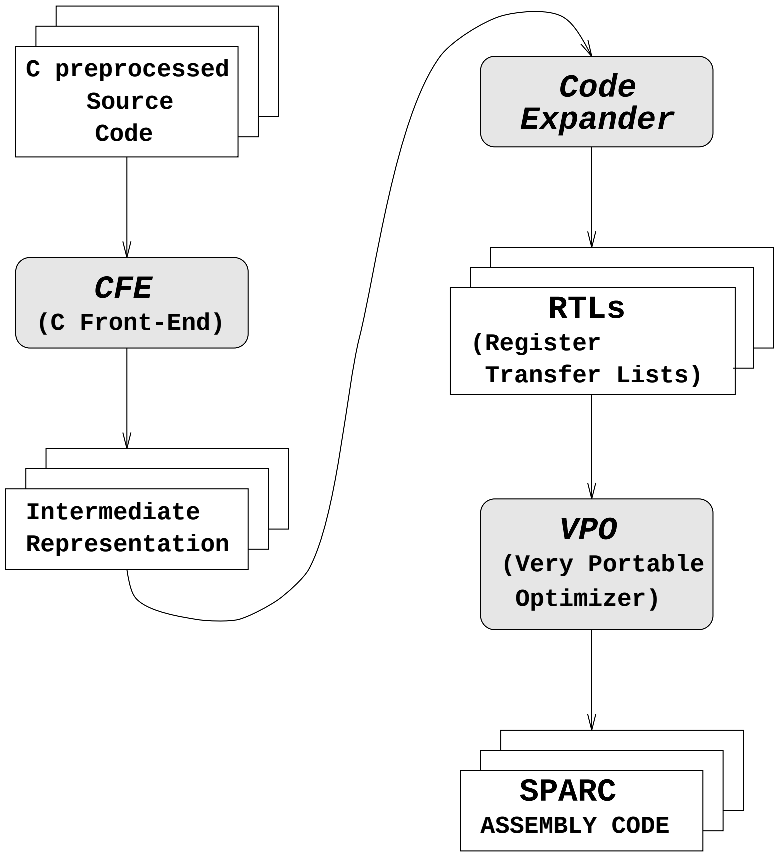


Figure 2.1: *VPCC* (Very Portable C Compiler)

information about the order and control flow of the RTLs within a function. By manipulating RTLs as the sole intermediate representation, the following benefits can be achieved.

1. Most optimizations can be invoked in any order and can be allowed to iterate until no further improvement can be found. Therefore, many phase ordering problems are eliminated.[2]

2. The effect of a modification to the set of RTLs comprising a function is relatively simple to grasp.[3]

2.2 Modifications of the Compiler

In order to exploit the indirect jump operation the following modifications were made to the compiler. The front-end of the compiler, *cfe*, was modified to always produce a linear sequence of conditional branches when translating a C `switch` statement. An additional code-improving transformation phase to coalesce branches into an indirect jump from a table was added to the back-end of the compiler, *vpo*.

Coalescing of branches was treated as a transformation for a loop. Loop transformations, such as *loop-invariant code motion*, typically require extra reg-

[2] In contrast, a more conventional compiler system will perform optimizations on various different representations. For instance, machine-independent transformations are often performed on intermediate code and machine-dependent transformations, such as peephole optimizations, are often performed on assembly code.

[3] In contrast, most conventional compiler systems generate code after optimizations. Thus, the optimizations are actually performed on intermediate code. Since there is typically not a one-to-one mapping between an intermediate code operation and a machine instruction, the effect of a modification on the final code that will be generated may not be obvious in these systems.

isters. Most compiler optimizers perform these transformations starting with the innermost loops first to secure registers for the most frequently executed code segments. When an indirect jump occurs inside a loop, performing code motion on the loop-invariant instructions for calculating the jump table address requires a register. Thus, as depicted in Figure 2.2, the author coalesced branches from the innermost loop outward after all other transformations for a given loop have been initially attempted. Afterwards, branch coalescing was also attempted on the outermost level of an entire function.

```
Branch Chaining
Useless Jump Elimination
Dead Code Elimination
Eliminate Unconditional Jumps by Reordering Code
Instruction Selection
Evaluation Order Determination
Global Instruction Selection
Register Assignment
Jump Minimization
Instruction Selection
DO {
     Register Allocation
     Instruction Selection
     Common Subexpression Elimination
     Dead Variable Elimination
     Loop Optimizations:
        Code Motion
        Recurrences
        Loop Strength Reduction
        Induction Variable Elimination
        If (First Pass)
           BRANCH COALESCING
     Useless Jump Elimination
     Cheaper Instruction Replacement
     Instruction Selection
} WHILE (change)
BRANCH COALESCING
Setup Entry and Exit
Instruction Scheduling
Fill Slots
Useless Jumps
```

Figure 2.2: Modified *VPO*

Chapter 3
RELATED WORK

Several authors have suggested heuristics for deciding between different methods of translating multiway branch statements [25, 16, 5, 28]. These methods include a linear search (branch for each case value), binary search, hashing, and indirect jumps from tables. The approach used in this dissertation initially generates conditional branches to perform a linear search and relies on the code-improving transformation to coalesce these and other branches into indirect jumps. The techniques used in this dissertation to reduce the cost of performing an indirect jump from a table often make binary searches, hashing, and other alternative methods less beneficial.

There has been some research on other techniques for avoiding conditional branches. A superoptimizer will generate an exhaustive set of bounded sequences of instructions with the goal of finding a sequence that will produce the same effect as a more expensive sequence of instructions. The more expensive sequence can then be recognized in a traditional optimizer and replaced with the less expensive sequence. This technique has been used to eliminate conditional branches over short instruction sequences in many instances on the IBM RS/6000 [14]. Loop unrolling has been used to avoid executions of the conditional branch associated with a loop termination condition [11]. 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 [1]. Conditional branches have also been avoided by code duplication [20]. 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].

The approach in this dissertation is similar to the above techniques in that it improves performance despite the penalty of increasing code size. However, there are often situations where several branches can be coalesced into a single indirect jump to avoid the execution of branches that these other techniques could not. Our approach essentially provides early resolution of branches that may originally have been some distance away in the control flow from the point where the indirect jump is inserted.

Chapter 4

COALESCING A CONTIGUOUS SEQUENCE OF BRANCHES

Chapters 4 and 5 explain several algorithms to detect and replace more branches into a single indirect jumps than those from only considering indirect jumps when translating multiway statements. The approach for coalescing a contiguous sequence of conditional branches into an indirect jump from a table is explained in Chapter 4. The more general approach for coalescing a set of potentially noncontiguous conditional branches, which are often separated by blocks of intervening instructions, is described in the next chapter.

The task for coalescing a contiguous sequence of conditional branches was accomplished in the following manner. First, contiguous conditional branches, which can be potentially coalesced into an indirect jump, are identified. Second, the execution cost of the sequence of branches is estimated. Finally, when the indirect jump transformation is deemed beneficial, the original control flow is transformed to replace the detected branches by the instructions to perform an indirect jump.

4.1 Detecting Sequences of Coalescent Branches

A general algorithm for detecting a sequence of branches that can be coalesced together may provide additional opportunities that would not be available by generating indirect jumps only when translating multiway selection statements.

The analysis for the approach described in this dissertation to detect sequences of branches that can be coalesced into an indirect jump required the following conditions.

1. The branches must be contiguous in the control flow. In other words, the instructions implementing the comparisons and branches must be connected by control-flow transitions with no intervening instructions.

2. Each branch must compare the same variable (or register) with a constant.

3. At most one branch can have no incoming transitions from another branch in this set. Thus, at most one branch can be the head of the sequence.

The algorithm for detecting a sequence of branches that can be coalesced is given in Figure 4.1. The algorithm not only will detect a coalescent sequence of branches, but will also attempt to maximize the number of branches to be coalesced.

Figure 4.2 contains an example flow graph that is used to illustrate the algorithm. Assume that the blocks 2, 4, 20, 21, 22, and 23 contain branches that compare the same branch variable with a constant. Also assume that blocks 2, 4, 22, and 23 contain no other instructions besides a comparison and branch. Consider the case in which the detection of a sequence of branches is attempted at block 2. The algorithm recursively searches backwards and mark blocks 2, 23, 20, and 21 as visited. Assume block 20 is chosen as the head of the sequence since it is the first block detected that has no visited immediate predecessor. At this point the algorithm recursively searches forward and collects blocks 20, 22, 23, 2, and 4 as the sequence of branches to be coalesced.

```
PROCEDURE Detect_Sequence()
{
   FOR each block B DO {

      IF (B contains a branch that
          compares variable V with a constant) {

         Search_Back(B, V);
         H = Choose_Head();
         Collect_Blocks(B, V);
      }
   }
}


PROCEDURE Search_Back(B, V)
{
   mark B as visited;
   IF (B has no instructions
       preceding its compare and branch) {

      FOR each immediate predecessor P of B DO {
         IF (P has not been visited &&
             P has a branch &&
             P compares V with a constant) {

            Search_Back(P, V);
         }
      }
   }
}


PROCEDURE Choose_Head()
{
   FOR each block B marked as visited DO {

      IF (B has no immediate predecessor
          marked as visited) {

         RETURN B;
      }
   }

   RETURN visited block that dominates
          the most visited blocks;
}


PROCEDURE Collect_Blocks(B, V)
{
   mark B as collected;
   FOR each immediate successor S of B DO {

      IF (S has not been collected &&
          S starts with a compare and branch &&
          S compares V with a constant) {

         Collect_Blocks(S, V);
      }
   }
}
```

Figure 4.1: Algorithm for Detection of Potentially Coalescent Branches

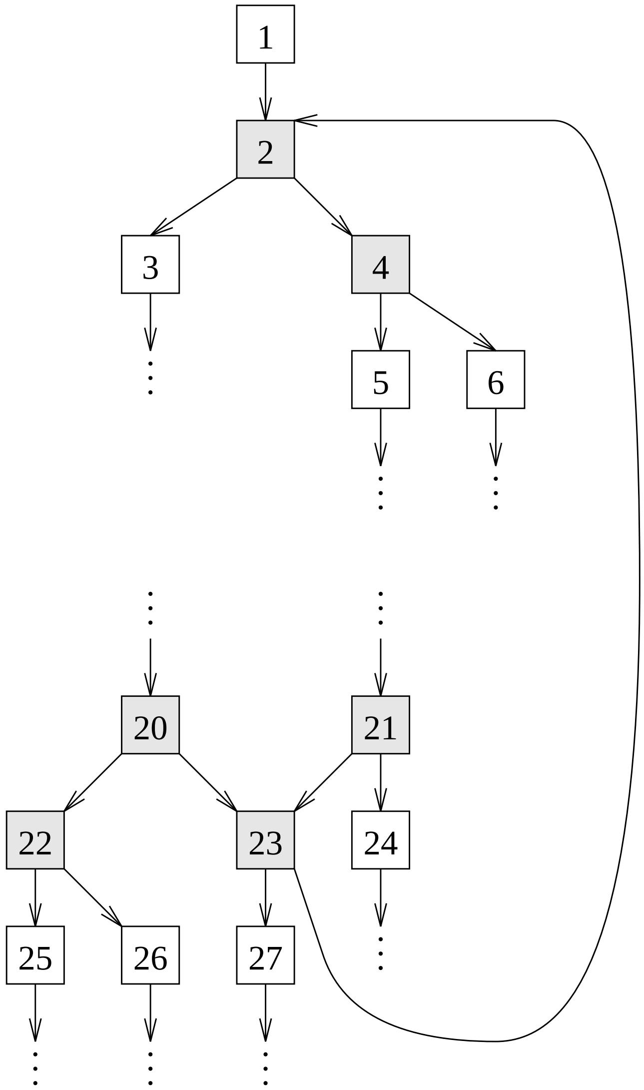


Figure 4.2: An Example Control Flow

4.2 Constructing the Jump Table

Once it has been determined that a set of conditional branches can be coalesced, a jump table must be constructed in order to perform the transformation. Construction of a jump table requires two steps.

1. Identify all possible targets for the indirect jump.

2. Associate each possible value of the branch variable with a single potential target.

To efficiently accomplish these steps, a DAG (Directed Acyclic Graph) is built as the blocks containing the coalescent branches are collected. Each node in the DAG represents one of the coalescent branches. Each edge represents either a transition between two such branches or a transition to a potential target of the indirect jump.

The benefits of using a DAG are as follows. First, all possible targets for the indirect jump can be quickly identified since they will be the targets of the transitions out of the DAG. Second, each nonoverlapping value range of the branch variable can be easily associated with a single target by propagating value ranges of the variable through the DAG. Each node will have two outgoing edges, one for the *true* (taken) transition and the other for the *false* (fall-through) transition. The possible range of values at each node is calculated by unioning the effect of applying the relational operator of each immediate predecessor node on its corresponding input range.

The use of a DAG allows coalescing of branches that check if a variable is within a specific range. For instance, the C code segment in Figure 4.3(a) checks if a character could be part of a C identifier. Figure 4.3(b) depicts the DAG that was built representing the control flow of the coalescent branches in the code segment. Nonoverlapping value ranges of the condition variable are mapped to the targets out of the DAG (**A**, **B**, **C**, **D**, and **E**). Note that at most one target from a transition out of the DAG will be permitted to have unbounded value ranges. For instance, only the D target has value ranges that cannot be represented in a jump table. Such a target would correspond to the default case of a C `switch` statement.

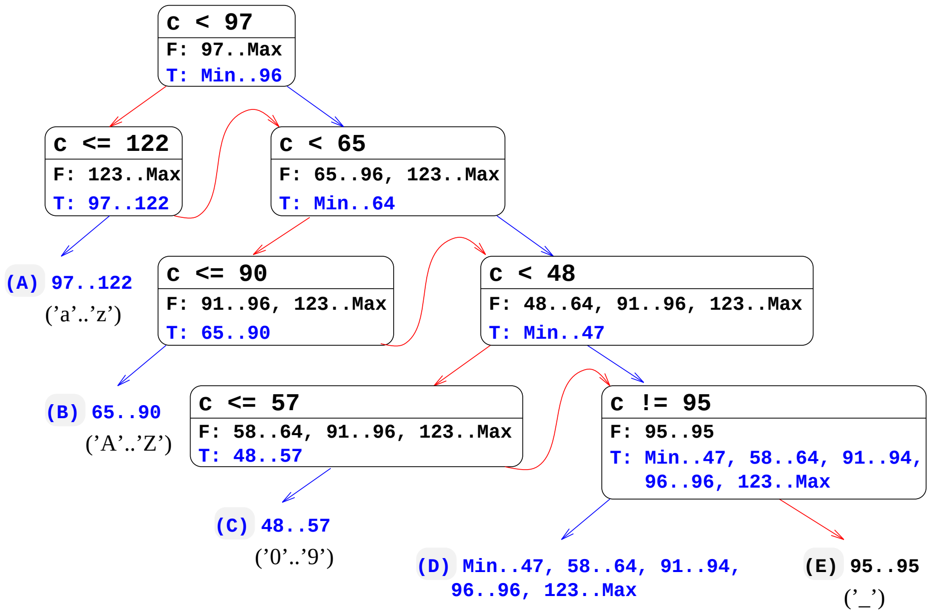
4.3 Estimating the Benefits of Coalescing a Set of Contiguous Branches

Before coalescing a set of contiguous branches, the compiler attempts to determine if the transformation is worthwhile. Our compiler inspects the DAG representing the branches to be coalesced. The number of instructions through each path in the DAG is calculated. The average number of instructions required to traverse the DAG is estimated by calculating a probability for each path through the DAG. The compiler also determines the number of instructions required to perform the indirect jump. If a benefit is predicted, then the branches are coalesced.

The probability of taking each path was estimated to obtain a more accurate prediction for the average number of instructions executed to traverse the DAG. Past studies always assumed that each case of a multiway selection statement, except for the default case, is equally likely [28]. However, the improving

```c
if ((c >= 'a' && c <= 'z') ||
    (c >= 'A' && c <= 'Z') ||
    (c >= '0' && c <= '9') ||
    (c == '_'))
{
  ...
}
```

(a) C Code Segment



(b) DAG Used for Value Range Analysis

Figure 4.3: Example of Checking If a Character Is Part of a C Identifier

transformation described in this dissertation coalesces branches that are generated from control statements other than multiway selection statements. Many studies have recently used heuristics [3], value range propagation [22], or empirical data from the execution of other programs [8] to predict the direction that branches will take. A different approach that is an extension of using value range propagation was found to be most effective by the author for the improving transformation in this dissertation. The range of values associated with the variable being compared at each node in the DAG was inspected when it was determined that the values being compared were within the range of possible character values. Each character value was also weighted according to an estimated frequency of common use. For instance, values representing ASCII letters and digits were assigned a higher weight than values representing control characters. The probability for the direction that a branch would take was calculated by using a ratio between the sum of the weights of the possible values of each of the two outgoing transitions from the branch. The probability of a path being taken through the DAG was simply the factor of the probability of each branch decision along that path. If the compiler could not determine that the comparisons were with character values, then each branch in the DAG was assumed to have an equal probability of being taken or falling through.

Figure 4.4 shows an example DAG with probabilities assigned to each transition. The DAG consists of three nodes, where each node represents two instructions, a comparison and conditional branch. There are five unique paths through the DAG. By using probabilities associated with the transitions, a weighted average number of instructions can be calculated as shown in Table 4.1.

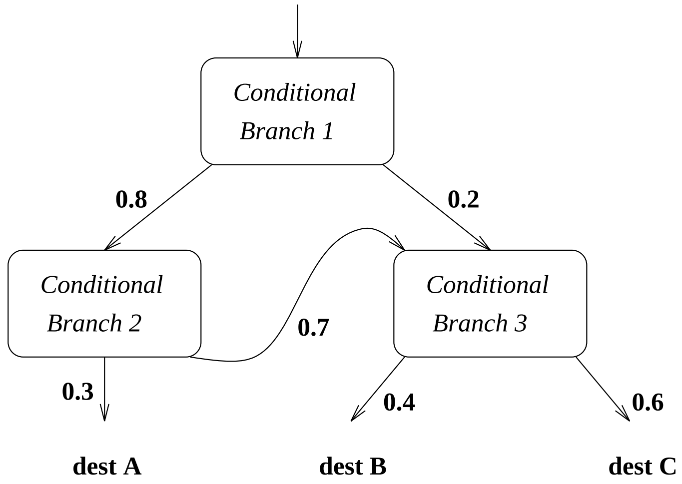


Figure 4.4: DAG with Weighted Edges

Table 4.1: Estimating the Number of Executed Instructions

Unique Path	Propagated Weight	Num of Instructions
1,2,A	0.8*0.3=0.24	4
1,2,3,B	0.8*0.7*0.4=0.224	6
1,2,3,C	0.8*0.7*0.6=0.336	6
1,3,B	0.2*0.4=0.08	4
1,3,C	0.2*0.6=0.12	4
Weighted		5.12

4.4 Transforming the Control Flow

After ensuring that the estimated execution cost of the detected sequence of coalescent branches outweighs the cost of performing an indirect jump operation, the branch at the head of the sequence being coalesced will be replaced by the instructions to perform the indirect jump. The original transitions from this head block will be deleted and replaced by transitions associated with the jump table targets. The other branches may or may not need to be deleted depending upon if transitions from other blocks can reach these branches.

Consider again the flow graph in Figure 4.2. The sequence of branches starting at block 20 (20, 22, 23, 2, 4) are coalesced into an indirect jump in Figure 4.5. The branch at block 20 was replaced by the indirect jump. The branch in block 22 was deleted after dead code elimination. The other branches will remain since there are transitions from block 21 and block 1 that can reach these branches. Figure 4.6 shows the effect of another coalescing transformation that replaces the branch in block 21 with an indirect jump. The branch in block 23 is deleted since its only other predecessor transition would be removed. Eventually, a coalescing transformation could be attempted on block 2 as well. The author did not coalesce a set of branches unless it was estimated that more than two branches would be executed on average. The detailed reason for this constraint is described later in Section 7.1.

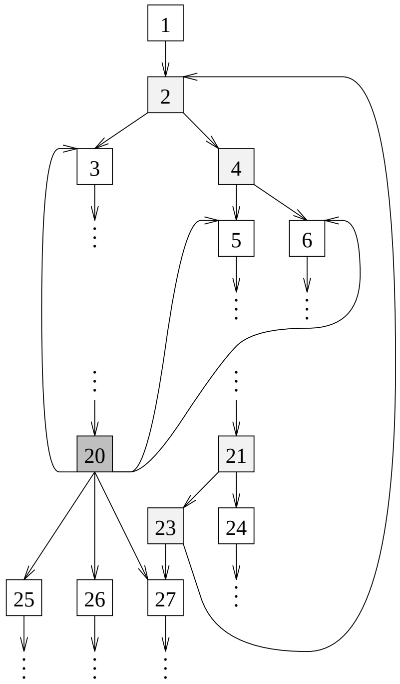


Figure 4.5: Coalesced Sequence Starting at Block 20

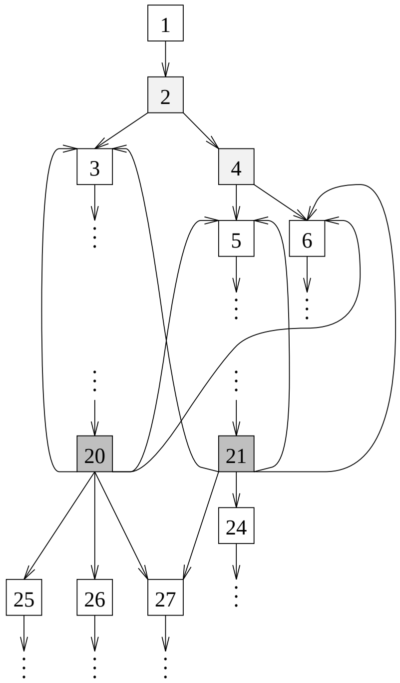


Figure 4.6: Coalesced Sequences Starting at Block 20 + 21

Chapter 5

COALESCING A SET OF NONCONTIGUOUS CONDITIONAL BRANCHES

The task for coalescing a set of potentially noncontiguous conditional branches, which are often separated by blocks of instructions, into an indirect jump was accomplished in the following manner. First, a set of coalescent conditional branches, which may or may not have intervening instructions, is identified. Second, a graph for the projected control flow is built to coalesce this set of conditional branches into an indirect jump. When the transformation is deemed beneficial, the original control flow is transformed according to the graph by duplicating basic blocks when necessary.

5.1 Finding A Set of Branches to Coalesce

A conditional branch is considered <u>***reachable***</u> from a point in a given control flow if there exists a path from that point to the conditional branch without the branch variable being affected. In order to find the largest set of coalescent branches, analysis is performed as follows. For each basic block B, the *reachable* branches from the exit point of B are determined. When B contains a conditional branch, the optimizer calculates the *reachable* branches that depend on the same branch variable as that of B. We denote such branches as ***related*** and denote B as the ***root*** block of these branches. After detecting all sets of *related* branches, the optimizer selects the set with the largest number of branches. The largest set should be chosen first since branch coalescing requires the allocation of registers.

The desired *reachability* information is collected by calculating the following state information for each basic block B.

- **in**: Set of blocks containing a *reachable* branch from the entry of B.
- **out**: Set of blocks containing a *reachable* branch from the exit of B. This includes the conditional branch in B, if one exists.
- **effect**: Set of blocks containing a branch instruction whose branch variable is updated by some instructions in B.

This state information is calculated by solving the following dataflow equations.

$$out[B] = out[B] \cup (\bigcup in[S]) \tag{5.1}$$

$$in[B] = out[B] \setminus effect[B] \tag{5.2}$$

Equations (5.1) and (5.2) were solved by the iterative algorithm shown in Figure 5.1. When the algorithm terminates, the *out* state of each basic block B contains the *reachable* branches from the exit point of B.[1]

Applying the iterative algorithm described in Figure 5.1 to the example control flow in Figure 5.2 produces the dataflow information as indicated by

[1]This algorithm is guaranteed to terminate since for any given control flow, (1) there exists a finite number of conditional branches, and (2) the *in* and *out* states of each block monotonically increase.

```
DO
  FOR each block B DO
      /* calculate B → OUT from the SUCCESSORS
         and its own branch */
     B->OUT := NULL.
     FOR each immediate successor S of B DO
        B->OUT := B->OUT ∪ S->IN.
     END FOR

     IF ( B contains a branch instruction) THEN
        B->OUT := B->OUT ∪ S->IN.

      /* calculate B → IN using B → OUT */
     B->IN := B->OUT \ B->EFFECT.
  END FOR
WHILE any changes
```

Figure 5.1: Reaching Algorithm

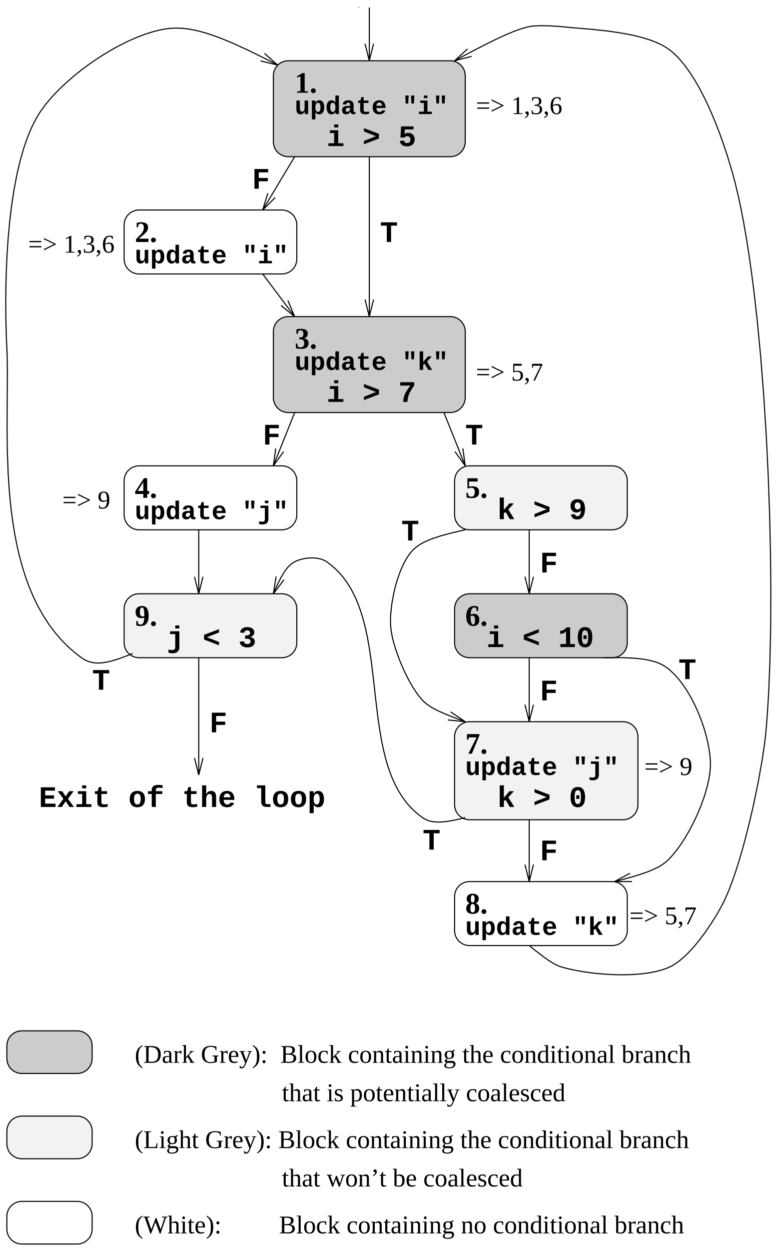


Figure 5.2: An Example Control Flow

Table 5.1. Since block 1 has the largest set of *related* branches, the compiler will first attempt to coalesce these branches by placing instructions to perform an indirect jump at the *root* block 1. However, it is possible that *related* branch sets of two or more blocks have the same cardinality. In this case the optimizer will choose the block that dominates the most blocks having branches in the same *related* set.

Table 5.1: Dataflow Information for the Example Control Flow

Block No.	***effect***	After The Algorithm		***Related*** Branches
		in	***out***	
block 1	1,3,6	null	1,3,6	{1,3,6}
block 2	1,3,6	null	3,6	null
block 3	5,7	3,6	3,5,6,7	{3,6}
block 4	9	null	9	null
block 5	null	5,6,7	5,6,7	{5,7}
block 6	null	6,7	6,7	{6}
block 7	9	7	7,9	{7}
block 8	5,7	null	null	null
block 9	null	9	9	{9}

5.2 Projecting the Restructured Control Flow

Once a set of *related* branches has been selected, the optimizer projects the revised control flow to coalesce these branches into a single indirect jump. The restructured control flow is calculated by recording states in each block for these *related* branches. The state associated with each *related* branch is defined to be a set of ***triples***, where each *triple* consists of the following components: (1) the block containing that branch, (2) whether the branch will be taken (T) or not taken (F), and (3) the value range of the branch variable to satisfy the condition that is specified by the second component.

The projected control flow is calculated in the following manner. For a given set of *related* branches and its associated *root* block, the optimizer propagates the state (*triples*) of each *related* branch backward through the control flow (toward its *root* block). When the propagation completes, the optimizer determines the sequence of *related* branches that would be executed starting from the root block for each nonoverlapping value range of the branch variable. At this point, cost-benefit analysis is performed to determine whether or not coalescing the set of *related* branches into an indirect jump is worthwhile. If it is deemed beneficial, then a graph is incrementally built to project the desired restructuring at the *root* block. If the optimizer determines that there will be no significant code-size increase, then the graph will later be used to modify the actual control flow.

As an illustration, consider the example control flow in Figure 5.2 with one additional assumption that the branch variable `i` was detected to contain an unsigned character value [0..255]. For the set of *related* branches at the *root* block 1, Table 5.2 shows the initial states associated with these branches.

Table 5.2: Initial States for *Related* Branches of Block 1

Related Branches	Initial States (*Triples*)
related branch in block 1	(1,T,[6..255]), (1,F,[0..5])
related branch in block 3	(3,T,[8..255]), (3,F,[0..7])
related branch in block 6	(6,T,[0..9]),(6,F,[10..255])

In order to propagate the *triples* for branch 6 toward the *root* block 1, this state information should be propagated through block 3. The transition from block 3 to block 6 can occur only when the value of branch variable `i` is in the range [8..255]. Similarly, the transition from block 1 to block 3 can occur only when the value of branch variable `i` is in the range [6..255]. Therefore, the value ranges of the *triples* for branch 6 should be properly adjusted during the propagation to reflect these two transitions. As shown in Figure 5.3, the value ranges of the triples for branch 6 are intersected with [8..255] at block 3, and the adjusted value ranges are intersected with [6..255] at the *root* block 1.

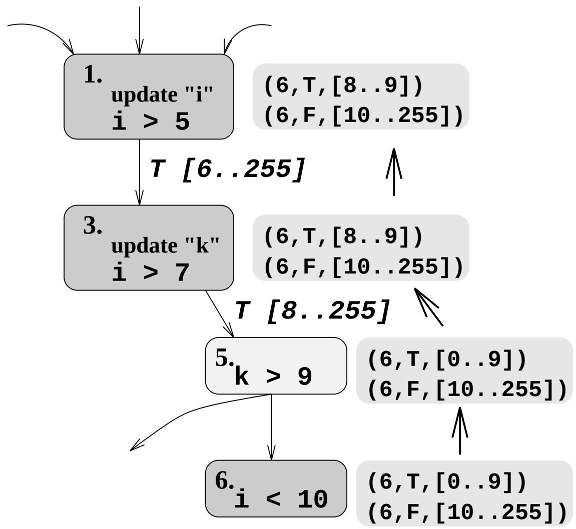


Figure 5.3: After Propagating *Triples* at the Block 6 toward the *Root* Block 1

Figure 5.4 shows the *triples* at the *root* block 1, after propagating all the states of these *related* branches. The value ranges of *triples* often overlap with some value ranges of other *triples*. This situation happens when more than one branch in the set of *related* branches can be executed for a given branch variable value. By properly reorganizing such overlapping value ranges, as depicted in Table 5.3, the optimizer can determine for each value range of `i` which *related* branches will be executed and whether these branches are taken (T) or not

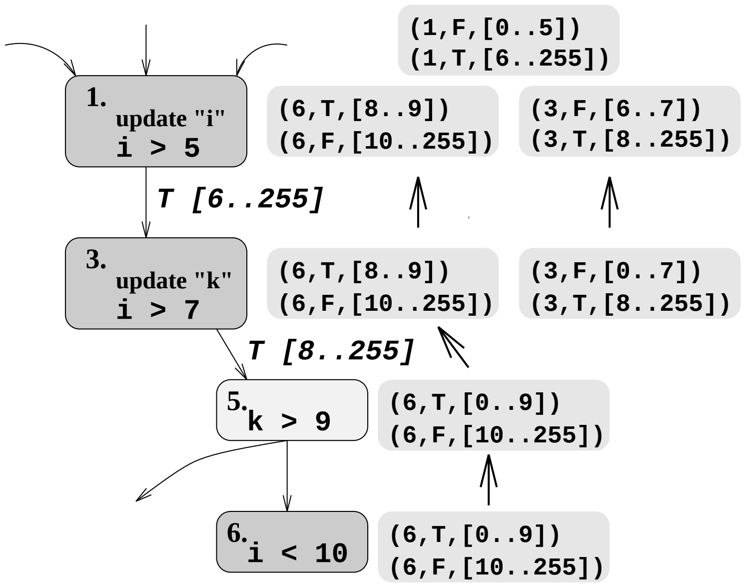


Figure 5.4: After Propagating *Triples* at the *Related* Blocks 1, 3, and 6 toward the *Root* Block 1

taken (F).

Similar cost-benefit analysis, as described in Chapter 4.3, is performed at this point to determine whether or not it is beneficial to coalesce the *related* branches of the *root* block into an indirect jump. The optimizer first checks if the values being compared are characters (represented in a byte). The optimizer weights the character values according to an estimated frequency of common use. For instance, values representing ASCII letters were assigned a higher weight than values representing control characters. The cost of executing the branches was calculated as a sum of products, where each product was obtained by multiplying the weights of the characters in each value range and the number of branches associated with that range. If the optimizer could not determine that the comparisons were with characters, then each value was given the same

Table 5.3: States of *Related Branches* Associated with Nonoverlapping Value Ranges of `i` at the *Root* Block 1

Value Range of `i`	States of *Related* Branches
[0..5]	1,F
[6..7]	1,T and 3,F
[8..9]	1,T and 3,T and 6,T
[10..255]	1,T and 3,T and 6,F

weight. The cost of executing the branches is compared to the cost of performing the indirect jump, which is described in the next section.

If the analysis determines that branch coalescing is worthwhile, then the restructuring algorithm shown in Figure 5.5 will produce a graph to efficiently represent the revised control flow to coalesce these *related* branches into an indirect jump at the root block. The central idea is that a new node will be added when no current node for that block exists with the same states for the *related* branches. The projected graph of the restructured control flow for Figure 5.2 is shown in Figure 5.6. The *related* branch in *root* block 1 will be replaced in the restructured code by instructions to perform an indirect jump. Note that a basic block represented with a dashed box indicates that the *related* branch is unnecessary and will not be placed in the restructured code.

```
PROCEDURE Build_Graph_From_Root(root_node,root_block)
{
   root_node = NewNode(NULL,root_block,NULL);

   FOR each non-overlapping value range VRANGE of
       the branch variable DO {
      current_states = related branch states associated with VRANGE;

      IF (current_states indicate related branch in root_block is taken)
         Build_Graph(root_node,root_block->taken,
                     current_states,root_block);
      ELSE
         Build_Graph(root_node,root_block->not_taken,
                     current_states,root_block);
   }
}

PROCEDURE Build_Graph(pred_node,successor_block,
                      current_states,root_block)
{
   /* Do not allow a cycle back to root_block */
   IF (successor_block == root_block)
     RETURN;

   /* Calculate new states */
   new_states = intersection between current_states and related
                branch states associated with successor block;

   IF (successor_block with new_states already
       exists in the graph) {
      Connect pred_node to the existing node;
      RETURN;
   }

   /* Create a new node for successor_block and
       append it to pred_node */
   new_node = NewNode(pred_node,successor_block,new_states);

   IF (successor_block contains related branch) {
       Mark new_node that the branch can be eliminated;
      IF (new_states indicate that successor of new node will be
          the branch target)
         Build_Graph(new_node,successor_block->taken,
                     new_states,root_block);
      ELSE
         Build_Graph(new_node,successor_block->not_taken,
                     new_states,root_block);
   }
   ELSE
      FOR each immediate successor block SUCC of successor_block DO {
         Build_Graph(new_node,SUCC,
                     new_states,root_block);
      }
}
```

Figure 5.5: Restructuring Algorithm

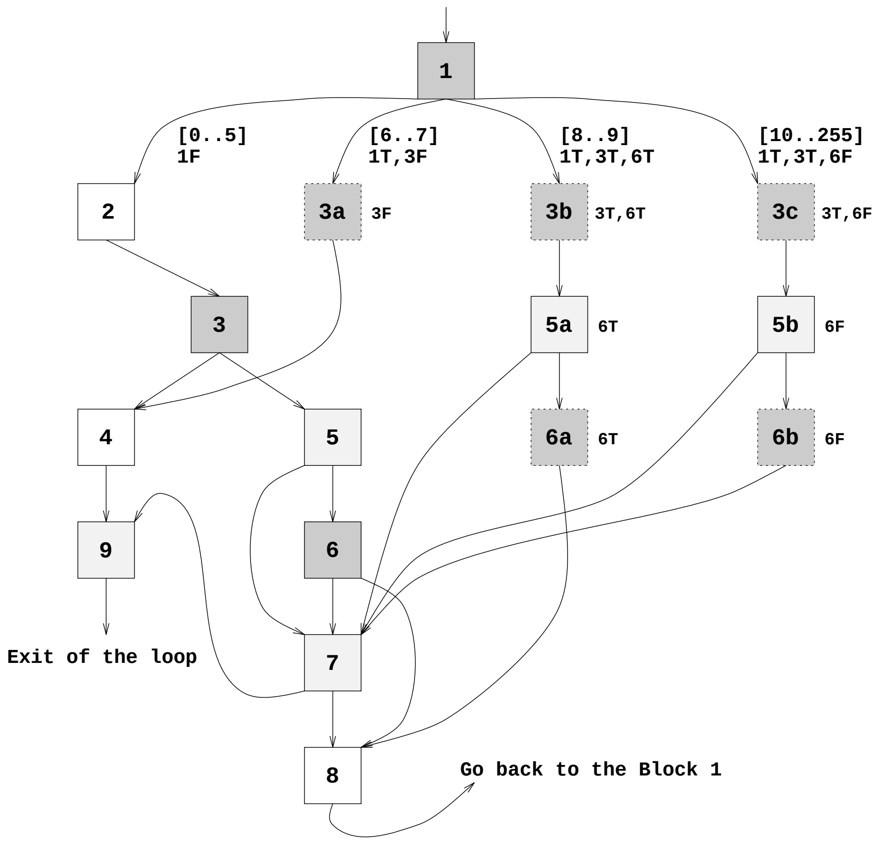


Figure 5.6: Graph Representing Restructured Control Flow for Figure 5.2

Chapter 6

EFFICIENTLY PERFORMING THE INDIRECT JUMP OPERATION

Compiler writers have long considered performing an indirect jump from a jump table as a very expensive operation. The tasks associated with performing an indirect jump includes the following:

1. checking if the value being compared is within a bounded range,
2. calculating the address of the jump table,
3. calculating the offset used to index into the table,
4. loading the target address from the table, and
5. performing the indirect jump.

The number of instructions required to perform an indirect jump from a jump table can vary depending upon a number of factors. For the C `switch` statement shown in Figure 6.1(a), Figure 6.1(b) depicts SPARC instructions represented as RTLs that are used to implement a corresponding indirect jump (disregarding the instruction in the delay slot of the indirect jump).[1] Similar

[1]These SPARC instructions are generated by the *pcc* [18], *gcc* [29], and *vpcc* [4] compilers.

instructions are available on most RISC machines. It would appear that at least 5 pairs of conditional branches must be executed to make coalescing branches into an indirect jump operation worthwhile on the SPARC since 8 instructions are used to implement an indirect jump.

(a)

```
switch (c) {
       case 'a':
          ...
       case 'b':
          ...
       case 'c':
          ...
       case 'd':
          ...
       case 'e':
          ...
       default:
          ...
     }
```

(b)

```
r[8]=r[8]-97;        # 1. Subtract the lowest case value
IC=r[8]?4;           # 2. Compare with (highest-lowest)
PC=ICh0,L27;         # 3. Perform unsigned > branch to
                          ensure the value is within range
                          (L27 is the default address)
r[20]=HI[L01];       # 4. Get High portion of address of
                          jump table
r[20]=r[20]|LO[L01]; # 5. Get Low portion of the address
r[8]=r[8]<<2;        # 6. Align value on a word boundary
                          so can index into jump table
r[8]=M[r[8]+r[20]];  # 7. Load target destination out of
                          jump table
PC=r[8];             # 8. Perform an indirect jump
L01:
.WORD  L22           # Target address for case 'a'
.WORD  L23           # Target address for case 'b'
.WORD  L24           # Target address for case 'c'
.WORD  L25           # Target address for case 'd'
.WORD  L26           # Target address for case 'e'
L27:
```

Figure 6.1: RTLs to Perform an Indirect Jump from a Jump Table

By statically analyzing the code surrounding an indirect jump operation, the optimizer can significantly reduce the cost of performing an indirect jump. Many optimizers can detect that instructions 4 and 5 are loop invariant and therefore can move these instructions out of a loop. The author implemented techniques that often avoid the execution of instructions 1-3 and 6 as well.

6.1 Padding the Front of the Table

Instructions 1-3 in Figure 6.1(b) are used to check if the expression is in the range of possible case values. Instruction 1 can be avoided when the lowest case value is positive and relatively close to zero. The jump table can be padded with the addresses corresponding to the default target. This technique is illustrated in Figure 6.2, which contains the instructions of Figure 6.1(b) with the modifications resulting from padding the front of the jump table. Instruction 2 in Figure 6.2 uses the highest case value in the comparison when padding is applied. Note also that instructions 4 and 5 in Figure 6.1(b) were removed in Figure 6.2 since it was assumed they are loop invariant for this example.

```
IC=r[8]?103;               # 2. Compare with (highest-lowest)
PC=ICh0,L27;               # 3. Perform unsigned > branch to
                                ensure the value is within range
                                (L27 is the default address)
r[8]=r[8]<<2;              # 6. Align value on a word boundary
                                so can index into jump table
r[8]=M[r[8]+r[20]];        # 7. Load target destination out of
                                jump table
PC=r[8];                   # 8. Perform an indirect jump
L01:
.word  L27                 # Target Address for 0
.word  L27                 # Target Address for 1
...                          ...
.word  L27                 # Target Address for 96 ('a'-1)
.word  L22                 # Target Address for 'a'
.word  L23                 # Target Address for 'b'
.word  L24                 # Target Address for 'c'
.word  L25                 # Target Address for 'd'
.word  L26                 # Target Address for 'e'
L27:
```

Figure 6.2: RTLs after Padding the Front of the Table

6.2 Using Value-Range Analysis to Avoid the Initial Range Check

The initial range check (instructions 1-3 in Figure 6.1(b)) can be completely avoided if a bounded range of case values is known and an entry can be stored in the table for each value [28]. Assume that the value range of the variable `c` in Figure 6.1(a) is [0..255]. The indirect jump operation associated with the known value range of the branch variable is depicted in Figure 6.3.[2]

Once a set of *related* branches has been selected, the optimizer *vpo* uses demand-driven analysis to recursively search all the possible paths backward from the *root* block to determine if the range of case values is bounded. In the following subsections, a general algorithm for such range determination is depicted, and several cases that can be handled by the algorithm are illustrated.

```
r[8]=r[8]<<2;             # 6. Align value on a word boundary
                               so can index into jump table
r[8]=M[r[8]+r[20]];       # 7. Load target destination out of
                               jump table
PC=r[8];                  # 8. Perform an indirect jump
L01:
.word  L27                # Target Address for 0
.word  L27                # Target Address for 1
...                         ...
.word  L27                # Target Address for 96 ('a'-1)
.word  L22                # Target Address for 'a'
.word  L23                # Target Address for 'b'
.word  L24                # Target Address for 'c'
.word  L25                # Target Address for 'd'
.word  L26                # Target Address for 'e'
.word  L27                # Target Address for 102 ('e'+1)
.word  L27                # Target Address for 103
...                         ...
.word  L27                # Target Address for 255
L27:
```

Figure 6.3: SPARC Instructions with a Bounded Range of Values

[2]Note that 256 targets are listed in the table. Often this space is reduced by a factor of four as described in the next section.

6.2.1 General Algorithm to Determine Bounded Value Ranges

A general algorithm for determining if the range of case values is bounded is shown in Figure 6.4. The essence of this algorithm is as follows.

1. Expand a branch variable using previous effects on the variable by recursively searching all the possible paths backward from the root block.

2. Whenever an expansion occurs, parse and evaluate the expanded expression to determine whether or not the range of case values can be determined.

The algorithm returns a state with a detected range of case values if one of the following conditions exists.

- **bounded**: The value ranges of a branch variable can be enumerated in a jump table.

- **unbounded**: The value ranges of the branch variable cannot be enumerated in a jump table.

- **duplicated**: The value ranges of the branch variable can be enumerated in a certain execution path. This state provides an extra opportunity for the optimizer to perform an indirect jump more efficiently in the **bounded** execution path by duplicating some blocks of instructions.

6.2.2 Analyzing Effects

For a given *root* block, a bounded value range of the branch variable can often be determined by examining each effect backward from the root. Consider the C code depicted in the left column of Figure 6.5 with an assumption that the block

```
PROCEDURE Bounded_Path(RTL_pointer, Register)
{
  current_block = basic block containing RTL_pointer.
  expanded_expr = Register.
  Value_Range_State =  None.
  Set_of_value_range = NULL.
  Set_of_duplicated_block = NULL.

  /* Expand and evaluate expanded_expr within current_block
     If the expanded_expr is determined to be BOUNDED, add
     the bounded value range to Set_of_value_range and return */

  WHILE (RTL_pointer = previous_rtl(RTL_pointer)) {
      Expand_and_Evaluate(RTL_pointer, expanded_expr,
                          Set_of_value_range,
                          Value_Range_State).

      IF (expanded_expr is either BOUNDED or UNBOUNDED)
         RETURN Value_Range_State.

      /* Alias effect such as r[9]=r[8] */
      ELSE IF (expanded_expr == Register &&
               RTL_pointer points to the instruction
               that assigns Register to New_Register)
         expanded_expr = Register = New_Register.
  }

  /* Neither BOUNDED nor UNBOUNDED state can be determined by
     evaluating expanded_expr. Expand and evaluate the expression
     by recursively looking back all predecessor blocks */

  FOR each predecessor block of current_block DO {
     temp_expr = expanded_expr.
     Recursively expand and evaluate temp_expr starting from
         the predecessor until temp_expr is determined to be
         either BOUNDED or UNBOUNDED.

     IF (temp_expr is determined to be BOUNDED)
        Add the associated value range to Set_of_value_range.
  }

  IF (Value_Range_State is both BOUNDED and UNBOUNDED &&
      there exists a single execution path along which
      the value range is BOUNDED) {

      Calculate Set_of_duplicated_block by taking intersection
        among sets of blocks along all possible execution
        paths to the root block.
      RETURN REPLICATED.
  }

  ELSE IF (Value_Range_State == BOUNDED)
      RETURN BOUNDED.

  ELSE
      RETURN UNBOUNDED.
}
```

Figure 6.4: Detection Algorithm for Bounded Value Ranges

containing the condition "`c == a`" has been selected as the *root*. The bounded value range of the condition variable `c` was detected by expanding register `r[8]`, which contains the temporary value of `c`, with previous effects on that register. The right column of Figure 6.5 depicts the RTLs when the branch coalescing analysis was about to be performed. The expansion of `r[8]` was accomplished as follow.

```
1. r[8]              # register containing values of 'c'
2. r[8]}24           # instruction 2: right shift('}') by 24 bits
3. (b[16]{24)}24     # instruction 1: left shift('{') by 24 bits
```

After the above expansion, the value range of `r[8]` was determined as bounded to the interval [-128..127], since the resulted effect from 24 bit left-shift followed by 24 bit right-shift is to mask the signed 8 bit value from `r[8]`. In a similar manner, the value range of a branch variable is determined as bounded to [0..255] when the variable can be expanded as the effect of unsigned byte load or conversion to an unsigned character value. Some other useful bounds were obtainable from the C mask operation, `'&'`.

6.2.3 Analyzing Effects for All Possible Paths

For a given *root* block, a bounded value range of the branch variable was often determined by recursively searching all the possible paths backward from the *root*. Consider the C code segment shown in Figure 6.6 with an assumption that the block containing the condition "`flag == 0`" has been selected as the *root*. The value range of the variable `flag` was determined by recursively searching all the possible paths backward from the *root* block. The optimizer determines

Example C source	**RTLs**
char c; ... if (c == ’a’) A(); else if (c == ’b’) B(); else if (c == ’c’) C(); ...	r[8]=b[16]{24; # 1. sll %l0,24,%o0 r[8]=r[8]}24; # 2. sra %o0,24,%o0 ... *# Block for c == ’a’* ... IC=r[8]?97; # 3. cmp %o0,97 PC=IC!0,L18; # 4. bne L18 *# Block for A()* ... *# Block for c == ’b’* L18: IC=r[8]?98; # 5. cmp %o0,98 PC=IC!0,L22; # 6. bne L22 *# Block for B()* ... *# Block for c == ’c’* L22: IC=r[8]?99; # 7. cmp %o0,99 PC=IC!0,L25; # 8. bne L25 *# Block for C()* ...

Figure 6.5: Example Case for Bounding Value Range

that the value of `flag` is bounded by the interval [0..4], since the value of `flag` is set to a certain constant in that interval for every possible path reaching the *root*.

```
int flag;
...
switch (*s) {
case 'f': case 'e': case 'g':
        flag = 1;
        break;

case 'd':
        flag = 2;
        ...
        break;

case 'o': case 'x':
        flag = *(s-1) == '1' ?  2 : 3;
        break;

case 's':
        flag = 4;
        break;

default:
        flag = 0;
        break;
}
if (flag == 0) {
   ...
}
...
```

Figure 6.6: Code Segment from *format()* in *awk*

6.2.4 Code Duplication for Bounded Value Range Path

Often a path of blocks is detected where the range of values is bounded and one or more paths are detected where the range is unbounded. Code is duplicated when deemed worthwhile to allow coalescing of branches to occur on the path with the bounded range. For example, Figure 6.7(a) shows a C code segment in *wc*, and the effects of the C statements in the shaded area are represented as RTLs with the control flow in Figure 6.7(b). The reaching algorithm in Figure 5.1 determined block 20 as the most beneficial *root* block. Note that the conditional branches in block 20 and block 24 were considered to be *related* since `r[10]` is an alias of `r[8]` by the RTL `r[10]=r[8]`.

Blocks 17 to 19 contain RTLs generated from invoking the `getc()` macro. Block 18 contains an RTL (`r[8]=B[r[9]]&255;`) that loads an unsigned character from a buffer and bounds the range of values from 0..255. Block 19 contains a call to `_filbuf`, which results in the value associated with `r[10]` being unbounded since no interprocedural analysis was performed. The optimizer recursively searches backwards and finds that blocks 20 and 18 are within a path back to the point where the range of values is bounded. Likewise, the compiler finds that blocks 20 and 19 are within a path where the range of values is unbounded. The intersection between the blocks in a bounded path and the blocks within any unbounded paths results in the block(s) that must be duplicated to distinguish the bounded path. Figures 6.7(c) shows the RTLs with the modified control flow after duplication of the block 20 and coalescing of the set of *related* branches. Coalescing can occur at the duplicated *root* (block 20') without an initial range check since the range of values is now bounded. Limits were placed on the amount of code allowed to be duplicated to prevent large code size increases.

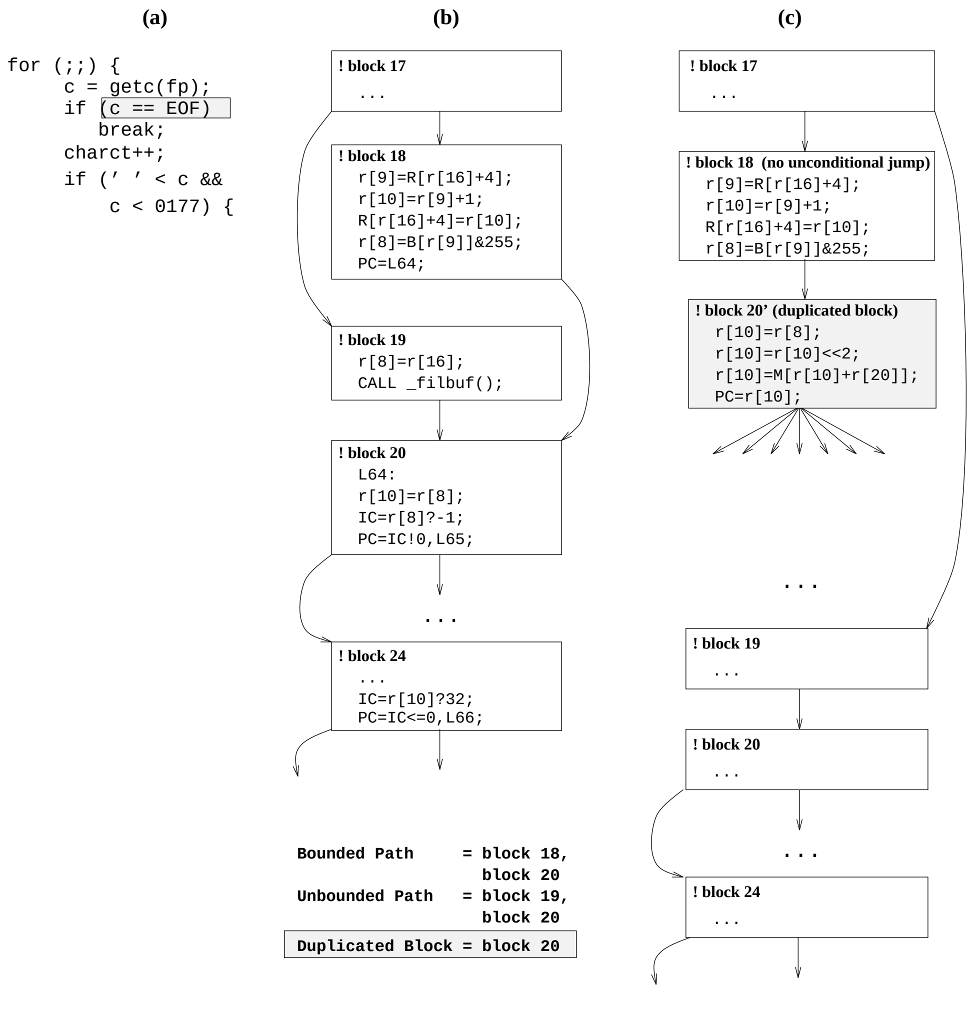


Figure 6.7: Using Duplication to Distinguish Paths for Coalescing

6.3 Efficiently Indexing into the Jump Table

Instruction 6 in Figure 6.3 left shifts the value by 2 since each element of the jump table contains a complete target address requiring 4 bytes. Consider tables containing byte displacements instead of complete word addresses. For instance, Figure 6.8 shows how the code in Figure 6.3 can be transformed to use byte displacements. There are two advantages for using byte displacements. First, the left shift will no longer be necessary. Second, the table only requires one fourth the amount of space. Thus, a jump table for a value range associated with a character can be compressed from 256 to 64 words.

```
                    # r[20] is the jump table address (L01)
                    # r[22] is the base address (L02) for the displacement
r[8]=M[r[8]+r[20]];         # 7. Load target destination out of jump table
PC=r[8]+r[22];              # 8. Perform an indirect jump
.seg  ''data''
L01:
.byte  L27-L02              # Target Address for 0
.byte  L27-L02              # Target Address for 1
...                           ...
.byte  L27-L02              # Target Address for 96 ('a'-1)
.byte  L22-L02              # Target Address for 'a'
.byte  L23-L02              # Target Address for 'b'
.byte  L24-L02              # Target Address for 'c'
.byte  L25-L02              # Target Address for 'd'
.byte  L26-L02              # Target Address for 'e'
.byte  L27-L02              # Target Address for 102 ('e'+1)
.byte  L27-L02              # Target Address for 103
...                           ...
.byte  L27-L02              # Target Address for 255
.align 4
.seg  ''text''
L27:
```

Figure 6.8: SPARC Instructions with Byte Displacements in the Jump Table

The disadvantages include requiring an additional register to calculate the base address for the displacements and not always having displacements small enough to fit within a byte. There are two approaches that were used to help ensure that the displacements are not too large. First, a label for the base of the displacements was placed at the instruction that was the midpoint between the first and last indirect jump targets. The jump table is always placed in the data segment so it will not cause the distance between indirect jump targets to be increased. Note this requires the calculation of the addresses of two labels (the one at the beginning of the jump table and the one used for the base address of the displacements). Before applying this approach, the compiler first ensures that the indirect jump would be in a loop and registers are available to move the calculation of both addresses out of the loop.

Second, the targets of the indirect jump may be moved to reduce the distance between targets. The instructions within a program may be divided into relocatable segments. Each segment starts with a basic block that is not fallen into from another block and ends with a block containing an unconditional transfer of control. An example of relocatable code segments is given in Figure 6.9. Assume each of the labels in the figure are potential targets of one indirect jump. There are three ways segments can be moved to reduce the distance between targets.

1. A segment that does not contain any targets for a specific indirect jump can be moved when it is between segments containing such targets. For example, segment **D** can be moved to follow segment **A** since both segments contain no targets for the indirect jump.

2. The segment containing the most instructions preceding the first target label in a segment can be moved so it will be the first segment containing targets. For example, segment **C** has blocks of instructions preceding the

block containing its first target label (**L2**). By moving segment **C** to follow segment **D**, these instructions preceding **L2** will be outside the indirect jump target range.

3. Likewise, the segment containing the most instructions following the last target label in its own segment can be moved so it will be the last positional segment containing targets. For example, segment **B** has the most instructions following its last target label (**L1**) and is moved to follow segment **E**. Jump tables are only converted to tables containing byte displacements when all targets of the indirect jump will be within the range of a byte displacement after relocating segments of code.

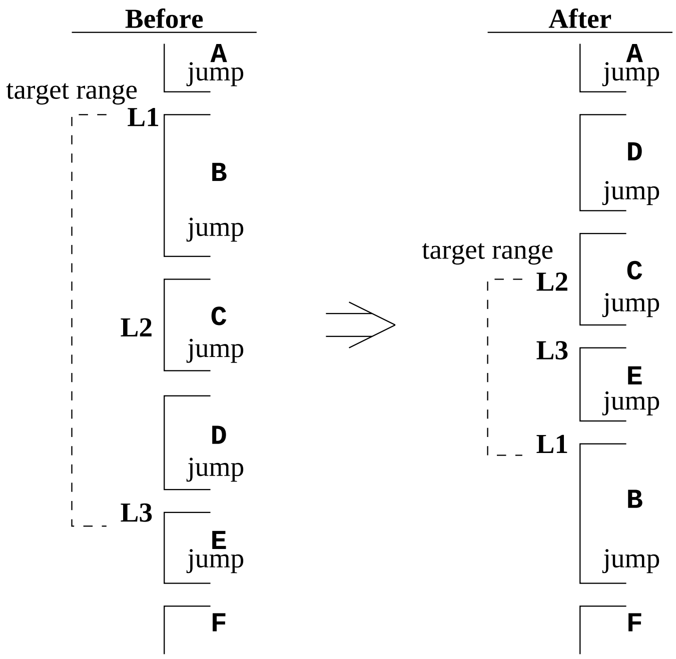


Figure 6.9: Relocating Segments of Code

6.4 Filling Delay Slots for Indirect Jumps

The optimizer *vpo* previous to this work only filled delay slots of indirect jumps with instructions that precede the jump. This approach was reasonable since indirect jumps with tables occurred infrequently and filling the delay slot from one of several targets is more complicated than filling the delay slot of a branch instruction. After implementing the transformation to coalesce branches, indirect jumps occurred much more frequently. The compiler has been modified to fill the delay slot of an indirect jump with an instruction from one of the targets if it could not be filled with an instruction that preceded the jump. An instruction from a target block could only be used to fill the delay slot if it did not affect any of the live variables or registers entering any of the other target blocks.

Filling a slot for an indirect jump is less advantageous than that for a conditional branch (or unconditional jump) since more targets are associated with an indirect jump. Therefore, the optimizer *vpo* tried the following method to usefully fill slots for indirect jumps. Since each target of an indirect jump has been associated with certain range(s) of case values, the probability of the transition from an indirect jump to a certain target can be statically estimated. The optimizer *vpo* ranks the indirect jump targets based upon such estimation, and attempts to fill its slot with the instruction from the most probable target.

When a set of branches that are originally separated by some intervening instructions is selected for branch coalescing, the actual transformation is accomplished by duplicating these intervening instructions. In such a case, the usefulness of filling slots for indirect jumps can be significantly improved. For example, consider the following C code in Figure 6.10(a). The detection and restructuring algorithms in Chapter 4 allow the optimizer to detect all the branches in the shaded area as coalescent and transform these branches into an indirect jump by duplicating the effects of `wd++` to several destinations of the coalescent branches. The restructured code in Figure 6.10(b) shows the comparable C code after the transformation.

(a) Original Code

```
while (*wd)
  switch (*wd++) {
  case 'l':
          ipr(linect);
          break;
  case 'w':
          ipr(wordct);
          break;
  case 'c':
          ipr(charct);
          break;
  }
```

(b) Restructured Code after Duplicating "wd++"

```
again:
switch (*wd) {
case '\0':
         break;
case 'l':
        wd++;
        ipr(linect);
        goto again;
case 'w':
        wd++;
        ipr(wordct);
        goto again;
case 'c':
        wd++;
        ipr(charct);
        goto again;
default:
        wd++;
        goto again;
}
```

Figure 6.10: Code Segment from *wcp()* in *wc*

After coalescing with code duplication, most of targets of the indirect jump have identical effects of `wd++` as depicted in Figure 6.10(b). Thus, the code duplication from branch coalescing potentially provides extra opportunities to fill the delay slot of the indirect jump with a useful instruction. However, there is one more complication that should be resolved for successfully filling an instruction of `wd++` for the delay slot of the indirect jump. The RTLs shown in

Figure 6.11(a) depict the restructured code after branch coalescing transformation occurs for the example C code. Note that hardware register `r[24]` contains the temporary value of `wd`. It appears that `r[24]=r[24]+1` cannot be filled for the delay slot, since `r[24]` is both set and referenced among targets of the indirect jump. However, `r[24]=r[24]+1` can be filled for the following reasons:

- `r[24]=r[24]+1` has no dependency with other instructions within the indirect jump target block containing the same instruction.

- `r[24]=r[24]+1` has no set-and-reference conflict when the analysis is performed by considering the targets containing that instruction as one conceptual target.

In order to fill such an indirect jump delay slot as described in the situation above, the following extra steps were added to *vpo*.

1. For each target of the indirect jump, evaluate the probability that the target may be taken using the associated case values in the jump table. When the range of case values is bound to values representing ASCII letters, the probability is further weighted using estimated character frequency distribution of common use.

2. Sort the indirect jump targets based on the evaluated probabilities.

3. Starting from the most probable jump target to the least, make a list of all the instructions that can be potentially filled for the indirect jump without considering effects from other jump targets. Whenever an identical instruction is found in an other target block, add the associated probability to that of the instruction.

(a) Before filling delay slot for indirect jump

```
r[8]=(B[r[24]]{24)}24;
r[9]=(B[r8]+r[20]]{24)}24;
PC=r[9]+r[21];
delay slot
.seg "data"
.byte L0019-L0017
...
L0020:
.byte L82-L0017
...
.byte L0016-L0017
...
.byte L0017-L0017
...
.byte L0018-L0017
...
.align 4
.seg "text"

L0019:
r[24]=r[24]+1;
PC=L83;

L0018:
r[24]=r[24]+1;
r[8]=r[26];

L0017:
r[24]=r[24]+1;
PC=L87;

L0016:
r[24]=r[24]+1;
r[8]=r[25];

L82:
PC=RT;
NL=RS[];
```

(b) After filling delay slot for indirect jump

```
r[8]=(B[r[24]]{24)}24;
r[9]=(B[r8]+r[20]]{24)}24;
PC=r[9]+r[21];
r[24]=r[24]+1;
.seg "data"
.byte L0019-L0017
...
L0020:
.byte L82-L0017
...
.byte L0016-L0017
...
.byte L0017-L0017
...
.byte L0018-L0017
...
.align 4
.seg "text"

L0019:
! filled for the indirect jump
PC=L83;

L0018:
! filled for the indirect jump
r[8]=r[26];

L0017:
! filled for the indirect jump
PC=L87;

L0016:
! filled for the indirect jump
r[8]=r[25];

L82:
PC=RT;
NL=RS[];
```

Figure 6.11: RTLs after Filling Delay Slot of the Indirect Jump for Example C Code in Figure 6.10(a)

   (a) If an instruction does not exist in the list, then insert the RTL with its associated block address and probability.

   (b) else (the same RTL already found on other target block), add the associated probability to that of the existing RTL in the list and add the associated block address to the block address list of the existing RTL.

4. Starting from the most probable instruction, determine if this instruction sets any variables or registers that could be live when entering any of the target blocks that do not have this instruction. If there is no conflict, then fill the delay slot with this instruction and delete it from the appropriate target blocks.

Chapter 7

OTHER ARCHITECTURAL ISSUES FOR COALESCING BRANCHES

The cost of performing an indirect jump from a jump table can vary on different machines. Not only can the number of instructions required to perform this operation vary, but indirect jump instructions (as well as conditional branches) can also result in pipeline stalls on many machines.

7.1 Dual Loop Test

To realistically estimate the pipeline impact on RISC architectures from replacing several conditional branches into an indirect jump, a dual loop test [10, 2] has been conducted on a SPARCstation-IPC, SPARCstation-5, SPARCstation-20, and UltraSPARC-1.

- First, an optimized executable[1] for the C code in Figure 7.1 has been generated to estimate the execution time involved with loop overhead. Let E_{loop} denote such an executable.

---

[1] Appendix A shows the optimized SPARC assembly code.

- Second, two optimized executables[2]with linear branches and with an indirect jump from a table, were generated for the C code shown in Figure 7.2. Let E_{linear} and $E_{indirect}$ denote such executables, respectively. Note that E_{linear} requires the execution of 2.5 branches on average for each loop iteration. Note that $E_{indirect}$ has been generated such that all the conditions in the loop body have been coalesced into an indirect jump operation requiring only two SPARC instructions as shown in Figure 6.8.

- Third, the author ran each executable 20 times, and chose the shortest execution time for each executable. Let $\tau_{E_{loop}}$, $\tau_{E_{branches}}$, and $\tau_{E_{indirect}}$ represent such shortest execution times respectively. ($\tau_{E_{linear}}$ - $\tau_{E_{loop}}$) gives a relative estimate of the total time required to execute the the conditional branches over all iterations. ($\tau_{E_{indirect}}$ - $\tau_{E_{loop}}$) gives a relative estimate of the time that is required to perform an indirect jump operation as shown in Figure 6.8, over all iterations.

- Finally, by varying the number of conditions in the loop, the relative impact of conditional branches versus an indirect jump has been measured as shown in Table 7.1.

From the dual loop test as described above, the author found that an indirect jump as depicted in Figure 6.1(c) required about the same execution time as two pairs of compare and branch instructions for most SPARCstations except the UltraSPARC-1. Therefore, the indirect jump transformation is only applied when it is estimated that more than two coalescent branches in the set will on

[2]Appendices B and C show the optimized SPARC assembly code respectively.

```
int i;
main()
{
   long int j, k, l;
   struct timeval before,after;

   gettimeofday(&before, (struct timezone *)NULL);
   k = 0;
   l = 0;
   for (j=0; j<10000000; j++) {
      i = j & 3;
   }
   gettimeofday(&after, (struct timezone *)NULL);
   after.tv_sec -= before.tv_sec;
   after.tv_usec -= before.tv_usec;
   if (after.tv_usec < 0)
      after.tv_usec--, after.tv_usec += 1000000;
   ...
   ...
   printf(``The elapsed time: %9ld.%02ld\n'',
            after.tv_sec, after.tv_usec/10000);
}
```

Figure 7.1: Code to Measure the Execution Time for Loop Overhead

```
   ...
   gettimeofday(&before, (struct timezone *)NULL);
   k = 0;
   l = 0;
   for (j=0; j<10000000; j++) {
      i = j & 3;

      /* 2.5 DYNAMIC NUMBER OF BRANCHES */

      if (i == 0) {
         k = k + 4;
         l = 4;
      }
      else if (i == 1) {
         k = k + 1;
         l = 1;
      }
      else if (i == 2) {
         k = k + 2;
         l = 2;
      }
      else if (i == 3) {
         k = k - 3;
         l = 3;
      }
  }
  gettimeofday(&after, (struct timezone *)NULL);
  ...
  printf(``The elapsed time: %9ld.%02ld\n'',
           after.tv_sec, after.tv_usec/10000);
  }
```

Figure 7.2: Code to Measuring the Execution Time for Loop Overhead and Loop Body

Table 7.1: Dual-Loop Test (10,000,000 iterations)

Machine Type	*Loop Cost*	*Linear Search*			*Indirect Jump*		
		2.5 br	4.5 br	8.5 br	2.5 br	4.5 br	8.5 br
SPARCstation-IPC	3.65s	3.82s	5.53s	8.82s	2.61s	2.71s	2.76s
SPARCstation-5	0.88s	1.03s	1.65s	2.74s	0.63s	0.76s	0.76s
SPARCstation-20	0.51s	0.93s	1.60s	2.65s	0.87s	0.93s	0.93s
UltraSPARC-1	0.40s	0.50s	1.16s	1.56	1.50s	1.51s	1.51s

average be executed. For the UltraSPARC-1, an indirect jump as depicted in Figure 6.1 required about the same execution time as eight pairs of compare and branch instructions. The major reason is that the UltraSPARC-1 (a Superscalar architecture) provides the hardware branch target/prediction buffer support for branches, but no hardware support for indirect jumps. In the following section, the author argues that, with a comparable hardware branch target/prediction buffer support, such unbalanced execution time discrepency can be eliminated.

7.2 Branch Target Buffer(BTB) Support for Branches and Indirect Jumps

One characteristic feature of RISC machines is pipelining. Pipelining divides the execution of each instruction into several stages. Different stages can be overlapped in execution to increase processor throughput. However, there are several obstacles that limit the full exploitation of pipelining. One of the most serious obstacles is branch instructions. If the current instruction turns out to be a branch, then the CPU should predict in advance whether or not the branch is taken and what the target address will be in order to preserve a steady flow through the pipeline. However, the execution path of a branch cannot be easily resolved in advance. Thus, branches typically cause delays in the pipeline [30, 23, 9, 15].

A Branch Target Buffer (BTB) can reduce these pipeline disruptions by predicting the path of the branch and caching information used by the branch. Various pieces of information can be kept in the BTB, including tags associated with the branch address, the branch target address, and branch prediction information [23]. However, it has been reported that BTB-based prediction schemes perform poorly for indirect jumps, since the target of an indirect jump can change with every dynamic instance of that branch [9, 30]. In fact, some compilers provide techniques that insert extra conditional branches that check for likely targets to avoid the execution of indirect jumps from a table [17] or indirect calls [7].

Most modern architectures seldom support indirect jumps in BTB due to such poor misprediction ratios for indirect jumps. However, consider the results shown in Figure 7.1. An UltraSPARC-1 could execute about eight pairs of compare and branch instructions in the time required to perform an indirect jump operation. One reason for the lower relative performance for indirect jumps on the UltraSPARC-1 was that this machine uses a BTB to provide architectural support for branches. There was no target buffer support on the UltraSPARC-1 for indirect jumps, which resulted in all indirect jumps being treated as mispredictions.

In the following sections, the author claims that, with comparable BTB support for indirect jumps, the branch coalescing transformation can be beneficial in reducing the total number of dynamic branch mispredictions. First, a conceptual design of BTBs is proposed that can provide comparable target buffer support for indirect jumps. Second, various branch prediction approaches will be described. Using more sophisticated branch prediction approaches as well as increasing the number of entries in BTBs is known to improve BTB performance [23]. Third, issues will be presented about how to manage BTBs that support branches and indirect jumps. Finally, with comparable BTB support for indirect jumps, the author will provide arguments describing why the total number of branch mispredictions can be reduced by the branch coalescing transformation. In addition, another compiler technique will be introduced that can potentially reduce the number of dynamic indirect jump mispredictions.

7.2.1 A Conceptual BTB Supporting Branches and Indirect Jumps

Target buffers are available to reduce the cost of indirect jumps on some machines. These buffers are typically specialized to support indirect jumps generated from *return* statements since indirect jumps from tables are not generated frequently by most compilers [15](see page 276). However, BTBs can be easily extended to support indirect jumps from tables by considering an indirect jump as another PC-relative branch instruction [15](see page 274). For instance, Figure 7.3 shows one conceptual view of a BTB, which, like a cache, can have several alternative designs. If the appropriate tag is not found in the buffer, then the hardware predicts that the branch will not be taken. If the appropriate tag is found in the buffer and a branch predictor indicates the branch as taken, then the hardware predicts that the branch will be taken. Otherwise, the branch is predicted as not taken.

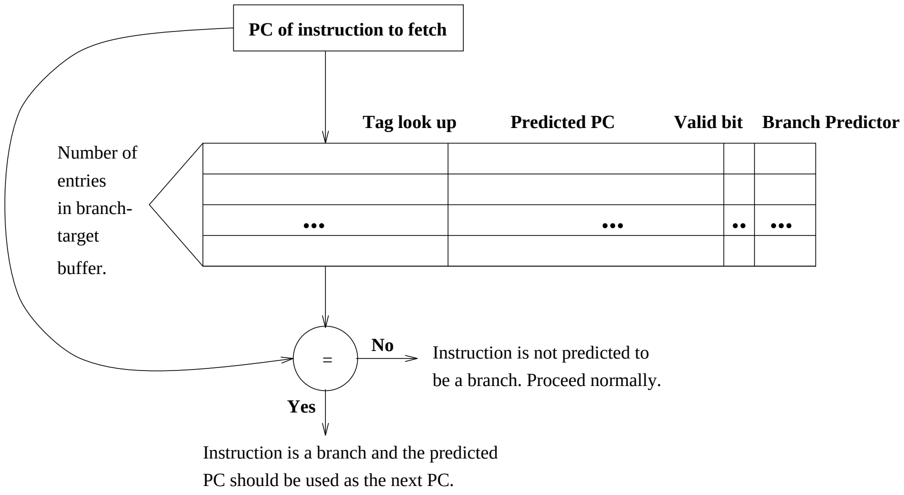


Figure 7.3: A Branch Target Buffer

7.2.2 Branch Predictors

The n-bit predictor scheme predicts the outcome of the branch using 2^n state diagram. When n is equal to one, the predictor predicts the next execution path of a branch based upon the previous outcome of the branch. This predictor has a performance drawback such that, when a loop branch is almost taken, the same branch will likely be predicted incorrectly twice, rather than once. As an illustration for such a mispredicted branch, consider the example code fragment shown in Figure 7.4. Assume that one-bit prediction information is in the BTB for `branch 2`. Mispredicting the tenth iteration of `branch 2` is inevitable since one-bit prediction information indicates that `branch 2` will be taken. However, when `branch 2` is accessed again after entering the inner loop for the second time, `branch 2` will be mispredicted as not taken. Thus, the prediction accuracy for `branch 2` that is taken in 90% of the iterations turns out to be only 80%.

In order to remedy this, two-bit predictor are often used. Consider the two-

```
i = 1;
while (i < 10) {        /* branch 1 */
   j = 1;
   while (j < 10) {     /* branch 2 */
      ...
      j++;
   }
   i++;
}
...
```

Figure 7.4: An Example for a Mispredicted Branch

bit state diagram shown in Figure 7.5. By having intermediate branch prediction states, such as **State 1** and **State 2**, the above performance shortcoming of one-bit predictor can be resolved. The two-bit predictor approach has been reported to do almost as well as the more general n-bit predictors [15](see page 263), and most machines rely on the two-bit predictor instead of the more general n-bit predictor.

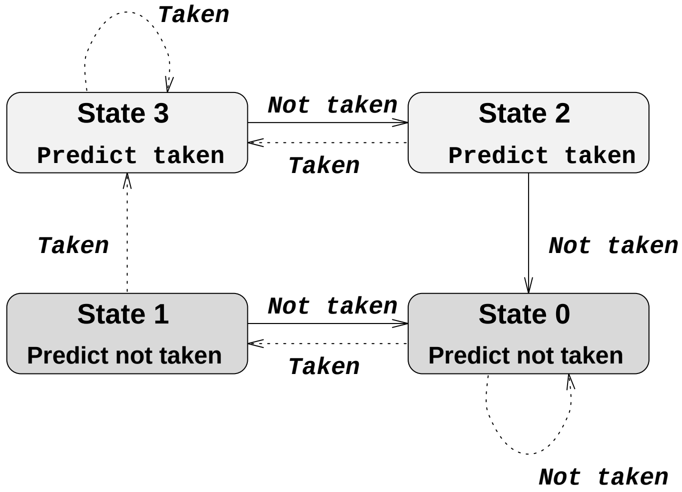


Figure 7.5: The states in a two-bit predicton scheme

In many cases, the execution path of a branch can be easily determined by observing the outcomes of the previous branch executions [21]. Consider the code fragment in Figure 7.6. If the branch 1 and 2 are taken, then the

```
if (aa == 2)          /* branch 1 */
   aa = 0;
if (bb == 2)          /* branch 2 */
   bb = 0;
if (aa != bb) {       /* branch 3 */
   ....
}
```

Figure 7.6: An Example Code Fragment for Branch Correlation

execution path of branch 3 can be easily predicted as not taken. The n-bit predictors can be further improved to make a prediction by using the outcomes of other branches. Such predictors are known as (m,n) correlation predictors. They use the outcome of the previous m branches to choose from 2^m branch predictors, each of which is a n-bit predictor for a single branch. The (m,n) predictors require one m-bit shift register to store the outcomes of the last m branch execution (0 for not taken, 1 for taken). This shift register can identify 2^m different contexts of a branch. Studies reported that (m,n) correlation predictors provide more accuracy than that of n-bit predictors [21, 15].

7.2.3 BTB Management

The target address for a branch is only placed in the buffer once the branch is taken. An indirect jump can be considered not taken (and therefore not placed in the buffer) if the target is the instruction following the indirect jump. If a branch (or indirect jump) is not in the buffer and it was not taken, then no delay is necessary since the not taken address is already calculated by the CPU. To maximize the performance of BTB, a branch (or an indirect jump), which is not in the BTB and is not taken, never replaces an entry in the buffer [23]. This approach has the effect of never replacing an entry in the buffer with a branch (or an indirect jump) that is not taken. Remember that a branch (or an indirect jump) is predicted as not taken if it is not found in the buffer. If the actual target of the indirect jump does not match the target in the buffer, then the branch target buffer is updated to contain the last target of the jump unless the same target is still predicted as taken. Note that, when the BTB uses correlating information from a (m,n) correlation predictor, the m-bit shift register does not reflect the outcome of previous indirect jump executions. The major reason is that there are several targets of the indirect jump that can be considered as taken addresses [30]. However, indirect jumps still use correlating information from the previous m executed branches.

7.2.4 Expected Benefits from Branch Coalescing Transformation with BTBs

Indirect jumps typically have higher misprediction rates than conditional branches since an indirect jump may have many possible targets [9]. It is the author's contention that higher misprediction rates do not necessarily mean worse performance. One must remember that several branches are being coalesced into a single indirect jump. Thus, the total number of mispredictions instead of the misprediction rate should be used when trying to measure branch target buffer performance with and without branch coalescing.

The author argues that with comparable branch target buffer support, an indirect jump will cause no more mispredictions than the set of conditional branches it replaced. If the target of an indirect jump is mispredicted, then the target of the indirect jump changed from the last time it was executed. Likewise, at least one of the conditional branches that would have been executed instead of the indirect jump must have had different behavior and would also likely result in a misprediction. There are actually two reasons why fewer mispredictions would occur after branch coalescing. First, an indirect jump can cause at most one misprediction when executed. The execution of a sequence of the replaced conditional branches may cause multiple mispredictions. Second, there should be less contention for entries in the branch target buffer since there will be only one indirect jump as compared to the set of branches the indirect jump replaced.

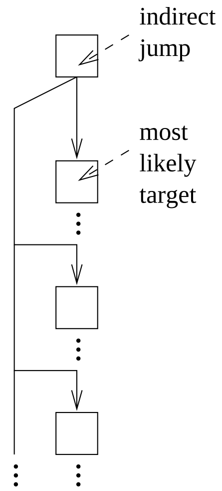


Figure 7.7: Placing the Most Likely Target

Architectural and compiler support can be used to further reduce the number of mispredictions from indirect jumps. Indirect jump history and a target cache containing the targets of the indirect jump that have been encountered have been used to improve prediction accuracy [9]. The author used compiler support to reduce the number of mispredictions. Often targets of an indirect jump have the block containing the indirect jump as their only predecessor.

Value range analysis was performed to predict the most likely target for each indirect jump, which was placed immediately following the indirect jump block as shown in Figure 7.7. Thus, jumps to this target will result in no delay when the tag for the indirect jump is not found in the buffer since this address will be treated as the not taken address. Note that the author does not suggest that the described approach is the best BTB design and configuration to support indirect jumps. Instead, the author is simply showing that, with comparable BTB support for indirect jumps, aggressively coalescing branches into indirect jumps can result in improved branch prediction performance. The branch prediction simulation results from various configurations will be shown in Chapter 8.1.3. With specialized BTB support for indirect jumps [9], even better results should be obtained.

Some machines provide other special architectural support for speculative execution of instructions dependent on branches, such as boosting [27] and predicted execution [24, 19]. The relative cost of an indirect jump versus the set of branches it replaces will be affected by such support. The compiler writer must use appropriate cost estimates based on the architectural support available for branches and indirect jumps on the target machine. An optimizer could also later convert indirect jumps into a sequence of conditional branches to exploit such architectural support.

Chapter 8

RESULTS

Various measurements are given in this chapter that shows the benefits of applying the branch coalescing transformation. Several common Unix utilities, as shown in Table 8.1, were selected as benchmarks since such non-numerical applications tend to have complex conditional control flow.

First, the following dynamic measurements were obtained by instrumenting the code generated for the SPARC by *vpo* (Very Portable Optimizer [4]) with all conventional optimizations applied.

1. Number of instructions executed

2. Cache work

Second, actual *execution time* measurements on SPARCstations were obtained to determine the soundness of the above measurements. Finally, *compile-time* measurements on the SPARCstation were collected to measure the additional time required to perform the branch coalescing transformation as an extra optimization phase.

Table 8.1: Benchmark Test Files

PROGRAM	DESCRIPTION
awk	pattern scanning and processing language
cb	a simple C program beautifier
cpp	C Compiler Preprocessor
ctags	generate tag file for emacs, vi
deroff	remove nroff, troff, tbl and eqn constructs
grep	search a file for a string or regular expression
hyphen	search a file for hyphenated words and lists the words
join	relational database operator
lex	lexical analysis program generator
nroff	format documents for display or line-printer
pr	prepare file(s) for printing, perhaps in multiple columns
ptx	generate permuted index
sdiff	contrast two text files by displaying them side-by-side
sed	stream editor
sort	sort and collate lines
wc	display a count of lines, words and characters
yacc	parser generator

8.1 Dynamic Measurements by Instrumenting Code

The following measurements were collected on code generated by compiler *vpo* (Very Portable Optimizer) using EASE (Environment for Architectural Study and Experimentation [13]) on the SPARC architecture for the Unix utilities described in Table 8.1.

8.1.1 Number of Instructions Executed

Table 8.2 shows the number of instructions executed for each benchmark. The *None* column contains the number of instructions executed, which was obtained by modifying the C front end, VPCC, to never translate a C `switch` statement using an indirect jump. The *Original* column shows the percentage change as compared to *None* when indirect jumps from tables were only generated by the front end of the original compiler. This front end only coalesces branches into indirect jumps when translating some C `switch` statements using conventional heuristics. Note that the *Original* measurements included filling delay slots for indirect jumps from target blocks specified in jump tables to fairly compare the impact of branch coalescing. The measurements show that a substantial benefit was obtained by conventional translation of multiway selection statements into jump tables. The *Cont* column shows the results when coalescing sequences of only *contiguous* branches using the techniques described in Chapter 4. The *Noncont* column shows the results when coalescing a set of (*contiguous* and *noncontiguous*) branches, that are often separated by blocks of intervening instructions using the techniques described in Chapter 5. These frequency measurements indicate that branch coalescing after code generation can effectively reduce the dynamic number of instructions. Coalescing had a negative impact on performance when performance estimates were overly optimistic or pessimistic, which occurred for *join* and *nroff*.

Table 8.3 contrasts the number of branches executed between the compiler (*None*) that strictly translates a multiway statement into a linear sequence of branches and the compiler (*Orig*) that translates each multiway statement into a linear sequence of branches, a heap tree of branches, or an indirect jump from a table depending on simple heuristics used in the compiler front end. Note that the benefits shown in the column **% Fewer Branch** came from both using an indirect jump and a heap tree of branches as code generation alternatives. Table 8.4 contrasts the number of branches that executed between the compiler (*None*) and the compiler (*Noncont*) that performs the branch coalescing as a

Table 8.2: Dynamic Instruction Frequency Measurements

Program	None	Original	Cont	**Noncont**
awk	13,666,952	–0.294%	–2.145%	**–3.118%**
cb	19,739,127	–12.976%	–20.613%	**–21.204%**
cpp	30,985,306	–37.421%	–37.960%	**–38.538%**
ctags	81,040,455	–0.545%	–10.984%	**–24.160%**
deroff	15,511,056	–0.193%	–1.011%	**–1.153%**
grep	11,810,070	–21.620%	–24.370%	**–24.370%**
hyphen	19,535,372	0.000%	–0.783%	**–2.187%**
join	3,552,801	0.000%	0.102%	**0.325%**
lex	10,052,031	–0.230%	–0.566%	**–0.689%**
nroff	25,118,855	–0.155%	–0.015%	**–0.017%**
pr	78,106,755	0.000%	–7.801%	**–7.760%**
ptx	20,059,920	0.000%	–8.921%	**–10.196%**
sdiff	17,582,760	0.000%	0.022%	**–0.017%**
sed	17,321,920	–6.578%	–6.839%	**–7.600%**
sort	18,921,766	0.000%	–32.862%	**–33.053%**
wc	17,860,086	0.000%	-17.853%	**–27.590%**
yacc	25,658,688	–0.194%	–0.303%	**–0.307%**
average	25,036,387	–4.718%	–10.171%	**–11.861%**

code-improving transformation. The column **Branches/Indirect** represents the average dynamic number of conditional branches that were replaced by an indirect jump from a table. Note that the benefits shown in the column **% Fewer Branch** solely came from coalescing set of branches into an indirect jump.

Table 8.5 shows the proportional benefit of the different techniques used to coalesce branches (*Noncont*) as compared to the *Original* (not the *None*) measurements. *After Code Generation* shows the benefits obtained by performing coalescing in the back end of a compiler as a general improving transformation instead of a code generation decision. These benefits indicate that a compiler back end can exploit more opportunities for branch coalescing and make better coalescing decisions. *Front Padding* includes padding the front of jump tables

Table 8.3: Reduced Number of Dynamic Conditional Branches by Generating Indirect Jumps as a Translation Decision of Multiway Statement (Orig)

Prog	*None*	*Original*	
	Branches	Fewer Branches	% Fewer Branches
awk	2213455	88579	–4.00%
cb	3538146	655680	–18.53%
cpp	6730186	4087213	–60.73%
ctags	17462573	97743	–0.56%
deroff	2722789	9572	–3.50%
grep	2526865	1211778	–47.96%
hyphen	2831171	0	0.00%
join	983936	0	0.00%
lex	1771795	8594	–4.90%
nroff	3654565	14622	–4.00%
pr	12078585	8	0.00%
ptx	3310268	0	0.00%
sdiff	2784468	5	0.00%
sed	3014722	479742	–15.91%
sort	4679991	11	0.00%
wc	3636505	0	0.00%
yacc	4877751	18286	–0.37%
average			–8.78%

to avoid subtracting the lowest value compared. *Avoid Initial Range Check* represents when value range analysis was also used to completely eliminate the initial range check. This technique resulted in a substantial decrease since 2 or 3 instructions were avoided each time it was applied. Also, many more sets of branches were now coalesced since the cost/benefit analysis would indicate that the coalescing transformation was worthwhile, when the initial range check could be avoided. *Efficient Indexing* includes using byte displacements in jump tables. Using byte displacements was possible since relocating code segments quite effectively compressed the target range of indirect jumps. Note that the last three techniques were often applied on coalesced branches not associated

Table 8.4: Reduced Number of Dynamic Conditional Branches by Branch Coalescing (Noncont)

Prog	*None*	*Noncont*		
	Branches	Fewer Branches	% Fewer Branches	Branches/Indirect
awk	2213455	247368	–11.18%	4.57
cb	3538146	1744080	–49.29%	10.71
cpp	6730186	4184501	–62.18%	42.87
ctags	17462573	9612642	–55.05%	9.17
deroff	2722789	114612	–4.21%	4.25
grep	2526865	1352497	–53.52%	9.61
hyphen	2831171	887400	–31.34%	3.07
join	983936	19	0.00%	4.75
lex	1771795	30415	–1.72%	8.39
nroff	3654565	3497	–0.10%	3.36
pr	12078585	4737895	–39.23%	3.51
ptx	3310268	997270	–30.13%	3.81
sdiff	2784468	4408	–0.16%	1.74
sed	3014722	520839	–15.91%	12.18
sort	4679991	3805448	–81.31%	2.95
wc	3636505	2463844	–67.75%	3.39
yacc	4877751	49660	–1.02%	6.14
average			–29.73%	7.95

with multiway selection statements.

Table 8.5: Reducing the Cost of Coalescing

Techniques	Proportional Benefit
After Code Generation	22.61%
Front Padding	8.97%
Avoid Initial Range Check	56.31%
Efficient Indexing	12.11%

8.1.2 Total Cache Work

The branch coalescing impact on caching was a concern since misses from jump table loads could potentially have negative impact on performance. Table 8.6 shows the average effect *Noncont* had on instruction caching, data caching, and total cache work as compared to the *Original* cache measurements. The cache work cycles were calculated using Equation 8.1, where a cache hit and a cache miss are counted as one cycle and ten cycles respectively [26]. Note that it was assumed that d-cache accesses could be performed simultaneously with i-cache accesses.

$$\text{CacheWork} = \text{i\_CacheHits} + 10 * (\text{i\_CacheMisses}) + 9 * (\text{d\_CacheMisses}) \quad (8.1)$$

The i-cache work of *Noncont* was reduced since the number of instructions referenced were diminished as compared to the *Original* measurements. As expected, the d-cache work of *Noncont* was increased since jump table loads after branch coalescing are more frequently performed as compared to the *Original* compiler. The total cache work was decreased since i-cache accesses are more frequent than d-cache accesses.

Table 8.6: Cache Work Improvement with a **Direct-Mapped Cache with 32 Byte Line Size**

CACHE SIZE	*Instruction*	*Data*	**CACHE WORK**
1K	–7.095%	+6.680%	–**5.125%**
2K	–7.220%	+7.162%	–**5.614%**
4K	–4.909%	+5.066%	–**4.288%**
8K	–7.930%	+2.598%	–**7.460%**
16K	–8.231%	+3.995%	–**7.289%**
32K	–7.947%	+4.290%	–**7.328%**

8.1.3 Other Measurements

Some other measurements not given in the tables provide useful information. There were on average about 0.901 more instructions executed between branches after *Noncont* as compared to the *Original* measurements. Thus, the opportunities for scheduling on superscalar and superpipelined machines may be improved. In addition, coalescing only caused a 2.566% code size increase.

8.2 Execution Time Measurements

Execution time measurements were also collected on a SPARCstation-IPC, a SPARCstation-20, and an UltraSPARC-1. The first two machines did not provide any branch target/prediction buffer support. The third machine only provides target/prediction buffer support for branches, but no support for indirect jumps.

The time measurements were collected using the C run-time library function *times()* that uses the unit of time as a tick (1 second = 60 ticks). The execution times were obtained from the sum of reported *user* times of ten executions of each program. Note that these results not only varied significantly during each measurement trial, but also the results seems to be affected by the different versions of operating systems, such as SunOS 4.x.x and SunOS 5.x.x. Thus, the reader should probably not view these execution time measurements as a reliable indicator of performance.

8.2.1 Measurements on SPARCstation-IPC and SPARCstation-20

The measurement results on these two machines are shown in Tables 8.7 and 8.8. There are a couple of reasons why the execution time decrease probably was not as significant as the reduction obtained from the number of instructions executed and total cache work. First, the execution time of an indirect jump operation required about the same time as two conditional branches. The author anticipates that the relative cost of an indirect jump would decrease with target/prediction buffer support for branches and indirect jumps since the load delay for fetching the indirect jump target address could be avoided and fewer mispredictions would occur. Second, Tables 8.7 and 8.8 only show the measurements from the code compiled by our compiler, which did not include the C run-time library code. However, the library code did contribute to the execution time measurements.

Table 8.7: Execution Time Measurements for SPARCstation IPC

Program	**None**	**SPARCstation IPC**			
		Orig	**Change**	*Noncont*	**Change**
awk	2121 ts	2113 ts	–0.38%	2254 ts	5.90%
cb	1442 ts	1364 ts	–5.41%	1320 ts	–8.46%
cpp	1484 ts	1004 ts	–32.35%	1010 ts	–31.94%
ctags	4392 ts	4374 ts	–0.41%	4058 ts	–7.61%
deroff	917 ts	912 ts	–0.55%	911 ts	–0.65%
grep	442 ts	357 ts	–19.23%	340 ts	–23.08%
hyphen	741 ts	737 ts	–0.54%	736 ts	–0.68%
join	296 ts	296 ts	0.00%	303 ts	2.31%
lex	504 ts	503 ts	–0.20%	496 ts	–1.59%
nroff	1097 ts	1100 ts	0.27%	1118 ts	1.88%
pr	2854 ts	2857 ts	0.11%	2702 ts	–5.33%
ptx	3015 ts	3027 ts	0.40%	2962 ts	–1.76%
sdiff	9263 ts	9454 ts	2.01%	9280 ts	0.22%
sed	4670 ts	4449 ts	–4.76%	4403 ts	–5.72%
sort	680 ts	683 ts	0.44%	574 ts	–15.59%
wc	777 ts	778 ts	0.13%	678 ts	–12.74%
yacc	1163 ts	1281 ts	9.21%	1295 ts	10.19%
average			**–3.01%**		**–5.57%**

8.2.2 Measurements on UltraSPARC-1

The same exeuction time measurements were also conducted on a UltraSPARC-1. As shown in Table 8.9, the executables from *Noncont* compiler compared to those from *None* compiler resulted in worse performance (even for the *Origi-*

Table 8.8: Execution Time Measurements for SPARCstation-20

Program	None	SPARCstation-20			
		Orig	**Change**	*Noncont*	**Change**
awk	617 ts	622 ts	0.80%	611 ts	–0.97%
cb	375 ts	363 ts	–3.20%	341 ts	–9.07%
cpp	493 ts	349 ts	–29.21%	372 ts	–29.41%
ctags	1267 ts	1208 ts	–4.66%	1109 ts	–12.47%
deroff	275 ts	274 ts	–0.36%	272 ts	–1.09%
grep	182 ts	162 ts	–10.99%	157 ts	–13.74%
hyphen	289 ts	289 ts	0.00%	282 ts	—2.42%
join	141 ts	142 ts	0.70%	144 ts	2.08%
lex	209 ts	206 ts	–1.44%	201 ts	–3.83%
nroff	381 ts	384 ts	0.78%	385 ts	1.04%
pr	874 ts	878 ts	0.46%	830 ts	–5.03%
ptx	1429 ts	1425 ts	–0.28%	1385 ts	–3.08%
sdiff	7520 ts	7479 ts	–0.55%	7475 ts	–0.60%
sed	1401 ts	1332 ts	–4.93%	1320 ts	–5.78%
sort	259 ts	258 ts	–0.39%	256 ts	–1.16%
wc	252 ts	250 ts	–0.79%	246 ts	–2.38%
yacc	414 ts	436 ts	5.05%	428 ts	3.27%
average			**–2.88%**		**–4.98%**

nal). The author strongly suspects that such disimprovement stems from no comparable target/prediction buffer support for the indirect jumps. In order to properly estimate the execution time impact on this machine by applying the branch coalescing transformation, EASE (Environment for Architectural Study and Experimentation [13]) was extended to be able to simulate branch prediction with BTB support as shown in Figure 7.3.

8.2.3 Branch Prediction Simulation with BTB (Branch Target Buffer)

Indirect jumps from tables are generally considered to cause poorer branch prediction performance. The reason for this view is that indirect jumps typically have higher misprediction rates than conditional branches since an in-

Table 8.9: Execution Time Measurements for Ultra-SPARCstation

Program	**None**	**Ultra-SPARCstation**			
		Orig	**Change**	*Noncont*	**Change**
awk	479 ts	483 ts	0.83%	488 ts	1.84%
cb	261 ts	268 ts	2.61%	266 ts	1.88%
cpp	305 ts	286 ts	–6.23%	279 ts	–8.53%
ctags	936 ts	943 ts	0.74%	1008 ts	7.14%
deroff	199 ts	205 ts	2.93%	200 ts	0.50%
grep	123 ts	120 ts	–2.44%	118 ts	–4.07%
hyphen	206 ts	208 ts	0.96%	225 ts	8.44%
join	104 ts	104 ts	0.00%	106 ts	1.89%
lex	129 ts	135 ts	4.44%	137 ts	5.84%
nroff	239 ts	243 ts	1.65%	242 ts	1.24%
pr	529 ts	534 ts	0.94%	581 ts	8.95%
ptx	1013 ts	1016 ts	0.30%	1020 ts	0.69%
sdiff	6651 ts	6676 ts	0.37%	6660 ts	0.14%
sed	922 ts	900 ts	0.37%	893 ts	–3.15%
sort	178 ts	177 ts	–0.56%	207 ts	14.01%
wc	171 ts	170 ts	–0.59%	208 ts	17.79%
yacc	284 ts	293 ts	3.07%	285 ts	0.35%
average			**0.67%**		**3.23%**

direct jump may have many possible targets. However, the essence of branch coalescing transformation is to replace several conditional branches into an indirect jump. Thus, it was contended that the total number of mispredictions instead of the misprediction rate should be used when trying to measure branch target/prediction buffer performance with and without branch coalescing transformation.

Tables 8.10, 8.11, and 8.12 show the decrease in the number of mispredictions from *Noncont* (branch coalescing) as compared to the *Original* (not the *None*) branch target/prediction buffer measurements. As contended by the author, even though the misprediction ratio went up after performing the branch coalescing transformation, the total number of mispredictions was decreased. Note that both the *Original* and *Noncont* buffer measurements supported prediction for indirect jumps.

8.3 Compile-Time Overhead

Initially, the *compile-time* overhead of branch coalescing was quite excessive. Two improvements were made to increase compile-time efficiency for the branch coalescing transformation. These improvements were decreasing the number of basic blocks used to represent jump tables and avoiding unnecessary attempts to coalesce branches.

8.3.1 Reducing the Number of Basic Blocks

The complexity for both data and control-flow analysis for code improving transformations is proportional to the number of basic blocks. In fact, the author found that most of the compile-time overhead was due to the detrimental effect that additional basic blocks had on subsequent analysis and transformations.

Originally, VPO (Very Portable Optimizer) represented each entry in the jump table as a separate basic block. This representation scheme was a concern to the author since most of the techniques in Chapter 6 to make indirect jumps more efficient were applied at the cost of duplicating jump table entries. In order to avoid excessive generation of basic blocks from those techniques, an alternative scheme has been designed and implemented to compactly represent the control flow for a jump table.

As an illustration, consider the RTLs shown in Figure 6.3, which is the snapshot after eliminating the value range check instructions for the indirect jump by enumerating 256 jump table entries into the jump table. However,

Table 8.10: Branch Misprediction Ratio and Number of Mispredicted Branches with a **Direct-Mapped BTB** with (0,1) **Correlation Predictor**

Entries in BTB	**Branch Misprediction Ratio**			**Percentage Reductions in Mispredicted Branches**
	Orig	*Noncont*	**Difference**	
32	0.1182	0.1365	**0.0183**	**–5.60%**
64	0.1050	0.1152	**0.0102**	**–9.09%**
128	0.0935	0.1042	**0.0107**	**–9.52%**
256	0.0892	0.0988	**0.0096**	**–10.35%**
512	0.0871	0.0964	**0.0094**	**–10.67%**
1024	0.0811	0.0961	**0.0149**	**–4.43%**

Table 8.11: Branch Misprediction Ratio and Number of Mispredicted Branches with a **Direct-Mapped BTB** with (0,2) **Correlation Predictor**

Entries in BTB	**Branch Misprediction Ratio**			**Percentage Reductions in Mispredicted Branches**
	Orig	*Noncont*	**Difference**	
32	0.1118	0.1252	**0.0134**	**–8.11%**
64	0.0971	0.1014	**0.0043**	**–12.28%**
128	0.0848	0.0899	**0.0051**	**–12.81%**
256	0.0804	0.0841	**0.0038**	**–14.11%**
512	0.0779	0.0824	**0.0045**	**–14.10%**
1024	0.0720	0.0817	**0.0097**	**–7.20%**

Table 8.12: Branch Misprediction Ratio and Number of Mispredicted Branches with a **Direct-Mapped BTB** with (2,2) **Correlation Predictor**

Entries in BTB	**Branch Misprediction Ratio**			**Percentage Reductions in Mispredicted Branches**
	Orig	*Noncont*	**Difference**	
32	0.1131	0.1271	**0.0140**	**–8.33%**
64	0.0969	0.1024	**0.0055**	**–12.07%**
128	0.0840	0.0902	**0.0062**	**–12.53%**
256	0.0788	0.0836	**0.0048**	**–13.56%**
512	0.0758	0.0817	**0.0059**	**–13.30%**
1024	0.0695	0.0809	**0.0114**	**–6.15%**

256 blocks for the jump table, as shown in Figure 8.1(a), could be efficiently represented into fewer blocks when consecutive jump table entries that contain the same target address can be grouped into a single basic block. Figure 8.1(b) shows a compact representation of the original control flow. Note that each basic block containing a jump table entry has another field to indicate the repetition count such that the jump table entries can be restored while SPARC assembly code is being produced.

8.3.2 Avoiding Unnecessary Coalescing Attempts

In *vpo*, several loop transformations are iteratively applied until no further improvement (change(es)) can be made, as depicted in Figure 2.2. Coalescing of branches was treated as a transformation for a loop since the transformation typically requires extra registers. The author coalesced the branches from the innermost loop outward after all other transformations for a given loop have been initially attempted. Within such a loop optimization framework, unnecessary branch coalescing analysis could be avoided. For a given loop, if the branch coalescing analysis has been already applied without any transformation, then there is typically no need to re-apply the analysis for the same loop. Most of transformations from the branch coalescing are typically completed during the first pass of a loop optimization process. The author found that other improving transformations rarely provided new opportunities for branch coalescing. Therefore, the branch coalescing transformation was not applied during the second pass of the same loop optimization process.

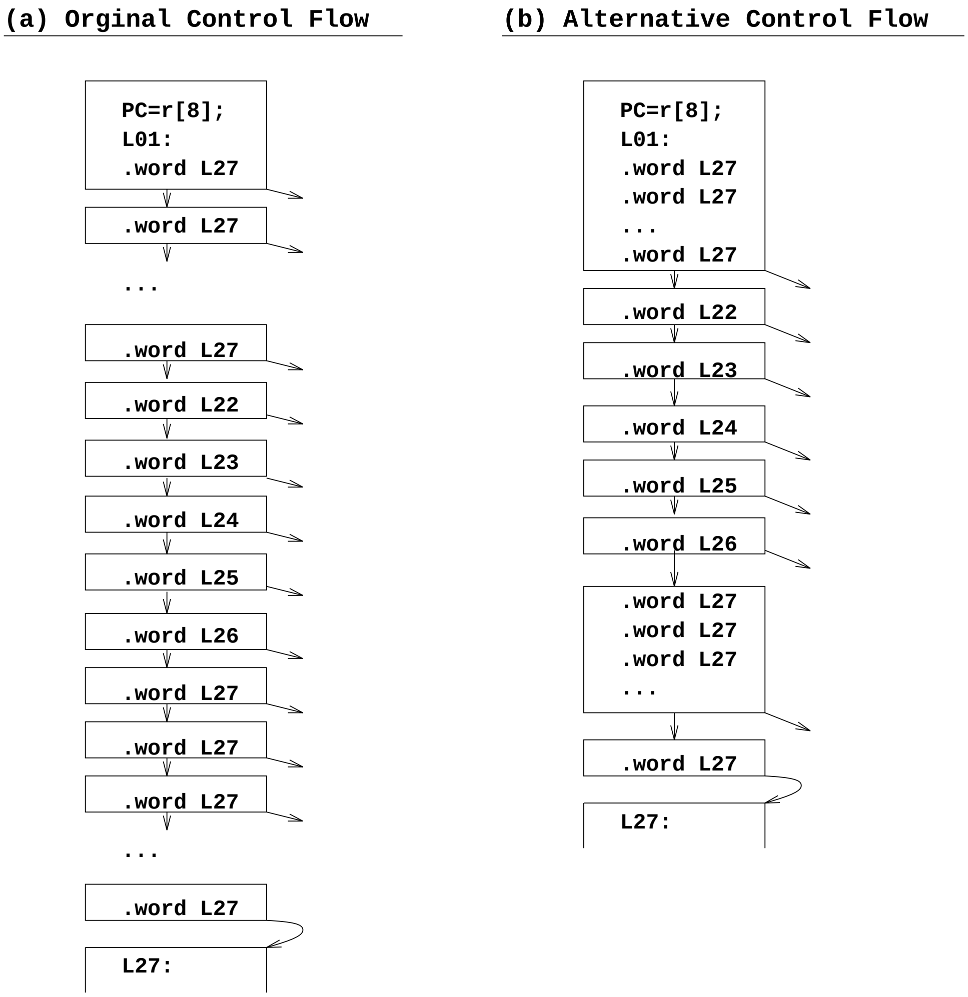


Figure 8.1: Control Flow Representations for Indirect Jump Table shown in Figure 6.3

8.3.3 Compilation Overhead

Compile time measurements were collected on a SPARCstation-20 using the C run-time library function *times()*. The compile times were obtained from the average of the sum of the reported *user* and *system* times of 10 compilations of each benchmark. Table 8.13 compares the results with *None* and *Noncont* (that is, with and without the branch coalescing transformation.) The author suspects that the compilation overhead can be reduced with some additional tuning. Some portion of compilation overhead in system time is due to `I/O` in producing jump table entries when generating SPARC assembly code. This overhead can be avoided when an assembler supports a directive that specifies a repetition factor for consecutive values that are identical (e.g. `.word <value><repetition factor>`).

Table 8.13: Compile Time Measurements

Program	None		Noncont		**Extra Overhead**
	user	*system*	*user*	*system*	
awk	39.40 sec	7.18 sec	76.35 sec	7.88 sec	+80.82%
cb	4.83 sec	0.72 sec	5.48 sec	0.77 sec	+12.61%
cpp	23.02 sec	3.17 sec	36.28 sec	3.40 sec	+51.56%
ctags	9.60 sec	0.72 sec	14.07 sec	0.93 sec	+45.40%
deroff	33.68 sec	1.03 sec	38.17 sec	1.10 sec	+13.11%
grep	4.68 sec	0.67 sec	6.53 sec	0.78 sec	+36.76%
hyphen	1.37 sec	0.60 sec	1.53 sec	0.60 sec	+9.32%
join	3.58 sec	0.62 sec	4.25 sec	0.67 sec	+17.06%
lex	41.40 sec	3.78 sec	49.22 sec	4.08 sec	+17.96%
nroff	43.25 sec	6.13 sec	45.83 sec	6.32 sec	+5.60%
pr	6.03 sec	0.82 sec	6.52 sec	0.85 sec	+7.54%
ptx	6.42 sec	0.78 sec	7.13 sec	0.78 sec	+9.95%
sdiff	8.37 sec	0.78 sec	12.40 sec	0.98 sec	+45.47%
sed	20.52 sec	2.27 sec	24.65 sec	2.45 sec	+18.95%
sort	9.30 sec	0.68 sec	10.38 sec	0.68 sec	+10.85%
wc	0.95 sec	0.50 sec	1.50 sec	0.53 sec	+40.23%
yacc	34.87 sec	2.67 sec	43.47 sec	2.93 sec	+23.62%
average	17.13 sec	1.95 sec	22.57 sec	2.10	**+26.28%**

Chapter 9

FUTURE WORK

There are other areas that could be investigated to provide additional opportunities for coalescing conditional branches. One factor that limited the opportunities for coalescing branches into indirect jumps was not performing interprocedural analysis to more effectively determine value ranges. Often `int` arguments being compared to constants in one function are loaded from memory as a byte in a different function. Interprocedural analysis would allow the first three instructions in Figure 6.1(b) comprising the initial range check to be avoided more frequently.

Profiling could also be used to help determine when coalescing was worthwhile. The author statically estimated the average number of branches that would be executed through a set of related branches. Coalescing can have a negative impact on performance when these estimates are overly optimistic or pessimistic. Profiling would provide more accurate estimates for coalescing decisions. In general, detecting bounded ranges and using an estimated frequency for character values provided good heuristics when making coalescing decisions. This approach has promising implications for conventional branch prediction.

Chapter 10

CONCLUSIONS

This dissertation has described compiler support for effectively exploiting indirect jumps. The general improving transformation presented for coalescing branches after code generation provided benefits that otherwise would not be available.

Two general approaches were designed and implemented to aggressively replace a set of branches into a single indirect jump as opposed to only considering indirect jumps when translating multiway statements. The first approach allows the compiler to detect and coalesce a contiguous sequence of branches into an indirect jump. The second approach provides a more general algorithm that can coalesce a set of potentially noncontiguous conditional branches, which are often separated by blocks of intervening instructions. Thus, better code can be generated by using the second approach instead of the first since a greater number of branches per indirect jump can be coalesced. However, the first approach is relatively simpler to implement and it requires relatively less complex analysis than the second.

Various techniques were developed and implemented to efficiently perform the indirect jump operation by analyzing the context of the given machine instructions. Applying these techniques often resulted in the execution of only two instructions on the SPARC. In order to provide an effective branch coalescing transformation, two cost/benefit analyses were designed and applied by estimating the average number of branches executed for the detected set of coalescent branches. In order to coalesce a set of conditional branches, which are often separated by blocks of intervening instructions, a restructuring algorithm using code duplication was designed and implemented. Furthermore, the original delay slot filling scheme was extended to usefully fill the delay slots of indirect jumps. Thus, a code-improving transformation was designed and implemented in order to essentially provide early resolution of conditional branches that may originally have been some distance from the point where the indirect jump is inserted.

BTBs (Branch Target Buffer) are available to reduce the cost of branches on many machines. The branch coalescing impact on branch mispredictions was a concern to the author. The author's contention was that with comparable target buffer support for indirect jumps, the total number of branch mispredictions should be reduced since several branches are being coalesced into a single indirect jump. To justify the contention, the author accomplished the following tasks. First, the EASE environment [13] was extended to be able to simulate effects on branch mispredictions with BTB support for branches and indirect jumps [15](see page 276). Second, in order to better exploit a BTB for indirect jumps, a compiler analysis technique was implemented to locate the most probable target of the indirect jump immediately after the jump as a fall-through destination. Thus, if an indirect jump is not in the buffer, then no delay is necessary since the next address of the indirect jump is already calculated by the CPU.

Finally, various measurements were collected to demonstrate the benefit of applying the branch coalescing transformation. The additional benefits from coalescing noncontiguous branches were contrasted with the simpler analysis required for only coalescing contiguous branches.

The results showed reductions in the number of instructions executed and branch mispredictions, total cache work, and execution time at the cost of tolerable compile-time overhead.

Appendix A

Optimized SPARC Assembly Code for Loop Overhead

```
! block 1
        .seg    ''data''
        .align  8
        .global _i
_i:
        .word   0
        .seg    ''text''
        .global _main
_main:
.1_after = 96
.1_before = 104
        save    %sp,(-112),%sp
        add     %sp,.1_before,%o0
        call    _gettimeofday,2
        mov     %g0,%o1
        mov     %g0,%l2
        mov     %g0,%l1
        sethi   %hi(10000000),%o0
        or      %o0,%lo(10000000),%o0
        cmp     %g0,%o0
        bge     L40
        mov     %g0,%l0
! block 2
        sethi   %hi(_i),%o2
        mov     %o0,%o4
! block 3
L42:
        and     %l0,3,%o1
! block 4
        add     %l0,1,%l0
! block 5
        cmp     %l0,%o4
        bl      L42
        st      %o1,[%o2 + %lo(_i)]
! block 6
L40:
        add     %sp,.1_after,%o0
        call    _gettimeofday,2
        mov     %g0,%o1
        ld      [%sp + .1_after],%o3
        ld      [%sp + .1_before],%o4
        sub     %o3,%o4,%o3
        st      %o3,[%sp + .1_after]
        ld      [%sp + (.1_after + 4)],%o3
        ld      [%sp + (.1_before + 4)],%o4
        sub     %o3,%o4,%o3
        cmp     %o3,%g0
        bge     L44
```

```
        st      %o3,[%sp + (.1_after + 4)]
! block 7
        sub     %o3,1,%o1
        sethi   %hi(1000000),%o2
        or      %o2,%lo(1000000),%o2
        add     %o1,%o2,%o1
        st      %o1,[%sp + (.1_after + 4)]
! block 8
L44:
        sethi   %hi(L46),%o0
        add     %o0,%lo(L46),%o0
        call    _printf,2
        mov     %l0,%o1
        sethi   %hi(L47),%o2
        add     %o2,%lo(L47),%o0
        call    _printf,2
        mov     %l2,%o1
        sethi   %hi(L48),%o2
        add     %o2,%lo(L48),%o0
        call    _printf,2
        mov     %l1,%o1
        sethi   %hi(L49),%l0
        ld      [%sp + .1_after],%l1
        sethi   %hi(10000),%o2
        or      %o2,%lo(10000),%o1
        call    .div,2
        ld      [%sp + (.1_after + 4)],%o0
        mov     %o0,%o2
        add     %l0,%lo(L49),%o0
        call    _printf,3
        mov     %l1,%o1
! block 9
        ret
        restore
! block 1
        .seg    ``data''
L49:
        .ascii ``The elapsed time: %9ld.%02ld\12\0''
L48:
        .ascii ``The value of l = %d\12\0''
L47:
        .ascii ``The value of k = %d\12\0''
L46:
        .ascii ``The value of j = %d\12\0''
```

Appendix B

Optimized SPARC Assembly Code for Linear Sequence of Branches

```
! block 1
        .seg    ''data''
        .common _i,4,''data''
        .seg    ''text''
        .global _main
_main:
.1_after = 96
.1_before = 104
        save    %sp,(-112),%sp
        add     %sp,.1_before,%o0
        call    _gettimeofday,2
        mov     %g0,%o1
        mov     %g0,%l1
        mov     %g0,%l2
        sethi   %hi(1000000),%o0
        or      %o0,%lo(1000000),%o0
        cmp     %g0,%o0
        bge     L40
        mov     %g0,%l0
! block 2
        sethi   %hi(_i),%o2
        mov     %o0,%o4
        and     %l0,3,%o1
! block 3
L42:
        cmp     %o1,%g0
        bne     L43
        st      %o1,[%o2 + %lo(_i)]
! block 4
        add     %l1,4,%l1
        ba      L39
        mov     4,%l2
! block 5
L43:
        cmp     %o1,1
        bne,a   L45
        cmp     %o1,2
! block 6
        add     %l1,1,%l1
        ba      L39
        mov     1,%l2
! block 7
L45:
        bne,a   L47
        cmp     %o1,3
! block 8
        add     %l1,2,%l1
```

```
        ba      L39
        mov     2,%l2
! block 9
L47:
        bne,a   L41
        add     %l0,1,%l0
! block 10
        add     %l1,3,%l1
        mov     3,%l2
! block 11
! block 12
L39:
        add     %l0,1,%l0
! block 13
L41:
        cmp     %l0,%o4
        bl,a    L42
        and     %l0,3,%o1
! block 14
L40:
        add     %sp,.1_after,%o0
        call    _gettimeofday,2
        mov     %g0,%o1
        ld      [%sp + .1_after],%o3
        ld      [%sp + .1_before],%o4
        sub     %o3,%o4,%o3
        st      %o3,[%sp + .1_after]
        ld      [%sp + (.1_after + 4)],%o3
        ld      [%sp + (.1_before + 4)],%o4
        sub     %o3,%o4,%o3
        cmp     %o3,%g0
        bge     L51
        st      %o3,[%sp + (.1_after + 4)]
! block 15
        sub     %o3,1,%o1
        sethi   %hi(1000000),%o2
        or      %o2,%lo(1000000),%o2
        add     %o1,%o2,%o1
        st      %o1,[%sp + (.1_after + 4)]
! block 16
L51:
        sethi   %hi(L53),%o0
        add     %o0,%lo(L53),%o0
        call    _printf,2
        mov     %l0,%o1
        sethi   %hi(L54),%o2
        add     %o2,%lo(L54),%o0
        call    _printf,2
        mov     %l1,%o1
        sethi   %hi(L55),%o2
        add     %o2,%lo(L55),%o0
        call    _printf,2
        mov     %l2,%o1
        sethi   %hi(L56),%l0
        ld      [%sp + .1_after],%l1
        sethi   %hi(10000),%o2
        or      %o2,%lo(10000),%o1
        call    .div,2
        ld      [%sp + (.1_after + 4)],%o0
        mov     %o0,%o2
        add     %l0,%lo(L56),%o0
        call    _printf,3
        mov     %l1,%o1
```

```
! block 17
        ret
        restore
! block 1
        .seg    ‘‘data’’
L56:
        .ascii ‘‘The elapsed time: %9ld.%02ld\12\0’’
L55:
        .ascii ‘‘The value of l = %d\12\0’’
L54:
        .ascii ‘‘The value of k = %d\12\0’’
L53:
        .ascii ‘‘The value of j = %d\12\0’’
```

Appendix C

Optimized SPARC Assembly Code for Indirect Jump

```
! block 1
        .seg    ''data''
        .common _i,4,''data''
        .seg    ''text''
        .global _main
_main:
.1_after = 96
.1_before = 104
        save    %sp,(-112),%sp
        add     %sp,.1_before,%o0
        call    _gettimeofday,2
        mov     %g0,%o1
        mov     %g0,%l1
        mov     %g0,%l2
        sethi   %hi(1000000),%o0
        or      %o0,%lo(1000000),%o0
        cmp     %g0,%o0
        bge     L40
        mov     %g0,%l0
! block 2
        sethi   %hi(L008),%o4
        or      %o4,%lo(L008),%o4
        sethi   %hi(_i),%g1
        mov     %o0,%o2
        and     %l0,3,%o1
        sethi   %hi(L007),%o5
        or      %o5,%lo(L007),%o5
! block 3
L42:
        st      %o1,[%g1 + %lo(_i)]
        ldsb    [%o1 + %o4],%o0
        jmp     %o0 + %o5
        mov     3,%l2
.seg    ''data''
.align  4
L008:
.byte   L004-L007
! block 4
.byte   L007-L007
! block 5
.byte   L006-L007
! block 6
.byte   L005-L007
.align  4
.seg    ''text''
! block 7
L004:
        add     %l1,4,%l1
        ba      L39
```

```
        mov     4,%l2
! block 8
L007:
        add     %l1,1,%l1
        ba      L39
        mov     1,%l2
! block 9
L006:
        add     %l1,2,%l1
        ba      L39
        mov     2,%l2
! block 10
L005:
        add     %l1,3,%l1
! block 11
! block 12
L39:
        add     %l0,1,%l0
! block 13
        cmp     %l0,%o2
        bl,a    L42
        and     %l0,3,%o1
! block 14
L40:
        add     %sp,.1_after,%o0
        call    _gettimeofday,2
        mov     %g0,%o1
        ld      [%sp + .1_after],%o3
        ld      [%sp + .1_before],%o4
        sub     %o3,%o4,%o3
        st      %o3,[%sp + .1_after]
        ld      [%sp + (.1_after + 4)],%o3
        ld      [%sp + (.1_before + 4)],%o4
        sub     %o3,%o4,%o3
        cmp     %o3,%g0
        bge     L51
        st      %o3,[%sp + (.1_after + 4)]
! block 15
        sub     %o3,1,%o1
        sethi   %hi(1000000),%o2
        or      %o2,%lo(1000000),%o2
        add     %o1,%o2,%o1
        st      %o1,[%sp + (.1_after + 4)]
! block 16
L51:
        sethi   %hi(L53),%o0
        add     %o0,%lo(L53),%o0
        call    _printf,2
        mov     %l0,%o1
        sethi   %hi(L54),%o2
        add     %o2,%lo(L54),%o0
        call    _printf,2
        mov     %l1,%o1
        sethi   %hi(L55),%o2
        add     %o2,%lo(L55),%o0
        call    _printf,2
        mov     %l2,%o1
        sethi   %hi(L56),%l0
        ld      [%sp + .1_after],%l1
        sethi   %hi(10000),%o2
        or      %o2,%lo(10000),%o1
        call    .div,2
        ld      [%sp + (.1_after + 4)],%o0
        mov     %o0,%o2
```

```
        add     %l0,%lo(L56),%o0
        call    _printf,3
        mov     %l1,%o1
! block 17
        ret
        restore
! block 1
        .seg    ``data''
L56:
        .ascii ``The elapsed time: %9ld.%02ld\12\0''
L55:
        .ascii ``The value of l = %d\12\0''
L54:
        .ascii ``The value of k = %d\12\0''
L53:
        .ascii ``The value of j = %d\12\0''
```

References

[1] F. Allen and J. Cocke. *Design and Optimization of Compilers*. Prentice-Hall, Englewood Cliffs, NJ, 1971.

[2] N. Altman and N. Weiderman. Timing variation in dual-loop benchmarks. Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, October 1987.

[3] T. Ball and J.R. Larus. Branch prediction for free. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 300–313, June 1993.

[4] M. E. Benitez and J. W. Davidson. A portable global optimizer and linker. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 329–338, June 1988.

[5] R. L. Bernstein. Producing good code for the case statement. *Software-Practice and Experience*, 15:1021–1024, October 1985.

[6] R. Bodik, R. Gupta, and Mary L. Soffa. Interprocedural conditional branch elimination. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

[7] B. Calder and D. Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 397–408, December 1994.

[8] B. Calder, D. Grunwald, and D. Lindsay. Corpus-based static branch prediction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 79–92, June 1995.

[9] Po-Yung Chang, Eric Hao, and Yale N. Patt. Target prediction for indirect jumps. In *The 24th Annual International Symposium on Computer Architecture*, June 1997.

[10] R. M. Clapp, L. Duchesneau, R. A. Volz, T. N. Mudge, and T. Schultze. Toward real-time performace benchmarks for ADA. *Communications of the ACM*, 29(8):760–778, August 1986.

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

[12] J. W. Davidson and D. B. Whalley. Quick compilers using peephole optimizations. *Software Practice & Experience*, 19(1):195–203, January 1989.

[13] J. W. Davidson and D. B. Whalley. A design environment for addressing architecture and compiler interactions. *Microprocessors and Microsystems*, 15(9):459–472, November 1991.

[14] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the gnu c compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 341–352, June 1992.

[15] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, second edition, 1996.

[16] J. L. Hennessy and N. Mendelsohn. Compilation of the Pascal case statement. *Software-Practice and Experience*, 12:879–882, September 1982.

[17] A. M. Holler. Optimization for a superscalar out-of-order machine. In *Proceedings of the 29th International Symposium on Microarchitecture*, pages 336–348, December 1996.

[18] S. C. Johnson. *A Tour Through the Portable C Compiler*. Unix Programmer's Manual 7th Edition Section 33, January 1979.

[19] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. W. Hwu. Characterizing the impact of predicated execution on branch prediction. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 217–227, December 1994.

[20] F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 56–66, June 1995.

[21] S. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Architectural Support for Programming Languages and Operating Systems*, pages 76–84, September 1992.

[22] J. Patterson. Accurate static branch prediction by value range propagation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 929–942, June 1995.

[23] C. H. Perleberg and A. J. Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, April 1993.

[24] D. N. Pnevmatikatos and G. S. Sohi. Guarded execution and branch prediction in dynamic ILP processors. In *Proceedings of the 21th International Symposium on Computuer Architecture*, pages 120–129, April 1994.

[25] Arthur Sale. The implementation of case statements in Pascal. *Software-Practice and Experience*, 11:929–942, September 1981.

[26] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[27] M. D. Smith, M. S. Lam, and M. A. Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 344–353, May 1990.

[28] D. A. Spuler. Compiler code generation for multiway branch statements as a static search problem. Technical report, Dept. of Computer Science, James Cook University, Townsville, 4811, Australia, 1994.

[29] R. M. Stallman. *Using and Porting GNU CC (version 1.37.1).* Free Software Foundation, Inc., Cambridge, MA, February 1990.

[30] D. W. Wall. Limits of instruction-level parallelism. In *Architectural Support for Programming Languages and Operating Systems*, pages 176–188, April 1991.

Biographical Sketch

Gang-Ryung Uh was born in Seoul, South Korea in 1960. He earned a Bachaelor of Arts degree in Economics from Hankuk University of Foreign Studies in 1987, and a Master of Science degree in Computer Science from Florida State University in 1992. After obtaining the Ph.D degree in Computer science, he plans to continue on various compiler backend optimization studies.