Coalescing Conditional Branches into Efficient Indirect Jumps

Gang-Ryung Uh and David B. Whalley

Computer Science Dept., Florida State University, Tallahassee, FL 32306-4019, USA

Abstract. Indirect jumps from tables are traditionally only generated by compilers as an intermediate code generation decision when translating multiway selection statements. However, making this decision during intermediate code generation poses problems. The research described in this paper resolves these problems by using several types of static analysis as a framework for a code improving transformation that exploits indirect jumps from tables. First, control-flow analysis is performed that provides opportunities for coalescing branches generated from other control statements besides multiway selection statements. Second, the optimizer uses various techniques to reduce the cost of indirect jump operations by statically analyzing the context of the surrounding code. Finally, path and branch prediction analysis is used to provide a more accurate estimation of the benefit of coalescing a detected set of branches into a single indirect jump. The results indicate that the coalescing transformation can be frequently applied with significant reductions in the number of instructions executed and total cache work. This paper shows that static analysis can be used to implement an effective improving transformation for exploiting indirect jumps.

1 Introduction

Indirect jumps from tables can be used to replace sequences of branches comparing the same register or variable to constants. Traditionally, indirect jumps from tables are only generated by compiler front ends when translating multiway selection statements, such as the Pascal `case` or C `switch` statements. Making this decision early during intermediate code generation poses problems. First, it is difficult to determine when a particular method can be effectively used in a machine-independent fashion since an accurate cost can only be known after generating machine instructions. Even if a compiler front end is updated to contain machine-dependent information, the approach used will be affected by the context of the code surrounding the multiway selection statement. Second, many code-improving opportunities may be missed by only considering the translation of multiway selection statements. The authors propose that these problems can be more effectively addressed by coalescing conditional branches into indirect jumps from tables as a general improving transformation after code generation.

There are many opportunities for coalescing branches into indirect jumps from tables when performed as an improving transformation as opposed to an intermediate code generation decision. Consider the following code fragment from

ctags (C tags generator) shown in Figure 1(a). A typical C compiler would generate an indirect jump with a table for the `switch` statement 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 comparisons of the same variable (or register) with constants and be able to coalesce the comparison with zero and the conditional branch into the indirect jump and jump table. Note that one can view this branch as another case for the `switch` statement as shown in Figure 1(b). Other common instances may occur due to programming style. Figure 2(a) shows a code segment from *grep* program that has a series of `if` statements comparing the same variable to different constants. Compilers will 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 Figure 2(b). Thus, the control flow for both code segments can be accomplished with an indirect jump. Use of multiple macros may also result in several consecutive comparisons being performed. Therefore, coalescing of branches as a general improving transformation has the appealing aspect that performance is affected by the program logic (control flow) and not the program style (whether or not multiway selection statements are used).

(a) Original Code

```
for (sp = line; *sp; sp++)
  switch (*sp) {
    ...
  }
```

(b) Equivalent Code as a Switch Statement

```
for (sp = line; ; sp++)
  switch (*sp) {
     case 0: /* exit the loop */
        goto out;
     ...
  }
out:
```

Fig. 1. Code Fragment from *ctags*

(a) Original Code

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

(b) Equivalent Code as a Switch Statement

```
c = *sp++;
switch (c) {
   case 0: goto cerror;
   case '<': ...
   case '>': ...
   case '(': ...
   case ')': ...
   case '1': ...
   ...
   case '9': ...
   default: ...
}
```

Fig. 2. Code Fragment from *grep*

This paper describes a general approach for decreasing the number of conditional branches executed by coalescing branches into indirect jumps from tables. First, a control-flow analysis algorithm is used to detect potential sequences of

branches that can be transformed into a single indirect jump. Test conditions other than ones for equality and inequality can also be coalesced into indirect jumps, including testing if a variable is within a bounded range of constants. Second, value-range propagation analysis is performed to determine the range of values associated with each potential target of the indirect jump. Third, various techniques are used that statically analyze the code surrounding the sequence of branches being coalesced to reduce the cost of performing an indirect jump from a table. Applying these techniques often results in the execution of only two instructions on a SPARC for this operation. Fourth, path and branch prediction analysis is also performed to estimate the benefit of coalescing a set of branches into an indirect jump. The number of instructions executed through each path of the set of branches and the probability of each path being taken is considered versus the cost of performing an indirect jump. Finally, the control flow is transformed from the sequence of branches to an indirect jump when the transformation is deemed beneficial. Results are given indicating that the code improving transformation could be frequently applied and often resulted in significant performance improvements.

2 Related Work

Several authors have suggested heuristics for deciding between different methods of translating multiway selection statements [12, 14]. These methods include a linear search (branch for each case value), binary search, hashing, and indirect jumps from tables. The approach used in this paper 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 paper 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. Loop unrolling has been used to avoid executions of the conditional branch associated with a loop termination condition [6]. Loop unswitching moves a conditional branch with a loop-invariant test condition before the loop and replicates the loop in each of the two paths of the branch [1]. A more general method has recently been developed that avoids conditional branches by code replication [10]. This method determines if there are paths where the result of a conditional branch will be known and replicates code to avoid execution of the branch. Coalescing branches into indirect jumps from tables will avoid the execution of branches that these other techniques could not.

3 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 paper 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 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 3. The algorithm not only will detect a coalescent sequence, but will also attempt to maximize the number of branches to be coalesced.

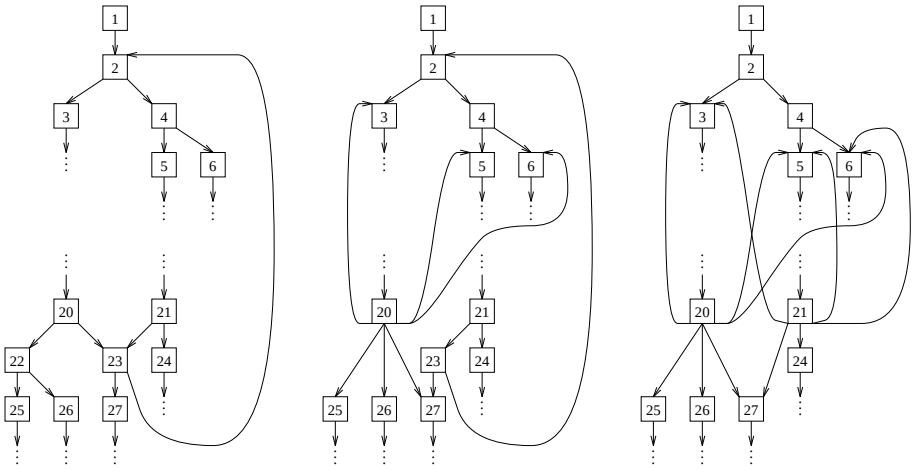
```
PROCEDURE Detect_Sequence()
  FOR each block B DO
    IF (B contains a branch that compares
        variable V with a constant) THEN
      Search_Back(B, V);
      H = Choose_Head();
      Collect_Blocks(H, V);

PROCEDURE Search_Back(B, V)
  mark B as visited;
  IF (B has no instructions
      preceding its compare and branch) THEN
    FOR each immediate predecessor P of B DO
      IF (P has not been visited AND
          P has a branch AND
          P compares V with a constant) THEN
        Search_Back(P, V);

PROCEDURE Choose_Head()
  FOR each block B marked as visited DO
    IF (B has no immediate predecesor
        marked as visited) THEN
      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 AND
        S starts with a compare
          and branch AND
        S compares V with
          a constant) THEN
      Collect_Blocks(S, V);
```

Fig. 3. Algorithm for Detection of Potentially Coalescent Branches



(a) Before Coalescing (b) Coalesced Sequence Starting at Block 20 (c) Coalesced Sequences at Blocks 20 + 21

Fig. 4. Transforming Branches into Indirect Jumps

Figure 4(a) contains an example flow graph that is used to illustrate the algorithm. Assume that the blocks 2, 4, 20, 21, 22, and 23 contain a branch that compares the same condition variable with a constant. Also assume that blocks

2, 4, 22, and 23 contain no other instructions besides a comparison and branch. Consider if 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.

4 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 condition 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 condition 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 5(a) checks if a character could be part of a C identifier. Figure 5(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.

5 Reducing the Cost of Performing an Indirect Jump

Compiler writers have long considered performing an indirect jump from a jump table as a very expensive operation. The tasks associated with such an indirect jump operation include (1) performing an initial range check to ensure that 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.

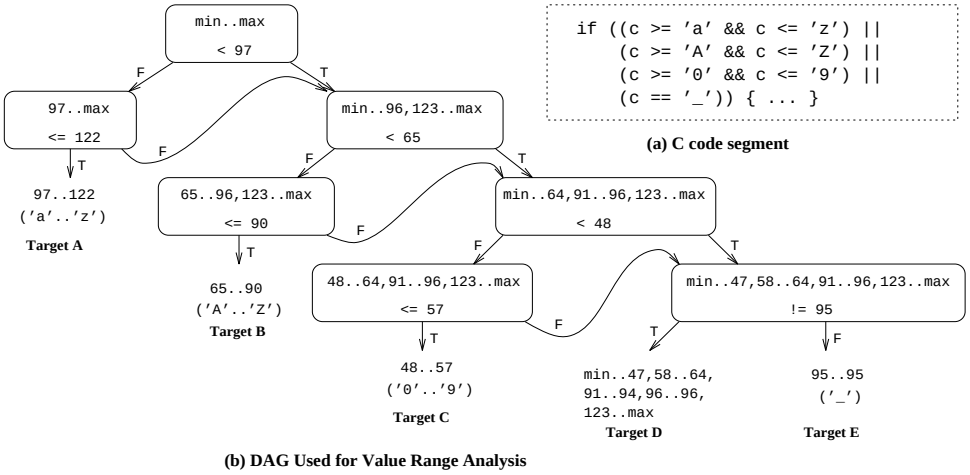


Fig. 5. Example of Checking If a Character Is Part of a C Identifier

Compiler writers have made little attempt at reducing the cost of this operation since indirect jumps from tables occurred relatively infrequently.

The number of instructions required to perform an indirect jump from a jump table can vary depending upon a number of factors. Figure 6 depicts SPARC instructions represented as RTLs that are used to implement an indirect jump (disregarding the instruction in the delay slot of the indirect jump) by the *pcc* [9], *gcc* [15], and *vpo* [4] compilers. Similar instructions are available on most RISC architectures. It would appear that at least 5 pairs of compare and branch instructions 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. However, instructions 4 and 5 are loop invariant and therefore can often be moved out of a loop. The following subsections describe other techniques that will often avoid the execution of instructions 1-3 and 6 as well.

```
r[8]=r[10]-32;              # sub %o2,32,%o0          1. Subtract lowest case value
IC=r[8]?45;                 # cmp %o0,45              2. Compare with the (highest-lowest) case value
PC=ICh,L18;                 # bgu L18                 3. Perform unsigned > branch to ensure value is in range
r[20]=HI[L01];              # sethi %hi(L01),%l4      4. Get high portion of address of jump table
r[20]=r[20]|LO[L01];        # or %l4,%lo(L01),%l4     5. Or in the low portion of the address
r[8]=r[8]<<2;               # sll %o0,2,%o0           6. Left shift value so can index into jump table
r[8]=M[r[8]+r[20]];         # ld [%o0+%l4],%o0        7  Load target destination out of jump table
PC=r[8];                    # jmp %o0                 8. Perform an indirect jump
.seg  "data"
L01:
.word  L27                  # Target address of lowest case value m.
...
.word  L24                  # Target address of highest case value n.
```

Fig. 6. SPARC Instructions Implementing an Indirect Jump from a Jump Table

5.1 Padding the Front of the Table

Instructions 1-3 in Figure 6 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 7, which contains the instructions of Figure 6 with the modifications resulting from padding the front of the jump table. Instruction 2 in Figure 7 uses the highest case value in the comparison when padding is applied. Note also that instructions 4 and 5 in Figure 6 were removed in Figure 7 since it was assumed they are loop invariant for this example.

```
IC=r[8]?77              # 2. Compare with highest case value
PC=ICh0,L18;            # 3. Perform unsigned > branch
r[8]=r[8]<<2;           # 6. Left shift value
r[8]=M[r[8]+r[20]];     # 7. Load target destination
PC=r[8];                # 8. Perform an indirect jump
.seg  "data"
L01:
.word  L18              # Default target for case value 0
...
.word  L18              # Default target for case value m-1
.word  L27              # Target address of lowest case value m
...
.word  L24              # Target address of highest case value n
```

Fig. 7. SPARC Instructions after Padding the Front of the Table

```
r[8]=r[8]<<2;           # 6. Left shift value
r[8]=M[r[8]+r[20]];     # 7. Load target destination
PC=r[8];                # 8. Perform an indirect jump
.seg  "data"
L01:
.word  L18              # Default target for case value 0
...
.word  L18              # Default target for case value m-1
.word  L27              # Target address of lowest case value m
...
.word  L24              # Target address of highest case value n
.word  L18              # Default target for case value n+1
...
.word  L18              # Default target for case value 255
```

Fig. 8. SPARC Instructions with a Bounded Range of Values

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

The initial range check (instructions 1-3 in Figure 6) 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 [14]. Consider a variable loaded from memory as a byte value and compared against characters. The decimal value of the variable can either be from -128..127 or 0..255, depending upon if the value was loaded as a signed or unsigned byte. The instructions associated with this approach is depicted in Figure 8 when an unsigned byte was loaded from memory. 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. Unfortunately, characters in C are often stored in integer variables to compare with EOF. The authors used demand-driven analysis to search backwards from the head of the sequence to determine if the range of case values is bounded by the RTLs representing byte loads or conversions to unsigned or signed character values.

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 replicated when deemed worthwhile to allow coalescing of branches to occur on the path with the bounded range. For example, Figures 9(a) and 9(b) show RTLs and the control flow corresponding to a code segment in *wc*, where block 24 contains the head of a sequence of conditional branches comparing the same register to constants. Blocks 17 to 20 contain RTLs generated from invoking the `getc()` macro. Block 18 contains an RTL 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 inter-procedural analysis was performed. The compiler recursively searches backwards

and finds that blocks 24, 20, and 18 are within a path back to the point where the range of values is bounded. Likewise, the compiler finds that blocks 24, 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 blocks that must be replicated to distinguish the bounded path. Figures 9(c) and 9(d) show the control flow and RTLs and after replication of the blocks 20 and 24 and coalescing of the sequence of branches. Coalescing can occur at the replicated head (block 24') without an initial range check since the range of values is now bounded. Limits were placed on the amount of code allowed to be replicated to prevent large code size increases.

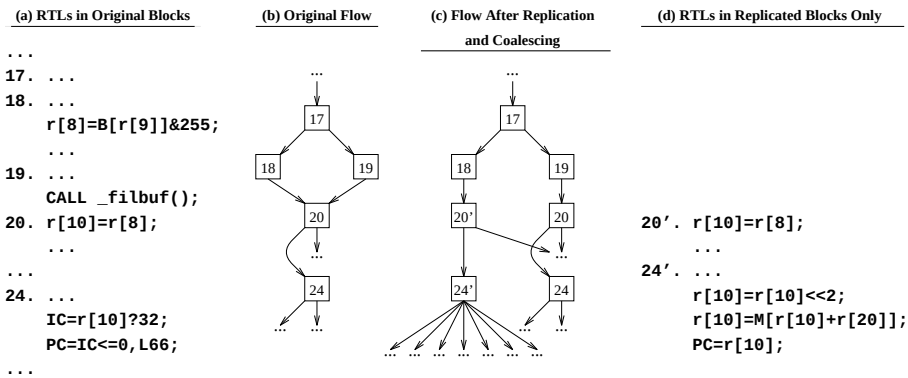


Fig. 9. Using Replication to Distinguish Paths for Coalescing

5.3 Efficiently Indexing into the Jump Table

Instruction 6 in Figure 8 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 10 shows how the code in Figure 8 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.

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 11. 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.

```
                          # 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 displacement out of jump table
PC=r[8]+r[22];      # 8. Perform indirect jump
.seg  "data"
L01:
   .byte  L18-L02     # Default target for case value 0
   ...
   .byte  L18-L02     # Default target for case value m-1
   .byte  L27-L02     # Target address of lowest case value m
   ...
   .byte  L24-L02     # Target address of highest case value n
   .byte  L18-L02     # Default target for case value n+1
   ...
   .byte  L18-L02     # Default target for case value 255
```

Fig. 10. SPARC Instructions with Byte Displacements in the Jump Table

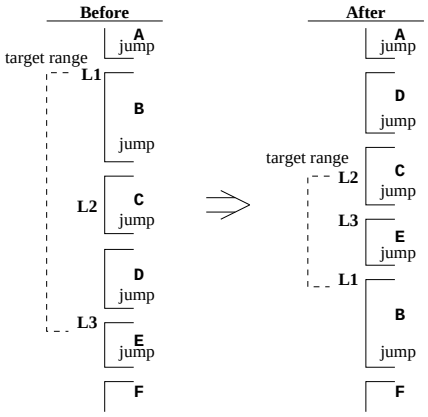


Fig. 11. Relocating Segments of Code

6 Estimating the Benefits of Coalescing a Set of Branches

Before coalescing a set of branches, the compiler attempts to determine if the coalescing was 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 [14]. However, the improving transformation described in this paper coalesces branches that are generated from control statements other than multiway selection statements. Many studies have recently used heuristics [3], value range propagation [11], or empirical data from the execution of other programs [5] 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 authors for the improving transformation in this paper. 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 12 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 Figure 13.

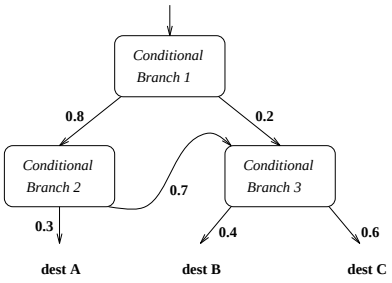


Fig. 12. DAG with Weighted Edges

Unique Path	Propagated Weight	Num of Insts
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

Fig. 13. Estimating the Number of Executed Instructions

7 Transforming the Control Flow

After ensuring that the transformation will be beneficial, 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(a). The sequence of branches starting at block 20 (20, 22, 23, 2, 4) are coalesced into an indirect jump in Figure 4(b). 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(c) 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 may be attempted on block 2 as well. Coalescing is only performed when the estimated benefit outweighs the estimated cost. Note that 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. The dual loop method [2] was used by the authors to obtain a more accurate estimation of the execution time required for a sequence of conditional branches versus indirect jumps. We found that an indirect jump as shown in Figure 10 required about the same execution time as two pairs of compare and branch instructions for a SPARCstation-IPC, SPARCstation-5, SPARCstation-10, and SPARCstation-20. Therefore, the transformation is only applied when it is estimated that more than two coalescent branches in the set will on average be executed.

One issue that affected how often branches could be coalesced was the point at which this improving transformation was performed. Calculating the address of the jump table requires a register to be allocated. If the indirect jump is inside of a loop, then it is desirable to move instructions for this calculation outside of the loop since they are loop invariant. Allocating a register for this purpose will compete with other improving transformations that allocate registers. The improving transformation described in this paper was implemented in the *vpo* compiler with all conventional optimizations being applied [4]. Loop transformations in *vpo* are performed on the innermost loops first. The authors decided to coalesce branches within a loop after all other loop transformations for that loop have been initially attempted. Calculating the base address for byte displacements also requires allocating a register. However, the authors converted complete addresses in jump tables to byte displacements after all other improving transformations (including filling delay slots) have been applied. Performing this conversion later allows the compiler to know exactly the number of instructions between the first and last indirect jump targets. If it is estimated that the target range can be represented in a byte at the point the branches are coalesced, then a register was reserved in the loop for calculating the base address for the byte displacements.

The *vpo* compiler 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 jump 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 from tables occurred much more frequently. The compiler was 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. The path analysis described in the previous section was used to order the indirect jump targets most likely to be taken for selecting the instruction to fill the delay slot.

8 Results

Measurements were collected on the code generated by *vpo* using *ease* (Environment for Architectural Study and Experimentation) [8] on the SPARC architecture. Non-numerical applications tend to have complex conditional control flow. Table 1 shows the number of instructions executed from several common Unix utilities. The *None* column contains the number of instructions executed, which was obtained by modifying the C front end, called *vpcc* [7], 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 original *vpcc* used with *vpo* [4]. This front end only coalesces branches into indirect jumps when translating some C `switch` statements using the same heuristics as *pcc*. 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 *Coalescing* column shows the results when coalescing of branches is performed using the techniques described in this paper. These frequency measurements indicate that performing indirect jumps from tables 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*, *nroff*, and *sdiff*.

Table 2 shows the proportional benefit of the different techniques used to coalesce branches 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 to avoid subtracting the lowest value compared. This technique could be applied frequently since most coalescing of branches involved comparisons with character constants, which have values that are nonnegative and close to zero. *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. *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 with multiway selection statements.

Table 1. Dynamic Instruction Frequency Measurements

Program	Description	None	Original	Coalescing
awk	pattern language	13,666,952	-0.294%	-2.145%
cb	C program beautifier	19,739,127	-12.976%	-20.613%
cpp	C preprocessor	30,985,306	-37.421%	-37.960%
ctags	C tags generator	74,316,425	-0.536%	-10.974%
deroff	remove nroff cmd lines	15,511,507	-0.195%	-1.028%
grep	pattern search	11,810,070	-21.620%	-24.370%
hyphen	lists hyphenated words	19,535,372	0.000%	-0.783%
join	relational join on files	3,552,801	0.000%	0.102%
lex	scanner generator	10,052,031	-0.230%	-0.566%
nroff	document formatter	25,118,855	-0.155%	-0.015%
pr	prepare for printing files	78,016,755	0.000%	-7.801%
ptx	generate permuted index	22,679,653	0.000%	-7.891%
sdiff	side-by-side file diffs	17,582,760	0.000%	0.022%
sed	stream editor	17,872,507	-6.375%	-6.629%
sort	sort or merge files	18,921,766	0.000%	-31.289%
wc	word counter	17,860,086	0.000%	-17.853%
yacc	parser generator	25,658,688	-0.194%	-0.303%
average		24,880,627	-4.706%	-10.006%

Table 2. Reducing the Cost of Coalescing

Techniques	Proportional Benefit
After Code Generation	9.303%
Front Padding	32.760%
Avoid Initial Range Check	48.084%
Efficient Indexing	9.853%

Table 3. Cache Work Measurements

Cache Size	Instruction	Data	Total
1K	-5.761%	8.438%	-3.746%
2K	-7.591%	6.673%	-6.172%
4K	-5.988%	3.077%	-5.065%
8K	-5.877%	1.310%	-5.611%
16K	-5.997%	2.230%	-5.448%
32K	-5.587%	2.891%	-5.554%

The impact of branch coalescing on caching was a concern since misses from jump tables loads could potentially have a negative impact on performance. Table 3 shows the average effect coalescing has on instruction caching, data caching, and total cache work as compared to the *Original* cache measurements. Each cache configuration used was direct-mapped with a 32 byte line size. The cache work cycles were calculated by counting a cache hit as one cycle and a cache miss as ten [13]. The i-cache work was reduced since the number of instructions referenced were diminished. For high hit ratios this reduction is close to the decrease in instructions executed from the *Original* measurements. As expected, the d-cache work was increased due to the jump table loads performed. The total cache work was obtained from the sum of the cycles associated with i-cache hits, i-cache misses, and d-cache miss penalties. It was assumed that d-cache accesses could be performed simultaneously with i-cache accesses. The total cache work was decreased since i-cache accesses are more frequent than d-cache accesses.

Some other measurements not given in the tables provide useful information. When all the techniques are used to reduce the cost of coalescing branches, 24.269% of the executed branches were avoided by coalescing as a general improving transformation. There was one indirect jump executed for every 8.579 branches avoided. There were on average 10.344 branches coalesced into each jump table.

9 Future Work

There are several areas that could be investigated to provide additional opportunities for coalescing conditional branches. Often there are paths between conditional branches that compare the same variable to constants without the variable being updated. These noncontiguous branches could be coalesced and the other instructions that were to be executed between the branches can be replicated when necessary. By resolving many of these conditional branches early, many basic blocks can be merged and result in greater opportunities for instruction-level parallelism.

Another factor that limited 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 three instructions comprising the initial range check to be avoided more frequently.

Profiling could also be used to help determine when coalescing was worthwhile. The authors statically estimated the average number of branches that would be executed through the DAG. 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, which the authors are currently exploring.

Finally, a more detailed study is needed about the effect branch coalescing and related architectural enhancements have on pipelining. For instance, the stalls from mispredictions for a branch target buffer could be less than or greater than stalls from mispredictions associated with the sequence of branches that were coalesced. The authors suspect that the total number of branch target buffer mispredictions should decrease after branch coalescing since fewer transfers of control will be encountered.

10 Conclusions

This paper described an approach for coalescing conditional branches into indirect jumps. Static analysis techniques were used to perform this improving transformation more effectively. Control-flow analysis was used to detect sequences of branches that can be coalesced together and to select a head branch for the location of the indirect jump. Coalescing branches after code generation as a general improving transformation provided additional opportunities that would otherwise not be available. Methods for reducing the cost of performing indirect jumps from tables were investigated and shown to be effective. Value-range

analysis was used to avoid the initial range check when possible. Target-range analysis was used to compress the target range, which avoided the execution of a shift instruction and reduced the size of the jump table. Path and branch prediction analysis was performed to predict the average number of instructions executed through the set of branches to be coalesced and obtain more accurate estimation of the benefit of coalescing. The results indicate that indirect jumps from tables have been underutilized in the past since reductions in both the number of instructions executed and cache work can be obtained.

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. 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.
6. 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.
7. J.W. Davidson and D.B. Whalley. Quick compilers using peephole optimizations. *Software Practice & Experience*, 19(1):195–203, January 1989.
8. 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.
9. S.C. Johnson. *A Tour Through the Portable C Compiler.* Unix Programmer's Manual 7th Edition Section 33, January 1979.
10. 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.
11. 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.
12. Arthur Sale. The implementation of case statements in pascal. *Software-Practice and Experience*, 11:929–942, September 1981.
13. A.J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.
14. 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.
15. R.M Stallman. *Using and Porting GNU CC (version 1.37.1).* Free Software Foundation, Inc., Cambridge, MA, February 1990.