Branch Elimination by Condition Merging

WILLIAM KREAHLING∗, DAVID WHALLEY∗, MARK BAILEY∗†,
XIN YUAN∗, GANG-RYUNG UH‡, AND ROBERT VAN ENGELEN∗
∗Computer Science Department, Florida State University, Tallahassee, FL 32306-4530; e-mail: whalley@cs.fsu.edu
†Computer Science Department, Hamilton College, Clinton, NY 13323; e-mail: mbailey@hamilton.edu
‡Agere Systems, 1247. S. Cedar Crest Blvd., Allentown, PA 18103-6209; e-mail: uh@agere.com

ABSTRACT

Conditional branches are expensive. Branches require a significant percentage of the execution cycles since they occur frequently and can cause pipeline stalls. In addition, branches result in forks in the control flow, which can prevent other code-improving transformations from being applied. In this paper we describe profile-based techniques for replacing the execution of a set of two or more branches with a single branch on a conventional scalar processor. First, we gather profile information to detect the frequently executed paths in a program. Second, we detect sets of conditions in frequently executed paths that can be merged together into a single condition. Third, we estimate the benefit of merging each set of conditions. Finally, we restructure the control flow to merge the sets that are deemed beneficial. The results show that eliminating branches by merging conditions can significantly reduce the number of conditional branches performed in non-numerical applications.

1. INTRODUCTION

Conditional branches occur frequently in programs, particularly in non-numerical applications. Branches are an impediment to improving performance since they consume a significant percentage of the execution cycles, cause pipeline stalls when mispredicted, and can inhibit the application of other code-improving transformations. Techniques to reduce the number of executed branches or remove branches in the control flow have the potential for significantly improving performance.

Sometimes a set of conditions can be merged together. Consider Figure 1(a), which shows conditions being tested in basic blocks 1, 3, and 5. The wider transitions between blocks shown in figures in this paper represent the more frequently executed path, which occurs in Figure 1 when conditions *a*, *b*, and *c* are all satisfied. Figure 1(b) depicts the three conditions being merged together. If the merged condition is true, then the original conditions need not be tested. Note merging conditions results in the elimination of both comparison and branch instructions.[1] The elimination of the forks in the control flow between blocks 2, 4, and 6 may enable additional code-improving transformations to be performed. If the merged condition is not satisfied, then the original conditions are tested. Figure 1(c) shows that sometimes branches become redundant after merging conditions. In this case, condition *c* must be false if *(a && b && c)* is false and *(a && b)* is true. Thus, the branch in block 5 can be replaced by an unconditional transition to block 9. We call this the *breakeven* path since the same number of branches will be executed.

[1] Blocks 2 and 3 could have been represented as a single block. Throughout the paper we represent basic blocks containing a condition as having no other instructions besides a comparison and a conditional branch so the examples may be more easily illustrated.

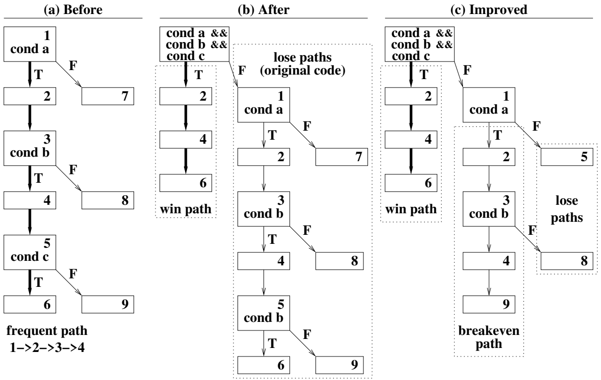

Figure 1: Merging Two Conditions

In this paper we describe techniques to replace the execution of a set of two or more branches with a single branch. Figure 2 presents an overview of the compilation process for merging conditions. A first compilation pass produces an executable file that is instrumented to collect path profiling information. A second compilation pass uses the profile data to merge conditions in the frequently executed paths. The C front end used in this research is *lcc* [FrH95] and the backend used is *vpo* [BeD88] targeted to the SPARC architecture.

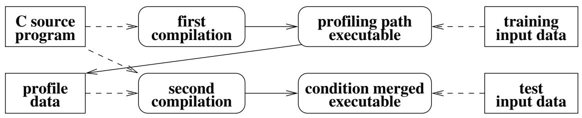

Figure 2: Compilation Process for Merging Conditions

2. RELATED WORK

There have been numerous techniques that have been used to reduce the number of conditional branches executed. Loop unrolling has long been used to avoid executions of the conditional branch associated with a loop termination condition [DoH79]. Loop unswitching moves a conditional branch with a loop-invariant condition before the loop and duplicates the loop in each of the two destinations of the branch [AlC71]. Superoptimizers have been used to find a bounded sequence of instructions that have the same effect as a conditional branch [GrK92]. Conditional branches have been avoided by using static analysis and code duplication [MuW95, BGS97]. Conditional branches have been coalesced together into an indirect jump from a jump table [UhW97]. Sequences of branches have been reordered to allow the sequence to be exited earlier, which reduces the number of branches executed [YUW98, YUW02]. Finally, there has recent work on eliminating branches using ILP architectural features. If conversion uses predicated execution to eliminate branches by squashing the result of an instruction when a predicate is not satisfied [HeP96]. A technique called control CPR has been used to merge branches in frequent paths by performing comparisons that set predicate bits in parallel and testing multiple predicates in a single bypass branch [ScK95, SMJ99]. Control CPR not only reduces the number of executed branches, but also enables other code-improving transformations. Among the techniques mentioned, control CPR is the most similar to the work that we describe in this paper since both approaches merge conditions by using path-based profile data. However, our approach differs in several aspects, including that it can be used on a conventional scalar processor without ILP architectural support.

3. MERGING CONDITIONS THAT INVOLVE A SINGLE VARIABLE

In this section, we describe techniques to merge a set of conditions where each branch compares the same variable to invariant values (e.g. constants). In each case, the variable's value must not be unpredictably updated between the branches in the path.

3.1. Eliminating Logically Redundant Branches

Sometimes the conditions associated with two or more branches are logically correlated. In other words, one branch result (taken or fall through) may imply the result of another. Consider Figure 3(a). If the condition for the branch in block 1 is satisfied, then `v > 5`. If `v` is not affected between the execution of the two branches, then `v > 2` is guaranteed to be satisfied and the branch in block 3 can be deleted, as shown in Figure 3(b). Other techniques using static analysis and code duplication could also eliminate the branch in block 3 [MuW95, BGS97].

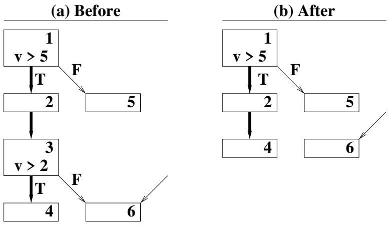

Figure 3: Merging Logically Redundant Conditions

A path-based approach can be used to merge conditions that could not be merged using static analysis alone. Consider the flow graph in Figure 4(a). Satisfying the condition in block 1 will not guarantee the result of the branch in block 3. However, when the condition of the branch in block 3 is satisfied, then the condition in block 1 is guaranteed to be also satisfied. If the path 1→2→3→4 is the frequent path, then the second branch testing the condition `v > 5` can be tested first, as shown in Figure 4(b). When the path 0→1→2→3 is taken, the condition of the branch in block 3 is guaranteed to be false since the same condition has already been shown to be false in block 0. The restructured flow graph in Figure 4(c) depicts how the code can be further improved by eliminating the branch in block 3 since the test now is redundant. Performing the code-improving transformation results in one less branch executed when path 0→2→4 is taken, the same number of branches executed when the path 0→1→2→6 is taken, and one additional branch executed when the path 0→1→5 is taken. The actual improvement would depend on the frequency that each path is taken. However, a benefit can be obtained when the *win* path (0→2→4), which reduces the number of branches, is taken more frequently than the *lose* path (0→1→5), which increases the number of branches. Thus, improvements may be obtained even when the *break-even* path (0→1→2→6) is the most frequently executed path.

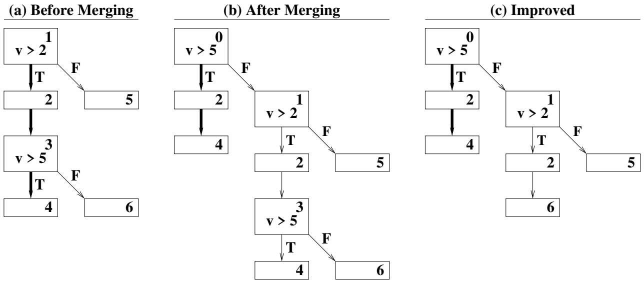

Figure 4: Merging Other Logically Redundant Conditions

3.2. Merging Not Equal Tests Using Range Checks

Sometimes a single variable is checked to determine if it is equal to one of a set of constants. For example, Figure 5(a) shows the variable `c` being compared to three different character constants. Figure 5(b) depicts the flow graph representing these tests. It is often the case that a variable involved in such tests is not equal to any of the constants [YUW98, YUW02]. Profile data is collected to determine not only the paths that are frequent, but also the reason that a particular transition was taken. For instance, when a variable is involved in a test to see if it is equal or not equal to a constant and the not equal transition is taken, then profile data is collected to determine if the variable was greater than or less than the constant. The compiler uses this profile information during the second compilation pass to determine the most likely range of values that a particular variable may have when it is not equal to any of the specified constants. If this range is greater than or less than all of the specified constants, then the set of branches can be replaced with a single branch. Figure 5(c) shows that the three branches testing the variable `c` can be bypassed by checking if `c` is larger than the largest specified constant. Note that when the merged condition is not satisfied, the original set of branches must be tested since the variable could still be equal to any of the specified constants. By performing condition merging on frequent paths, we can merge conditions separated by other intervening branches, which is not possible using a non-path based approach [YUW98, YUW02].

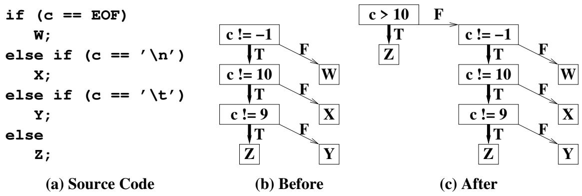

Figure 5: Merging Not Equal Tests Using Range Checks

3.3. Merging Bit Tests

Sometimes different bits in a single variable are tested to see if they are clear or set. Figure 6(a) shows two different bits in the same variable being tested to see if they are clear. Given that the profile data indicates that one combination of bits is very likely, then that combination of bits can be tested in a single comparison, as depicted in Figure 6(b). When the merged condition is not satisfied, both bits cannot be clear. If we reach block 1 from the block containing the merged condition, then we know that satisfying both conditions (case 4 in Table 1) cannot occur since the merged condition failed. If we reach block 2 from block 1, then we know that the first condition is true. Thus, the second condition must be false (case 3 in Table 1 is the only alternative) and the branch in block 3 can be eliminated.

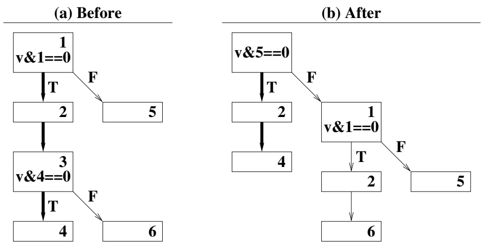

Figure 6: Merging Bit Tests

case	v&1 == 0	v&4 == 0	v&5 == 0
1	F	F	F
2	F	T	F
3	T	F	F
4	T	T	T

Table 1: Truth Table for the Conditions in Figure 6

Table 2 shows different sequences of SPARC instructions represented as RTLs (register transfer lists) that can be used to test a set of bits. The first portion of the table shows a general sequence of three instructions that can be used to test if a set of bits has a specific combination of bits set. `r[v]` is a register containing the value of the variable. `r[t]` is a temporary. *Mask* is a constant that indicates the set of bits to be tested. *Desired_value* is the value of the bits that will result in the frequent path being taken. The branch is taken when the set of bits in the variable does not have the desired value. The second portion shows that the sequence can be reduced to two instructions when the desired value is to have all of the bits clear. All integer ALU operations requiring a single cycle on the SPARC also have the option of setting the integer condition codes, which results in a comparison to zero. The third portion depicts the sequence of instructions we used when the desired value has all of the specified bits in the mask set. The SPARC has ANDNOT and ORNOT instructions that perform a bitwise NOT of the second operand before performing the logical operation. If the variable has all of the desired bits set, then the bitwise NOT will cause all of the desired bits to be clear. Hence, the ANDNOT operation allows a comparison with zero. The first instruction assigning the mask is loop invariant and the cost can be reduced by performing loop-invariant code motion if the branch is inside a loop and a register is available to hold the value of the mask.

General Bit Test	
`r[t]=r[v]&`*mask*`;`	# bitwise AND
`IC=r[t]?`*desired_value*`;`	# comparison
`PC=IC!=0,`*off_trace*`;`	# branch
Testing If Multiple Bits Are Clear	
`IC=(r[v]&`*mask*`)?0;`	# bitwise AND comparison
`PC=IC!=0,`*off_trace*`;`	# branch
Testing If Multiple Bits Are Set	
`r[t]=`*mask*`;`	# loop-invariant assignment
`IC=(r[t]&~r[v])?0;`	# bitwise ANDNOT comparison
`PC=IC!=0,`*off_trace*`;`	# branch

Table 2: SPARC Code Generation Strategies for Merged Bit Tests

One may question how effective merging bit tests will be since they are not used frequently in the source code of applications. We increase the frequency of bit tests by reducing integer global and local scalar variables to bits within a global or local flags variable when such a scalar is only assigned constant values and is only dereferenced for == 0 and != 0 tests. In effect, such variables are used as a boolean, which requires only a single bit to be represented.

We accomplish the reduction of integer scalars to bits in a flag variable by examining and updating the *lcc* intermediate code files comprising the program. The assignment of nonzero constants to such a variable is replaced by a bitwise OR operation that sets a specific bit of a flag variable. The assignment of zero to such a variable is replaced by a bitwise AND operation that clears a specific bit of a flag variable. The == 0 and != 0 tests of the scalar variables are replaced by bitwise AND tests of the appropriate bit of the flag variable.

There are several advantages to reducing integer scalar variables to bits within a flag variable besides being able to merge conditions performing bit tests. Consider the SPARC instructions shown in Figure 7(a). Multiple assignments that set specific bits are merged by assigning the bitwise OR of the constants, as shown in Figure 7(b). Likewise, Figure 7(c) depicts that multiple assignments that clear specific bits are also merged. Redundant loads of the same flag variable are eliminated, as illustrated in Figure 7(d). Finally, Figure 7(e) shows that redundant stores are also eliminated. Note that the elimination of redundant loads and stores would not be possible if separate integer scalar variables were used. Performance conscious programmers often employ such techniques by hand, which is a tedious and error prone task. We are not aware of any prior work that automates this approach.

```
r[5] = M[flag];
r[5] = r[5] | 4;
r[5] = r[5] | 16;
M[flag] = r[5];
r[5] = M[flag];
r[5] = r[5] & ~2;
r[5] = r[5] & ~8;
M[flag] = r[5];
```

(a) Original Instructions

```
r[5] = M[flag];
r[5] = r[5] | 20;
M[flag] = r[5];
r[5] = M[flag];
r[5] = r[5] & ~2;
r[5] = r[5] & ~8;
M[flag] = r[5];
```

(b) After Merging Assignments Setting Bits

```
r[5] = M[flag];
r[5] = r[5] | 20;
M[flag] = r[5];
r[5] = M[flag];
r[5] = r[5] & ~10;
M[flag] = r[5];
```

(c) After Merging Assignments Clearing Bits

```
r[5] = M[flag];
r[5] = r[5] | 20;
M[flag] = r[5];
r[5] = r[5] & ~10;
M[flag] = r[5];
```

(d) After Eliminating Redundant Loads

```
r[5] = M[flag];
r[5] = r[5] | 20;
r[5] = r[5] & ~10;
M[flag] = r[5];
```

(e) After Eliminating Redundant Stores

Figure 7: Other Benefits from Reducing Scalars to Single Bits

4. MERGING CONDITIONS THAT INVOLVE MULTIPLE VARIABLES

The previous section described techniques for merging sets of conditions that compare a single variable to invariant values. In this section we will describe techniques for merging sets of conditions that involve multiple variables.

4.1. Merging into a Single Equivalent Condition

Sometimes logical operations can be used to efficiently merge conditions that compare multiple variables to 0 or -1 into a single equivalent condition. Consider the flow graph shown in Figure 8(a). The frequent path checks if the two variables are equal to zero. Figure 8(b) depicts the two conditions being merged together using a bitwise OR operation. Only when both variables are equal to zero will the result of the OR operation be zero. Again, if the merged condition is not satisfied, then testing the condition `v2 == 0` is unnecessary since it cannot be true.

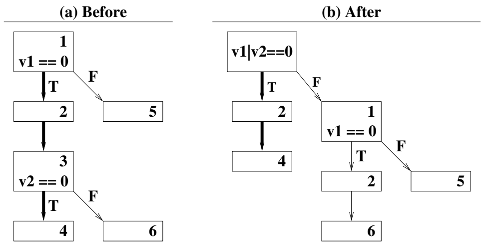

Figure 8: Merging Conditions That Test If Different Variables Are Equal to Zero

Unlike the methods presented in the previous section, the number of instructions required to merge conditions involving multiple variables is proportional to the number of different variables being compared. Figure 9 shows the general code generation strategy we used for merging such a set of conditions. `r[1]..r[n]` represent the registers containing the values of n different variables. `r[t]` represents a register containing the temporary results. When merging conditions comparing n multiple variables, n-1 logical operations and a single branch replace $2n$ instructions (n pairs of comparisons and branches).

```
r[t] = r[1] | r[2];
r[t] = r[t] | r[3];
...
IC = (r[t] | r[n]) ? 0;
PC = IC != 0, <off-trace target>;
```

Figure 9: Code Generation for the Merged Condition That Checks If n Variables Are Equal to Zero

Table 3 shows how sets of conditions comparing multiple variables (*v1..vn*) to 0 and -1 can be merged into a single condition. Rule 1 has been illustrated in Figures 8 and 9. Rule 2 uses the SPARC ORNOT instruction to perform a bitwise NOT on an operand before performing a bitwise OR operation. A word containing the value -1 has all of its bits set in a two's complement representation. Thus, if the operand was a -1, then the result of the bitwise NOT would be 0 and at that point the first rule can be used. The merged condition in rule 3 performs a bitwise AND operation on the variables. A negative value in a two's complement representation has its most significant bit set. Only if the most significant bit is set in all of the variables will the most significant bit be set in the result of the bitwise AND operation. If the most significant bit is set in the result, then the result value will be negative. The merged condition in rule 4 performs a bitwise OR operation on the variables. A nonnegative value in a two's complement representation has its most significant bit clear. Only if the most significant bit is clear in all the variables will the most significant bit be clear in the result of the bitwise OR operation. The last two rules perform a bitwise NOT on an operand, which flips the most significant bit along with the other bits in the value. This allows < and >= tests to be merged together.

<table>
<tr><th>Rule</th><th>Original Conditions</th><th>Merged Condition</th></tr>
<tr><td>1</td><td>v1== 0 &&..&& vn== 0</td><td>(v1|..| vn)==0</td></tr>
<tr><td>2</td><td>v1== 0 &&..&& vn==-1</td><td>(v1|..|~vn)==0</td></tr>
<tr><td>3</td><td>v1 < 0 &&..&& vn < 0</td><td>(v1&..& vn)< 0</td></tr>
<tr><td>4</td><td>v1>= 0 &&..&& vn>= 0</td><td>(v1|..| vn)>=0</td></tr>
<tr><td>5</td><td>v1 < 0 &&..&& vn>= 0</td><td>(v1&..&~vn)< 0</td></tr>
<tr><td>6</td><td>v1>= 0 &&..&& vn < 0</td><td>(v1|..|~vn)>=0</td></tr>
</table>

Table 3: Rules for Merging Conditions Comparing Multiple Variables into a Single Equivalent Condition

Notice that > 0 and <= 0 tests are not listed in the table. A > 0 test would have to determine that both the most significant bit is clear and that one or more of the other bits are set. These types of tests cannot be efficiently performed using a single logical operation on a conventional scalar processor.

Additional opportunities for merging conditions are available when conditions across loop boundaries are considered. It would appear that in Figure 10(a) there is no opportunity for merging conditions. However, Figure 10(b) depicts that after loop unrolling there are multiple branches that use the same relational operator. Furthermore, the block containing the test of `a[i+1]` postdominates the block containing the test of `a[i]`. Thus, loading the value of `a[i+1]` early will never introduce a new exception since the load of `a[i+1]` will be performed anyway.

(a) Original Code

```
for (i = 0; i < 10000;
    i++)
   if (a[i] < 0)
      x;
```

(b) After Loop Unrolling

```
for (i = 0; i < 10000; i += 2) {
   if (a[i] < 0)
      x;
   if (a[i+1] < 0)
      x;
}
```

Figure 10: Increasing Merging Opportunities by Unrolling Loops

4.2. Merging into a Sufficient Condition

Sometimes logical operations can be used to efficiently merge conditions that compare multiple variables into a single sufficient condition. In other words, the success of the merged condition will imply the success of the original conditions. However, the failure of the merged condition does not imply the original conditions were false.

There were no ≠ 0 tests listed in Table 3. Yet tests to determine if a variable is not equal to zero occur frequently in programs. We can determine if two or more variables are guaranteed to be not equal to zero by performing a bitwise AND operation on the variables and checking if the result does not equal to zero, as shown in rule 7 of Table 4. Note that failure of the merged condition does not imply that the variables are all not equal to zero.

<table>
<tr><th>Rule</th><th>Original Conditions</th><th>Merged Condition</th></tr>
<tr><td>7</td><td>v1≠ 0 &&..&& vn≠ 0</td><td>(v1&..&vn) ≠0</td></tr>
<tr><td>8</td><td>v1≠c1 &&..&& vn≠cn</td><td>(v1&..&vn) &~ (c1|..|cn)≠0</td></tr>
<tr><td>9</td><td>v1≠c1 &&..&& vn≠cn</td><td>~(v1|..|vn) & (c1&..&cn)≠0</td></tr>
<tr><td>10</td><td>v1<c1 &&..&& vn<cn</td><td>(v1|..|vn) u< min(c1,..,cn)</td></tr>
</table>

Table 4: Rules for Merging Conditions Comparing Multiple Variables into a Sufficient Condition

One may ask how often can such conditions be successfully merged in practice. Consider the code segment:

```
if (p1 != NULL && p2 != NULL)
```

where two pointer variables, `p1` and `p2`, are tested to see if they are both not null. In most applications, a pointer variable is only null in an exceptional case (e.g. end of a linked list). It is extremely likely that two non-null pointer values will have one or more corresponding bits both set due to address locality. Figure 11 shows how two or more conditions checking if multiple variables are not equal to zero can be merged. Note that if the merged condition is not satisfied, then the entire original set of branches still needs to be tested.

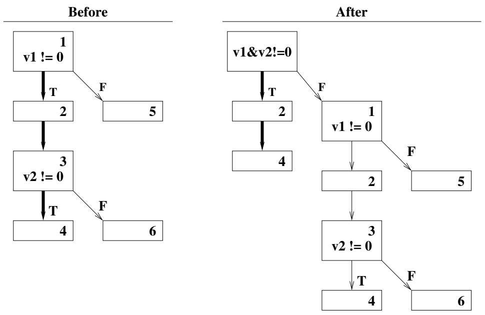

Figure 11: Merging Conditions That Check If Different Variables Are Not Equal to Zero

We are also able to merge conditions that check if multiple variables are all not equal to a specified list of constants. One method that can be used is to check if any bits set in all of the variables are always clear in all of the constants. Rule 8 of Table 4 depicts how this is accomplished, where *c1..cn* are constants. A bitwise AND operation is performed on all of the variables to determine which bits are set in all of these variables. The complement of the bitwise OR of the constants is taken, which results in the bits being set that are clear in all of the constants. Note that determining which bits in the constants are always clear is determined at compile time. If any bits set in all of the variables are clear in all of the constants, then it is known that all of the variables will not be equal to all of the constants.

Rule 8 in Table 4 is also used when the constants are all zero. Figure 12 shows how the merged condition in rule 8 simplifies to the merged condition in rule 7 when *c1..cn* are all zero.

```
(v1&..&vn)&~(c1|..|cn) ≠ 0
 (v1&..&vn)&~(0|..|0) ≠ 0
       (v1&..&vn)&~(0) ≠ 0
(v1&..&vn)&0xFFFFFFFF ≠ 0
           (v1&..&vn) ≠ 0
```

Figure 12: Rule 7 Is Implied by Rule 8

Another method to determine if multiple variables are not equal to a list of constants is to check if one or more bits that are clear in all of the variables are set in all of the constants. Merging conditions using this method is shown in rule 9 of Table 4. The decision to use rule 8 or 9 is determined by checking the success of these rules during the profile run. Given an equal likelihood that either rule could be successfully applied, rule 8 is preferable since rule 9 requires an extra instruction to perform the bitwise NOT operation, which is accomplished at compile time for rule 8.

Sometimes we found that rules 8 or 9 are used when a single variable was checked to see if it was not equal to a set of constants. For instance, Figure 13 illustrates how rule 8 is simplified to only require a single bitwise AND operation to set the condition codes when only a single variable was involved. Thus, both the range checking method illustrated in Figure 5 and the bit testing methods using rules 8 or 9 are checked to estimate which would be most profitable.

```
(v1&..&v1)&~(c1|..|cn) ≠ 0
        v1&~(c1|..|cn) ≠ 0
```

Figure 13: Using Rule 8 Efficiently with a Single Variable

We are sometimes able to merge conditions checking if multiple variables are less than (or less than or equal to) constants. Rule 10 in Table 4 depicts how this is accomplished, where *c1..cn* have to be positive constants and the `u<` in the merged condition represents an unsigned less than comparison. If the result of the bitwise OR on the variables is less than the minimum constant, then the original conditions have to be satisfied. The unsigned less than operator is necessary since one of the variables could be negative and the result of the bitwise OR operation would be treated as a negative value if a signed less than operator is used.

Sometimes conditions can be merged comparing multiple variables to values that are not constants. Consider the original loop and unrolled loop shown in Figure 14(a) and Figure 14(b), respectively. It would appear there is no opportunity for merging conditions. However, `x` is loop invariant. Thus, the bits set in `a[i]` and `a[i+1]` can be ANDed with `~x` to determine if both array elements are not equal to `x`, as shown in Figure 14(c). The expression `~x` is loop invariant and the compiler will move it out of the loop when loop-invariant code motion is performed. Likewise, the loads of the two array elements in the `else` case will be eliminated after applying common subexpression elimination. Note we are also able to merge conditions checking if multiple variables are not equal to multiple loop invariant values. The profitability of using rules 8 and 9, where *c1..cn* could be loop invariant values, is estimated during the profile run.

(a) Original Code

```
for (i=0; i < 10000; i++)
   if (a[i] == x)
      num++;
```

(b) After Loop Unrolling

```
for (i=0; i < 10000;
     i += 2) {
   if (a[i] == x)
      num++;
   if (a[i+1] == x)
      num++;
}
```

(c) After Condition Merging

```
for (i=0; i < 10000;
     i += 2) {
   if ((a[i]&a[i+1])
       & ~x)
      continue;
   else {
      if (a[i] == x)
         num++;
      if (a[i+1] == x)
         num++;
   }
}
```

Figure 14: Merging Conditions That Check If Multiple Variables Are Not Equal to Loop-Invariant Values

Figure 15 shows another example where conditions comparing multiple unsigned variables to nonconstants can be merged. When finding the largest value in an array, it is very likely that most of the elements examined will not be greater than the maximum value found so far. When merging the less than or equal tests, the value (`max` in this case) being compared to the variables needs to be the same value (otherwise one would not know which would be the minimum) and has to be invariant in the path between the conditions.

5. ESTIMATING THE BENEFIT OF MERGING A SET OF CONDITIONS

In order to apply a condition merging transformation, the compiler needs to know which paths are frequently executed in the program. We extended the *ease* environment [DaW91] in the

(a) Original Code

```
max=0;
for (i=0; i < 10000; i++)
   if (a[i] > max)
      max=a[i];
```

(b) After Loop Unrolling

```
max = 0;
for (i=0; i < 10000;
     i += 2) {
   if (a[i] > max)
      max=a[i];
   if (a[i+1] > max)
      max=a[i+1];
}
```

(c) After Condition Merging

```
max=0;
for (i=0; i < 10000;
     i += 2) {
   if ((a[i] | a[i+1]) <=
       max)
      continue;
   else {
      if (a[i] > max)
         max=a[i];
      if (a[i+1] > max)
         max=a[i+1];
   }
}
```

Figure 15: Merging Conditions That Check If Multiple Unsigned Variables Are Less Than an Unsigned Invariant Value

vpo compiler [BeD88] to collect path profile information. First, we had to define what type of sequences of blocks would constitute a path. We decided to detect paths within a function that are either terminated by edges that crossed loop boundaries or when a return block is reached. Consider the example flow graph that is shown in Figure 16. The edges that cross loop boundaries are 3→4 (entering the loop), 6→4 (backedge), and 6→7 (exiting the loop). Statically enumerating all of the paths in a function according to this definition can sometimes result in an excessive number of paths. Thus, we decided to detect paths dynamically during the execution of the profile run. During the first compilation pass we modify the generated assembly to insert a call to an instrumentation routine at the beginning of each basic block. When this routine is invoked, the current block number is appended to the list of blocks that have been encountered for the current path. When we detect a block that could not be in the same path, then we record the list of blocks as a path if it was the first time the path was encounted and we increment a counter for that path. We encountered many functions that had a large number of static paths, but relatively few paths were actually executed. Whenever the dynamic number of unique paths in a function exceeded a specified limit (100 in our experiments), we would stop collecting measurements and not attempt to merge sets of conditions for that function. When this limit was exceeded, which rarely occurred in our test programs, we felt that it was unlikely to find a frequent path within such a function since so many paths are executed.

We also allowed sets of conditions across the backedges of loops to be merged. However, conditions to be merged were at most allowed to span two consecutive paths to simplify the analysis.

We use the path profile information for merging conditions in the following manner. We only examine paths whose execution contributed to more than 0.1% of the total instructions executed in a program. Consider the flow graph in Figure 17. Assume that the conditions in blocks 2 and 6 can be merged together, where both branch conditions are assumed to be satisfied. This example illustrates two interesting points. First, a path starting at block 2 and ending at block 6 comprises only a subpath of any path represented in the path profile information. Second, there are two possible subpaths (2→4→5→6 and 2→4→6) connecting these two branches even when the condition in the first branch is satisfied. We duplicated code associated with subpaths rather than entire paths when merging sets of conditions. The compiler collects all possible subpaths that comprise a set of conditions. It then uses the path frequency information to estimate the benefit of merging each set. The compiler determines the likelihood that the path comprising these branches with the appropriate transitions will be taken given that the first branch in the set will be reached. The benefit is estimated based on the instructions saved when these conditions are satisfied and the extra instructions executed when the conditions are not satisfied.

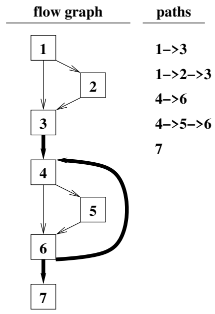

Figure 16: Paths Do Not Cross Loop Boundaries

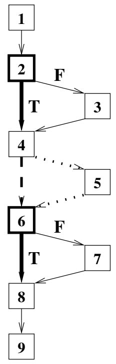

Figure 17: A Set of Conditions to Be Merged May Be Connected by Multiple Subpaths

6. APPLYING THE TRANSFORMATION

Figure 18 gives a high-level overview of the algorithm used to merge sets of conditions. After finding all of the sets of conditions that can be merged, the compiler sorts these sets in the order of the highest estimated benefit first. There are two reasons for merging the most beneficial sets first. (1) Merging sometimes requires the generation of loop-invariant expressions that can be moved out of the loop after applying loop-invariant code motion. This transformation requires the allocation of registers for which there are only a limited number available on a target machine. (2) Merging conditions changes the control flow within a function. If two sets of conditions overlap in the paths of blocks that connect them, then the estimated benefit is invalid after the first set is merged and the second set of conditions will not be merged.

```
FOR each path executing > 0.1% of the total insts DO
   FOR each branch in the path DO
      FOR each of the remaining branches in the path DO
         Determine if the set of branches can be merged.
   FOR each mergeable set of conditions in the path DO
      Estimate the benefit for that set based on the
         expected gain.
Sort the sets in descending order of benefit.
FOR each set with a benefit whose blocks are
    not affected by a previously merged set DO
   IF the set has the needed registers for the
      merging transformation to be applied THEN
      Merge the set of conditions.
      Mark blocks that are affected.
Reapply other code improving transformations.
```

Figure 18: Overview of Condition Merging Optimization

After merging sets of conditions, we apply a number of code-improving transformations in an attempt to improve the modified control flow. For instance, the elimination of the branch in block 3 in Figure 4 simplifies the control flow in the *win* and *breakeven* paths. Our general strategy was to generate the control flow in a simple manner and rely on other code-improving transformations to improve the code. For instance, the code shown in Figures 14(c) and 15(c) can be improved by applying loop-invariant code motion and common subexpression elimination. We also invoke a number of code-improving control-flow transformations, such as branch chaining, code positioning, and unreachable code elimination, to improve the code layout.

7. RESULTS

Table 5 shows the test programs on which condition merging was applied. We chose these non-numerical applications since they have complex control flow, a higher density of conditional branches, and have branches testing integer values. Control dependences in such applications often inhibit many types of compiler optimizations. For each test program we used training and test data that were available with the SPEC benchmarks or were similar to the examples found in the man pages describing the other applications. In each case the training data was smaller than the test data, resulting in fewer instructions executed in the training run than in the test runs reported in this section.

SPEC Benchmarks	Description
compress	Compresses and decompresses files.
go	AI game playing program.
ijpeg	Graphic compression and decompression
li	LISP interpreter.
m88ksim	Motorola 88K simulator.
perl	Practical extraction and report language.
vortex	Database program.
Other Applications	**Description**
ctags	Generates a tag file for *vi*.
dd	Copies a file to another file with possible conversions.
deroff	Removes *nroff* constructs from a file.
diff	Displays line-by-line differences between two text files.
grep	Searches a file for a string or regular expression.
lex	Lexical analysis program generator.
nroff	Text formatter.
od	Dumps files in a variety of formats.
othello	Game playing program.
pr	Prepares file(s) for printing.
sed	Stream editor.
sort	Sorts and collates lines.
tbl	Formats tables for *nroff*.
tr	Substitutes or deletes selected characters in text.
uniq	Report or filter out repeated lines in a file.
yacc	Yet another compiler-compiler.

Table 5: Test Programs

Table 6 shows the overall impact that each of the condition merging techniques had on the number of branches performed and the total number of instructions executed. The baseline measurements included the reduction of local and global scalar variables to bits within flag variables, which achieved slight reductions in the number of instructions executed as illustrated in Figure 7. The different techniques are shown in the same order in which they were presented in the paper (Sections 3.1, 3.2, 3.3, and 4). Techniques A and B can eliminate conditions that could not be eliminated using non-path based approaches. For instance, Figure 4 shows an example of eliminating a logically redundant condition using technique A that could not be eliminated using static analysis alone [MuW95]. Technique B can eliminate conditions that are separated by other intervening branches, which is also not feasible using a non-path based approach [YUW98, YUW02]. Note that techniques A, B, and C merge sets of conditions that compare a single variable to constants and technique D merges sets of conditions that can compare multiple variables. However, technique C, which merges bit tests, relies on the reduction of different scalar variables to bits within flag variables. Thus, technique C in effect merges conditions that compare multiple variables specified in the source code. In fact, many of the sets of conditions merged using technique C would have been merged using technique D if reducing scalars to bits had not been applied. Table 6 shows techniques C and D reduced the number of branches by 6.42% over what was accomplished by techniques A and B. In fact, technique D alone reduced more branches than techniques A and B combined. Overall, 15.87% of the executed branches were eliminated.

Label	Description	Branches	Insts
A	merging using logical correlation	95.832%	98.580%
B	merging using range checks	93.537%	96.354%
C	merging of bit tests	98.008%	99.399%
D	merging using logical operations	88.467%	96.330%
A-B	applying techniques A and B	89.961%	95.468%
A-C	applying techniques A, B, and C	88.279%	95.016%
A-D	applying techniques A, B, C, and D	84.190%	94.256%

Table 6: Dynamic Results from Applying the Individual Techniques

The primary goal of our study was to reduce the number of conditional branches executed. However, it is interesting to note the effect on other performance measures, even though we did not tune the compiler to exploit the modified control flow. Table 6 also shows the effect that condition merging had on the number of instructions executed. Overall, there was a 5.74% average reduction in the number of instructions executed. While most of the reduction was directly due to fewer executed comparisons and branches, occasionally the compiler was able to apply additional code-improving transformations on the modified control flow, such as common subexpression elimination, to obtain additional improvements.

Figures 20-22 displays condition merging results for each program. Figure 20 displays the effects of each individual technique on the number of branches. While using logical operations (technique D) and range checks (technique B) were the most beneficial, all of the techniques can be usefully applied. Figures 21 and 22 show the cumulative results of applying the techniques. *Sort* had unusually large benefits since most of the execution was spent in tight loops where conditions could be merged using techniques B and D.

There are reasons why the cumulative benefits was less than the sum of the benefits from the individual techniques. (1) Sometimes sets of conditions merged by one technique could also be merged by using a different technique. For instance, sets of $\neq$ branches in *sort* could be merged using logical operations (technique D), as described in Section 4.2, or by using range checks (technique B), as described in Section 3.2. (2) Sometimes different sets of conditions to be merged overlapped in the control-flow graph. When two sets of conditions overlap and it was estimated that both could be merged separately with benefits, we only merged the set deemed most beneficial. In this case the merging of the first set of conditions will change the paths associated with the second set of branches. (3) Merging conditions sometimes required the allocation of registers for loop-invariant values. Merging one set of branches sometimes consumed the remaining available registers that are needed to merge another set.

Figure 23 shows the effect that condition merging had on execution time for each of the test programs. For the UltraSPARC II, we found that there was a high variance in the *user* time using

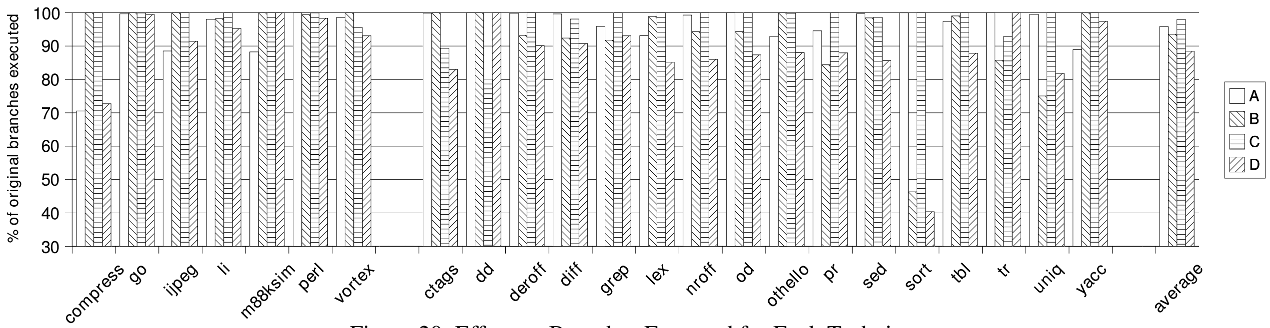

Figure 20: Effect on Branches Executed for Each Technique

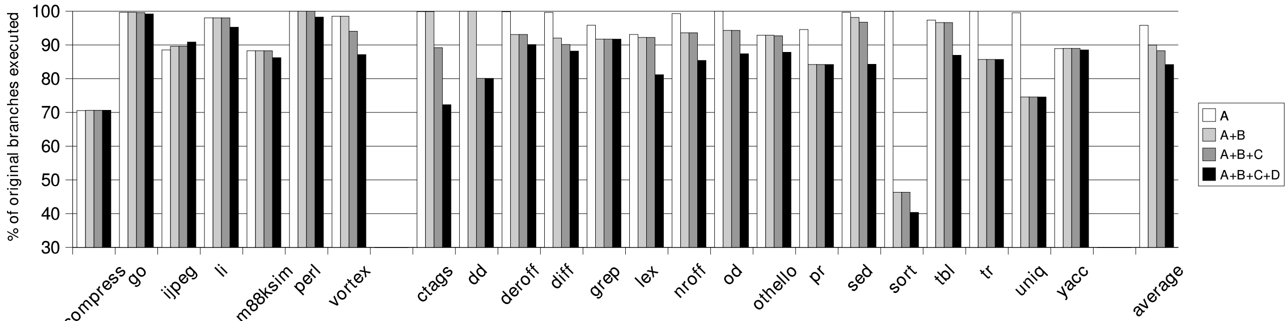

Figure 21: Cumulative Effect on Branches Executed

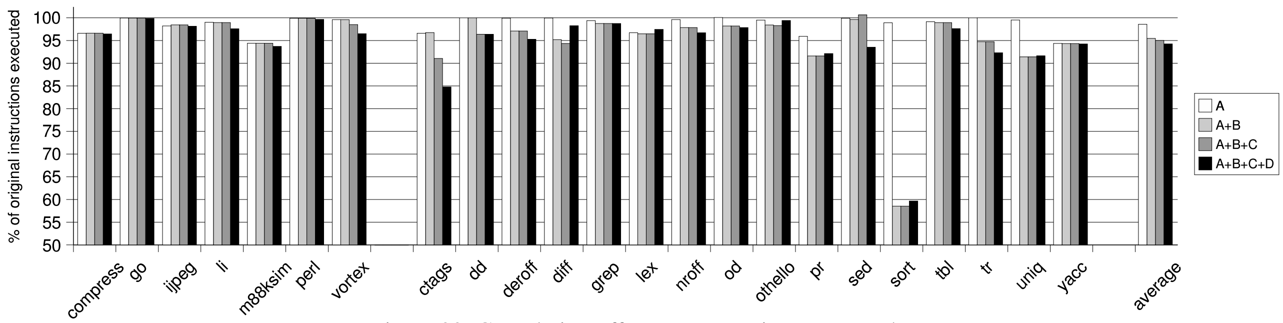

Figure 22: Cumulative Effect on Instructions Executed

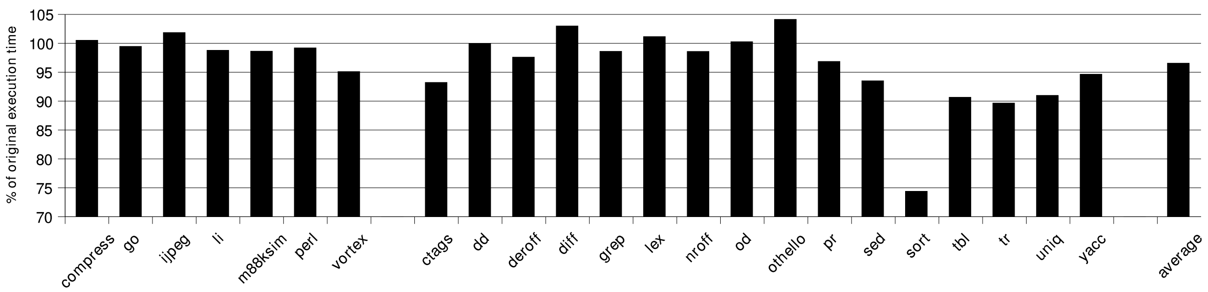

Figure 23: Effect on Execution Time

the Unix utilities available (e.g. *ptime*). Thus, we instead measured execution time by reading the 64-bit clock register (*%tick*) that is accessible on the UltraSPARC architectures. We read the value in this clock register before and after each external call to a run-time library routine, which allowed us to obtain for each program a fairly consistent execution time that does not include the time spent in the run-time library. We used the minimum time of 200 executions of each program, which represents the execution time with the fewest cycles spent in other processes due to interrupts. The figure shows that on average that there was a **3.43**% reduction on the UltraSPARC II.

Applying all the techniques resulted in a **6.99**% increase in static code size after condition merging. This static increase compares favorably with the 5.74% decrease in instructions executed. The code size would have increased more if we had not required that the paths we inspected comprise at least 0.1% of the total

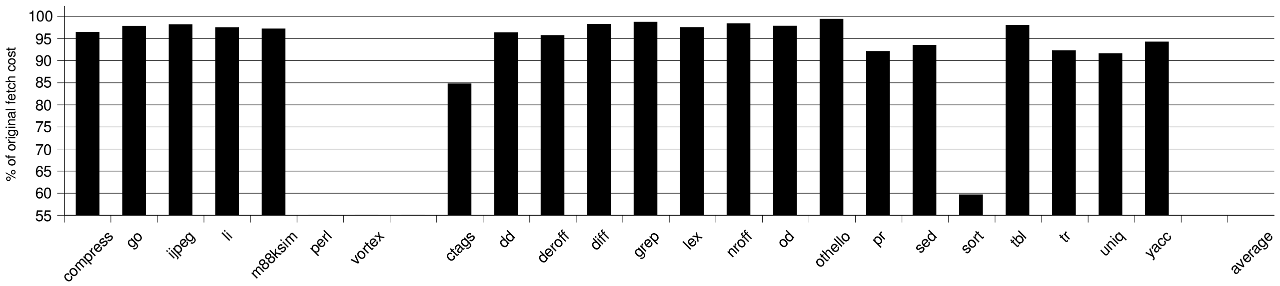

Figure 24: Effect on Estimated Fetch Cost for a 16KB Two-Way Associative Instruction Cache

instructions executed.

Figure 24 shows the effect that condition merging had on instruction cache performance for a 16KB two-way set associative cache with 32 byte lines, which is the configuration used on the UltraSPARC II. We found that the average miss ratio increased slightly (**+0.000??**) after merging conditions due to the increase in the static number of instructions. However, the miss ratio is not an appropriate measurement given that the number of instructions executed was affected. In the figure we show the effect on the overall fetch cost, which we calculated by counting each hit as a single cycle and each miss as ten cycles [Smi82]. The average fetch cost decreased by almost **?**%. As expected, there is a very strong correlation between instructions executed (A+B+C+D result in Figure 22) and fetch cost (Figure 24) since the instruction cache hit ratios were high. Figure 25 shows the effect condition merging had on the fetch cost for a variety of cache sizes, where each configuration was two-way associative with a 32 byte line size. In each case the average fetch cost was improved.

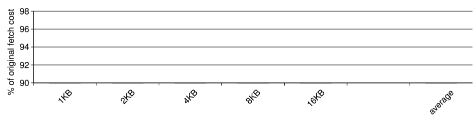

Figure 25: Average Fetch Cost for a Variety of Cache Sizes

Branch prediction measurements were obtained for the SPARC Ultra II when executing all four techniques. This machine has 2048 branch prediction entries, where a (0,2) predictor is used [HeP96]. Given that the number of executed branches decreased by 15.87%, one would suspect that the number of branch mispredictions would also decrease. We found this was often not the case. **????** of the test programs have fewer mispredictions after condition merging and the remaining **????** have increases. Overall, the average number of mispredictions increased by **?.??**%. Past studies on control CPR have not investigated the impact on branch prediction [ScK95, SMJ99].

One may question the value of condition merging since the number of branch mispredictions is sometimes increased. We found that the average number of fewer instructions executed to each additional branch misprediction was **???.??** for the **????** programs that exhibited an increased number of mispredictions. Thus, the increase in mispredictions was on average far outweighed by the benefit of reducing the number of instructions executed. A similar phenomenon was discovered when reordering contiguous sequences of branches [YUW98, YUW02].

On average there were only 10.13 sets of conditions merged per test program. While some sets were not merged due to overlapping regions of code, most beneficial sets were not merged due to not being executed frequently enough. The number of conditions merged per set ranged from 2 to 6. 82.83% of the sets that were merged contained 2 conditions, 9.01% contained 3 conditions, 6.01% contained 4 conditions, 0.86% contained 5 conditions, and 1.29% contained 6 conditions. Due to these sets of conditions being frequently executed, significant control height reduction is achieved despite only merging a small number of sets and conditions per set in each test program.

8. FUTURE WORK

One area we plan to explore is the use of more aggressive analysis to detect when speculatively executed loads would not introduce exceptions. We perform condition merging in a compiler back end using a representation that was equivalent to machine instructions. One advantage of performing condition merging in a back end is that more accurate estimates of the performance benefits can be made, which is very important when applying transformations based on path profile data. However, a disadvantage of performing a transformation in a back end is that much of the semantic information which one could use to determine if a load instruction can cause an exception is not immediately available in a low level representation. We conservatively merged sets of conditions that required loads to be speculatively executed only when it was obvious that these loads would not introduce new exceptions. By performing more aggressive analysis, we should be able to detect more sets of conditions to be merged. This will be particularly useful for merging conditions that cross loop boundaries.

Value range analysis may allow condition merging to be applied more frequently. Consider if `a` and `max` in Figure 15 were signed instead of unsigned variables. `max` is guaranteed to be nonnegative in this example since it is initialized to zero and it is only updated with a value that is greater than itself. Merging conditions in this case can be applied using an unsigned ≤ operator. If either `a[i]` or `a[i+1]` is negative, then the merged condition will fail and the original conditions will be tested.

We also sometimes found that the merging of one set of conditions would inhibit the merging of another set. Code is duplicated and the paths within a function are modified when conditions are merged. This code duplication invalidates the path profile data on which condition merging is based. Thus, merging a set of conditions is not currently allowed whenever the control flow changed in the subpaths associated with the set of conditions to be merged. With careful analysis we may be able to infer new path frequency measurements for these duplicated paths.

9. CONCLUSIONS

In this paper we described techniques to perform condition merging on a conventional scalar processor. We replaced the execution of two or more branches with a single branch by merging conditions. Path profile information is gathered to determine the frequency that paths are executed in the program. Sets of conditions that can be merged are detected and the benefit of merging each set is estimated. The control flow is then restructured to merge the sets of conditions deemed beneficial. The results show that significant reductions can be achieved in the number of branches performed for non-numerical applications. In addition, we showed benefits in the number of instructions executed, instruction cache performance, execution time, and branch prediction performance relative to dynamic instruction counts.

There are several contributions that we presented in this paper. First, we have shown that the reduction of integer scalar variables to bits within a flag variable can be automated and results in opportunities to merge conditions. Second, we were able to merge conditions comparing multiple variables to constants or invariant values. Unlike prior work on control CPR, we were able to accomplish our techniques through innovative use of available instructions on a conventional scalar processor. Finally, we have shown there are benefits to be obtained by merging conditions in paths that are not the most frequent. In many cases we were able to generate a *breakeven* path that allows a set of conditions to be merged when the *win* path was not the most frequently executed path upon reaching the first condition.

We believe that condition merging may be useful in other settings. Condition merging may be a very good fit for run-time optimization systems, which optimize frequently executed paths during the execution of a program. Condition merging may also be useful for low power embedded systems processors where architectural support for ILP is not available.

REFERENCES

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

[BeD88] M. E. Benitez and J. W. Davidson, "A Portable Global Optimizer and Linker," *Proceedings of the SIGPLAN '88 Symposium on Programming Language Design and Implementation*, pp. 329-338 (June 1988).

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

[DaW91] J. W. Davidson and D. B. Whalley, "A Design Environment for Addressing Architecture and Compiler Interactions," *Microprocessors and Microsystems* **15**(9) pp. 459-472 (November 1991).

[DoH79] J. Dongarra and A. Hinds, "Unrolling Loops in FORTRAN," *Software Practice & Experience* **9** pp. 219-226 (1979).

[FrH95] C. W. Fraser and D. R. Hanson, *A Retargetable C Compiler: Design and Implementation,* Addison-Wesley (1995).

[GrK92] T. Granlund and R. Kenner, "Eliminating Branches using a Superoptimizer and the GNU C Compiler," *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pp. 341-352 (June 1992).

[HeP96] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, Second Edition,* Morgan Kaufmann, San Francisco, CA (1996).

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

[ScK95] M. Schlansker and V. Kathail, "Critical Path Reduction for Scalar Programs," *International Symposium on Microarchitecture*, pp. 57-69 (December 1995).

[SMJ99] M. Schlansker, S. Mahlke, and R. Johnson, "Control CPR: A Branch Height Reduction Optimization for EPIC Architectures," *Proceedings of the SIGPLAN '99 Symposium on Programming Language Design and Implementation*, pp. 155-168 (May 1999).

[Smi82] A. J. Smith, "Cache Memories," *Computing Surveys* **14**(3) pp. 473-530 (September 1982).

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

[YUW98] M. Yang, G. Uh, and D. B. Whalley, "Improving Performance by Branch Reordering," *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pp. 130-141 (June 1998).

[YUW02] M. Yang, G. Uh, and D. B. Whalley, "Efficient and Effective Branch Reordering Using Profile Data," *ACM Transactions on Programming Languages and Systems* **24**(6)(November 2002).