SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2003; **00**:1–25 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

Branch elimination by condition merging‡

William C. Kreahling[1], David Whalley[1], Mark W. Bailey[2], Xin Yuan[1], Gang-Ryung Uh[3], Robert van Engelen[1]

[1] *Florida State University, Tallahassee, FL 32306* [2] *Hamilton College, Clinton, NY 13323*
[3] *Boise State University, Boise, ID 83725*

SUMMARY

Conditional branches are expensive. Branches require a significant percentage of execution cycles since they occur frequently and cause pipeline flushes when mispredicted. 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. These sets of branches can include tests of multiple variables. For instance, the test `if (p1 != 0 && p2 != 0)`, which is testing for NULL pointers, can be replaced with `if (p1 & p2 != 0)`. Program profiling is performed to target condition merging along frequently executed paths. The results show that eliminating branches by merging conditions can significantly reduce the number of conditional branches executed in non-numerical applications.

KEY WORDS: compiler, condition merging, profiling, code duplication

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 execution cycles, cause pipeline flushes 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.

Contract/grant sponsor: National Science Foundation; contract/grant number: CCR-9904943, EIA-0072043, CCR-0208892, CCR-0105422, CCR-0312493
Contract/grant sponsor: Department Of Energy; contract/grant number: DEFG02-02ER25543
Contract/grant sponsor: National Aeronautics and Space Administration, EPSCoR; contract/grant number: -
‡This is an preprint accepted for publication in Software: Practice and Experience, ©John Wiley & Sons Ltd

Copyright © 2003 John Wiley & Sons, Ltd.

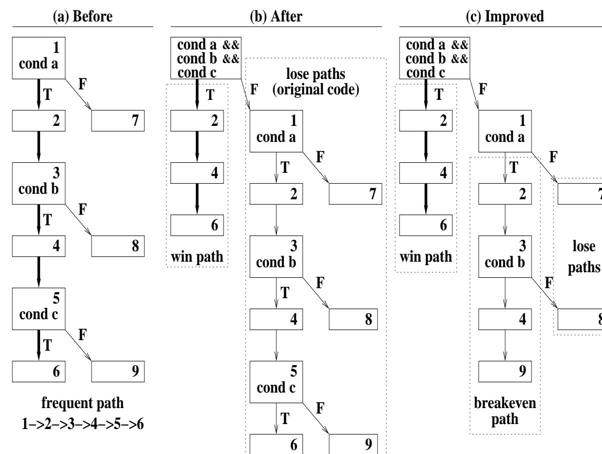

Figure 1. Merging Three Conditions

One approach to reducing the cost of branches is to attempt to merge a set of conditions 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*. 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 branches can 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 in this path as were executed in the original path. We only apply the condition merging transformation when we estimate that the total instructions executed will be decreased.

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

*Blocks 2 and 3 in Figure 1(a) 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.

Copyright © 2003 John Wiley & Sons, Ltd.

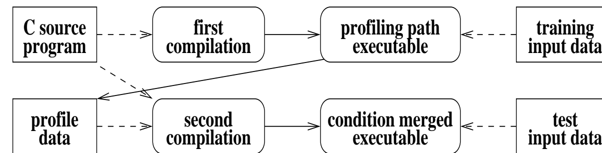

Figure 2. Compilation Process for Merging Conditions

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* [7] and the back-end used is *vpo* [3] targeted to the SPARC architecture. For our test programs these techniques on average decrease the number of branches by 15.81% (0.82% to 59.62%), the number of instructions by 5.74% (0.14% to 40.34%), and the execution time by 3.43% (-4.15% to 25.59%). These improvements were automatically obtained by the compiler on a conventional scalar processor.

RELATED WORK

There are numerous techniques used to decrease the number of conditional branches executed. Loop unrolling has long been used to reduce the number of times the conditional branch associated with a loop termination condition is executed [6]. Loop unswitching moves a conditional branch with a loop-invariant condition before the loop and duplicates the loop in each of the two destinations of the branch [1]. Superoptimizers have been used to find a bounded sequence of instructions that have the same effect as a conditional branch [8]. Conditional branches have been avoided by using static analysis and code duplication [10, 4]. Conditional branches have been coalesced together into an indirect jump from a jump table [15]. Sequences of branches have been reordered to allow the sequence to be exited earlier, which reduces the number of branches executed [17, 18]. A technique, which shares some similarities to the work described in this paper, applies conjunctions to selection conditions for records in database queries [11]. However these conjunctions were applied manually as opposed to our optimization which is applied automatically by a compiler. Finally, there has been recent work on eliminating branches using architectural features designed to increase ILP (instruction level parallelism). *If conversion* uses predicated execution to eliminate branches by squashing the result of an instruction when a predicate is not satisfied [9]. Another technique eliminates branches by performing if-conversion as if the machine supported predicated execution, and then applies reverse-if conversion to return the code to an acyclic control flow representation [16]. This technique shares some similarities to ours, however our approach differs in several aspects, including that we can merge conditions involving multiple variables. A technique called control CPR (critical path reduction) 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

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

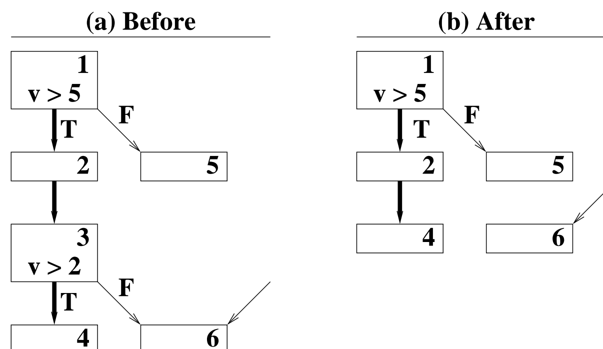

Figure 3. Merging Logically Redundant Conditions

bypass branch [12, 13]. 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 described 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.

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.

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 [10, 4].

A path-based approach using profile data 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

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

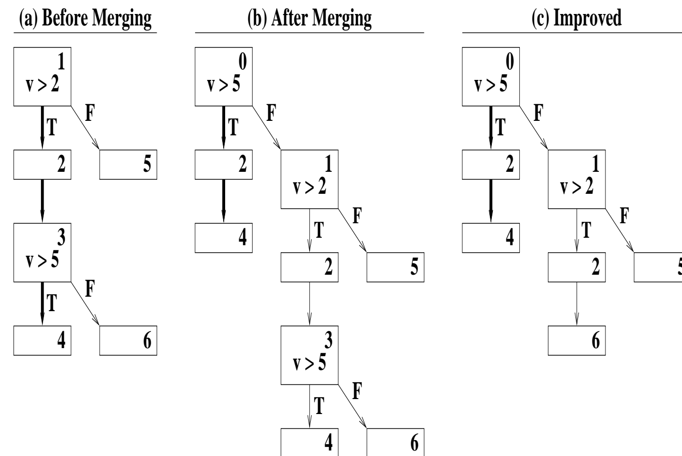

Figure 4. Merging Other Logically Redundant Conditions

be also satisfied. If the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ 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 $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$ 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.

Dynamo, a dynamic optimization system [2], could also merge conditions like the ones depicted in Figure 4(a). However, in many cases, our technique is able to further improve the code by restructuring the flow graph. For example, the branch in block 3 is eliminated, as shown in Figure 4(c), since the test is now redundant.

Performing the code-improving transformation results in one less branch executed when path $0 \rightarrow 2 \rightarrow 4$ is taken, the same number of branches executed when the path $0 \rightarrow 1 \rightarrow 2 \rightarrow 6$ is taken, and one additional branch executed when the path $0 \rightarrow 1 \rightarrow 5$ 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 $0 \rightarrow 2 \rightarrow 4$, which reduces the number of branches, is taken more frequently than the *lose* path $0 \rightarrow 1 \rightarrow 5$, which increases the number of branches. Thus, improvements may be obtained even when the *break-even* path $0 \rightarrow 1 \rightarrow 2 \rightarrow 6$ is the most frequently executed path.

Merging Not Equal Tests Using Range Checks

A single variable is often 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 [17, 18]. Profile data

Copyright © 2003 John Wiley & Sons, Ltd.
Prepared using speauth.cls

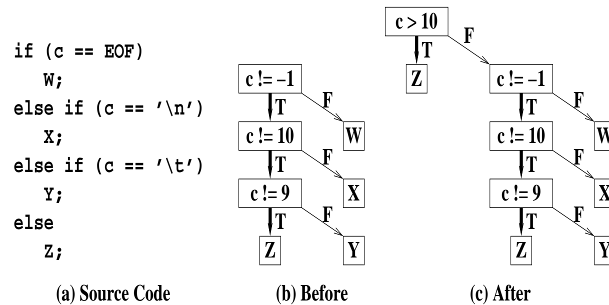

Figure 5. Merging Not Equal Tests Using Range Checks

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, which 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 [17, 18].

Merging Bit Tests

Different bits in a single variable are sometimes 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. If the profile data indicates that one combination of bits is very likely, then that combination of bits can be tested in a single comparison by changing the constant being compared, 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 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 and the branch in block 3 can be eliminated. Table I 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

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

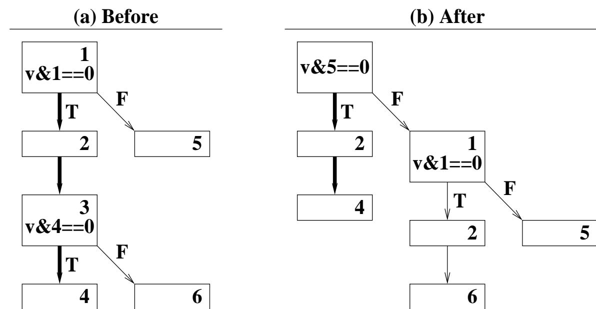

Figure 6. Merging Bit Tests

Table I. SPARC Code Generation Strategies for Merged Bit Tests

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

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 of the table 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 performing a comparison to zero to set the integer condition codes. 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

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

(a) Original Instructions	(b) After Merging Assignments Setting Bits

`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];`
	`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];`

(c) After Merging Assignments Clearing Bits	(d) After Eliminating Redundant Loads	(e) After Eliminating Redundant Stores

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

Figure 7. Other Benefits from Reducing Scalars to Single Bits

with zero. The first instruction assigning the mask value 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.

Integer flags are commonly used in many applications. In effect, such variables are used as booleans, which requires only a single bit to be represented. Our system automatically reduces 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 '$\neq 0$' tests.

We accomplish the reduction of integer scalars to bits in a flag variable by examining and updating the *l*cc 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 '$\neq 0$' 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.

Copyright © 2003 John Wiley & Sons, Ltd.
Prepared using speauth.cls

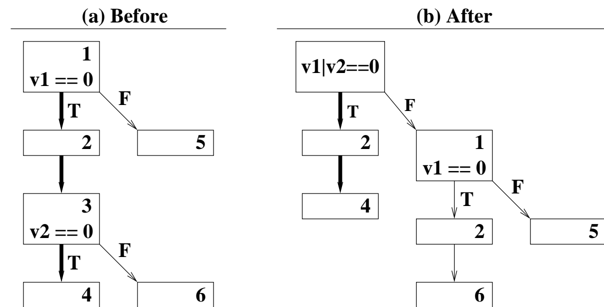

Figure 8. Merging Conditions That Test If Different Variables Are Equal to Zero

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 describe techniques for merging sets of conditions that involve multiple variables.

Merging into a Single Equivalent Condition

We use logical operations 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.

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. $r[1] \ldots r[n]$ 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).

Table II shows how sets of conditions comparing multiple variables $(v_1 \ldots v_n)$ to 0 and -1 can be merged into a single condition. Column one gives the rule number, column two depicts the original condition, column three depicts the merged condition, and column four represents the percentage of time the rule was selected during testing compared to the other rules shown in

Copyright © 2003 John Wiley & Sons, Ltd.

```
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 Generated for the Merged Condition That Checks If n Variables Are Equal to Zero

Table II. Rules for Merging Conditions Comparing Multiple Variables into a Single Equivalent Condition

Rule	Original Conditions	Merged Condition	% Applied
1	$v_1 = 0$ &&..&& $v_n = 0$	$(v_1$ |..| $v_n) = 0$	42.7%
2	$v_1 = 0$ &&..&& $v_n =$ -1	$(v_1$ |..| $\sim v_n) = 0$	0.0%
3	$v_1 < 0$ &&..&& $v_n < 0$	$(v_1$ &..& $v_n) < 0$	0.0%
4	$v_1 \geq 0$ &&..&& $v_n \geq 0$	$(v_1$ |..| $v_n) \geq 0$	4.5%
5	$v_1 < 0$ &&..&& $v_n \geq 0$	$(v_1$ &..& $\sim v_n) < 0$	0.9%
6	$v_1 \geq 0$ &&..&& $v_n < 0$	$(v_1$ |..| $\sim v_n) \geq 0$	0.0%

Table II and Table III. 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 is 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 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.

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.

Copyright © 2003 John Wiley & Sons, Ltd.

(a) Original Code	(b) After Loop Unrolling

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

```
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

Additional opportunities for condition merging become available when sets of conditions, which cross 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. Our system merges sets of conditions across loop iterations. In order to simplify the analysis, we only merge conditions that span two consecutive iterations of a loop.

Merging into a Sufficient Condition

Our system also uses logical operations to efficiently merge conditions, which 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. Table III shows rules for merging conditions containing multiple variables into a single sufficient condition. The columns in this table similar information to the columns in table II.

There were no $\neq 0$ tests listed in Table II. 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 III. Note that failure of the merged condition does not imply that the variables are all not equal to zero.

One may ask how often such conditions can 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 non-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

Copyright © 2003 John Wiley & Sons, Ltd.

Table III. Rules for Merging Conditions Comparing Multiple Variables into a Single Sufficient Condition

Rule	Original Conditions	Merged Condition	% Applied
7	$v_1 \neq 0$ &&..&& $v_n \neq 0$	$(v_1$ &..& $v_n) \neq 0$	18.2%
8	$v_1 \neq c_1$ &&..&& $v_n \neq c_n$	$(v_1$ &..& $v_n)$ & $\sim(c_1$ |..| $c_n) \neq 0$	15.5%
9	$v_1 \neq c_1$ &&..&& $v_n \neq c_n$	$\sim(v_1$ |..| $v_n)$ & $(c_1$ |..| $c_n) \neq 0$	16.4%
10	$v_1 < c_1$ &&..&& $v_n < c_n$	$(v_1$ |..| $v_n)$ $u<$ $\min(c_1, .., c_n)$	1.8%

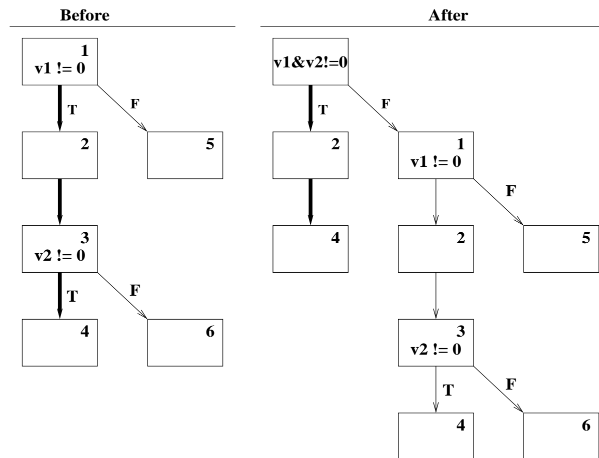

Figure 11. Merging Conditions That Check If Different Variables Are Not Equal to Zero

multiple variables are not equal to zero can be merged. If the merged condition is not satisfied, then the entire original set of branches still needs to be tested.

We are also able to merge conditions that check if multiple variables are all not equal to a specified list of constants. One method we 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 III depicts how this is accomplished, where $c_1 \ldots c_n$ 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 III is also used

Copyright © 2003 John Wiley & Sons, Ltd.
Prepared using speauth.cls

$$
\begin{aligned}
(v_1\&..\&v_n)\& \sim (c_1|..|c_n) &\neq 0 \\
(v_1\&..\&v_n)\& \sim (0|..|0) &\neq 0 \\
(v_1\&..\&v_n)\& \sim (0) &\neq 0 \\
(v_1\&..\&v_n)\&0xFFFFFFFF &\neq 0 \\
(v_1\&..\&v_n) &\neq 0
\end{aligned}
$$

Figure 12. Rule 7 is Implied by Rule 8

$$
\begin{aligned}
(v_1\&..\&v_1)\& \sim (c_1|..|c_n) &\neq 0 \\
v_1\& \sim (c_1|..|c_n) &\neq 0
\end{aligned}
$$

Figure 13. Using Rule 8 Efficiently with a Single Variable

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 $c_1 \ldots c_n$ are all zero.

Another method to determine if multiple variables are not equal to a list of constants is to check if one or more bits, which 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 III. 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.

We found that rules 8 or 9 can be 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.

We are also able to merge conditions checking if multiple variables are less than (or less than or equal to) constants. Rule 10 in Table III depicts how this is accomplished, where $c_1 \ldots c_n$ must 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 operation is used.

Our system merges conditions 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, bits that are set in both $a[i]$ and $a[i+1]$ can be ANDed with $\sim x$ to determine if the array elements are not equal to x, as shown in Figure 14(c). The expression

Copyright © 2003 John Wiley & Sons, Ltd.

(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

$\sim x$ 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 $c_1 \ldots c_n$ could be loop invariant values, is estimated during the profile run.

Figure 15 shows another example where conditions comparing multiple unsigned variables to non-constants 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.

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 [5] in the *VPO* compiler [3] to collect path profile information. We define a path as a sequence of blocks within a function that are either terminated by edges that cross 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 $3 \rightarrow 4$ (entering the loop), $6 \rightarrow 4$ (backedge), and $6 \rightarrow 7$ (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

Copyright © 2003 John Wiley & Sons, Ltd.

(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

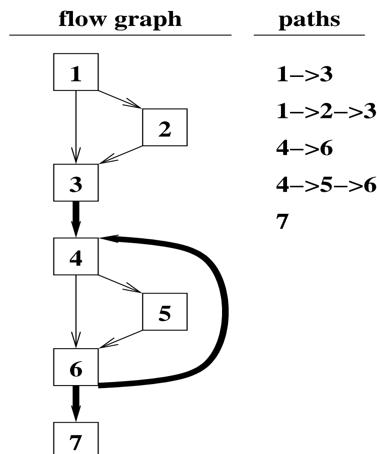

Figure 16. Paths Do Not Cross Loop Boundaries

Copyright © 2003 John Wiley & Sons, Ltd.
Prepared using speauth.cls

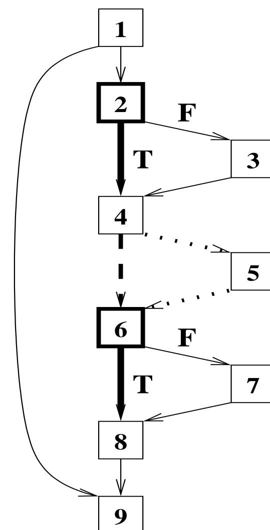

Figure 17. A Set of Conditions to be Merged May be Connected by Multiple Subpaths

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 and it is the first time the path is encountered, we record the list of blocks as a path and we increment a counter for that path. We found that many functions had a large number of static paths, but often relatively few of these paths were actually executed. Whenever the dynamic number of unique paths in a function exceeded a specified limit (100 in our experiments), we stopped collecting measurements and no longer attempted 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 attempted to merge sets of conditions across the backedges of loops. We only attempted to merge conditions that span two consecutive executions of the same path by treating it as a single path. We believe this restriction allows most of the beneficial sets of conditions across path boundaries to be merged without significantly increasing the number of sets of conditions to be evaluated.

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

Copyright © 2003 John Wiley & Sons, Ltd.

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 ($2 \rightarrow 4 \rightarrow 5 \rightarrow 6$ and $2 \rightarrow 4 \rightarrow 6$) connecting these two branches even when the condition in the first branch is satisfied. Rather than duplicating entire paths when merging sets of conditions, we instead duplicated subpaths. The compiler collects all possible subpaths that contain the specified 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 containing 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.

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, which can be merged, the compiler sorts these sets in descending order, according to estimated benefit. There are two reasons for merging the most beneficial sets first:

1. Merging may require 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, which connect them, then the estimated benefit is invalid after the first set is merged and the second set of conditions will not be merged.

After merging sets of conditions, we apply a number of code-improving transformations in an attempt to improve the modified control flow. For instance, Figure 1(c) shows that merging conditions 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, loop inversion, and unreachable code elimination, to improve the code layout. Branch chaining simplifies the control flow when the target block of a transfer of control contains a single unconditional jump, which may also have a target block containing a single unconditional jump. The first transfer of control is re-written to jump to the last target in the chain of unconditional jumps. Code positioning can reorder the blocks in the control flow. For example if the target of an unconditional jump does not have a fall through predecessor, then the target block, along with any contiguous blocks that follow it, can be placed directly after the block with the unconditional jump. The unconditional jump can then be removed. Loop inversion places a loop exit test at the end of the loop instead of the beginning of the loop, saving the execution of an unconditional jump in the loop.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls

```
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

RESULTS

Table IV 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 SPEC benchmark we used training and test data that were available with the benchmark. For the other applications we used input data similar to the examples found in the man pages describing these applications. In each case the training data was smaller than the test data, resulting in fewer instructions executed in the training run than in the test runs reported in this section.

The number of times that each rule was selected is shown in column four of Table II and Table III. The majority of the rules selected involve merging multiple variables testing for inequality to the value zero or a set of constants. However the rule that was selected most was rule 1, which involves merging variables testing for equality to zero.

Table V shows the overall impact that each 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. Techniques A and B 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 [10]. Technique B eliminates conditions that are separated by other intervening branches,

Copyright © 2003 John Wiley & Sons, Ltd.
Prepared using speauth.cls

Table IV. Test Programs

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 with possible conversions.
deroff	Removes *nroff* constructs from a file.
diff	Displays line-by-line differences between two text files.
grep	Searches files 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 V. Dynamic Results from Applying the Individual Techniques

Label	Description	Branches	Insts.
A	merging using logical correlation	95.899%	98.539%
B	merging using range checks	93.537%	96.431%
C	merging using bit tests	98.008%	99.284%
D	merging using logical operations	88.462%	96.066%
A-B	applying techniques A and B	89.961%	95.556%
A-C	applying techniques A, B, and C	88.279%	95.089%
A-D	applying techniques A, B, C, and D	84.189%	94.256%

Copyright © 2003 John Wiley & Sons, Ltd.

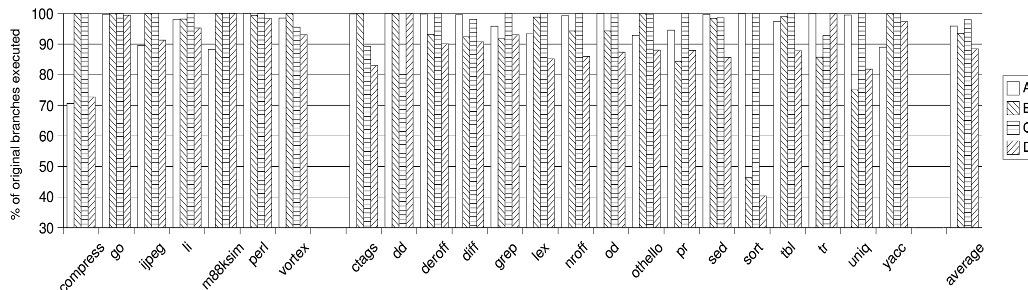

Figure 19. Effect on Branches Executed for Each Technique

which is also not feasible using a non-path based approach [17, 18]. Note that techniques A, B, and C merge sets of conditions that compare a single variable 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 V also shows techniques C and D reduced the number of branches by 6.42% over techniques A and B. In fact, technique D alone reduced more branches than techniques A and B combined. Overall, 15.81% of the executed branches were eliminated.

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 V 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 code-improving transformations on the modified control flow to obtain additional improvements.

Figures 19–21 display condition merging results for each program. Figure 19 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, branches were merged by applying rules from all of the techniques. Figures 20 and 21 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.

In a few cases, the application of an additional technique increased the number of branches and/or instructions executed. This increase was due in part to using different training and test data, which affected the accuracy of our estimated benefits. The number of instructions

Copyright © 2003 John Wiley & Sons, Ltd.

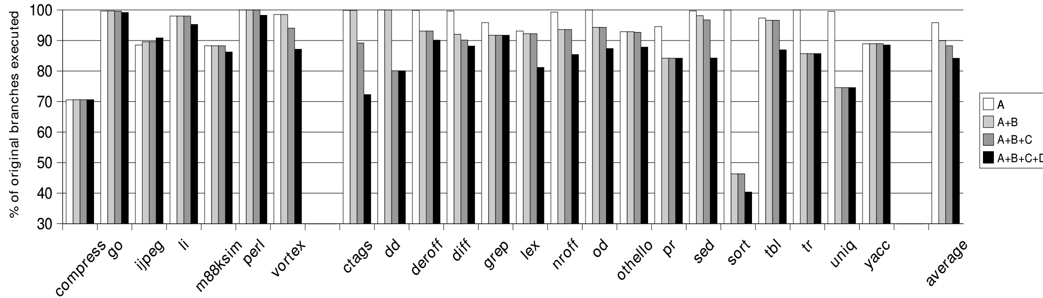

Figure 20. Cumulative Effect on Branches Executed

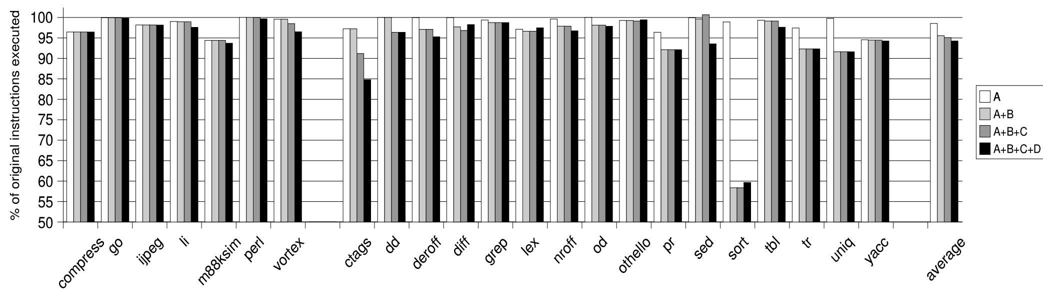

Figure 21. Cumulative Effect on Instructions Executed

executed increased more frequently since we could not predict the effect that the modified control flow would have on subsequently applied optimizations.

There are reasons why the cumulative benefits were less than the sum of the benefits from the individual techniques. (1) Sets of conditions merged by one technique may also be merged by using a different technique. For instance, sets of $\neq$ conditions in *sort* could be merged using logical operations (technique D) or by using range checks (technique B). (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 may require 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.

Copyright © 2003 John Wiley & Sons, Ltd.

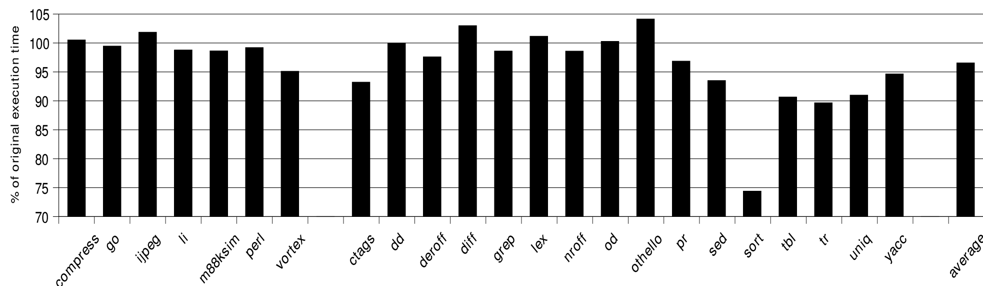

Figure 22. Effect on Execution Time

Figure 22 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 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. Figure 22 shows that on average that there was a 3.43% execution time reduction on the UltraSPARC II. While the execution time of most of the applications improved, there was a performance degradation for a couple of the applications. The reason for the degradation is difficult to explain fully given the complexity of the UltraSPARC II implementation.

Applying all the techniques resulted in a roughly 7% 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 instructions executed.

Figure 23 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 (+0.00092) after merging conditions. 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 [14]. The average fetch cost decreased by almost 5%. As expected, there is a very strong correlation between instructions executed (A+B+C+D result in Figure 21) and fetch cost (Figure 23) since the instruction cache hit ratios were high. The only program whose fetch cost increased was *perl*. A frequently executed loop, which involved code from multiple

Copyright © 2003 John Wiley & Sons, Ltd.
Prepared using speauth.cls

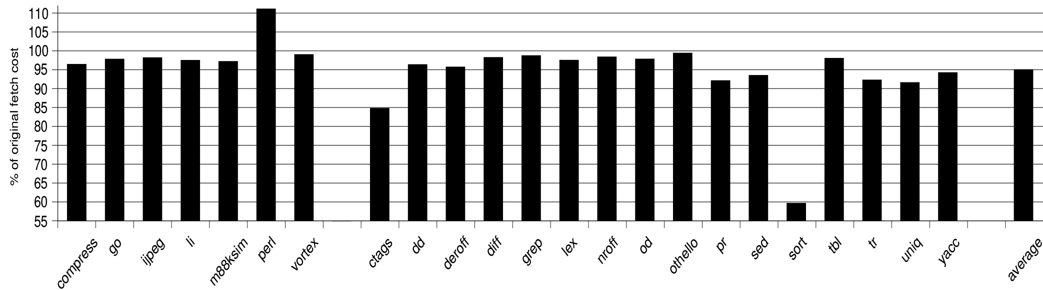

Figure 23. Effect on Estimated Fetch Cost for a 16KB Two-Way Associative Instruction Cache

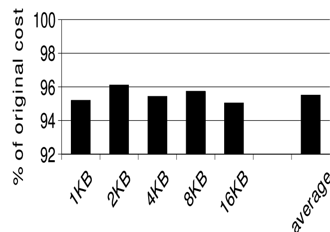

Figure 24. Average Fetch Cost for a Variety of Cache Sizes

routines, was aligned slightly differently after merging conditions and this alignment caused the performance to degrade since the routines were then mapped to the same cache lines. After merging *perl* actually had slightly fewer instructions executed and its code size only increased by about 0.1%. Figure 24 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.

Branch prediction measurements were obtained for the UltraSPARC II when executing all four techniques. This machine has 2048 branch prediction entries, where a (0,2) predictor is used [9]. Given that the number of executed branches decreased by almost 16%, one would suspect that the number of branch mispredictions would also decrease. We found this was often not the case. Five of the test programs have fewer mispredictions after condition merging and the remaining ones exhibited increases. A similar phenomenon was discovered when reordering

Copyright © 2003 John Wiley & Sons, Ltd.
Prepared using speauth.cls

Table VI. The Number of Conditions Merged.

Conditions Merged	Percentage of Sets
2	82.83%
3	9.01%
4	6.01%
5	0.86%
6	1.29%

contiguous sequences of branches [17, 18]. While condition merging may sometimes adversely affect branch prediction performance, the average execution time of the test programs still decreases. This is probably due to the large reduction of instructions executed.

On many machines there is a pipeline stall associated with every taken branch. Merging conditions resulted in approximately 28% average reduction in the number of taken branches for the test programs. This reduction was due in part to decreasing the number of executed branches. The transformation also makes branches within the *win* path more likely to fall through since the sequence of blocks representing each frequent path is now contiguous in memory. Overall, the average number of control transfers (taken branches, unconditional jumps, calls, returns, and indirect jumps) was reduced by roughly 20%.

On average there were only 10.13 sets of conditions merged per test program. Some sets, while beneficial, were not merged due to overlapping regions of code. However the main reason that some beneficial sets are not merged is infrequent execution. Table VI shows the number conditions, per set, that were merged. Due to these sets of conditions being frequently executed, significant control height reduction is achieved despite only merging on average 10.13 sets of conditions per test program and 2.29 conditions per set.

FUTURE WORK

One area 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 is 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. We conservatively merged sets of conditions, which required loads to be speculatively executed, only when these loads could not introduce new exceptions. By performing more aggressive analysis, one should be able to detect more sets

Copyright © 2003 John Wiley & Sons, Ltd.
Prepared using speauth.cls

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 $\leq$ operator. If either $a[i]$ or $a[i+1]$ is negative, then the merged condition will fail and the original conditions will be tested.

Past studies on control CPR have not investigated the impact on branch prediction [12, 13]. One could more thoroughly investigate the effect condition merging has on branch misprediction to discover the reasons why additional mispredictions sometimes occur. It may be possible to perform additional optimizations that may reduce the number of mispredictions.

We also found that the merging of one set of conditions may 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 one may be able to infer new path frequency measurements for these duplicated paths.

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.

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. We showed a reduction in the number of branches executed, by an approximate average of 16%, while on average, there were about 6% fewer instructions executed. Execution time for the benchmark programs was reduced by roughly 3%, and the average fetch cost decreased by almost 5%.

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

Copyright © 2003 John Wiley & Sons, Ltd.
Prepared using speauth.cls

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.

REFERENCES

1. F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, Englewood Cliffs, NJ, USA, 1971. Transformations.
2. Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.
3. Manuel E. Benitez and Jack W. Davidson. A portable global optimizer and linker. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 329–338, Atlanta, GA, USA, June 1988. ACM Press.
4. Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Interprocedural conditional branch elimination. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 146–158, New York, June 15–18 1997. ACM Press.
5. 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.
6. J. J. Dongarra and A. R. Hinds. Unrolling loops in FORTRAN. *Software, Practice and Experience*, 9(3):219–226, March 1979.
7. Chris W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Pub. Co., Reading, MA, USA, 1995.
8. Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimiser and the GNU C compiler. In Christopher W. Fraser, editor, *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 341–352, San Francisco, CA, June 1992. ACM Press.
9. John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, second edition, 1996.
10. F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 56–66, La Jolla, CA, June 1995. ACM Press.
11. Kenneth A. Ross. Conjunctive selection conditions in main memory. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 109–120. ACM Press, 2002.
12. Michael Schlansker and Vinod Kathail. Critical path reduction for scalar programs. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 57–69, Ann Arbor, Michigan, November 29–December 1, 1995. IEEE Computer Society TC-MICRO and ACM SIGMICRO, IEEE Computer Society Press.
13. Michael Schlansker, Scott Mahlke, and Richard Johnson. Control CPR: A branch height reduction optimization for EPIC architectures. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 155–168, Atlanta, Georgia, May 1–4, 1999. ACM Press.
14. Alan Jay Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.
15. G.-R. Uh and D. B. Whalley. Coalescing conditional branches into efficient indirect jumps. In *Proceedings of the International Static Analysis Symposium*, pages 315–329, September 1997.
16. Nancy J. Warter, Scott A. Mahlke, Wen-Mei W. Hwu, and B. Ramakrishna Rau. Reverse If-Conversion. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 290–299, 1993.
17. Minghui Yang, Gang-Ryung Uh, and David B. Whalley. Improving performance by branch reordering. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 130–141, Montreal, Canada, 17–19 June 1998. ACM Press.
18. Minghui Yang, Gang-Ryung Uh, and David B. Whalley. Efficient and effective branch reordering using profile data. volume 24, pages 667–697, November 2002.

Copyright © 2003 John Wiley & Sons, Ltd.