Branch Elimination via Multi-Variable Condition Merging.*

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

[1] Florida State University, Tallahassee FL 32306, USA,
{kreahlin, whalley, yuan, engelen}@cs.fsu.edu
[2] Hamilton College, Clinton NY 13323, USA,
mbailey@hamilton.edu
[3] Boise State University, Boise ID 83725, USA,
uh@cs.boisestate.edu

Abstract. 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. 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 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 execution cycles, cause pipeline flushes when mispredicted, and can inhibit the application of other code-improving transformations. Techniques to reduce or eliminate the number of executed branches in the control flow have the potential for significantly improving performance.

Sometimes a set of conditions can be merged together. Consider Fig. 1(a), which shows conditions being tested in basic blocks 1 and 3. The wider transitions between blocks shown in figures in this paper represent the more frequently executed path, which occurs in Fig. 1 when conditions a and b are both satisfied. Figure 1(b) depicts the two conditions being merged together. If the merged

* This research was supported in part by NSF grants CCR-9904943, EIA-0072043, CCR-0208892, CCR-0105422, and by DOE grant DEFG02-02ER25543.

condition is true, then the original conditions need not be tested. Note merging conditions results in the elimination of both comparison and branch instructions. [4] The elimination of the fork in the control flow between blocks 2 and 4 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 b must be false if (a && b) is false and (a) is true. Thus, the branch in block 3 can be replaced by an unconditional transition to block 6. We call this the *breakeven* path since the same number of branches will be executed. We only apply the condition merging transformation when we estimate that the total instructions executed will be decreased.

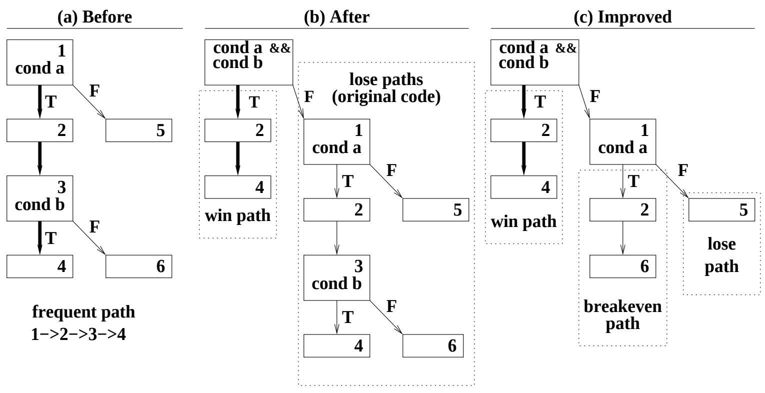

Fig. 1. Merging Three Conditions

In this paper we describe techniques to replace the execution of a set of two or more branches with a single branch. The performance improvements were obtained automatically by our compiler on a conventional scalar processor.

2 Related Work

There are numerous techniques that have been used to decrease the number of conditional branches executed. Loop unrolling has long been used to reduce execution of the conditional branch associated with a loop termination condition[1]. Loop unswitching moves a conditional branch with a loop-invariant condition before the loop and duplicates the loop in each of the two destinations

[4] Blocks 2 and 3 in Fig. 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.

of the branch[2]. Superoptimizers have been used to find a bounded sequence of instructions that have the same effect as a conditional branch[3]. Conditional branches have been avoided by using static analysis and code duplication[4, 5]. Conditional branches have been coalesced together into an indirect jump from a jump table[6]. Sequences of branches have been reordered to allow the sequence to be exited earlier, which reduces the number of branches executed[7, 8]. There has been 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[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[10]. 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 bypass branch[11, 12]. 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. There are several difference in the two approaches, including that our approach can be used on a conventional scalar processor without any ILP architectural support.

3 Merging Conditions

We have developed techniques that merge conditions where either single variables or multiple variables are compared to invariant values.

3.1 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 Fig. 2(a). The frequent path checks if the two variables are equal to zero. Figure 2(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. If the merged condition is not satisfied, then testing the condition $v2 == 0$ is unnecessary since it cannot be true.

The number of instructions required to merge conditions involving multiple variables is proportional to the number of different variables being compared. Figure 3 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).

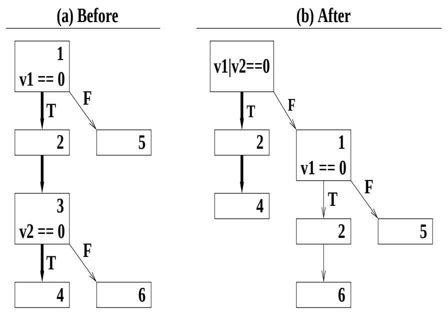

Fig. 2. Merging Conditions That Test If Different Variables Are Equal to Zero

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

Fig. 3. Code Generated for the Merged Condition That Checks If n Variables Are Equal to Zero

The top portion of table 1 shows how sets of conditions comparing multiple variables $(v1 \ldots vn)$ to 0 and -1 can be merged into a single condition. The first column shows the original conditions, the second column shows the conditions after merging, and the third column shows the static percentage each rule was applied, when merging sets involving multiple variables. Rule 1 has been illustrated in Figures 2 and 3. 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. Rules 5 and 6 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.

Additional opportunities for condition merging become available when sets of conditions that cross loop boundaries are considered. It would appear that in Fig. 4(a) there is no opportunity for merging conditions. However, Fig. 4(b) depicts that after loop unrolling there are multiple branches that use the same relational operator. Our compiler merges sets of conditions across loop iterations.

Table 1. Rules for Merging Conditions Comparing Multiple Variables Into a Single Equivalent Condition

Rule	Original Conditions	Merged Condition	% Applied
1	v1 == 0 &&..&& vn == 0	(v1 |..| vn) == 0	42.7%
2	v1 == 0 &&..&& vn == -1	(v1 |..| ~vn) == 0	0.0%
3	v1 < 0 &&..&& vn < 0	(v1 &..& vn) < 0	0.0%
4	v1 >= 0 &&..&& vn >= 0	(v1 |..| vn) >= 0	4.5%
5	v1 < 0 &&..&& vn >= 0	(v1 &..& ~vn) < 0	0.9%
6	v1 >= 0 &&..&& vn < 0	(v1 |..| ~vn) >= -1	0.0%
7	v1 $\neq$ 0 &&..&& vn $\neq$ 0	(v1 &..& vn) $\neq$ 0	18.2%
8	v1 $\neq$ c1 &&..&& vn $\neq$ cn	(v1 &..& vn) & ~(c1 |..| cn) $\neq$ 0	15.5%
9	v1 $\neq$ c1 &&..&& vn $\neq$ cn	~(v1 |..| vn) & (c1 |..| cn) $\neq$ 0	16.4%
10	v1 < c1 &&..&& vn < cn	(v1 |..| vn) $u<$ min (c1 ,.., cn)	1.8%

(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;
}
```

Fig. 4. Increasing Merging Opportunities by Unrolling Loops

3.2 Merging into a Sufficient Condition

Our compiler uses logical operations to efficiently merge conditions that compare multiple variables into a single sufficient condition. In other words, the success of the merged condition implies the success of the original conditions. However, the failure of the merged condition does not imply the original conditions were false.

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 1.

One may ask how often such conditions can be successfully merged in practice. Consider the code in Fig. 5(a), 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 5(b) shows code with a merged condition, testing the pointers. Note that if the merged condition is not satisfied, then the entire original set of branches still needs to be tested.

```
   (a) Before        (b) After
if (p1 != 0)      if (p1 & p2 != 0){
   w;                w;
else                 y;
   x;             }else{
if (p2 != 0)         if (p1 != 0)
   y;                   ...
else                 if (p2 != 0)
   z;                   ...
                  }
```

Fig. 5. Checking 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 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 1 depicts how this is accomplished, where $c1 \ldots cn$ 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.

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 1 depicts how this is accomplished, where $c1 \ldots cn$ 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.

Our compiler merges conditions comparing multiple variables to values that are not constants. Consider the original loop and unrolled loop shown in Fig. 6(a) and Fig. 6(b), respectively. It would appear there is no opportunity for merging conditions. However, x is loop invariant. Thus, the bits that are set in both $a[i]$ and $a[i+1]$ can be ANDed with $\tilde{}x$ to determine if the array elements are not equal to x, as shown in Fig. 6(c). The expression $\tilde{}x$ 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.

(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++;
   }
}
```

Fig. 6. Merging Conditions That Check If Multiple Variables Are Not Equal to Loop-Invariant Values

4 Applying the Transformation

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 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 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.

After merging sets of conditions, we apply a number of code-improving transformations in an attempt to improve the modified control flow. For instance, Fig. 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 Fig. 6(c) can be improved by applying loop-invariant code motion and common subexpression elimination.

5 Results

Condition merging on multiple variables reduced the average number of branches executed by 11.54% and reduced the total number of instructions executed by an average of 3.93%. Figure 7 displays the results of condition merging when the techniques that merged conditions involving single variables were combined with the techniques involving multiple variable conditions. All the benchmarks were run on the UltraSPARC II. When merging multiple variable conditions, the branches executed were reduced by an average of 11.54%. For space considerations the techniques that merge single variable conditions are not shown here but are discussed, in detail, in a technical report[13]. The average number of branches executed when using both techniques was reduced by 15.81% while the average number of instructions executed was reduced by 5.74%. Execution time was reduced an average of 3.43%. 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. *Sort* had unusually large benefits since most of the execution was spent in tight loops where conditions could be merged. The execution time for a few of the benchmarks got worse after the condition merging techniques were applied. The increase in execution time could be caused by several factors that could include inaccurate performance estimates due to subsequent optimizations and unforeseen effects due to multiple issue, instruction caching and branch prediction.

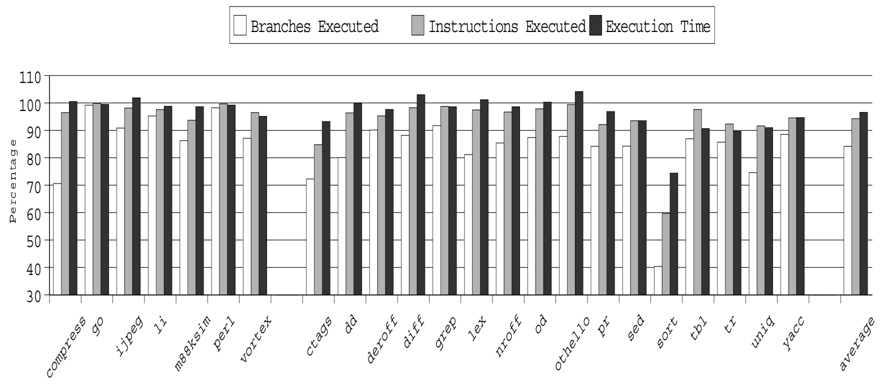

Fig. 7. Effect on Branches, Instructions, and Execution Time on the UltraSPARC II

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 instructions executed.

On many machines there is a pipeline stall associated with every taken branch. Merging conditions resulted in an average 27.90% 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 20.18%.

6 Future Work

One area to explore is the use of more aggressive analysis to detect when speculatively executed loads would not introduce exceptions. 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.

Past studies on control CPR have not investigated the impact on branch prediction[11, 12]. A more thorough investigation of the effect condition merging has on branch misprediction is needed 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 it may be possible to infer new path frequency measurements for these duplicated paths.

7 Conclusions

In this paper we described a technique 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, and execution time.

There contributions presented in this paper are twofold. First, we were able to merge conditions comparing multiple variables to constants or invariant values through innovative use of instructions on a conventional scalar processor. Second, we have shown that these techniques may still provide benefits even when the *win* path is not the most frequently executed. In these cases it may be possible to generate and merge conditions along a *breakeven* path.

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

1. Dongarra, J.J., Hinds, A.R.: Unrolling loops in FORTRAN. Software, Practice and Experience **9** (1979) 219–226
2. Allen, F.E., Cocke, J.: A catalogue of optimizing transformations. In Rustin, R., ed.: Design and Optimization of Compilers. Prentice-Hall, Englewood Cliffs, NJ, USA (1971) 1–30. Transformations.
3. Granlund, T., Kenner, R.: Eliminating branches using a superoptimiser and the GNU C compiler. In Fraser, C.W., ed.: Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation, San Francisco, CA, ACM Press (1992) 341–352
4. Mueller, F., Whalley, D.B.: Avoiding conditional branches by code replication. In: Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation, La Jolla, CA, ACM Press (1995) 56–66
5. Bodík, R., Gupta, R., Soffa, M.L.: Interprocedural conditional branch elimination. In: Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation. Volume 32, 5 of ACM SIGPLAN Notices., New York, ACM Press (1997) 146–158
6. Uh, G.R., Whalley, D.B.: Coalescing conditional branches into efficient indirect jumps. In: Proceedings of the International Static Analysis Symposium. (1997) 315–329
7. Yang, M., Uh, G.R., Whalley, D.B.: Improving performance by branch reordering. In: Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation, Montreal, Canada, ACM Press (1998) 130–141
8. Yang, M., Uh, G.R., Whalley, D.B.: Efficient and effective branch reordering using profile data. Volume 24. (2002) 667–697
9. Hennessy, J., Patterson, D.: Computer Architecture: A Quantitative Approach. Second edn. Morgan Kaufmann Publishers Inc., San Francisco, CA (1996)
10. Warter, N.J., Mahlke, S.A., Hwu, W.M.W., Rau, B.R.: Reverse If-Conversion. In: Proceedings of the Conference on Programming Language Design and Implementation. (1993) 290–299
11. Schlansker, M., Kathail, V.: Critical path reduction for scalar programs. In: Proceedings of the 28th Annual International Symposium on Microarchitecture, Ann Arbor, Michigan, IEEE Computer Society TC-MICRO and ACM SIGMICRO, IEEE Computer Society Press (1995) 57–69
12. Schlansker, M., Mahlke, S., Johnson, R.: Control CPR: A branch height reduction optimization for EPIC architectures. In: Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation, Atlanta, Georgia, ACM Press (1999) 155–168
13. Kreahling, W., Whalley, D.W., Bailey, M., Yuan, X., Uh, G.R., van Engelen, R.: Branch elimination by condition merging. Technical report, Florida State University (2003) URL: `http://websrv.cs.fsu.edu/research/reports/TR-030201.ps`.