CS253 Final Project
Conditional Branch Elimination
Preliminary Version

Eric Feigin (feigin@fas.harvard.edu)
Razvan Surdulescu (surdules@fas.harvard.edu)

April 27, 1998

Abstract

This paper describes a method for avoiding conditional branches by replicating code along known execution paths in a program. The first part describes the algorithms used to determine what branches are avoidable and under what circumstances; the second part describes a restructuring algorithm that replicates parts of the graph, taking advantage of the information gathered in the first part. Statistics indicate reductions in total as well as conditional branch instruction count.

There is little original work in this paper; most of it is based on [MuW92].

1 Introduction

1.1 What are avoidable branches?

Avoidable branches are conditional branches for which there exists some execution path through the program along which the result of the branch is known.

A simple example serves to illustrate the purpose of our project. Consider the simple control-flow graph presented in figure 1.

There is a conditional branch in node 1, branching on the condition `a>2` (i.e. if `a>5` is true, then the branch will take and jump to node 3, otherwise it will fall-through to node 2.) Node 2 contains a procedure call. Node 3 contains the instruction `a++`, which is the only instruction in node 3 which affects `a`. Nodes 4 and 5 contain no instructions that affect `a`.

On entrance to node 2 we knew that `a>5` was false, because to get to node 2, the branch in node 1 must have fallen-through. However, node 2 contains a procedure call, which may have some unknown effect on the value of `a`. Therefore, we know nothing at the exit of node 2 about whether `a>5` is true or false.

On entrance to node 3, we know that the condition `a>5` is true, because to get to node 3, the branch in node 1 must have taken. The only instruction in

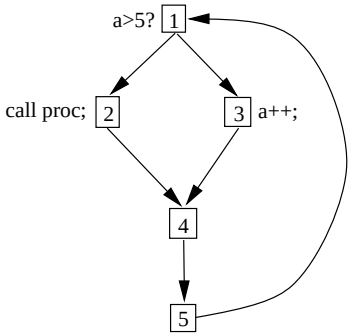

Figure 1: Original CFG

node 3 that affects `a` is `a++`. The instruction `a++` has the effect of increasing `a` (assuming no overflow–an issue that we will address later.) Thus, after the instruction `a++` executes, the condition `a>5` will remain true; at the exit of node 3 we know that the condition `a>5` is true, because it was true on entrance to node 3, and no instruction in node 3 changed this fact. Therefore, at the exit to node 3 *we still know that the conditional branch in node 1 will take.*

Nodes 4 and 5 do not affect `a`. So, when node 5 jumps back to node 1, we know that the branch in node 1 will take if we reached node 1 along the execution path $1 \to 3 \to 4 \to 5 \to 1$. Thus the branch in node 1 is an *avoidable branch.*

1.2 How can avoidable branches be eliminated?

With the control flow graph shown above, we can't take advantage of the information we have, because, when we reach node 1, we don't know what execution path we took to get there.

However, by using code duplication, we can restructure the control flow graph to avoid this branch along the execution path where its result is known.

We first need to split node 4 into two identical copies–one (node 4a) whose predecessor is node 2, the other (node 4b) whose predecessor is node 3. Likewise, we duplicate node 5 into a node 5a whose predecessor is node 4a and a node 5b whose predecessor is 4b.

We have now separated the two divergent execution paths. At the exit of node 4a, we know nothing about the branch in node 1, whereas at the exit of node 4b we know the branch in node 1 will still take. This is the point of duplicating nodes: we can have node 5a jump back to our original node 1, which contains the branch, since we still have to check the branch condition in this case (our execution path made the condition unknown.) However, node 5b can jump to a new copy of node 1 (called node 1b) that *does not contain a branch* since we know that we reached node 1b along an execution path where we know the branch to be taken and, therefore, we can simply eliminate the branch and

replace it with a jump to the branch's take target (in this case, node 3.)

The restructured control-flow graph looks like this (figure 2):

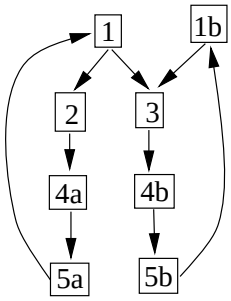

Figure 2: Restructured CFG

We have now eliminated the branch in node 1 along a particular execution path. There are clear drawbacks with increased code size (which we will discuss in more detail later); despite that, this is often a worthwhile optimization because, we remove the branch *and* the comparison associated with it; additionally, branches are good to eliminate because they often have a large cpi[1] due to the time required to flush the instruction buffer if the hardware guesses the branch direction incorrectly.

1.3 Our approach to avoiding conditional branches

There at two steps to our approach. First, we analyze the code and determine which branches can be eliminated; additional state is computed, whenever possible, so as to predict *which direction* branches would go along certain paths in the program. Second, we use this information to duplicate blocks (and eliminate branches) along paths where we know the direction of eliminable branches.

Our approach is very similar in many ways to the approach in the original paper by Mueller and Whalley [MuW92] (henceforth abbreviated as MW.) The reason we decided to examine and implement their paper is because we feel we could do better in some respects:

1. We first discuss a shortcoming in MW which might produce buggy optimized code in some (rare) cases.

2. The heuristics described by MW in their paper for determining when (and how) branches can be avoided and their direction predicted, are rather poorly explained and structured. We present a much more modular and easily extendable method for performing intraprocedural branch analysis, based on a bit vector data flow analysis.

[1] number of cycles per instruction

3. The MW restructuring algorithm suffers from similar ambiguity. MW claim that the restructuring needs to be performed on an individual loop basis, while we don't feel that this is necessary (see *Future Work* section for details on this.)

4. Last but not least, we wanted to explore implementation details and difficulties for this optimization pass on the SUIF[2] platform. We can only hope that this work will be useful as well as used and extended later on.

2 Related Work

Please refer to the *Related Work* section in [MuW92] for all related preceding work.

[YoS94] performed a profile history based branch prediction and code transformation.

Bodik, Gupta and Soffa extended [MuW92] to perform not only intraprocedural, but interprocedural conditional branch analysis and elimination as well. [BoGuS97]

3 Motivation

Please refer to the *Motivation* section in [MuW92] for three very good examples.

We choose to expand on one of their examples in order to reveal what we consider a potential bug and comment on it. The example in question is the second one in [MuW92]:

Original code:

```
while (i < 100 || somecmd) {
    A;
    i++;
}
```

After restructuring:

```
while (i < 100) {
    A;
    i++;
}

while (somecmd) {
    A;
    i++;
}
```

Clearly, in most cases, the optimization will work correctly: rather than checking both conditions in the loop (once `i` goes over 100), we can check them successively and therefore have to perform roughly half as many comparisons (depending on the life span of `somecmd`).

[2]Stanford University Intermediate Form

However, what may just as well happen is that, while `somecmd` is being successively computed, `i` will overflow and come back down to 0 (or -2^{31}, depending on the type)–in any event, *less than 100*! This means that the loop may continue to run because the *first* condition is now true, and not the second one. In the restructured code, once the first condition is "exhausted", there is no way to go back to it, meaning that if it is indeed the case that the original code assumes overflow in its execution, the optimizer will not catch it and produce incorrect (optimized) code.

In general, it is rarely (if ever) the case that "correct" code will make such assumptions, but if it does, the optimizer will produce semantically different optimized code. We have not reached a conclusion on what may be a "fix" for this problem. One possible approach is to consider type information on the variables examined and try to see (in the branch analysis section) what may happen if overflow occurs upon various arithmetic operations.

4 Avoidable Branch Detection

4.1 Overview

The idea behind avoidable branch detection is twofold:

1. Computation of the information necessary to tell how a branch result is affected when the instructions of a particular node are executed. This information is necessary for the restructuring phase.

2. Computation of what branches are avoidable, so that the restructuring algorithm does not needlessly copy nodes in an attempt to avoid branches that do not meet the conditions of being avoidable (i.e. branches for which there is no path along which they are known.)

As previously mentioned, we have implemented the MW section on *Determining Whether Branches Can Be Avoided* as a forward bit-vector dataflow analysis.

The dataflow analysis will search all execution paths in an attempt to find branches for which there exists an execution path to the branch along which the branch result is known.

It is important to make a distinction between branches that are known to be taken and branches that are known to fall through. This is a distinction that MW alludes to, but never fully explains. The reason it is important to make this distinction is because certain instructions only affect the branch if the branch result is known to be one way or the other. Recalling the example in the introduction, `a++` keeps the branch on `a>5` known to be taken if it was originally known to be taken, because incrementing `a` preserves the condition that `a>5`. However, if this branch were known to fall through, the instruction `a++` would not preserve this condition, because incrementing `a` would not necessarily preserve the condition `a<5`. Thus it is not only important to know that the

branch result is known–it is also important that we know the direction of the branch.

Keeping this in mind, the goal of the dataflow analysis is to produce the following four sets for every node **n** in our control flow graph:

TAKEN_IN(n) : The set of all branches which could be known to take when we enter **n**.

FALLTHROUGH_IN(n) : The set of all branches which could be known to fall through when we enter **n**.

TAKEN_OUT(n) : The set of all branches which could be known to take when we exit **n**.

FALLTHROUGH_OUT(n) : The set of all branches which could be known to fall through when we exit **n**.

In order to compute the effect of certain execution paths on a branch, it is necessary to somehow determine the effects that the instructions on the path will have on the branch condition. To this end, we define the following four terms, for the effect an instruction **i** can have on a branch **b**:

1. **i** is said to **taken-generate b** if, after executing **i**, **b** is always known to take. For example, the instruction `a=7` taken-generates a branch whose take condition is `a>5`, since setting `a` to be `7` clearly makes the take condition `a>5` be true.

2. Analogously, **i** is said to **fallthrough-generate b** if, after executing **i**, **b** is always known to fall through. For example, the instruction `a=3` fallthrough-generates a branch whose take condition is `a>5`, since setting `a` to be `3` clearly makes the take condition `a>5` be false.

3. **i** is said to **taken-kill b** if, when we know **b** is going to take, executing **i** makes it so we don't necessarily still know that `b` is going to take. For example, the instruction `a--` taken-kills a branch whose take condition is `a>5`, since decrementing `a` might cause the take condition `a>5` to be false, even though we had known it to be true previously.

4. **i** is said to **fallthrough-kill b** if, when we know **b** is going to fall through, executing **i** makes it so we don't necessarily still know that `b` is going to fall through. For example, the instruction `a++` fallthrough-kills a branch whose take condition is `a>5`, since incrementing `a` might cause the take condition `a>5` to be true, even though we had known it to be false previously.

Note that one instruction can fall into more than one of these categories. For example, an instruction that taken-generates a particular branch must also necessarily fallthrough-kill it, so it belongs to both "classes".

It is fairly straightforward to extend these four categorizations to pertain to nodes rather than individual instructions. Thus, we associate the following four sets with each node **n** in our control flow graph.

TAKEN_GEN(n) : The set of all branches which are taken-generated by an instruction in **n** and are not subsequently taken-killed by any instruction in **n**.

FALLTHROUGH_GEN(n) : The set of all branches which are fallthrough-generated by an instruction in **n** and are not subsequently fallthrough-killed by any instruction in **n**.

TAKEN_KILL(n) : The set of all branches which are taken-killed by an instruction in **n** and are not subsequently taken-generated by any instruction in **n**.

FALLTHROUGH_KILL(n) : The set of all branches which are fallthrough-killed by an instruction in **n** and are not subsequently fallthrough-generated by any instruction in **n**.

Discussion of exactly how these sets are constructed is deferred until the next section of this paper.

Given these sets, however, we can now describe how the dataflow analysis will be performed. First, the TAKEN_GEN, FALLTHROUGH_GEN, TAKEN_KILL, and FALLTHROUGH_KILL sets are computed for each node. Then, the TAKEN_IN, FALLTHROUGH_IN, TAKEN_OUT, and FALLTHROUGH_OUT sets are created for each node, and initialized to be empty. Then, while there are any changes in any node's sets, the following updating functions are performed (in a standard, forward dataflow analysis manner) to each node **n**:

$$\begin{aligned}
\texttt{TAKEN_IN}(n) &= \bigcup_{x \in \text{pred(n)}} (\texttt{TAKEN_OUT}(x) \cup \{\text{branches in x which } \textit{take} \text{ to n}\}) \\
\texttt{FALLTHROUGH_IN}(n) &= \bigcup_{x \in \text{pred(n)}} (\texttt{FALLTHROUGH_OUT}(x) \cup \{\text{branches in x which } \textit{fallthrough} \text{ to n}\}) \\
\texttt{TAKEN_OUT}(n) &= (\texttt{TAKEN_IN}(n) - \texttt{TAKEN_KILL}(n)) \cup \texttt{TAKEN_GEN}(n) \\
\texttt{FALLTHROUGH_OUT}(n) &= (\texttt{FALLTHROUGH_IN}(n) - \texttt{FALLTHROUGH_KILL}(n)) \cup \texttt{FALLTHROUGH_GEN}(n)
\end{aligned}$$

Once this analysis has run to completion, all that remains is to determine which branches are avoidable (or, equivalently, which nodes contain avoidable branches). A node **n** contains an avoidable branch if it meets the following conditions:

1. **n** contains a conditional branch **b**.
2. **b** $\in$ TAKEN_IN(**n**) and **b** $\notin$ TAKEN_KILL(**n**), or **b** $\in$ FALLTHROUGH_IN(**n**) and **b** $\notin$ FALLTHROUGH_KILL(**n**). Basically, this condition stipulates that it's possible for us to know the result of the branch coming into **n**, and no instruction in **n** kills this result. (If there were such an instruction, no amount of control flow graph restructuring could possibly allow us to know the branch result before the branch is executed, because it wouldn't matter what execution path we'd taken to get to **n**.)

4.2 Computing the gen and kill sets

We will now discuss how we compute the `TAKEN_GEN`, `FALLTHROUGH_GEN`, `TAKEN_KILL` and `FALLTHROUGH_KILL` sets for each node **n**. The main problem is determining the effects of individual instructions on branches; once this information has been determined, it is fairly simple to construct the four sets for each node.

We acknowledge that the way in which our current implementation performs this analysis is by no means optimal–these issues will be touched upon in the *Future Work* section.

We classify all instructions into one of the following categories:

1. **unknown effect**: The instruction has an unknown effect on program state. It may change any number of registers, variables, or memory locations. A call is an example of this type of instruction. (This is the default classification for an instruction.)

2. **no effect**: The instruction does not change program state–i.e. it affects no registers, variables, or memory. A label is an example of this type of instruction.

3. **writes memory**: The instruction writes to memory. A store instruction is an example of this type of instruction.

4. **changes destination**: This instruction changes its destination register or variable in some unknown way. A `mov` instruction is an example of this type of instruction.

5. **sets destination**: The instruction sets its destination to a constant value. A load constant instruction is an example of this type of instruction.

6. **increases destination**: The instruction increases the value of its destination. Adding a positive constant to a register is an example of this type of instruction.

7. **decreases destination**: The instruction decreases the value of its destination. Subtracting a positive constant from a register is an example of this type of instruction.

We then classify branch conditions into the different types, and use the type of the branch to determine the effect a particular instruction will have on that branch. The four types of branches are as follows: [3]

1. **generic**: The branch condition is arbitrarily complex. This is the default classification for a branch condition.

   Generic branches are both taken-killed and fallthrough-killed by:

   (a) Any instruction which has an unknown effect.

[3] Note that use of the term *register* in these classifications can refer also to a variable or virtual register.

(b) Any instruction which affects (changes, sets, increases, or decreases) a destination which is an operand of the branch comparison.

(c) Any instruction which writes memory, if the branch condition performs any loads.

Generic branches are never taken-generated or fallthrough-generated by an instruction.

2. **register**: The branch condition is a single register, and therefore the branch condition depends on whether or not that register is equal to zero.

   Register branches are both taken-killed and fallthrough-killed by:

   (a) Any instruction which has an unknown effect.

   (b) Any instruction which changes, increases, or decreases the register used for the comparison.

   Register branches are taken-generated (and thus fallthrough-killed) by any instruction that sets the register to a non-zero value. [4]

   Register branches are fallthrough-generated (and thus also taken-killed) by any instruction which sets the register to be zero.

3. **register-register**: The branch condition compares two registers. `r1>r2` is an example of this type of branch condition.

   Register-register branches are both taken-killed and fallthrough-killed by:

   (a) Any instruction which has an unknown effect.

   (b) Any instruction which changes either register.

   Register-register branches are taken-killed (but not fallthrough-killed) by:

   (a) Any instruction which increases whichever register needs to be smaller for the branch to take.

   (b) Any instruction which increases whichever register needs to be larger in the for the branch to take.

   Analogously, register-register branches are fallthrough-killed (but not taken killed) by:

   (a) Any instruction which increases whichever register needs to be larger for the branch to take.

[4] This is actually somewhat of an oversimplification. It is not true that the branch will always take when the branch condition is true–there are also branches which take when their branch condition is *false*. But this kind of branch can be handled analogously to the way a true branch is handled, and thus, for purposes of clarity, we will only consider true branches throughout this section of the paper.

(b) Any instruction which decreases whichever register needs to be smaller for the branch to take.

(If the two registers need to be equal or not equal for the branch to take, any increase or decrease to one register will both taken-kill and fallthrough-kill the branch.)

Register-register branches are never taken-generated or fallthrough-generated in the current implementation.

4. **register-constant**: The branch condition compares a register and a constant. `r1>5` is an example of this type of branch condition.

   Register-constant branches are both taken-killed and fallthrough-killed by:

   (a) Any instruction which has an unknown effect.
   (b) Any instruction which changes the register.

   Register-constant branches are taken-killed (but not fallthrough-killed) by:

   (a) Any instruction which increases the register if the register needs to be smaller than the constant for the branch to take.
   (b) Any instruction which decreases the register if the register needs to be larger than the constant for the branch to take.

   Analogously, register-constant branches are fallthrough-killed (but not taken-killed) by:

   (a) Any instruction which decreases the register if the register needs to be smaller than the constant for the branch to take.
   (b) Any instruction which increases the register if the register needs to be larger than the constant for the branch to take.

   (If the register needs to be equal to or not equal to the constant for the branch to take, then increasing or decreasing the register both taken-kills and fallthrough-kills the branch.)

   Register-constant branches are taken-generated (and thus fallthrough-killed) by any instruction which sets the register to a constant which has the proper relation to the constant in the comparison in order to make the branch take.

   Register-constant branches are fallthrough-generated (and thus taken-killed) by any instruction which sets the register to a constant which has the proper relation to the constant in the comparison in order to make the branch fall through.

Once the effects of every instruction in a node on every branch in the program have been computed, the `TAKEN_GEN`, `FALLTHROUGH_GEN`, `TAKEN_KILL`, and `FALLTHROUGH_KILL` sets can be constructed according to the definitions in the previous subsection.

5 Graph Restructuring

5.1 An example

Consider the following piece of C code:

```
void main()
{
        int i, j, k, q;
        int nop;

        do {
          if (i > 0) {
            i = q;
          } else {
            if (j > 0) {
              nop = 0;
            } else {
              j = q;
            }
          }

          k++;
        } while (k > 0);
}
```

The code itself simply runs in an infinite loop. We exhibit it only because it has a very interesting flow graph, which we would like to analyze (see figure 3.)

The notation `>4` next to some of the nodes (such as node 6 for instance) means that the code in node 6 taken-kills and fallthrough-kills the branch in node 4.

Note that the branches in nodes 2 and 4 are avoidable, whereas the branch in node 9 is not (see the Introduction and the DFA section on how this information is obtained.)

The very basic skeleton of the restructuring algorithm is as follows:

1. Enumerate all the avoidable branches in the graph and mark their state as unknown; call this list of branches and states L.

2. Copy the root of the cfg, call it C and set its instate to be L.

3. Using the gen and kill sets from the data flow analysis (the information about which branches are affected by which node), compute the outstate of C, from its instate (that is, examine how this block of instructions affects the state of the avoidable branches upon exit from the block.)

4. Make copies of C's successors. Set their instate to be the outstate of C. Iterate through all successors. Call the current successor S.

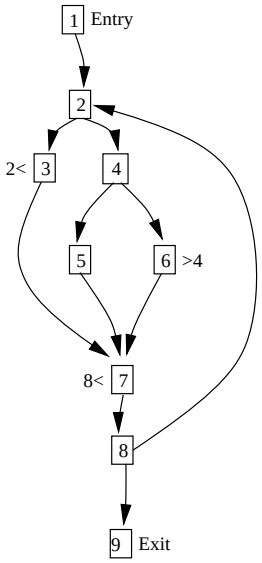

Figure 3: Original CFG

5. If C contains a conditional branch, mark the instate of S with the direction followed by that branch to get from C to S.

6. Look through the newly created graph so far: if there is another copy of S with the same instate as S *for all avoidable branches that can be reached from* S, then connect C to that node. Otherwise connect C to S and stick S in the graph.

7. Continue until there are no nodes left to analyze.

The restructured graph is in figure 4. The blocks whose box is dotted are instances of eliminated branches; their name (`4a` etc.) indicates the number of the blocks of which they are clones (both in the dummy graph as well as in the original graph.) The information outside certain blocks indicates the branches and the direction of those branches that was followed (and known) up to reaching the current block.

5.2 The algorithm

The fundamental difference between our algorithm and the one in [MuW92] is that we only compare instates (in the penultimate step above) for the avoidable branches that *are reachable* from the current node (the original paper compares the instate for *all* avoidable branches.)

The information required to determine if an (avoidable) branch is reachable from some given node can be easily precomputed in a simple data flow analysis

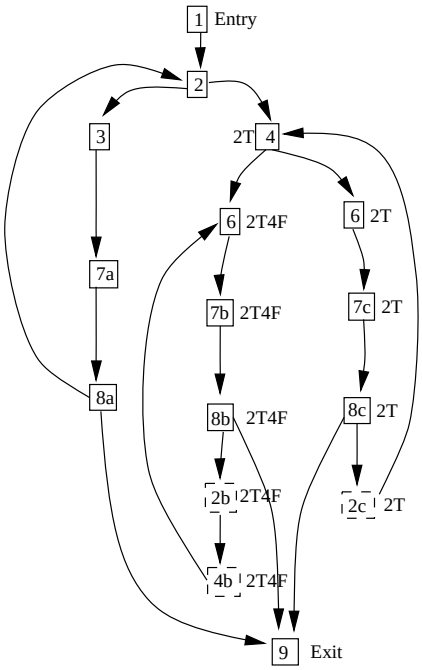

Figure 4: Restructured CFG

pass.

Here's the pseudocode for the algorithm. The algorithm is only executed if the graph has more than 2 nodes and there are avoidable branches in it. Variable names are, in general, indicative of their origin in the graphs (`c_` indicates a cloned node in the new graph, `o_` indicates a node in the original graph.)

```
LET o_root be the original root of the graph
COPY o_root into c_root
SET the instate of c_root to be a list of all avoidable branches,
    with their state set to UNKNOWN.

LET dummy_list be a list of nodes; put c_root on it

WHILE dummy_list is not empty
    LET c_elem be the first element on the list
    Compute the outstate of c_elem from its instate

    Try to remove a branch from the c_elem
        (if its instate contains a known state for the
         branch in c_elem)
```

```
    FOR each successor of c_elem
        LET the current successor be c_succ and make a copy
             of it in the new graph.

        SET c_succ's instate to be c_elem's outstate

        IF we got to c_succ by following a branch from c_elem,
             and that branch is avoidable, then set c_succ's
             instate to reflect the direction of that branch.
        ENDIF

        IF there is another node (in the newly created graph)
             with the same instate as c_succ
             for all avoidable branches reachable from c_succ
             THEN
                  delete the copy of c_succ and
                  use that node instead
             ELSE
                  keep the copy of c_succ in the graph and
                  attach it to the dummy_list
        ENDIF

        Mark c_succ as successor of c_elem
    ENDFOR
ENDWHILE
```

After the graph is restructured, standard cleanup routines are involved:

1. Optimize jumps: there could be blocks that consisted entirely of one avoidable branch; once removed, those blocks are now an unconditional branch: we want to replace any jump to that block with the ultimate destination of that block.

2. Merge block sequences: newly created blocks, especially the ones that come from blocks whose branches were remove, might be coalesced together if there are no branches in between them.

3. Remove dead code

6 Performance Results

First we ran the optimization pass on a set of tailor-made tests that we expected very good results out of. The tests are all in `./be/toy` (see section on *Source Tree Structure*).

Legend:

1. *avd* = avoidable

2. *cbr* = conditional branch
3. *instr* = instruction
4. *func* = function
5. *res* = restructured
6. *dyn* = dynamic

Here are the numbers we gathered; most of the programs consist of only one function, so there will be no difference between the global numbers and the per function numbers. In addition to counting conditional branches, we also counted unconditional branches (their number is relevant since we don't perform any code layout and the increase in nodes could produce an increase in unconditional branches due to the layout of blocks in the final code.)

First, static data:

Name	*Static avd cbr count*			*Static instr increase*	
	Total	Per Func	Per Res Func	Total	Res Func
toy1	1	1	1	-1.6%	-1.6%
toy2	1	1	1	60.2%	60.2%
toy3	2	2	2	-50.8%	-50.8%

Now, some dynamic numbers (positive is good, negative is bad):

Name	*Dyn* $\frac{\text{avd cbr}}{\text{cbr}}$	*Dyn instr decrease*		
		total	cbr	jmp
toy1	50.0%	33.2%	33.3%	∞%
toy2	33.3%	3.4%	50.0%	-33.6%
toy3	66.6%	59.8%	199.6%	∞%

Just for edification, here's the code for `toy3.c`:

```
void main()
{
  int i=0;
  int x=0, y=0, z=0;
  int zero=0, one=1;

  do {
    if(x) {
      if(z)
        z=zero;
      else
        zero=0;
    }
    else
      x=one;
```

```
    i++;
  } while (i < 1000);
}
```

Next, we tested some "real applications." Overall, the performance results weren't nearly as good. On the tests that we chose to run branch elimination on, the average dynamic conditional branch count went down, on average, by 1.7% (with a mode less than 0.5%)–a far cry from the [MuW92] average, which veered somewhere around 10%.

Let's look at the numbers first and then we'll talk about them. First, static data:

Name	*Static avd cbr count*			*Static instr increase*	
	Total	Per Func	Per Res Func	Total	Res Func
compress92	5	0.3	2.5	47.7%	55.3%
eqntott92	20	0.3	2.2	164.3%	182.25%
espresso92	58	0.16	1.48	14.3%	16.8%
gzip	32	0.32	2.28	71.6%	77.7%
li92	14	0.03	1.55	1.8%	3.2%
m88ksim95	38	0.13	1.52	13.2%	16.9%
sort	28	0.75	4	445.5%	430.0%
wc	1	0.11	0.11	1.66%	1.66%

Now, some dynamic numbers (positive is good, negative is bad):

Name	*Dyn* $\frac{\text{avd cbr}}{\text{cbr}}$	*Dyn instr decrease*				
		total	cbr	jmp	rdmem	wrmem
compress92	0.00%	0.00%	0.00%	-0.08%	0.00%	0.00%
eqntott92	0.03%	0.04%	0.02%	26.96%	0.00%	0.00%
espresso92	1.2%	-1.34%	0.42%	-126.71%	0.77%	0.08%
gzip	0.1%	0.33%	0.03%	19.31%	0.08%	0.08%
li92	5.1%	-0.54%	0.09%	-31.06%	-0.01%	0.00%
m88ksim95	0.6%	-1.02%	0.06%	-28.61%	0.00%	0.00%
sort	2.3%	-4.57%	1.19%	-504.35%	2.16%	0.55%
wc	11.4%	-2.13%	11.49%	-288.33%	0.00%	0.00%

There are a few items to note in this data:

1. The total dynamic instruction count went up in certain cases. Almost invariably, this is due to an increase in the dynamic `jmp` count due to poor layout in the restructured graph (see for instance the numbers for `espresso92`.) This is easily explained by the way we build the graph: the original graph is traversed and cloned in a breadth first fashion (whereas the block layout should follow a depth first traversal in order to decrease the number of `jmp`'s between successors.)

2. Another conjecture we had about the above was that there was more

register pressure due to code restructuring, and thus there more memory spills inserted by the register allocator. Therefore we gathered numbers for `rdmem` and `wrmem` instructions. There were almost no increases in these counts (with the one exception of `li92` which was insignificant) so it must be that the increase in `jmp`'s is the reason for dynamic instruction count increase.

3. The decrease in `cbr`'s is very small. In general, there is a strong correlation between the dynamic number of `cbr`'s and the ratio between dynamic number of *avoidable* `cbr`'s to total `cbr`'s (first column in the table above.) In many cases, the avoidable branches were hit very rarely during the execution of the program, so it's no surprise that we didn't see a strong reduction in their number when executing the optimized code.

Overall, despite the weak decrease in dynamic conditional branch count in "real applications", we are satisfied with the numbers. In most cases above, the poor performance is explained by the fact that avoidable branches are rarely hit along common execution paths in these programs. The overall dynamic instruction count can be readily improved by some rudimentary code layout, which would reduce the dynamic `jmp` count.

7 Future Work

7.1 DFA algorithms:

1. The algorithm would clearly benefit from a more advanced analysis in the construction of the `TAKEN_GEN`, `FALLTHROUGH_GEN`, `TAKEN_KILL`, and `FALLTHROUGH_KILL` sets.

   By considering only one instruction at a time, we clearly lose a lot of potential data. For instance, in our current implementation, the instruction `mov r1, r2` preceded by the instruction `ldc r2, 5`, will be analyzed merely as changing `r2`. It will not be recognized as setting `r2` to `5`, which it clearly does. One possible solution to this is trying to keep track of which registers are known to have constant values at a certain point in the program. Another possible solution is doing some sort of range analysis on the variables in the program, so that we can know something about what value a variable will have when it is assigned the value of another variable.

   Also, the generic branch type is much too limited, and doesn't really take advantage of all the information at our disposal: it assumes the branch to be killed by any instruction that modifies any of the operands of the comparison. Probably the correct way to do this sort of analysis is to construct the entire expression tree for the branch condition along each path, and see what we can then tell about how the branch condition is affected along certain paths. We note, however, that this sort of path analysis may be impossible in the current bit-vector dataflow analysis

model, and may make the dataflow analysis much slower, because of the exponential number of paths in a program.

2. A better representation of branches than is currently implemented would be very useful, both because of the problems stated above with the generic branch class, and because it would make branch subsumption calculations more versatile.

3. The current implementation of the dataflow analysis oversimplifies the known-unknown transitions somewhat. It is possible for an instruction to make the state of a branch go directly from being taken to being fallthrough (or vice-versa), a change which is not taken advantage of in our current implementation. For example, if the take-condition `r1==r2` is known to be true, incrementing `r1` makes it known to be false, and thus makes the branch be known to fall through. In our current implementation, an instruction that increments `r1` will taken-kill the branch but not fallthrough-generate it. We note that any solution to this problem will require reworking the nature of the bit-vector dataflow analysis (since the current system of gen and kill sets can't intelligently handle this type of effect), and the resulting increase in complexity may not be worth the effort, because it is doubtful that this will dramatically increase the number of avoidable branches.

7.2 Restructuring algorithms:

1. Loop Detection and Exploitation: MW claim that by performing this optimization on a *per-loop* basis (i.e. start with the innermost loop, optimize it, go to the next loop and optimize it but do *not* re-optimize the inner one, etc) will dramatically reduce code size. We feel that by ignoring loops and looking at the code globally, the performance improvements are greater. This, of course, comes at the increased cost of code blowup. It might be interesting to explore the other alternative as well and compare the two.

2. Code Repositioning: As this time, due to large increase in code size, there are very many unconditional jumps introduced in between successive blocks (there is no layout intelligence in the code at this point, so successive blocks in the cfg do not necessarily end up in consecutive positions in the final code.) In some pathological cases, the number of conditional branches eliminated is less than the number of unconditional jumps introduced. We are certain that this can be improved by a few simple layout heuristics.

3. Inlining: It would be interesting to experiment with some simple procedure inlining, and see how this affects the performance of branch elimination (determine whether more or less branches can be eliminated, inspect the growth in number of nodes and code size etc.)

8 SUIF Implementation Details

8.1 DFA algorithms

The DFA library is going through another revision; this section will be added as soon as the new version is checked in.

8.2 Restructuring algorithm

All the restructure code is in the file `./be/be.cc` (see the section on the *Source Tree Structure* for details.)

8.2.1 Hungarian notation

There was little attempt for a consistent (and cryptic) variable and function Hungarian notation. However, for ease, especially in the restructure algorithm, all `o` prefixes for variables (`o_root` etc.) indicate an original node in the original graph; all the `c` prefixes (`c_root` etc.) indicate clones in the restructured graph.

8.2.2 Node properties

One of the serious drawbacks encountered in writing the restructuring algorithm is that there are no annotations for the `cfg_node` class. As such, we had to implement a system of keeping track of and attaching certain properties to nodes (such as the instate, the outstate etc.)

Class `node_prop` implements properties for `cfg_node`s. The properties that can currently be attached are:

1. `branch_list * instate`
2. `branch_list * outstate`
3. `cfg_node_list * clone`

The `branch_list` is a linked list of `branch_state` objects, each of which can hold information about the id of a `cfg_node` that contains an avoidable branch and the direction of that branch at the instate or outstate of a block. The branch direction can be one of `TRUE` (taken), `FALSE` (fallthrough) or `UNKNOWN`.

The `clone` property is a list that keeps track of all the nodes that have been cloned out of this particular one. This is used to retrieve the "original" node in the "original" graph from a clone in the restructured graph.

The node properties live on a hash table, indexed by the `cfg_node`'s id; normally, the table achieves O(1) lookup times.

Relevant eponymous functions for accessing and modifying properties are:

1. `get_prop()`
2. `add_prop()`

3. `remove_prop()`
4. `connect_prop()` (connects two nodes that are clones of each other by adding relevant data to the `clone` property list for both of them.)

A related function is `get_o()` which obtains the *original* node from a *cloned* node by looking it up on the `clone` list in the clone's property.

8.2.3 New graph structure

In order to find a node with the same instate as a clone (see the end of the restructure algorithm and the function `find_same_instate()`), we somehow have to optimally the search through the newly restructured graph (which can get to be rather large in cases.) This is done by hashing all the nodes in the cloned graph by the index of the *original node* (in the original graph) where they came from. In this scenario, when looking up a (cloned) node, we simply have to search the bucket for other similar clones and see if there is one with the same instate.

8.2.4 Future work

It would be really nice to make `cfg_node` derived from `suif_object` so that annotations can be attached to it.

8.3 Source Tree Structure and Binaries

The source for the project is organized in the following tree:

./ab_test : Source for a debugging tool `ab_test` which displays, in detail, all the FALLTHROUGH and TAKE insets and outsets for all blocks of a program's cfg. Used to test the correctness of the dfa implementation; preserved for historical reasons only.

Executed with a low-suif `.sfl` file as input; produces output on `stderr`.

./ab_test/test : A few useful tests for `ab_test`.

./be : Source for the branch elimination pass `be`. The binary takes as input a low-suif `.sfl` file and outputs another low-suif file that's restructured according to the branch elimination algorithm.

The program understands either or both of two command line parameters:

1. `-count`: counts the total number of conditional branches and the number of avoidable branches in the program. It also annotates avoidable branches so that they can be dynamically counted later on. The *count* output is produced (or appended) in a file `LOG` in the current directory (if the file doesn't exist, it is created.)
2. `-restructure`: specifies whether to restructure the graph or not.

Additional scripts of interest in this directory are variants of `ascc-*`:

1. `ascc-S`: standard SUIF script
2. `ascc-be-S`: restructures the graph by executing `be -restructure` before `agen`
3. `ascc-be-halt-S`: restructures and instruments binaries (see `be-counts` below.)
4. `ascc-halt-S`: only instruments binaries without restructuring them (it does, however, execute `be -count` to gather statistics on the binary and dynamically count avoidable branches)

./be/sim : A rudimentary simulator for suif cfg trees that was used to implement a test the preliminary restructuring algorithm. Preserved for historical reasons only.

./be/toy : Simple, tailor-made, tests used to exhibit the amazing power of our optimization pass.

./dfa : Source for the extended dfa library, that contains the analysis for avoidable branches and reachable blocks. Compiles to a static and dynamic `libdfa.[a,so]`.

./doc : The `.tex`, `.fig`, `.eps` files that constitute this document. To compile the `.dvi` version of the paper, type `latex proj.tex`.

./haltsuif/be-counts : Source for a binary `be-counts` used to instrument the cfg of a program and prepare it for `halt`. The cfg is instrumented with information about the number of total instructions, conditional branches, avoidable conditional branches and unconditional jumps. The binary is executed as `be-counts file.af file.ae`.

./count : Source for a binary `count` which counts the number of effective machine instructions in a machine suif program (ignoring labels etc.) The program is executed as `count file.ae` and produces (or appends) its input to a file `LOG` in the current directory (if the file doesn't exist, it creates one.) Used to determine code size increase after restructuring.

All these binaries compile by simply typing `make` in their respective directories.

9 Conclusions

To be written when we're actually done with the project.

10 Acknowledgments

We would like to thank Prof. Mike Smith (`smith@deas.harvard.edu`) for the many hours spent with us discussing implementation approaches, optimizations and the many other issues that we encountered.

Just as many thanks go to Glenn Holloway (`holloway@eecs.harvard.edu`) for putting up with the flood of questions we had about SUIF, for his long and detailed replies and for helping us with many tools, among which a low SUIF version of the cfg library.

11 References

[MuW92] *"Avoiding Conditional Branches by Code Replication"* , Frank Mueller and David B. Whalley, *SIGPLAN Notices*, 30(6):56-66, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[YoS94] *"Improving the Accuracy of Static Branch Prediction Using Branch Correlation"*, Cliff Young and Michael D. Smith, *ASPLOS-VI, October 1994*

[BoGuS97] *"Interprocedural Conditional Branch Elimination"*, Ratislav Bodik, Rajiv Gupta, Mary Lou Soffa, *PLDI '97*