Using Conditional Branches to Improve Constant Propagation

Preston Briggs
Linda Torczon
Keith D. Cooper

CRPC-TR95533
April 1995

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Supported by Darpa through ONR grant N00014-91-J-1989

Using Conditional Branches to Improve Constant Propagation

Preston Briggs
Keith D. Cooper
Linda Torczon

Department of Computer Science
Rice University
Houston, Texas 77251–1892

1 Introduction

When writing or maintaining programs, programmers recognize the implications of conditionals or *guards* and use them to improve their code [7]. Consider this example:

do $x \neq y \rightarrow$
  if $x = 0 \rightarrow \langle$block $a\rangle$
  [] $x \neq 0 \rightarrow \langle$block $b\rangle$
  fi
od
$\langle$block $c\rangle$

In this code fragment (expressed in Dijkstra's notation), we are certain that $x = 0$ at the entrance of block a and that $x = y$ at the entrance of block c. We would like for a compiler to recognize the implications of guards and to take advantage of them to achieve better optimization. We are particularly concerned with the problems posed by machine-generated code; *e.g.*, the code resulting from inline substitution or extensive loop transformations [5, 10].

In their paper on global constant propagation,[1] Wegman and Zadeck mention that the quality of their results could be enhanced by taking advantage of constants implied by guards [11, Section 10]. This paper describes an efficient algorithm for discovering assertions of equality implied by conditional branches and discusses how to use these assertions to improve the results of Wegman and Zadeck's sparse constant propagation algorithms [12].

2 Assertions

While many kinds of assertions can be derived from the source code, we will limit our attention to assertions of equality: either the equality of two variables or the equality of a variable and a constant. Of course, many guards that arise in practice are more complex than a simple comparison for equality. In this example,

do $x = y$ **and** $p \rightarrow \langle$block $a\rangle$
od
$\langle$block $b\rangle$

we can assert that $x = y$ and that $p = \mathit{true}$ at the entrance of block a. On the other hand, no simple assertions are possible at the beginning of block b. Some guards yield no useful assertions. Consider the following case:

do $x = y$ **or** (p **and** q) $\rightarrow \langle$block $c\rangle$
od
$\langle$block $d\rangle$

This work has been supported by DARPA through ONR grant N00014-91-J-1989.

[1] Throughout this paper, "global" is used in the traditional sense of encompassing an entire routine.

This report was originally submitted as an extended abstract to the SIGPLAN 93 Conference on Programming Language Design and Implementation. It was submitted in November 1992; it was not accepted.

At the beginning of block c, we are uncertain which operand of the **or** was true. Similarly, at the entrance of block d, we are sure that $x \neq y$ (an unhelpful fact); but we cannot tell which of p and q was false.

So far, we have shown only high-level code; however, our optimizer, like many, works with a lower-level intermediate form. In these low-level forms, it may be difficult to recognize a guard. For example, the fragment

do $x = 1$ **and** $y = z \rightarrow$ ⟨block a⟩
od
⟨block b⟩

might be represented in our low-level intermediate code as

```
            LDI       1 ⇒ r1
label_1:    CMPeq     rx, r1 ⇒ r2
            CMPeq     ry, rz ⇒ r3
            AND       r2, r3 ⇒ r4
            BR        label_2, label_3, r4
label_2:    ⟨block a⟩
            JMP       label_1
label_3:    ⟨block b⟩
```

where r_1 and r_x are simply mnemonic replacements for register numbers. Note that there are no direct comparisons between variables and constants; `CMPeq` operates on registers. Similarly, the compare instructions are distinct from the branch instruction; therefore, we need to trace back from branches to the instructions responsible for the condition code. The compares and branches need not be adjacent in the code; indeed, they can be located in different basic blocks. For example, the comparison of y and z might be loop invariant. Thus, we must trace through the `AND` instruction to find the comparisons. This would create the assertions shown below.

```
            LDI       1 ⇒ r1
label_1:    CMPeq     rx, r1 ⇒ r2
            CMPeq     ry, r3 ⇒ r3
            AND       r2, r3 ⇒ r4
            BR        label_2, label_3, r4
label_2:    ASSERT    rx, r1 ⇒ r10
            ASSERT    ry, rz ⇒ r11
            ASSERT    true, r2 ⇒ r12
            ASSERT    true, r3 ⇒ r13
            ASSERT    true, r4 ⇒ r14
            ⟨block a⟩
            JMP       label_1
label_3:    ASSERT    false, r4 ⇒ r15
            ⟨block b⟩
```

The precise form of the assertions may be surprising. Consider the assertion at *label_2*. It indicates that $r_x = r_1$ in all the blocks dominated by the assertion; that is, all references in block a.[2] To make the equality explicit, we would also replace all dominated references to r_x or r_1 with r_{10}. The other form of assertion, shown at *label_3*, indicates that r_4 is *false* in all dominated blocks. Again, we would replace any dominated references to r_4 with r_{15}. Of course, the assertions could be represented in other forms; we have found this particular form most convenient for our work.

[2]Recall that block m *dominates* block n if and only if every path leading to n passes through m.

3 Finding Assertions

Our basic approach will be to trace back along use-def chains from conditional branches, through boolean operations, to comparisons. When an equality comparison is discovered, an assertion is inserted at the beginning of the appropriate successor block. After the assertions have been added, we walk forward over the dominator tree, renumbering registers as directed by the assertions.

3.1 Using Static Single Assignment

There are several difficulties that make the task of finding assertions more difficult. These problems are avoided through the use of a specific variant of the static single assignment form (SSA) [6]. In the next two sections, we give small examples illustrating the problems and show how to avoid them using SSA.

3.1.1 Naming Difficulties

Consider the following code fragment.

```
            CMPeq     r_x, r_y ⇒ r_p
            ADDI      1, r_x ⇒ r_x
            BR        label_1, label_2, r_p
label_1:    ⟨block a⟩
            JMP       label_2
label_2:    ⟨block b⟩
```

If we naively insert assertions based on the comparison,[3] we get the following result:

```
            CMPeq     r_x, r_y ⇒ r_p
            ADDI      1, r_x ⇒ r_x
            BR        label_1, label_2, r_p
label_1:    ASSERT    r_x, r_y ⇒ r_xy
            ⟨block a⟩
            JMP       label_2
label_2:    ⟨block b⟩
```

Note that the assertion at *label_1* is incorrect; by this point, r_x is no longer equal to r_y. The problem is that the compare refers to a different value than the value that reaches block a. To avoid this problem, we first convert the code into static single assignment form. After conversion to SSA, we get:

```
            CMPeq     r_x, r_y ⇒ r_p
            ADDI      1, r_x ⇒ r_x'
            BR        label_1, label_2, r_p
label_1:    ⟨block a⟩
            JMP       label_2
label_2:    ⟨block b⟩
```

This time, an assertion based on the comparison is correct.

```
            CMPeq     r_x, r_y ⇒ r_p
            ADDI      1, r_x ⇒ r_x'
            BR        label_1, label_2, r_p
label_1:    ASSERT    r_x, r_y ⇒ r_xy
            ⟨block a⟩
            JMP       label_2
label_2:    ⟨block b⟩
```

[3] For clarity, we omit assertions involving r_p and similar simple predicates in the rest of our examples.

Of course, block a will have no references to r_x after conversion to SSA form (they will have been modified to refer to $r_{x'}$); but there may be references to r_y. Thus, the assertion at *label_1* can still be useful in the event that r_x is determined to be constant.

3.1.2 Copy Instructions

A similar problem arises due to copies. Consider this example, representing a pair of nested **if** statements.

```
            BR      label_1, label_4, r_p
label_1:    MV      r_x ⇒ r_z'
            CMPeq   r_x, r_y ⇒ r_q
            BR      label_2, label_3, r_q
label_2:    ASSERT  r_x, r_y ⇒ r_xy
            ⟨block a⟩
            JMP     label_3
label_3:    ⟨block b⟩
            JMP     label_4
label_4:    PHI     r_z, r_z' ⇒ r_z''
            ⟨block c⟩
```

In this case, the code is already in SSA form and the assertion implied by the compare has been inserted at *label_2.* The difficulty arises during constant propagation. Suppose the compiler discovers that r_y is constant. Because the copy from r_x into $r_{z'}$ occurs before the assertion, the `MV` will not be rewritten. Thus, constant propagation has no way of recognizing that $r_{z'}$ contains the same constant as r_y.

To overcome this problem, we use a variant of the SSA form suggested by Zadeck [14]. This variation eliminates all copy instructions by folding them into existing ϕ-nodes during the renaming phase of SSA construction [6, Section 5.2]. In the case shown above, we would get the following code, where all mentions of $r_{z'}$ have been replaced by r_x to reflect the effect of the (deleted) copy instruction at *label_1*.

```
            BR      label_1, label_4, r_p
label_1:    CMPeq   r_x, r_y ⇒ r_q
            BR      label_2, label_3, r_q
label_2:    ASSERT  r_x, r_y ⇒ r_xy
            ⟨block a⟩
            JMP     label_3
label_3:    ⟨block b⟩
            JMP     label_4
label_4:    PHI     r_z, r_x ⇒ r_z''
            ⟨block c⟩
```

In the original paper, renaming is done in a routine called `SEARCH` [6, Figure 12]. Figure 1 gives a version of `SEARCH` modified to fold the effect of copy instructions into ϕ-nodes. The significant changes are in the handling of copy instructions. Rather than creating a new register number for the destination, we push the source register number, effectively replacing the destination in all dominated instructions. Since the intermediate values used in our low-level code are held in registers, the routine really does a renumbering of the registers. Rather than keeping stacks of counts, we stack actual register numbers. We number the registers from zero; therefore, the global variable *regs* is initially zeroed and is ultimately equal to the number of values (or definition points) in the routine. At the same time, we record the use-def chains to support tracing.

```
proc search(block)
    for each instruction inst in block do
        — renumber uses (except for φ-nodes)
        if inst.opcode = PHI →
            skip
        [] inst.opcode ≠ PHI →
            i ← 0
            do i < inst.refs →
                inst.ref[i] ← top(stack[inst.ref[i]])
                i ← i + 1
            od
        fi
        — push new register number for each definition
        if inst.opcode = MV →
            push(stack[inst.def], inst.ref[0])
        [] inst.opcode ≠ MV →
            push(stack[inst.def], regs)
            regs ← regs + 1
        fi
    od
    — renumber inputs in successor block's φ-nodes
    for each successor succ of block in the control-flow graph do
        j ← which_pred(succ, block)
        for each φ-node phi in succ do
            phi.ref[j] ← top(stack[phi.ref[j]])
        od
    od
    — walk over dominated blocks
    for each child of block in the dominator tree do
        search(child)
    od
    — restore stacks, delete copies, set defined registers, build UD chains
    for each instruction inst in block (in reverse order) do
        new ← pop(stack[inst.def])
        if inst.opcode = MV →
            delete inst
        [] inst.opcode ≠ MV →
            inst.def ← new
            def[new] ← inst
        fi
    od
end
```

Figure 1 The Modified `SEARCH` Procedure

```
proc trace_true(reg, block)
    inst ← def[reg]
    if inst.opcode = NOT →
        trace_false(inst.ref[0], block)
    [] inst.opcode = AND →
        trace_true(inst.ref[0], block)
        trace_true(inst.ref[1], block)
    [] inst.opcode = NOR →
        trace_false(inst.ref[0], block)
        trace_false(inst.ref[1], block)
    [] inst.opcode = CMPeq →
        assert_equal(inst.ref[0], inst.ref[1], block)
    [] inst.opcode ∉ {NOT, AND, NOR, CMPeq} →
        skip
    fi
    assert_true(reg, block)
end
```

```
proc trace_false(reg, block)
    inst ← def[reg]
    if inst.opcode = NOT →
        trace_true(inst.ref[0], block)
    [] inst.opcode = OR →
        trace_false(inst.ref[0], block)
        trace_false(inst.ref[1], block)
    [] inst.opcode = NAND →
        trace_true(inst.ref[0], block)
        trace_true(inst.ref[1], block)
    [] inst.opcode = CMPne →
        assert_equal(inst.ref[0], inst.ref[1], block)
    [] inst.opcode ∉ {NOT, OR, NAND, CMPne} →
        skip
    fi
    assert_false(reg, block)
end
```

Figure 2 Tracing Conditionals

3.2 Tracing through Boolean Operations

As mentioned in the discussion of an earlier example, it is possible to trace back through boolean operations to discover additional assertions; however, some care is required. Consider the instruction `AND` $r_1, r_2 \Rightarrow r_3$. If r_3 is true, then clearly both r_1 and r_2 must be true. On the other hand, if r_3 is false, either or both of the inputs may be false. Similar considerations apply to `OR`, `NAND`, and `NOR`.

Figure 2 illustrates a pair of mutually recursive routines that can be used to trace back from a conditional branch to discover assertions. For each conditional branch, we would call both *trace_true* and *trace_false*, each with the appropriate successor block; that is, we would pass the successor block along the true branch to *trace_true* and the other successor block to *trace_false*. The trace routines follow the use-def chains back as far as possible, placing assertions at the entrance of the successor block at every step.

We have not shown the routines *assert_equal, assert_true,* or *assert_false.* They simply insert the appropriate assertion at the entrance of a basic block. As each assertion is created, it defines a unique register number – the value of the global variable, *regs.* The variable *regs* is then incremented so that it continues to equal the number of definition points. To ensure that there is always a valid basic block to receive each assertion, we must sometimes split edges in the control-flow graph; that is, we insert an empty basic block along any edge whose source has more than one successor and whose destination has more than one predecessor. To simplify our optimizer passes, we routinely split all such edges during construction of the control-flow graph.

Note that no attempt is made to trace through ϕ-nodes. We have obtained a significant simplification of our algorithm through one of the basic properties of SSA: each use is reached by a single definition. The price of achieving this property is the introduction of ϕ-nodes. If we attempt to trace through ϕ-nodes, we must compare boolean formulas, potentially taking exponential time [1]. Note also that termination of our algorithm is assured since all cycles in the use-def chains must include a ϕ-node. DAGs can cause some extra assertions to be created; however, these do not affect correctness. They may be avoided without loss of asymptotic efficiency by marking instructions as they are visited.

3.3 Renumbering Registers

After inserting assertions, we must perform a second renumbering pass. Like *search, renumber* takes the form of a preorder walk over the dominator tree. It is outlined in Figure 3. Notice the similarities with *search*. In this case, only registers mentioned in assertions are renumbered. Simultaneously, def-use chains are accumulated for constant propagation.

The assertions are handled using a form of disjoint-set union. Effectively, the operands of an assertion are unioned to give a third set – the result of the assertion. This union holds in all the blocks dominated by the assertion, but must be undone after traversal of the current subtree is complete. Because of the need for backtracking and the short expected path length, we perform no path compression during *find.*[4]

4 Using Assertions

Wegman and Zadeck describe four algorithms for performing global constant propagation; we are concerned here with their *sparse* algorithms [12]. The two sparse algorithms, *sparse constant* and *sparse conditional constant,* both operate over the SSA graph, propagating lattice values forward along def-use chains. At each instruction (including ϕ-nodes), the inputs are evaluated and a new lattice value is computed and forwarded to each of its uses.

Wegman and Zadeck use the lattice *meet* operation to evaluate ϕ-nodes during constant propagation. The meet of two operands, $x \wedge y$, is their greatest lower bound in the lattice. For constant propagation, the meet operation is defined as:

$$
\begin{array}{lcll}
\text{any} & \wedge & \top & = \quad \text{any} \\
\text{any} & \wedge & \bot & = \quad \bot \\
c_i & \wedge & c_j & = \quad c_i \qquad \text{if } c_i = c_j \\
c_i & \wedge & c_j & = \quad \bot \qquad \text{if } c_i \neq c_j
\end{array}
$$

To evaluate assertions during propagation, we use the lattice *join* operation. The join of two operands $x \vee y$ is their least upper bound. For constant propagation, the join operation is defined as:

$$
\begin{array}{lcll}
\text{any} & \vee & \top & = \quad \top \\
\text{any} & \vee & \bot & = \quad \text{any} \\
c_i & \vee & c_j & = \quad c_i \qquad \text{if } c_i = c_j \\
c_i & \vee & c_j & = \quad \top \qquad \text{if } c_i \neq c_j
\end{array}
$$

The final case, $c_i \vee c_j$ where $c_i \neq c_j$, is interesting. Inconsistent inputs indicate that control will never reach this point; therefore, the answer is unimportant – a *don't care.* Furthermore, in the case of the sparse conditional constant algorithm, unreachable assertions will never be evaluated.

The join operation gives exactly the desired behavior: after both operands are available (*i.e.,* after they have been lowered from $\top$), if either is constant, then the result is constant.

5 Removing Assertions

After constant propagation, we must remove the ϕ-nodes and assertions to recover executable code. To remove a ϕ-node, we insert a copy in each predecessor block. The copy moves the value from the input register to the result register. Of course, we avoid inserting redundant copies (*i.e.,* a copy from r_x to r_x). Removing assertions is simple. Since both both operands must have the same value, we can replace a reference to the result of an assertion with the name of either operand. Once again, we use a preorder walk over the dominator tree to accomplish the required transformations.

[4] However, see Westbrook and Tarjan for discussion of asymptotically efficient approaches to the problem [13].

```
func find(x)
    do root[x] ≠ NULL →
        x ← root[x]
    od
    return x
end

proc renumber(block)
    for each instruction inst in block do
        — find uses and add to DU chains (except for φ-nodes)
        if inst.opcode = PHI →
            skip
        [] inst.opcode ≠ PHI →
            i ← 0
            do i < inst.refs →
                new ← find(inst.ref[i])
                inst.ref[i] ← new
                uses[new] ← uses[new] ∪ {inst}
                i ← i + 1
            od
        fi
        — perform unions for assertions
        if inst.opcode = ASSERT →
            root[inst.ref[0]] ← inst.def
            root[inst.ref[1]] ← inst.def
        [] inst.opcode ≠ ASSERT →
            skip
        fi
    od
    — find inputs in successor block's φ-nodes and add to DU chains
    for each successor succ of block in the control-flow graph do
        j ← which_pred(succ, block)
        for each φ-node phi in succ do
            new ← find(phi.ref[j])
            phi.ref[j] ← new
            uses[new] ← uses[new] ∪ {phi}
        od
    od
    — walk over dominated blocks
    for each child of block in the dominator tree do
        renumber(child)
    od
    — undo unions
    for each assertion assert in block do
        root[assert.ref[0]] ← NULL
        root[assert.ref[1]] ← NULL
    od
end
```

Figure 3 Renumbering After Adding Assertions

Note that this process can introduce many copies, since each ϕ-node is replaced by one or more copies. Furthermore, the *minimal* SSA produced by the efficient dominance frontier algorithm includes many dead ϕ-nodes – nodes whose results are never used [6]. To eliminate unneeded copies, we can either add a final phase to eliminate dead code or we can work with the *pruned* SSA form throughout [3]. This latter approach avoids introducing dead ϕ-nodes and thus avoids the problem.

In our implementation, we use the pruned SSA form. Advantages include quicker placement of ϕ-nodes (many less are required), lower memory requirements, and faster propagation of constants. The main disadvantage is the need for a phase to perform liveness analysis (dead code elimination is performed in a separate pass of our optimizer).

6 Summary

We have motivated and described each of the necessary phases; however, the overall picture may not be clear. Our implementation is organized in six phases.

1. Compute live registers to support construction of the pruned SSA form.
2. Build pruned SSA, folding copies into ϕ-nodes. At the same time, create use-def chains.
3. Find assertions using use-def chains.
4. Renumber to account for assertions. Simultaneously, create def-use chains.
5. Perform sparse conditional constant propagation using def-use chains.
6. Remove ϕ-nodes and assertions.

Thus, with a small amount of extra work, essentially an extra pass over the code in each of steps 3 and 4, we are able to find assertions that can be used to improve the results of Wegman and Zadeck's sparse constant propagation algorithms.

6.1 Limitations

Our approach does not find every possible assertion. For example, we miss some possibilities because we do not trace back through ϕ-nodes. Additionally, it is possible to construct certain complex examples requiring that the effect of assertions be merged for best results; however, these examples cannot be constructed without `goto` statements. More significantly, we limit ourselves to assertions concerning equality, ignoring the possibilities offered by inequalities. This seems consistent with the "philosophy" of constant propagation; working with inequalities would seem to fall within the scope of range propagation (see Section 6.3).

6.2 Experimental Results

We have implemented several variations of our technique in the context of Wegman and Zadeck's sparse conditional constant propagator. Unfortunately, our entire optimizer is undergoing a major overhaul (hence our reconsideration of constant propagation) and we are currently unable to collect adequate experimental results. In the near future, we expect to be able to measure performance on a variety of Fortran routines. We envision a comparison with Wegman and Zadeck's original approach. We would collect data comparing the costs of analysis and the effectiveness of the added assertions in real code. Interesting cost data would include the time required for our extra phases and the additional space required for assertions. We expect both costs to be small.

6.3 Related Work

Important work in this area has typically explored forms of range propagation [8, 9]. These works try to solve much harder problems and are significantly more complex. Our approach is a simple solution to a limited problem.

We continue to explore further uses for assertions. A natural possibility is to use them during value numbering [4, 2]; we some preliminary results in this area. A further possibility is finding an extension of value numbering, using assertions of inequalities, to accomplish a form of range propagation.

Acknowledgements

Cliff Click and Ken Kennedy contributed to this work through their interest and enlightened discussion. Tim Harvey helped with the implementation efforts.

References

[1] J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 177–189, January 1983.

[2] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, January 1988.

[3] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 55–66, January 1991.

[4] John Cocke and Jacob T. Schwartz. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University, 1970.

[5] Keith D. Cooper, Mary W. Hall, and Linda Torczon. An experiment with inline substitution. *Software – Practice and Experience*, 21(6):581–601, June 1991.

[6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[7] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[8] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[9] Harold Johnson. Data flow analysis for 'intractable' imbedded system software. *SIGPLAN Notices*, 21(7):109–117, July 1986. *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction.*

[10] Ken Kennedy and Kathryn S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing '90*, pages 407–416, November 1990.

[11] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 291–299, January 1985.

[12] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[13] Jeffery Westbrook and Robert E. Tarjan. Amortized analysis of algorithms for set union with backtracking. *SIAM Journal on Computing*, 18(1):1–11, February 1989.

[14] F. Kenneth Zadeck. Personal communication. Telephone conversation, September 1991.