Global Code Motion

Global Value Numbering

Cliff Click†
Hewlett-Packard Laboratories, Cambridge Research Office
One Main St., Cambridge, MA. 02142
cliffc@hpl.hp.com (617) 225-4915

1. Introduction

We believe that optimizing compilers should treat the machine-independent optimizations (*e.g.*, conditional constant propagation, global value numbering) and code motion issues separately.[1] Removing the code motion requirements from the machine-independent optimizations allows stronger optimizations using simpler algorithms. Preserving a legal schedule is one of the prime sources of complexity in algorithms like PRE [18, 13] or global congruence finding [2, 20].

We present a straightforward near-linear-time[2] algorithm for performing *Global Code Motion* (GCM). Our GCM algorithm hoists code out of loops and pushes it into more control dependent (and presumably less frequently executed) basic blocks. GCM is not *optimal* in the sense that it may lengthen some paths; it hoists control dependent code out of loops. This is profitable if the loop executes at least once; frequently it is very profitable. GCM relies only on dependences between instructions; the original schedule order is ignored. GCM moves instructions, but it does not alter the *Control Flow Graph* (CFG) nor remove redundant code. GCM benefits from CFG shaping (such as splitting control-dependent edges, or inserting loop landing pads). GCM allows us to use a simple hash-based technique for *Global Value Numbering* (GVN).

[1] This work has been supported by ARPA through ONR grant N00014-91-J-1989 and was done at the Rice University Computer Science Department

[1] Modern microprocessors with superscalar or VLIW execution already require global scheduling.

[2] We require a dominator tree, which requires $O(\alpha(n,n))$ time to build It is essentially linear in the size of the control flow graph, and very fast to build in practice.

SIGPLAN '95La Jolla, CA USA
© 1995 ACM 0-89791-697-2/95/0006...$3.50

GVN attempts to replace a set of instructions that each compute the same value with a single instruction. GVN finds these sets by looking for algebraic identities or instructions that have the same operation and identical inputs. GVN is also a convenient place to fold constants; constant expressions are then value-numbered like other expressions. Finding two instructions that have the same operation and identical inputs is done with a hash-table lookup. In most programs the same variable is assigned many times; the requirement for identical inputs is a requirement for identical *values*, not variable names. Hence GVN relies heavily on *Static Single Assignment* (SSA) form [12]. GVN replaces sets of value-equivalent instructions with a single instruction, however the single replacement instruction is not placed in any basic block (alternatively, think of GVN selecting a block at random; the resulting program is clearly incorrect). A following pass of GCM selects a correct placement for the instruction based solely on its dependence edges. Since GCM ignores the original schedule, GVN is not required to build a correct schedule. Finally, GVN is fast, requiring only one pass over the program.

1.1 Related Work

Loop-invariant code motion techniques have been around awhile [1]. Aho, Sethi, Ullman present an algorithm that moves loop-invariant code to a pre-header before the loop. Because it does not use SSA form significant restrictions are placed on what can be moved. Multiple assignments to the same variable are not be hoisted. They also lift the profitability restriction, and allow the hoisting of control dependent code. They note that lifting a guarded division may be unwise but do not present any formal method for dealing with faulting instructions.

Partial Redundancy Elimination (PRE) [18, 13] removes partially redundant expressions when profitable. Loop-invariant expressions are considered partially redundant, and, when profitable, will be hoisted to before

the loop. Profitable in this case means that no path is lengthened. Hoisting a computation that is used on some paths in a loop but not all paths may lengthen a path, and is not hoisted. GCM will hoist such control-dependent expressions. In general this is very profitable. However, like all heuristics, this method sometimes lengthens programs.

GCM will move code from main paths into more control dependent paths. The effect is similar to *Partial Dead Code Elimination* [16]. Expressions that are dead on some paths but not others are moved to the latest point where they dominate their uses. Frequently this leaves no partially dead code.

GVN is a direct extension of local hash-table-based value-numbering techniques [1, 11]. Since it is global instead of local, it finds all the redundant expressions found by the local techniques. Local value-numbering techniques have been generalized to extended basic blocks in several compilers.

GVN uses a simple bottom-up technique to build partitions of congruent expressions. Alpern, Wegman and Zadeck present a technique for building partitions top-down [2]. Their method will find congruences amongst dependences that form loops. Because GVN is a bottom-up technique it cannot find such looping congruences. However, GVN can make use of algebraic identities and constant folding, which A/W/Z's technique cannot. Also GVN runs in linear time with a simple one-pass algorithm, while A/W/Z use a more complex $O(n \log n)$ partitioning algorithm. Cliff Click presents extensions to A/W/Z's technique that include algebraic identities and conditional constant propagation [10]. While this algorithm finds more congruences than GVN it is quite complex. Compiler writers have to trade off the expected gain against the higher implementation cost.

PRE finds lexical congruences instead of value congruences [18, 13]. As such, the set of discovered congruences (whether they are taken advantage of or not) only partially overlaps with GVN. Some textually-congruent expressions do not compute the same value. However, many textually-congruent expressions are also value-congruent. Due to algebraic identities, many value congruences are not textually equal. In practice, GVN finds a larger set of congruences than PRE.

Rosen, Wegman and Zadeck present a method for global value numbering [19]. Their technique and the one presented here find a nearly identical set of redundant expressions. However, the algorithm presented here is simpler, in part, because the scheduling issues are handled by GCM. Also GVN works on irreducible graphs; the R/W/Z approach relies heavily on program structure.

For the rest of this section we will discuss the intermediate representation used throughout the paper. We present GCM next, in Section 2, both because it is useful as an optimization by itself and because GVN relies on it. In Section 3 we present our implementation of GVN. In Section 4 we present numerical results showing the benefits of the GVN-GCM combination over more traditional techniques (*e.g.*, PRE).

1.2 Program Representation

We represent programs as CFGs, directed graphs with vertices representing basic blocks, and edges representing the flow of control[3] [1]. Basic blocks contain sequences of instructions, elementary operations close to what a single functional unit of a machine will do. Instructions are represented as assignments from simple (single operator) expressions (*e.g.*, "$a := b + c$"). We require the program to be in SSA form [12].

We make extensive use of *use-def* and *def-use* chains. *Use-def* chains are explicit in the implementation. All variables are renamed to the address of the structure that represents the defining instruction; essentially variable names in expressions become pointers to the expression that defines the variable. This converts sequences of instructions into a directed graph, with the vertices being instructions and the edges representing data flow. It is not a DAG because SSA ϕ-functions are handled like other expressions and can have backedges as inputs. This is similar in spirit to the *value graph* [2] or an operator-level *Program Dependence Graph* [15]. In order to make the representation *compositional* [8], we need a ϕ-function input that indicates under what conditions values are merged. It suffices to make ϕ-functions keep a pointer to the basic block they exist in. We show an example in Figure 1.

Both GCM and GVN rely solely on dependences for correct behavior, the original schedule is ignored (although it may be used as an input to the code-placement heuristic). To be correct, then, all dependences must be explicit. LOADs and STOREs need to be threaded by a memory token or other dependence edge [3, 15, 23]. Faulting instructions keep an explicit control input. This input allows the instruction to be scheduled in its original basic block or after, but not before.

Most instructions exist without knowing the basic block they started in.[4] Data instructions are treated as not residing in any particular basic block. Correct behavior depends solely on the data dependences. An instruc-

[3] Our implementation treats basic blocks as a special kind of instruction similar to Ferrante, Ottenstein and Warren's *region* nodes [15]. This allows us to simplify several optimization algorithms unrelated to this paper. Our presentation will use the more traditional CFG.

[4] The exceptions are instructions which can cause exceptions: loads, stores, division, subroutine calls, etc.

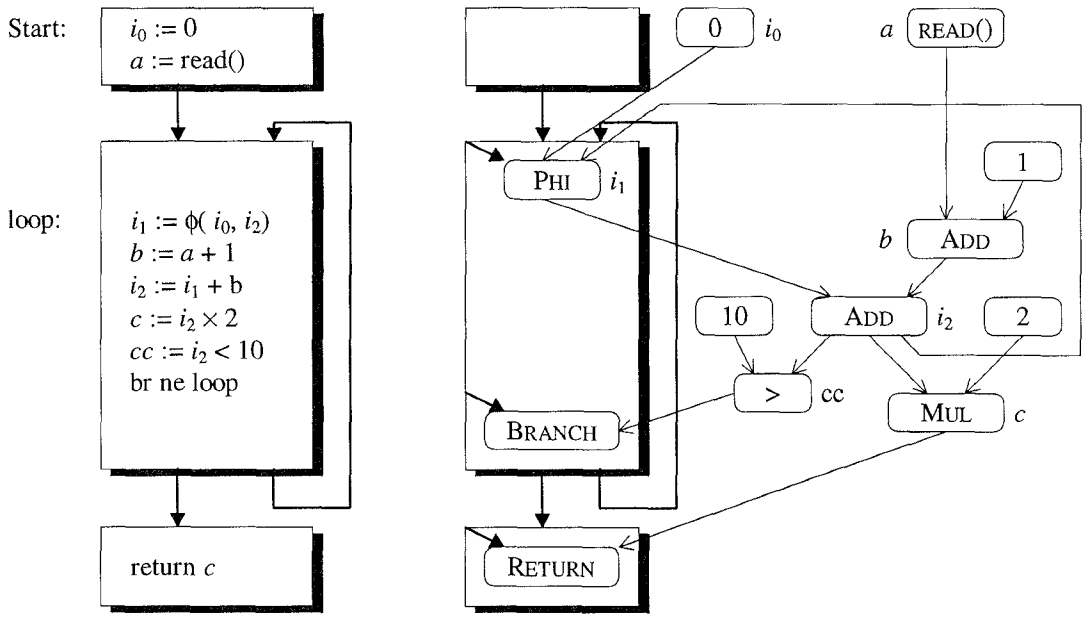

Figure 1 A loop, and the resulting graph to be scheduled

tion, the instructions that depend on it, and the instructions on which it depends exist in a "sea" of instructions, with little control structure.

The "sea" of instructions is useful for optimization, but does not represent any traditional intermediate representation such as a CFG. We need a way to serialize the graph and get back the CFG form. We do this with a simple global code motion algorithm.

The global code motion algorithm "schedules" the "free-floating" instructions by attaching them to the CFG's basic blocks. It must preserve any existing control dependences (divides, subroutine calls and other possibly faulting operations keep a dependence to their original block) and all data dependences. While satisfying these restrictions the scheduler uses a flexible and simple heuristic: place code outside as many loops as possible, then on as few execution paths as possible.

2. Global Code Motion

We use the following basic strategy:

1. Find the CFG dominator tree, and annotate basic blocks with the dominator tree depth.
2. Find loops and compute loop nesting depth for each basic block.
3. Schedule (select basic blocks for) all instructions early, based on existing control and data dependences. We place instructions in the first block where they are dominated by their inputs. This schedule has a **lot** of speculative code, with extremely long live ranges.
4. Schedule all instructions late. We place instructions in the last block where they dominate all their uses.
5. Between the early schedule and the late schedule we have a safe range to place computations. We choose the block that is in the shallowest loop nest possible, and then is as control dependent as possible.

We cover these points in more detail in the sections that follow. We will use the loop in Figure 1 as a running example. The code on the left is the original program. On the right is the resulting graph (possibly after optimization). Shadowed boxes are basic blocks, rounded boxes are instructions, dark lines represent control flow edges, and thin lines represent data-flow (data dependence) edges.

2.1 Dominators and Loops

We find the dominator tree by running Lengauer and Tarjan's fast dominator finding algorithm [17]. We find loops using a slightly modified Tarjan's "Testing flow graph reducibility" [21]. Chris Vick has an excellent write-up of the modifications and on building the loop tree [22]. We used his algorithm directly.

The dominator tree and loop nesting depth give us a handle on expected execution times of basic blocks. We use this information in deciding where to place instructions in the following sections. The running time of these algorithms is nearly linear in the size of the CFG. In practice, they can be computed quite quickly. We show the dominator tree and loop tree for our running example in Figure 2.

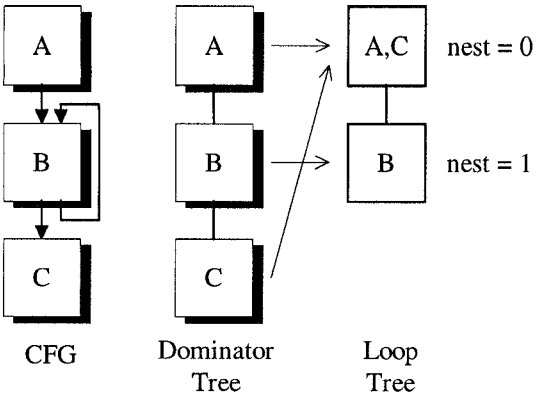

Figure 2 CFG, dominator tree and loop tree

2.2 Schedule Early

Our first scheduling pass greedily places instructions as early as possible. We place instructions in the first block where they are dominated by their inputs.[5] The only exception is instructions that are "pinned" into a particular block because of control dependences. These include PHI instructions, BRANCH/JUMP, STOP/RETURN instructions (these end specific blocks). Faulting instructions have an explicit control input, so they cannot be moved before their original basic block.[6] This schedule has a **lot** of speculative code, with extremely long live ranges.

We make a post-order DFS over the *inputs*, starting at instructions that already have a control dependence ("pinned" instructions). After scheduling all inputs to an instruction, we schedule the instruction itself. We place the instruction in the same block as its deepest dominator-tree depth input. If all input definitions dominate the instruction's uses, then this block is the earliest correct location for the instruction.

```
forall instructions i do
  if i is pinned then          // Pinned instructions remain
    i.visit := True;           // ... in their original block
    forall inputs x to i do    // Schedule inputs to pinned
      Schedule_Early( x );     // ... instructions

// Find earliest legal block for instruction i
Schedule_Early( instruction i ) {
  if i.visit = True then       // Instruction is already
    return;                    // ... scheduled early?
  i.visit := True;             // Instruction is being visited now
  i.block := root;             // Start with shallowest dominator
  forall inputs x to i do      //
    Schedule_Early( x );       // Schedule all inputs early
    if i.block.dom_depth < x.block.dom_depth then
      i.block := x.block;      // Choose deepest dominator input
}
```

[5] For some applications (*e.g.*, trace scheduling) the algorithm is not greedy enough. It does not move instructions past ϕ-functions.

[6] They can move after their original block, which only preserves partial correctness.

```
1.   int first := TRUE, x, sum0 := 0;
2.   while( pred() ) {
2.1    sum1 := φ( sum0, sum2 );
3.     if( first )
4.       { first := FALSE; x := ...; }
5.1    sum2 := sum1 + x;
5.2  }
```

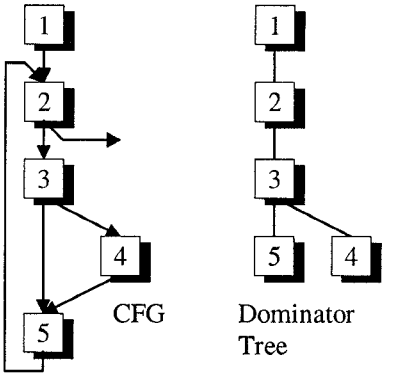

Figure 3 x's definition does not dominate it's use

One final problem: it is easy (and occurs in practice) to write a program where the inputs do not dominate all the uses. Figure 3 uses x in block 5 and defines it in block 4. The dominator building algorithm assumes all execution paths are possible; thus it may be possible to use x without ever defining it. When we find the block of the ADD instruction in line 5.1, we discover the deepest input comes from the assignment to x in block 4 (the other input, sum_1, comes from the PHI in block 2). Placing the ADD instruction in block 4 is clearly wrong; block 4 does not dominate block 5.

Our solution is to observe that translation to SSA leaves in some "extra" PHI instructions. We show the same code in Figure 4, but with the extra PHI instructions visible. In essence, the declaration of x_0 initializes it to the undefined value T.[7] This value is then merged with other values assigned to x, requiring PHI instructions for the merging. These PHI instructions dominate the use of x in block 5.

This problem can arise outside loops as well. In Figure 5, the PHI instruction on line 3 is critical. With the PHI instruction removed, the computation of $f(x)$ on line 4 will be scheduled early, in block 2. However, block 2 does not dominate block 4, and y is live on exit from block 4. In fact, there is no single block where the inputs to f dominate f's uses, and thus no legal schedule. The PHI instruction tells us that it's all right; the programmer wrote code such that it appears that we can reach f without first computing it's inputs.

[7] Pronounced "top"; this notation is taken from the literature on constant propagation.

```
1.   int first := TRUE, x0 := T, sum0 := 0;
2.   while( pred() ) {
2.1    sum1 := φ( sum0, sum2 );
2.2    x1 := φ( x0, x3);
3.     if( first )
4.       { first := FALSE; x2 := ...; }
5.     x3 := φ( x1, x2);
5.1    sum2 := sum1 + x3;
5.2  }
```

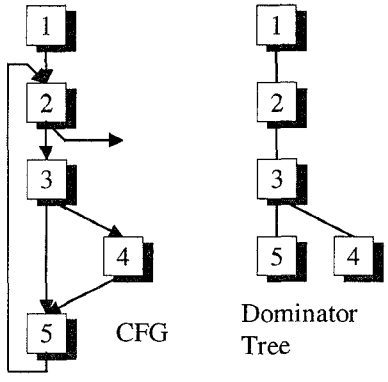

Figure 4 Now x's definitions dominate it's uses

These PHI instructions cannot be removed. They hold critical information on where the programmer has assured us that values will be available. In our implementation of GVN, the optimization that would remove this PHI instruction is the one where we make use of the following identity: $x \equiv \phi(x, \mathsf{T})$. We ignore this identity.

In Figure 6, scheduling early moves the add "$b := a + 1$" out of the loop, but it moves the multiply "$c := i_2 \times 2$" *into* the loop. The PHI instruction is pinned into the block. The add "$i_2 := i_1 + b$" uses the PHI, so it cannot be scheduled before it. Similarly, the multiply and the compare use the add, so they cannot be placed before the loop. The constants are all hoisted to the start of the program.

2.3 Schedule Late

Scheduling early hoists code too much. Much of the code is now speculative, with long live ranges. Witness the constants in Figure 6. They are all hoisted to the start of the program, no matter where their first use is. To correct this, we also schedule late: we find the last place in the CFG we can place an instruction. Between the early schedule and the late schedule will be a legal range to place the instruction.

We want to find the lowest common ancestor (LCA) in the dominator tree of all an instruction's uses. Finding the LCA could take $O(n)$ time for each instruction, but in practice it is a small constant. We use a post-order DFS over *uses*, starting from the "pinned" instructions. After visiting (and scheduling late) all of an instruction's children, we schedule the instruction itself. Before scheduling, the instruction is in its earliest legal block;

```
1.     int x0 := T, z0 := ...;
1.1    if( sometime() )
2.       x1 := ...;
3.     x2 := φ( x0, x1 );
3.1    if( sometime_later() )
4.       y := z0 + f(x2);
5.     ...y...
```

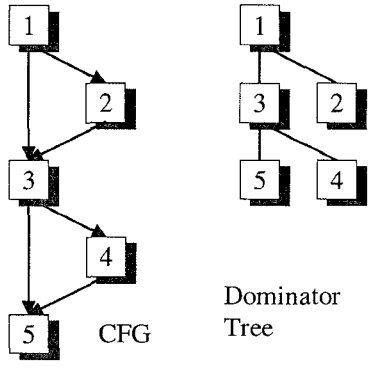

Figure 5 The PHI instruction on line 3 is critical

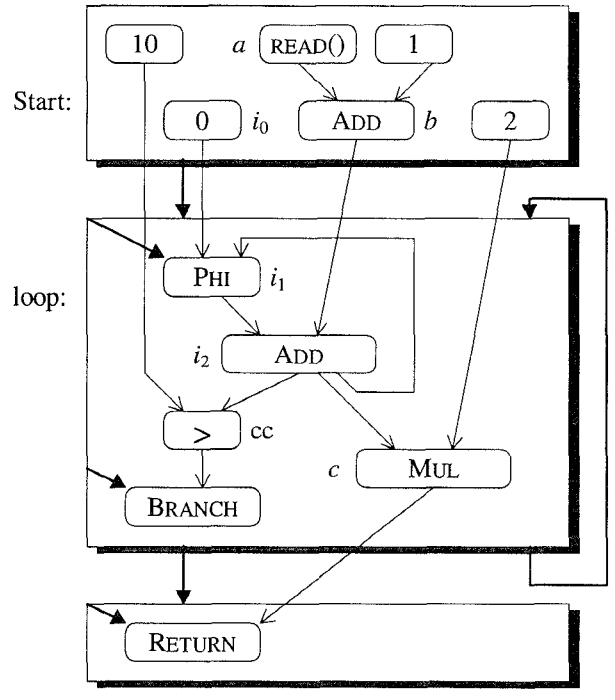

Figure 6 Our loop example, after scheduling early

the LCA of its uses represents its latest legal block. The earliest block dominates the latest block (guaranteed by the PHI instructions we did not remove). There is a line in the dominator tree from the latest to the earliest block; this represents a safe range where we can place the instruction. We can place the instruction in any block on this line. We choose the lowest (most control dependent) position in the shallowest loop nest.

For most instructions, uses occurs in the same block as the instruction itself. For PHI instructions, however, the use occurs in the previously block of the correspond-

ing CFG edge. In Figure 5 then, the use of x_0 occurs in block 1 (not in block 3, where the ϕ is) and the use of x_1 occurs in block 2.

```
forall instructions i do
  if i is pinned then          // Pinned instructions remain
    i.visit := True;           // ... in their original block
    forall uses y of i do      // Schedule pinned insts outputs
      Schedule_Late( y );

// Find latest legal block for instruction i
Schedule_Late( instruction i ) {
  if i.visit = True then       // Instruction is already
    return;                    // ... scheduled late?
  i.visit := True;             // Instruction is being visited now
  Block lca := NULL;           // Start the LCA empty
  forall uses y of i do {      // Schedule all uses first
    Schedule_Late( y );        // Schedule all inputs late
    Block use := y.block;      // Use occurs in y's block
    if y is a PHI then {       // ... except for PHI instructions
      // Reverse dependence edge from i to y
      Pick j so that the jth input of y is i
      // Use matching block from CFG
      use := y.block.CFG_pred[j];
    }
    // Find the least common ancestor
    lca := Find_LCA( lca, use );
  }
  ...use the latest and earliest blocks to pick final position
}
```

We use a simple linear search to find the least common ancestor in the dominator tree.

```
// Least Common Ancestor
Block Find_LCA( Block a, Block b ) {
  if( a = NULL ) return b;          // Trivial case
  // While a is deeper than b go up the dom tree
  while( a.dom_depth < b.dom_depth )
    a := a.immediate_dominator;
  // While b is deeper than a go up the dom tree
  while( b.dom_depth < a.dom_depth )
    b := b.immediate_dominator;
  while( a ≠ b ) {                  // While not equal
    a := a.immediate_dominator;     // ...go up the dom tree
    b := b.immediate_dominator;     // ...go up the dom tree
  }
  return a;                         // Return the LCA
}
```

2.4 Selecting a Block

In the safe range for an instruction we may have many blocks to choose from. The heuristic we use is to pick the lowest (most control dependent) position in the shallowest loop nest. Primarily this moves computations out of loops. A secondary concern is to move code off frequently executed paths into **if** statements. In Figure 5, if the computation of z_0 in block 1 is only used in block 4 then we would like to move it into block 4.

When we select an instruction's final position we affect other instructions' latest legal position. In Figure 7, when the instruction "$b := a + 1$" is scheduled before the loop, the latest legal position for "1" is also before the loop. To handle this interaction, we select instructions' final positions while we find the latest legal position. Here is the code to select an instruction's final position:

```
... Found the LCA, the latest legal position for this inst.
... We already have the earliest legal block.
  Block best := lca;          // Best place for i starts at the lca
  while lca ≠ i.block do      // While not at earliest block do...
    if lca.loop_nest < best.loop_nest then
      // Save deepest block at shallowest nest
      best := lca;
    lca := lca.immediate_dominator; // Go up the dom tree
  }
  i.block := best;            // Set desired block
```

The final schedule for our running example is in Figure 7. The MUL instruction's only use was after the loop. Late scheduling starts at the LCA of all uses (the last block in the program), and searches up the dominator tree for a shallower loop nest. In this case, the last block is at nesting level 0 and is chosen over the original block (nesting level 1). After the multiply is placed after the loop, the only use of the constant 2 is also after the loop, so it is scheduled there as well. The compare is used by the IF instruction, so it cannot be scheduled after the loop. Similarly the ADD is used by both the compare and the PHI instruction.

3. Global Value Numbering

Value numbering partitions the instructions in a program into groups that compute the same value. Our technique is a simple hash-table based bottom-up method [11]. We visit each instruction once, using a hash-table to determine if the instruction should be in the same partition as some previously visited instruction. Previously this method has been used with great success for local value-numbering. We extend to global value-numbering by simply ignoring basic block boundaries.

We use the following algorithm:

1) Visit every instruction using a *reverse postorder* (RPO) walk over the dependency edges.

2) At each instruction attempt to fold constants, use algebraic identities or find an equivalent instruction via a hash-table lookup. The hash table allows us to assign the same value-number to equivalent instructions in constant time. The hash table is not reset at basic block boundaries. Indeed, basic blocks play no part in this phase.

We use a RPO walk because, in most cases, the inputs to an instruction will have all been value-numbered before value-numbering the instruction itself. The exceptions arise because of loops. Because we do not reach a fixed point, running GVN a second time may be profitable. In our experience, one pass is usually sufficient.

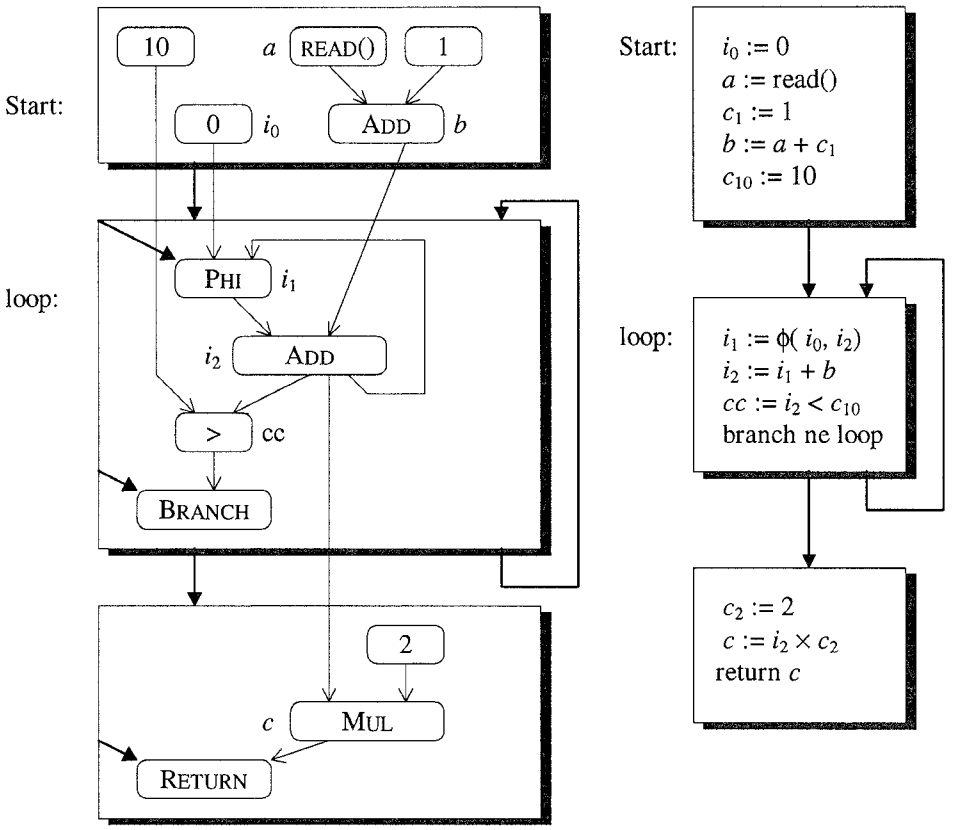

Figure 7 Our loop example, after scheduling late

3.1 Value-Numbering

Value-numbering attempts to assign the same value number to equivalent instructions. The bottom-up hash-table based approach is amenable to a number of extensions, which are commonly implemented in local value-number algorithms. Among these are constant folding and recognizing algebraic identities. When we value-number an instruction, we perform the following tasks:

1) Determine if the instruction computes a constant result. If it does, replace the instruction with one that directly computes the constant. Instructions that compute constants are value-numbered by the hash table below, so all instructions generating the same constant will eventually fold into a single instruction.

Instructions can generate constants for two reasons: either all their inputs are constant, or one of their inputs is a special constant. Multiplication by zero is such a case. There are many others and they all interact in useful ways.[8]

2) Determine if the instruction is an algebraic identity on one of its inputs. Again, these arise for several reasons; either the instruction is a trivial COPY, or one input is a special constant (add of zero) or both inputs are exactly the same (the ϕ or MAX of equal values). We replace uses of such instructions with uses of the copied value. The current instruction is then dead and can be removed.

3) Determine if the instruction computes the same value number as some previous instruction. We do this by hashing the instruction's operation and inputs (which together uniquely determine the value produced by the instruction) and looking in a hash table. If we find a hit, then we replace uses of this instruction with uses of the previous instruction. Again, we remove the redundant instruction. If the hash table lookup misses, then the instruction computes a new value. We insert it in the table. Commutative instructions use a hashing (and compare) function that ignores the order of inputs.

While programmers occasionally write redundant code, more often this arises out of a compiler's front end during the conversion of array addressing expressions into machine instructions.

3.2 Example

We present an extended example in Figure 8, which we copied from the Rosen, Wegman and Zadeck example [19]. The original program fragment is on the left and the SSA form is on the right. We assume every original

[8] By treating basic blocks as a special kind of instruction, we are able to constant fold basic blocks like other instructions. Constant folding a basic block means removing constant tests and unreachable control flow edges, which in turn simplifies the dependent ϕ-functions

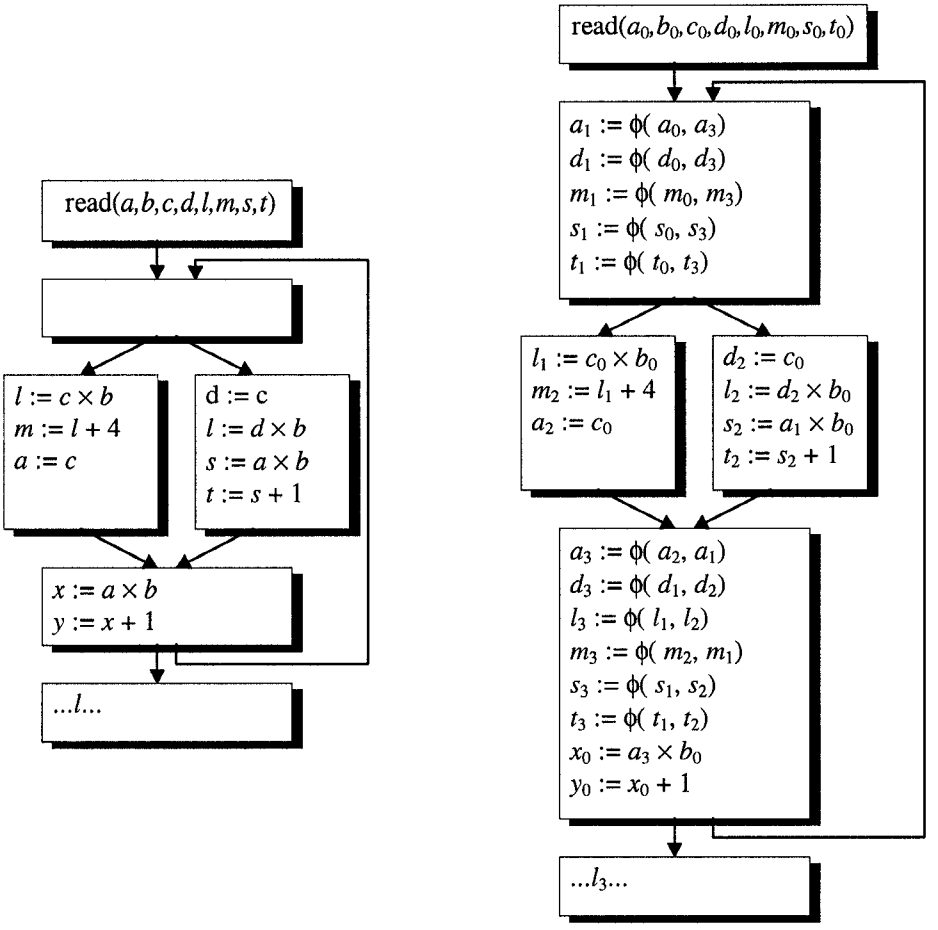

Figure 8 A larger example, also in SSA form

variable is live on exit from the fragment. For space reasons we do not show the program in graph form.

We present the highlights of running GVN here. We visit the instructions in reverse post order. The read() call obviously produces a new value number. The ϕ-functions in the loop header block (a_1, d_1, m_1, s_1, and t_1) are not any algebraic identities and they do not compute constants. They all produce new value numbers, as does "$l_1 := c_0 \times b_0$" and "$m_2 := l_1 + 4$". Next a_2 is an identity on c_0. We follow the def-use chains to remap uses of a_2 into uses of c_0.

Running along the other side of the conditional we discover that we can replace d_2 with c_0. Then "$l_2 := c_0 \times b_0$" has the same value number as l_1 and we replace uses of l_2 with l_1. The variables s_2 and t_2 produce new values. In the merge after the conditional we find "$l_3 := \phi(l_1, l_1)$", which is an identity on l_1. We replace uses of l_3 occurring after this code fragment with l_1. The other ϕ-functions all produce new values. We show the resulting program in Figure 9. At this point the program is incorrect; we require a round of GCM to move "$l_1 := c_0 \times b_0$" to a point that dominates its uses.

GCM moves the computations of l_1, m_2, x_0, and y_0 out of the loop. The final code, still in SSA form, is shown on the right. Coming out of SSA form requires a new variable name as shown on the right in Figure 9.

4. Experiments

We converted a large test suite into a low-level intermediate language, ILOC [6, 5]. The ILOC produced by translation is very naive and is intended to be optimized. We then performed several machine-independent optimizations at the ILOC level. We ran the resulting ILOC on a simulator to collect execution cycle counts for the ILOC virtual machine. All applications ran to completion on the simulator and produced correct results.

ILOC is a virtual assembly language. The current simulator defines a machine with an infinite register set and 1 cycle latencies for all operations, including LOADS, STORES and JUMPS. A single register can not be used for both integer and floating-point operations (conversion operators exist). ILOC is based on a load-store architecture; there are no memory operations other than LOAD and STORE. ILOC includes the usual assortment of logical, arithmetic and floating-point operations.

The simulator is implemented by translating ILOC into C. The C code is then compiled and linked as a na-

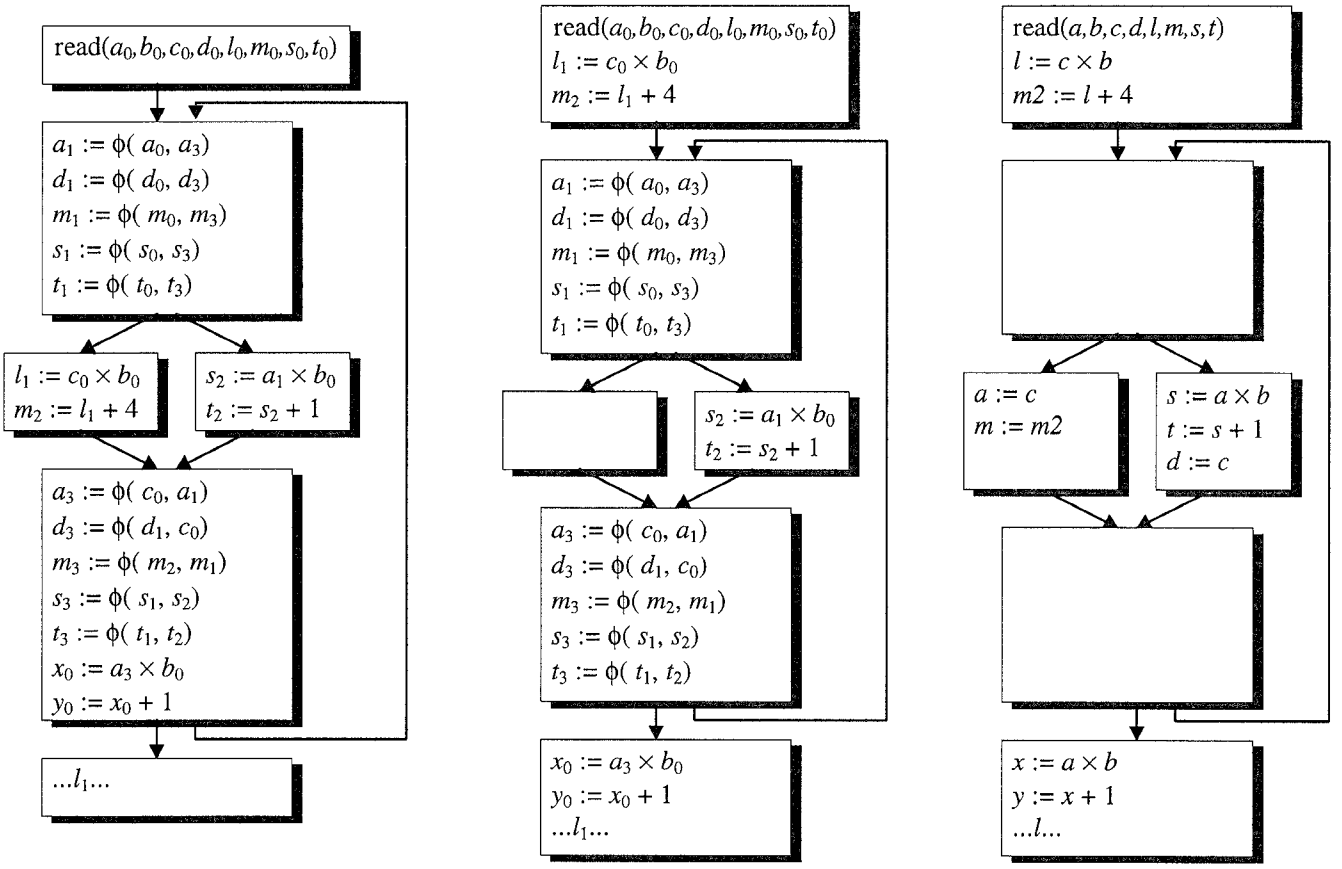

Figure 9 After GVN on the left, after GCM in the middle, back out of SSA form on the right

tive program. The executed code contains annotations that collect statistics on the number of times each ILOC routine is called and the dynamic cycle count.

The simulated ILOC clearly represents an unrealistic machine model. We argue that while the simulation is overly simplistic, it makes measuring the separate effects of optimizations possible. We are looking at machine-independent optimizations. Were we to include memory hierarchy effects, with long and variable latencies to memory and a fixed number of functional units we would obscure the effects we are trying to measure.

4.1 The Experiments

The test suite consists of a dozen applications with a total of 52 procedures. All the applications are written in FORTRAN, except for **cplex**. **Cplex** is a large constraint solver written in C. **Doduc**, **tomcatv**, **matrix300** and **fpppp** are from the Spec89 benchmark suite.[9] The remaining procedures come from the Forsythe, Malcom and Moler suite of routines [14].

All experiments start by shaping the CFG: splitting control-dependent edges and inserting loop landing pads. All experiments end with a round of *Dead Code Elimination* (DCE) [12]. Most of the phases use SSA form internally with a naive translation out of SSA at the end. Thus, most optimizations insert a large number of copies. We followed with a round of coalescing to remove the copies [4, 9]. This empties some basic blocks that only hold copies. We also split control-dependent blocks and some of those blocks go unused. We ran a pass to remove the empty blocks. Besides the optimizations already discussed, we also used the following:

CCP: *Conditional Constant Propagation* is an implementation of Wegman and Zadeck's algorithm [24]. It simultaneously removes unreachable code and replaces computations of constants with load of constants. After constant propagation, CCP folds algebraic identities that use constants, such as $x \times 1$ and $x + 0$, into simple copy operations.

GCF: *Global Congruence Finding* implements Alpern, Wegman, and Zadeck's global congruence finding algorithm [2]. Instead of using dominance information, *available expressions* (AVAIL) information is used to remove redundant computations. AVAIL is stronger than dominance, so more computations are removed [20].

PRE: *Partial Redundancy Elimination* implements Morel and Renvoise's algorithm. PRE is discussed in Section 1.1.

[9] At the time, our tools were not up to handling the entire Spec suite.

Reassociation: This phase reassociates and redistributes integer expressions allowing other optimizations to find more loop invariant code [6]. This is important for addressing expressions inside loops, where the rapidly varying innermost index may cause portions of a complex address to be recomputed on each iteration. Reassociation can allow all the invariant parts to be hoisted, leaving behind only an expression of the form "base+index×stride" in the loop.

Our experiments all follow this general strategy:

1. Convert from FORTRAN or C to naive ILOC. Reshape the CFG.
2. Optionally reassociate expressions.
3. Run either GVN-GCM or run GCF-PRE-CCP once or twice.
4. Clean up with DCE, coalescing and removing empty blocks.
5. Translate the ILOC to C with the simulator, then compile and link.
6. Run the compiled application, collecting statistics.

Variations on this strategy are outlined below in each experiment.

4.2 GVN-GCM *vs.* GCF-PRE-CCP

We optimized with the GVN-GCM combination and compared it to running one pass each of GCF, PRE and CCP. This represents an aggressive optimization strategy with reasonable compile times. Each of GCF, PRE and CCP can find some congruences or constants that GVN cannot. However, constants discovered with GVN can help find congruences and *vice-versa*. The separate passes have a phase-ordering problem; constants found with CCP cannot then be used to find congruences with PRE. In addition, GCM is more aggressive than PRE in hoisting loop invariant expressions.

The table in Figure 10 shows the results. The first two columns are the application and procedure name, the next column shows the runtime cycles for GVN-GCM. The fourth column is cycles for GCF-PRE-CCP. Percentage speedup is in the fifth column.

4.3 GVN-GCM *vs.* GCF-PRE-CCP repeated

We compared the GVN, GCM combination against two passes of GCF, PRE and CCP. We ran DCE after each set; the final sequence being GCF, PRE, CCP, DCE, GCF, PRE and CCP. As before we finish up with a pass of DCE, coalescing and removing empty blocks.

The sixth column in Figure 10 shows the runtime cycles for the repeated passes and column seven shows the percentage speedup over GVN-GCM. Roughly halve the gain over a single pass of GCF-PRE-CCP is lost. This indicates that the second pass of GCF and PRE could make use of constants found and code removed by CCP.

4.4 Reassociate, then GVN-GCM *vs.* GCF-PRE -CCP

We first reassociate expressions. Then we ran the GVN-GCM combination and compared it to running one pass of GCF, PRE and CCP. As before we finish up with a pass of DCE, coalescing and removing empty blocks.

In Figure 10, column eighth is the runtime cycle count for reassociating, then running the GVN-GCM combination. Column nine is reassociation, then GCF-PRE-CCP. Column ten is the percentage speedup. The average speedup is 5.9% instead of 4.3%. Reassociation provides more opportunities for GVN-GCM than it does for one pass of GCF, PRE and CCP.

4.5 Reassociate, then GVN-GCM *vs.* GCF-PRE -CCP repeated

We first reassociate expressions. Then we ran the GVN-GCM combination and compared it to running two passes of GCF, PRE and CCP. The final sequence being Reassociate, GCF, PRE, CCP, DCE, GCF, PRE and CCP. As before we finish up with a pass of DCE, coalescing and removing empty blocks.

Column eleven in Figure 10 shows the runtime cycles for the repeated passes and column twelve shows the percentage speedup over GVN-GCM. Again, roughly halve the gain over a single pass of GCF-PRE-CCP is lost. This indicates that the second pass of GCF and PRE could make use of constants found and code removed by CCP.

5. Conclusions

We have implemented two fast and relatively straightforward optimizations. We ran them both on a suite of programs with good results. By separating the code motion from the optimization issues we are able to write a particularly simple expression of *Global Value Numbering*. Besides finding redundant expressions, GVN folds constants and algebraic identities. After running GVN we are required to do *Global Code Motion*. Besides correcting the scheduled produced by GVN, GCM also moves code out of loops and down into nested conditional branches.

GCM requires the dominator tree and loop nesting depth. It then makes two linear-time passes over the program to select basic blocks. Running time is essentially linear in the program size, and is quite fast in practice. GVN requires one linear-time pass over the program and is also quite fast in practice. Together these optimizations provide a net gain over using GCF, PRE and CCP, and are very simple to implement.

Application	Routine	GVN-GCM	GCF-PRE-CCP		GCF-PRE-CCP		Reassociate GVN-GCM	Reassociate GCF-PRE-CCP		Reassociate GCF-PRE-CCP	
			Once	speedup	Repeated	speedup		Once	speedup	Repeated	speedup
doduc	debflu	689,059	898,775	23.3%	821,399	16.1%	689,931	929,271	25.8%	824,189	16 3%
doduc	paroi	536,685	637,325	15.8%	560,180	4.2%	513,005	689,495	25.6%	559,255	8.3%
rkf45	rkfs	56,024	66,254	15.4%	66,254	15.4%	58,945	61,654	4.4%	62,526	5.7%
fpppp	gamgen	134,227	158.644	15.4%	151,039	11.1%	79,057	102,630	23.0%	83,032	4.8%
doduc	prophy	553,522	646,101	14 3%	590,164	6.2%	617.902	834,681	26.0%	648,916	4.8%
doduc	pastem	625,336	721,850	13.4%	654,365	4 4%	598,399	769,328	22.2%	651,012	8.1%
doduc	ddeflu	1,329,989	1,485,024	10.4%	1,434,149	7.3%	1,332,394	1,541,800	13.6%	1,474,460	9 6%
doduc	yeh	342,460	379,567	9 8%	379,567	9 8%	342,460	362,090	5 4%	354,328	3.3%
doduc	debico	479,318	529,876	9.5%	497,174	3.6%	440,884	552,856	20.3%	508,216	13.2%
doduc	deseco	2,182,909	2.395,579	8.9%	2,249,694	3.0%	2,175,324	2,671,044	18.6%	2,324,804	6.4%
doduc	integr	389,004	426,138	8.7%	383,403	-1 5%	401,214	487,928	17.8%	391,913	-2.4%
doduc	cardeb	168,165	183,520	8.4%	157,620	-6.7%	159,840	169,090	5.5%	151,330	-5 6%
doduc	coeray	1,487,040	1,622,160	8 3%	1,622,160	8 3%	1,487,040	1,606,384	7.4%	1,606,384	7.4%
doduc	bilan	536,432	580,878	7.7%	545,127	1.6%	529,957	686.143	22.8%	582,127	9 0%
matrix300	sgemv	496,000	536,800	7.6%	536,400	7.5%	400,000	401,200	0 3%	400,400	0 1%
rkf45	rkf45	1,475	1,575	6.3%	1.575	6.3%	1,475	1,475	0.0%	1.475	0.0%
doduc	inithx	2,602	2,776	6.3%	2,606	0.2%	2,558	3,091	17 2%	2,591	1 3%
doduc	dcoera	921.068	981,505	6.2%	981,505	6.2%	921,068	975,215	5.6%	975,215	5.6%
fpppp	twldrv	76,291,270	81,100,769	5.9%	80,988,348	5.8%	79,039,949	81,989,205	3.6%	81,464,531	3.0%
seval	spline	882	937	5 9%	937	5.9%	801	837	4.3%	829	3 4%
doduc	heat	578,646	613,800	5 7%	613,800	5.7%	560,418	610,080	8.1%	608,964	8.0%
doduc	orgpar	23,692	24,985	5.2%	24,060	1 5%	21,287	22.950	7.2%	21,840	2.5%
doduc	drepvi	680,080	715,410	4.9%	689,325	1.3%	697,840	741,055	5.8%	720,890	3.2%
fpppp	fmtgen	926,059	973,331	4 9%	966,891	4 2%	923,419	960,776	3.9%	952,581	3.1%
doduc	repvid	486,141	507,531	4 2%	492,837	1.4%	453,033	514,041	11 9%	496,557	8.8%
doduc	inideb	863	898	3.9%	838	-3.0%	735	877	16.2%	844	12 9%
urand	urand	550	563	2.3%	563	2 3%	555	564	1.6%	564	1.6%
doduc	drigl	295,501	301,516	2.0%	298,926	1.1%	297,334	347,379	14.4%	347,379	14.4%
cplex	xload	3,831,744	3,899,518	1.7%	3,899,518	1.7%	3,831,744	3,899,518	1 7%	3,899,518	1.7%
doduc	colbur	751,074	763,437	1 6%	765,868	1 9%	761,478	804,580	5.4%	797,365	4.5%
matrix300	sgemm	8,664	8,767	1.2%	8,666	0 0%	7,928	8,022	1.2%	7,920	-0 1%
rkf45	fehl	134.640	135,960	1.0%	134,640	0.0%	130,416	137,808	5.4%	137,808	5.4%
cplex	xaddrow	16,345,264	16,487,352	0.9%	16,499,836	0.9%	15,539,173	16,487,352	5 8%	16,499,836	5.8%
doduc	si	9,924,360	9,981,072	0.6%	9,981,072	0 6%	9,300,548	9,470,684	1 8%	9.470,684	1 8%
cplex	xielem	19,738,262	19,791,979	0.3%	19,904,514	0.8%	19,755,998	19,718,353	-0.2%	19,956,683	1.0%
fpppp	fmtset	1,072	1,074	0.2%	1,074	0 2%	948	952	0.4%	952	0.4%
doduc	supp	2,564,382	2,567,544	0.1%	2,567,544	0.1%	2,564,382	2,561,220	-0 1%	2,567,544	0 1%
doduc	iniset	56,649	56.684	0.1%	56,655	0.0%	47,316	47,498	0.4%	47,469	0 3%
fpppp	fpppp	26,871,102	26,883,090	0 0%	26.877,096	0.0%	26,871,102	26,871,102	0.0%	26,865,108	0.0%
matrix300	saxpy	13,340,000	13,340,000	0.0%	13,340,000	0 0%	10,480,000	10,480,000	0 0%	10,480,000	0 0%
doduc	subb	1,763.280	1,763,280	0.0%	1.763,280	0.0%	1,763,280	1,763,280	0 0%	1,763,280	0 0%
doduc	x21y21	1,253,980	1,253,980	0.0%	1,355,263	7.5%	1,162,343	1.162.343	0.0%	1,162,343	0.0%
solve	decomp	642	641	-0.2%	631	-1 7%	655	651	-0.6%	601	-9 0%
svd	svd	4,596	4,542	-1.2%	4.547	-1.1%	4,612	4,407	-4.7%	4,135	-11.5%
cplex	chpivot	4,156,117	4,097,458	-1.4%	4,097,548	-1.4%	4,194,263	4,137,514	-1 4%	4,137,604	-1 4%
zeroin	zeroin	740	729	-1.5%	729	-1.5%	909	731	-24.4%	738	-23 2%
tomcatv	tomcatv	2 485E+08	2.436E+08	-2 0%	2 433E+08	-2 1%	1 952E+08	1.957E+08	0.3%	1.952E+08	0.0%
fmin	fmin	908	876	-3.7%	902	-0.7%	1,017	885	-14 9%	901	-12 9%
fpppp	efill	2,046,512	1,951,843	-4 9%	1,898,546	-7.8%	2,791,920	2,305,816	-21.1%	2,471,638	-13.0%
doduc	ihbtr	75,708	71,616	-5 7%	77.010	1.7%	71,430	84,266	15 2%	79,988	10 7%
doduc	saturr	63,426	59,334	-6.9%	58,962	-7 6%	63,426	60,078	-5 6%	59,706	-6.2%
	TOTAL	4.416E+08	4.432E+08	0.4%	4.423E+08	0 2%	3.873E+08	3.938E+08	1.6%	3.918E+08	1.1%
	Average	8,659,662	8,690,153	4 3%	8.672,843	2 4%	7,593,844	7,721.122	5.9%	7,681,987	2.2%

Figure 10 GVN-GCM *vs.* GCF-PRE-CCP

Bibliography

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, 1985.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, 1988.

[3] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein. The program dependence web: A representation supporting control-, data- and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Languages Design and Implementation*, June 1990.

[4] P. Briggs, *Register Allocation via Graph Coloring.* Ph.D. thesis, Rice University, 1992.

[5] P. Briggs. The Massively Scalar Compiler Project. Unpublished report. Preliminary version available via ftp://cs.rice.edu/public/preston/optimizer/shared.ps. Rice University, July 1994.

[6] P. Briggs and K. Cooper. Effective partial redundancy elimination. In *Proceedings of the SIGPLAN '94 Conference on Programming Languages Design and Implementation*, June 1994.

[7] P. Briggs and T. Harvey. Iloc '93. Technical report CRPC-TR93323, Rice University, 1993.

[8] R. Cartwright and M. Felleisen. The semantics of program dependence. In *Proceedings of the SIGPLAN '89 Conference on Programming Languages Design and Implementation*, June 1989.

[9] G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction,* June 1982.

[10] C. Click, *Combining Analyses, Combining Optimizations.* Ph.D. thesis, Rice University, 1995.

[11] J. Cocke and J. T. Schwartz. *Programming languages and their compilers.* Courant Institute of Mathematical Sciences, New York University, April 1970.

[12] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth ACM Symposium on the Principles of Programming Languages*, Jan. 1989.

[13] K. H. Drechsler and M. P. Stadel. A solution to a problem with Morel and Renvoise's "Global optimization by suppression of partial redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, Oct. 1988.

[14] G. E. Forstyhe, M. A. Malcom, and C. B. Moler. *Computer Methods for Mathematical Computations.* Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[15] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July, 1987.

[16] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the SIGPLAN '94 Conference on Programming Languages Design and Implementation*, June 1994.

[17] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July, 1979.

[18] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, Feb. 1979.

[19] B. K. Rosen., M. N. Wegman, and F. K. Zadeck, Global Value Numbers and Redundant Computations. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, Jan. 1988.

[20] T. Simpson, Global Value Numbering. Unpublished report. Available from ftp://cs.rice.edu/public/preston/optimizer/gval.ps. Rice University, 1994.

[21] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences,* 9:355–365, 1974.

[22] C. Vick, *SSA-Based Reduction of Operator Strength.* Masters thesis, Rice University, pages 11–15, 1994.

[23] D. Weise, R. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: Representation without taxation. In *Proceedings of the 21st ACM SIGPLAN Symposium on the Principles of Programming Languages*, 1994.

[24] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181-210, April 1991.