SCC-Based Value Numbering

Keith Cooper
Taylor Simpson

CRPC-TR95636-S
October 1995

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

SCC-Based Value Numbering

Keith D. Cooper
L. Taylor Simpson

Value numbering is an optimization that assigns numbers to values in such a way that two values are assigned the same number if and only if the compiler can prove they are equal. When this optimization discovers two computations that produce the same value, it can (under certain circumstances) eliminate one of them. There are two competing techniques for proving equivalences: hashing and partitioning. The hashing techniques are easy to understand and implement, and they can easily handle constant folding and algebraic identities (*i.e.*, $x+0=x$). Their prime drawback is that they are not global techniques. The partitioning techniques are global, but they cannot easily handle constant folding and algebraic identities. As a result of their shortcomings, both of these techniques can fail to discover some crucial equivalences. In this paper, we describe a new technique for assigning value numbers that combines the advantages of both techniques – it is easy to understand and implement; it can easily handle constant folding and algebraic identities, and it is global. We refer to this new technique as SCC-based value numbering because it is centered around the strongly connected components of the static single assignment graph. We will prove that our technique always finds at least as many equivalences as hashing or partitioning, and experimentally compare the improvements made in the context of an optimizing compiler.

1 Introduction

Value numbering is a code optimization designed to discover and eliminate redundant computations from a program. Although the name was originally applied to a method for improving single basic blocks, it is now used to describe a collection of optimizations that vary in power and scope. The compiler can only assign two expressions the same number if it can prove that they always produce equal values in their respective contexts. Two techniques for proving this equivalence appear in the literature: *hash-based methods* and *partitioning methods*. Both are described in Section 2.

Each of these techniques has both advantages and disadvantages. The hash-based approach is easy to understand and implement, and it can easily handle constant folding and algebraic identities. The technique can assign value numbers consistently over an entire routine, but is not global. This will be described in detail in Section 2.1. The partitioning techniques have the advantage of being global, but they are more difficult to implement and they are difficult to extend to handle constant folding and algebraic identities.

In this paper, we describe a new value numbering technique that combines the advantages of hash-based and partitioning techniques. Because the algorithm is centered around the strongly connected components (SCCs) of the routine's static single assignment (SSA) graph, we call it SCC-*based value numbering*. It is a global technique that is easy to understand and implement, and handles constant folding and algebraic simplification.

2 Previous Work

2.1 Hash-Based Value Numbering

Cocke and Schwartz describe a local technique that uses hashing to discover redundant computations and fold constants [7]. The algorithm was apparently discovered by Balke and his colleagues at CSC in the late 1960s. Each unique value is identified by its *value number*. Two computations in a block have the same value number if they are provably equal. This technique and its derivatives are called "value numbering."

Authors address: Rice University, 6100 South Main Street, Mail Stop 41, Houston, TX 77005. Address all correspondence to Taylor Simpson, `lts@cs.rice.edu`
This research has been supported by ARPA and by IBM Corporation.

The algorithm is relatively simple. In practice, it is very fast. For each instruction in the block, it hashes the operator and the value numbers of the operands to obtain the unique name that corresponds to its value. If the value has already been computed in the block, it will already exist in the table. The recomputation can be replaced with a reference to the earlier computation. Any operator with known-constant arguments is evaluated and the resulting value used to replace any subsequent references. The algorithm is easily extended to account for commutativity and simple algebraic identities without affecting its complexity.

As originally described, the technique works for single basic blocks. It can also be applied to larger scopes within a routine [5]. The most powerful of these approaches operates on SSA form [8]. Blocks are processed in reverse postorder to guarantee that all of a block's predecessors through non-back edges are processed before the block itself. The algorithm analyzes the ϕ-nodes in a block only if there are no incoming back edges. A ϕ-node can be eliminated if it is meaningless—all its parameters have the same value number—or if it is redundant—it computes the same value number as another ϕ-node in the block. At each instruction, the method overwrites each SSA name with its value number. After processing a block, it visits each successor block and updates any ϕ-node inputs that come from the current block. This algorithm is efficient, with an expected running time of $\mathbf{O}(N)$, where N is the number of SSA names. Despite its simplicity and effectiveness, this is not a global algorithm because it cannot handle values that travel through back edges in the control-flow graph (CFG).

2.2 Value Partitioning

Alpern, Wegman, and Zadeck presented a technique that uses a variation on Hopcroft's DFA-minimization algorithm to partition values into congruence classes [3, 1]. It operates on the SSA form of the routine. Two values are *congruent* if they are computed by the same opcode, and their corresponding operands are congruent. For all legal expressions, two congruent values must be equal. Since the definition of congruence is recursive, there will be routines where the solution is not unique. A trivial solution would be to set each value in the routine to be congruent only to itself; however, the solution we seek is the *maximal fixed point* – the solution that contains the most congruent values.

Initially, the partition contains a congruence class for the values defined by each operator in the program. The partition is iteratively refined by examining the uses of all members of a class and determining which classes must be further subdivided. This process runs in $\mathbf{O}(E \log_2 N)$ time, where N and E are the number of nodes and edges in the SSA graph. After the partition stabilizes, the registers and ϕ-nodes in the routine are renumbered based on the congruence classes. Because the effects of partitioning and renumbering are analogous to those of value numbering described in the previous section, we think of this technique as a form of global (or intraprocedural) value numbering.

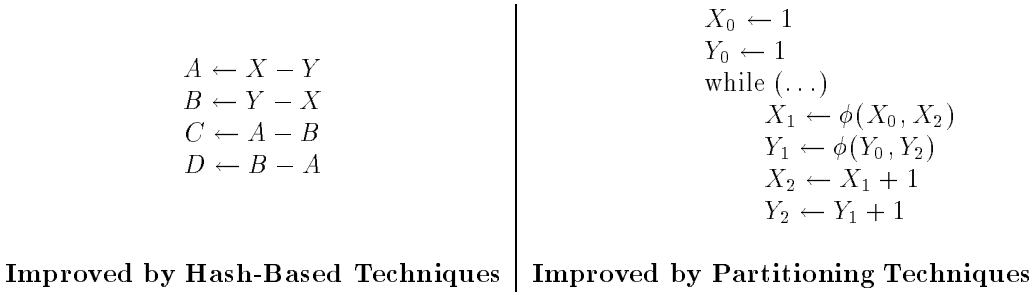


Figure 1 Comparing the Techniques

Click presents an extension to value partitioning that includes constant folding, algebraic simplification, and unreachable code elimination [6]. He presents two versions of the algorithm. The straightforward version runs in $\mathbf{O}(N^2)$ time, and the complex version runs in $\mathbf{O}(E \log_2 N)$ time, where N and E are the number of nodes and edges in the routine's intermediate representation graph. His intermediate representation contains the edges in the SSA graph plus some edges used for control dependences. The complex version can miss some congruences between ϕ-nodes that will be proven congruent by the straightforward algorithm. The problem occurs when the operands of a ϕ-node are assumed congruent and later proven not congruent. When the class containing the operands is split into two pieces, the algorithm arbitrarily places the ϕ-node in one of the pieces. Thus, another ϕ-node with the same operands might be placed in the other piece.

2.3 Comparing the Techniques

Assume that X and Y are known to be equal in the code fragment in the left column of Figure 1. Then the partitioning algorithm will find A congruent to B and C congruent to D. More careful reasoning would show that they are not just congruent by pairs, but also that they all have the value zero. Unfortunately, partitioning cannot discover that fact. On the other hand, the hash-based approach will easily conclude that if $X = Y$ then A, B, C, and D are all zero.

The critical difference between the hashing and partitioning algorithms identified by this example is their notion of equivalence. The hash-based approach proves equivalences based on values, while the partitioning technique considers only congruent computations to be equivalent. The code in this example hides the redundancy behind an algebraic identity. Only the techniques based on value equivalence will discover the common subexpression here.

Now consider the code fragment in the right column of Figure 1. If we apply any of the hash-based approaches to this example, none of them will be able to prove that X_1 is equal to Y_1. This is because at the time a value number must be assigned to X_1 and Y_1, none of these techniques have visited X_2 or Y_2. They must therefore assign different value numbers to X_1 and Y_1. However, the partitioning technique will prove that X_1 is congruent to Y_1 (and thus X_2 is congruent to Y_2). The key feature of the partitioning algorithm which makes this possible is its initial optimistic assumption that all values defined by the same operator are congruent. It then proceeds to disprove the instances where the assumption is false. In contrast, the hash-based approaches begin with the pessimistic assumption that no values are equal and proceeds to prove as many equalities as possible.

3 SCC-Based Value Numbering

The above comparison suggests a need for a value numbering algorithm that combines the features of the hash-based and the partitioning techniques. Such an algorithm would combine the ability to perform constant folding and algebraic simplification with the ability to make optimistic assumptions and later disprove them.

SCC-based value numbering is simpler to implement than value partitioning, and it runs in $\mathbf{O}(N \times D(\text{SSA}))$ time, where N is the number of SSA names, and $D(\text{SSA})$ is the *loop connectedness* of the SSA graph. The loop connectedness of a graph is the maximum number of back edges in any acyclic path. This number can be as large as $\mathbf{O}(N)$; Knuth showed that, for control-flow graphs of real Fortran programs, it is bounded, in practice by three [13]. We are concerned with the loop-connectedness of the SSA graph; we also expect it to be small. In our test suite, the maximum number of iterations required by the SCC algorithm is four.

We will first present a simplified version, called the RPO algorithm, that is easier to reason about. We will prove the correctness and time bounds for this algorithm, and then we will present SCC-based value numbering as an extension with the same asymptotic complexity. In practice, it is more efficient than the RPO algorithm.

```
for all SSA names i
    VN[i] ← ⊤
repeat
    done ← TRUE
    for all blocks b in reverse postorder
        for all definitions x in b
            temp ← lookup(x.op, VN[x[1]], VN[x[2]], x)
            if VN[x] ≠ temp
                done ← FALSE
                VN[x] ← temp
    Remove all entries from the hash table
until done
```

Figure 2 The RPO Algorithm

3.1 The RPO Algorithm

The algorithm in Figure 2 is called the RPO algorithm because it operates on the routine in reverse postorder. We will assume for simplicity that all definitions in the routine are of the form $x \leftarrow y$ op z, where *op* can be any operation in the intermediate representation or a ϕ-node. Let $x[i]$ represent the i^{th} operand of the expression defining x, and x.op represent the operator that defines x. Additionally, we say that $x[i]$ is a back edge if the value flows along a back edge in the CFG. The VN array maps SSA names to value numbers. Each value number represents a set of SSA names (*i.e.*, those names with the same entry in the VN array). Therefore, a value number is itself an SSA name. For clarity, we will surround an SSA name that represents a value number with angle brackets (*e.g.*, $\langle x \rangle$). The *lookup* function searches a hash table for the expression VN$[x[1]]$ x.op VN$[x[2]]$. If the expression is found, it returns the name of the expression. Otherwise, it adds the expression to the table with name $\langle x \rangle$.

The RPO algorithm computes a sequence of equivalence relations, $\cong_i$, that partition the set of SSA names. We say that $x \cong_i y$ if and only if after the i^{th} iteration of the RPO algorithm VN$[x]$ = VN$[y]$.

$$i = 0, \quad x \cong_0 y \quad \forall x, y$$

$$i > 0, \quad x \cong_i y \quad \text{iff} \begin{cases} x.\text{op} = y_{\text{op}} & \\ x[e] \cong_i y[e], & \forall x[e] \text{ that are non-back edges} \\ x[e] \cong_{i-1} y[e], & \forall x[e] \text{ that are back edges} \end{cases}$$

We refer to a partition by the equivalence relation that produces it. We say that one partition is a refinement of another ($\cong_i \preceq \cong_j$) if and only if there are no congruences in $\cong_i$ that are not in $\cong_j$ (*i.e.*, $\forall x, y \;\; x \cong_i y \Rightarrow x \cong_j y$). In other words, $\cong_i$ can be derived from $\cong_j$ by breaking congruences. Given the partition $\cong_i$, the algorithm computes $\cong_{i+1}$ in expected running time $\mathbf{O}(N)$, where N is the number of SSA names in the routine. The following theorem shows that each iteration refines the partition.

Theorem 1 $x \cong_i y \Rightarrow x \cong_{i-1} y$

Proof. The proof is by induction on i.

Basis ($i = 1$) By definition, $x \cong_0 y$.

Induction step ($i > 1$) Suppose not – let x be the SSA name with the smallest RPO number such that the assumption is false – $x \ncong_{i-1} y$ and $x \cong_i y$. Consider the reasons why $x \ncong_{i-1} y$:

Case 1 ($x.\mathrm{op} \neq y.\mathrm{op}$) This implies that $x \not\cong_i y$, a contradiction.

Case 2 ($x[e] \not\cong_{i-1} y[e]$ **for some non-back edge**) Since $x \cong_i y$, $x[e] \cong_i y[e]$ which means that $x[e]$ is a node where the assumption is false, and it has a smaller RPO number than x, a contradiction.

Case 3 ($x[e] \not\cong_{i-2} y[e]$ **for some back edge**) By the induction hypothesis, $x[e] \not\cong_{i-1} y[e]$, which implies that $x \not\cong_i y$, a contradiction. □

Corollary 1 *The* RPO *algorithm must terminate, and it finds the maximal fixed point of the congruence relation computed by value partitioning.*

Proof. Each step produces a refinement of the partition, and refinement cannot continue indefinitely. Further, value partitioning finds the maximal fixed point of the following equivalence relation:

$$x \cong y \quad \text{iff} \begin{cases} x.\mathrm{op} = y_{\mathrm{op}} \\ x[e] \cong y[e], \quad \forall e \end{cases}$$

Since the RPO algorithm begins with all SSA names congruent, we must converge to the same fixed point as value partitioning. □

To understand how quickly the algorithm terminates, we must understand how values are proven not to be congruent. Since we process the blocks in reverse postorder, back edges play a key role in determining the number of iterations required. The following lemma characterizes the iteration on which two SSA names are determined not to be congruent.

Lemma 1 *If $x \not\cong_i y$ and $x \cong_{i-1} y$, then there is a sequence of inputs (possibly empty):*

$$e_1, e_2, \ldots, e_n$$

with b_j = the number of back edges in $e_1, \ldots, e_j$ and $b_n = i - 1$ such that:

$$\begin{array}{rcl} x & \not\cong_i & y \\ x[e_1] & \not\cong_{i-b_1} & y[e_1] \\ & \vdots & \\ x[e_1]\ldots[e_n] & \not\cong_{i-b_n} & y[e_1]\ldots[e_n] \end{array}$$

Proof. The proof is by induction on i.

Basis ($i = 1$) Use the empty sequence.

Induction step ($i > 1$) Let $p_1, \ldots, p_m$ be the sequence of pairs x, y with $x \not\cong_i y$ and $x \cong_{i-1} y$, ordered by the minimum RPO number of the pair. We will proceed by induction on j, the index into this sequence.

Basis ($j = 1$) Consider the reasons why $x \not\cong_i y$:

Case 1 ($x.\mathrm{op} \neq y.\mathrm{op}$) This case cannot occur because we know that $x \cong_{i-1} y$.

Case 2 ($x[e] \not\cong_i y[e]$) **for some non-back edge)** This case cannot occur because either $x[e]$ will have a smaller RPO number than x or $y[e]$ will have a smaller RPO number than y.

Case 3 ($x[e] \not\cong_{i-1} y[e]$) **for some back edge)** The sequence consists of e followed by the sequence for the pair $x[e], y[e]$, which we know exists by the induction hypothesis for i.

Induction step($j > 1$) Consider the reasons why $x \not\cong_i y$:

Case 1 ($x.\mathrm{op} \neq y.\mathrm{op}$) This case cannot occur because we know that $x \cong_{i-1} y$.

Case 2 ($x[e] \not\cong_i y[e]$) **for some non-back edge)** The sequence consists of e followed by the sequence for the pair $x[e], y[e]$, which we know exists by the induction hypothesis for j.

Case 3 ($x[e] \not\cong_{i-1} y[e]$) for some back edge) The sequence consists of e followed by the sequence for the pair $x[e], y[e]$, which we know exists by the induction hypothesis for i. □

Now we can prove the algorithm's running time. It terminates in $D(\text{SSA}) + 2$ iterations, where $D(\text{SSA})$ is the *loop connectedness* (the maximum number of back edges on any acyclic path) of the SSA graph.

Theorem 2 $x \cong_{D(\text{SSA})+1} y \Rightarrow x \cong_{D(\text{SSA})+2} y$

Proof. Suppose not – let x be the SSA name with the smallest RPO number such that $x \cong_{D(\text{SSA})+1} y$ and $x \not\cong_{D(\text{SSA})+2} y$. According to Lemma 1, there is a sequence of inputs such that:

$$\begin{array}{rcl} x & \not\cong_{D(\text{SSA})+2} & y \\ x[e_1] & \not\cong_{D(\text{SSA})+2-b_1} & y[e_1] \\ & \vdots & \\ x[e_1]\ldots[e_n] & \not\cong_1 & y[e_1]\ldots[e_n] \end{array}$$

This sequence contains $D(\text{SSA}) + 1$ back edges, so it must contain a cycle. Since x has the smallest RPO number, it must be included in a cycle. Therefore, $x \not\cong_i y$ for some $i < D(\text{SSA}) + 2$. By Theorem 1, $x \not\cong_{D(\text{SSA})+1} y$, a contradiction. □

Corollary 2 *The* RPO *algorithm terminates in at most* $D(\text{SSA}) + 2$ *passes.*

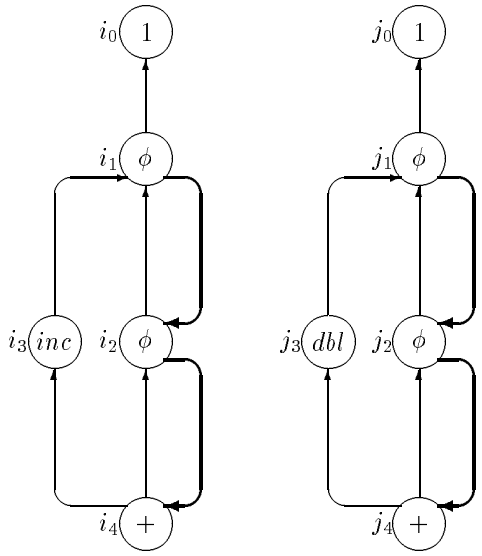
Proof. Since the partition $\cong_{D(\text{SSA})+2}$ is the same as the partition $\cong_{D(\text{SSA})+1}$, the *done* flag will remain `TRUE` throughout iteration $D(\text{SSA}) + 2$, and the algorithm will terminate. □

3.2 Extensions

Since our algorithm uses hashing, we can easily extend it to include constant folding and algebraic simplification. We do this by associating a value from the constant propagation lattice ($\{\top, \bot\} \cup \mathcal{Z}$) with each SSA name [15]. This framework will discover at least as many congruences as hash-based value numbering or value partitioning. Under this extended framework, an element can fall $D(\text{SSA}) + 1$ times with respect to the value numbering lattice and twice with respect to the constant propagation lattice. Therefore, the height of this aggregate lattice is $2D(\text{SSA}) + 2$. However, since each element falls in both frameworks on the first iteration, any element can fall at most $2D(\text{SSA}) + 1$ times. Therefore, the extended algorithm must terminate in $2D(\text{SSA}) + 2$ iterations.[1]

The example in Figure 3 requires $2D(\text{SSA}) + 2$ iterations for the algorithm to terminate. The back edges are shown with bold arrows. After the first iteration, all nodes are believed to be constants; notice that both i_3 and j_3 are assigned the constant 2. During the next iteration, i_2, j_2, i_4, and j_4 are determined not to be constant, but we still assume that $i_2 \cong j_2$ and $i_4 \cong j_4$. During the third iteration, we prove that i_1, i_3, j_1, and j_3 are not constant, and we prove that $i_4 \not\cong j_4$. On the fourth and fifth iteration, we prove that $i_2 \not\cong j_2$ and $i_1 \not\cong j_1$, respectively. On the sixth iteration, the partition stabilizes and the algorithm terminates. Intuitively, the algorithm takes $D(\text{SSA})$ passes to prove that i_3 and j_3 are not the constant 2, and thus cannot be equal; then it takes another $D(\text{SSA})$ passes to propagate this fact.

[1] The final iteration checks the stability of the analysis.



SSA graph

	0	1	2	3	4	5	6
i_0	$\top$	$i_0(1)$	$i_0(1)$	$i_0(1)$	$i_0(1)$	$i_0(1)$	$i_0(1)$
i_1	$\top$	$i_0(1)$	$i_0(1)$	i_1	i_1	i_1	i_1
i_2	$\top$	$i_0(1)$	i_2	i_2	i_2	i_2	i_2
i_3	$\top$	$i_3(2)$	$i_3(2)$	i_3	i_3	i_3	i_3
i_4	$\top$	$i_4(3)$	i_4	i_4	i_4	i_4	i_4
j_0	$\top$	$i_0(1)$	$i_0(1)$	$i_0(1)$	$i_0(1)$	$i_0(1)$	$i_0(1)$
j_1	$\top$	$i_0(1)$	$i_0(1)$	i_1	i_1	j_1	j_1
j_2	$\top$	$i_0(1)$	i_2	i_2	j_2	j_2	j_2
j_3	$\top$	$i_3(2)$	$i_3(2)$	j_3	j_3	j_3	j_3
j_4	$\top$	$i_4(3)$	i_4	j_4	j_4	j_4	j_4

Value numbers

Figure 3 Example requiring $2D(\text{SSA}) + 2$ iterations

3.3 Discussion

We have shown that the RPO algorithm finds at least as many congruences as hash-based value numbering or value partitioning in $\mathbf{O}(N \times D(\text{SSA}))$ time. Kam and Ullman showed that the iterative data-flow analysis used for a large class of data-flow frameworks requires $D(\text{CFG})$ passes over the CFG [11]. This number can be as large as $\mathbf{O}(B)$ where B is the number of blocks in the CFG, but it is believed that, in practice, this number is bounded by a small constant [13]. We expect that for most programs $D(\text{CFG}) = D(\text{SSA})$. However, the program in Figure 4 is an example where this is not true. The back edges are shown with bold arrows. Notice that $D(\text{CFG}) = 2$, but $D(\text{SSA}) = 6$. Further, we could make $D(\text{SSA})$ even larger by adding variables in the same pattern as j and k. Despite this potential, the maximum number of iterations required by SCC-based value numbering is four for any routine in our test suite.

3.4 The SCC Algorithm

To make the algorithm more efficient in practice, we operate on the SSA graph instead of the control-flow graph. We refer to the improved algorithm as the SCC algorithm because it concentrates on the *strongly connected components* of the SSA graph. The algorithm works in conjunction with Tarjan's depth-first algorithm for finding SCCs [14]. The algorithm uses a stack to determine which nodes are in the same SCC; nodes not contained in any cycle are popped singly, while all the nodes in the same SCC are popped together. Tarjan's algorithm has an interesting property: when a collection of nodes (possibly containing only a single node) is popped from the stack, all of the operands that are outside the collection have already been popped. Therefore, we assign value numbers as nodes are popped from the stack. When a single node is popped from the stack, we know that we have assigned value numbers to the operands of the corresponding expression. Thus, we can examine the expression and assign a value number to this node. When a collection of nodes representing an SCC is popped, we know that we have assigned value numbers to any operands outside the SCC. The members of the SCC require special handling in order to perform value numbering.

We assign value numbers to the nodes of an SCC by iterating over the SCC in reverse postorder (with

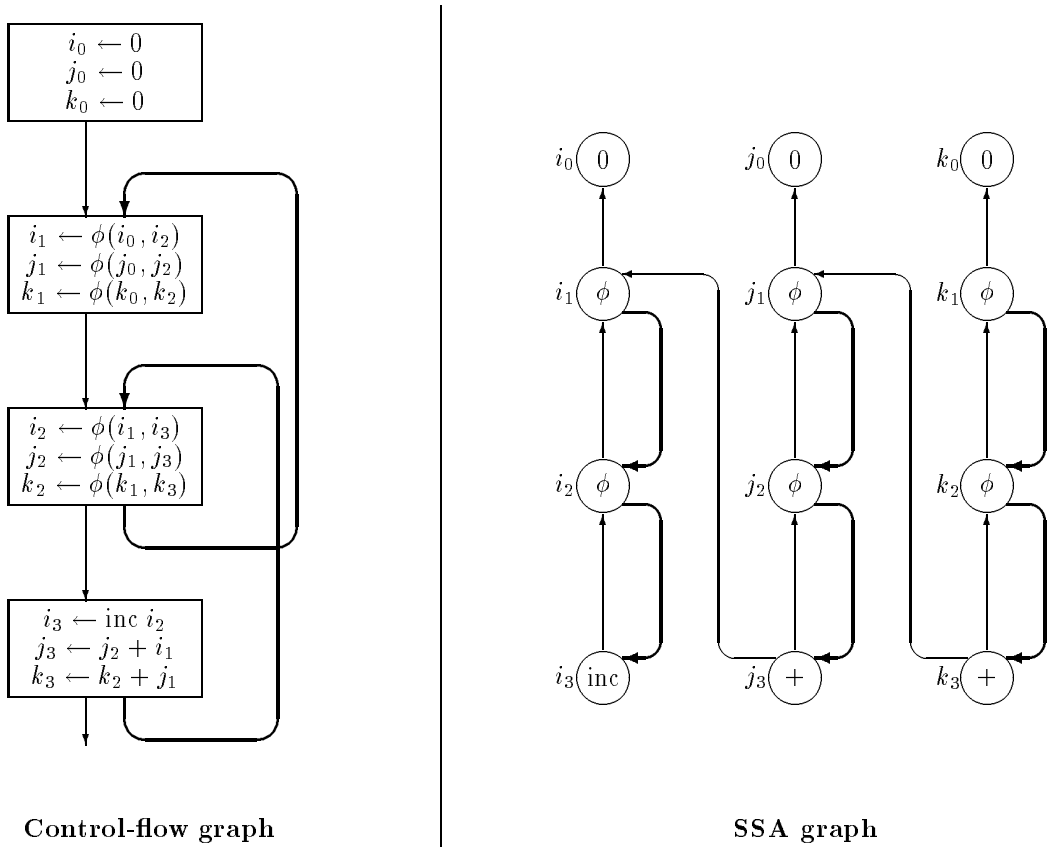


Figure 4 Example with $D(\text{CFG}) \neq D(\text{SSA})$

respect to the CFG). Initially, the value number for each member of the SCC is $\top$. A value number of $\top$ indicates that this value has not yet been examined. Since we cannot remove the entries from the hash table after each pass as the RPO algorithm does, we will use two hash tables. The iterative phase uses an *optimistic* table. Once the value numbers in the SCC stabilize, entries are added to the *valid* table. The value numbering of single values only uses the valid table.

3.5 Example

To further clarify the algorithm, consider how it would proceed if given the examples in Figure 1. Since none of the values in the code fragment in the left column are contained in a cycle, the straightforward hash-based value numbering will be applied to determine that A, B, C, and D are all equal to zero.

If the algorithm is applied to the code fragment in the right column, the values X_0 and Y_0 are not contained in any cycle, so they will be assigned value numbers before either of the SCCs. Assume they are each given the value number $\langle X_0 \rangle$ and that the SCC containing X_1 and X_2 is processed next. During the first pass over the SCC, the ϕ-node $\phi(\langle X_0 \rangle, \top)$ will be simplified (optimistically) to $\langle X_0 \rangle$, and X_1 will be given value number $\langle X_0 \rangle$.[2] Then, the expression defining X_2, $\langle X_0 \rangle + 1$, can be simplified to 2. An entry mapping the constant 2 to value number $\langle X_2 \rangle$ will be added to the optimistic table. During the second pass, the expression $\phi(\langle X_0 \rangle, \langle X_2 \rangle)$ cannot be simplified, so an entry mapping the expression to $\langle X_1 \rangle$ is added to

[2]Remember that expressions are formed from an operator and the value numbers of the operands, not the operands themselves.

the optimistic table. Next, an entry mapping $\langle X_1 \rangle + 1$ to $\langle X_2 \rangle$ will be added to the optimistic table. At this point the value numbers have stabilized, so we add entries mapping $\phi(\langle X_0 \rangle, \langle X_2 \rangle)$ to $\langle X_1 \rangle$ and mapping $\langle X_1 \rangle + 1$ to $\langle X_2 \rangle$ to the valid table. Notice that the optimistic entry mapping 2 to $\langle X_2 \rangle$ is not added to the valid table - this assumption has been disproven.

The next step is to process the SCC containing Y_1 and Y_2. During the first pass, the expression $\phi(\langle X_0 \rangle, \top)$ will be simplified to $\langle X_0 \rangle$, and Y_1 will be given the value number $\langle X_0 \rangle$. Next, the expression $\langle X_0 \rangle + 1$ can be simplified to 2; it will be found in the optimistic table with value number $\langle X_2 \rangle$. During the second pass, the expression $\phi(\langle X_0 \rangle, \langle X_2 \rangle)$ will be found in the optimistic table with value number $\langle X_1 \rangle$, and $\langle X_1 \rangle + 1$ will be found with value number $\langle X_2 \rangle$. At this point the value numbers have stabilized, so we process the SCC using the valid table. Since entries already exist mapping $\phi(\langle X_0 \rangle, \langle X_2 \rangle)$ to $\langle X_1 \rangle$ and $\langle X_1 \rangle + 1$ to $\langle X_2 \rangle$, no new entries will be added to the valid table. Thus, the algorithm has determined that $X_1 \cong Y_1$ and $X_2 \cong Y_2$.

The contents of the optimistic and valid tables is an important issue that merits further discussion. The primary function of the optimistic table is to hold assumptions that may later be disproven. In contrast, the valid table represents only those facts that are proven. Notice that in processing the example in the right column of Figure 1, entries for $\langle X_1 \rangle$ and $\langle X_2 \rangle$ were added to both the optimistic and valid tables. On the other hand, the entry mapping the constant 2 to $\langle X_2 \rangle$ was only added to the optimistic table. This entry represents an optimistic assumption that was disproven. It remains in the table because it is needed for the analysis of the second SCC – the one containing Y_1 and Y_2. In some sense, that entry "marks the trail" that the analysis must take in order to prove that the two SCCs are equivalent. It is also possible that the constant 2 appears somewhere else in the routine. If so, we cannot give it the name $\langle X_2 \rangle$; instead we add an entry to the valid table mapping 2 to a different name.

Recall that after the iteration stabilizes, we make one additional pass over the SCC using the valid table. The need to place expressions in both tables arises from constant folding and algebraic simplification. These transformations eliminate edges from the SSA graph. Thus, an SCC can be transformed into a collection of nodes that is no longer strongly connected. If this happens, we want to test the nodes in this new collection for congruence with other nodes that were processed using the valid table.

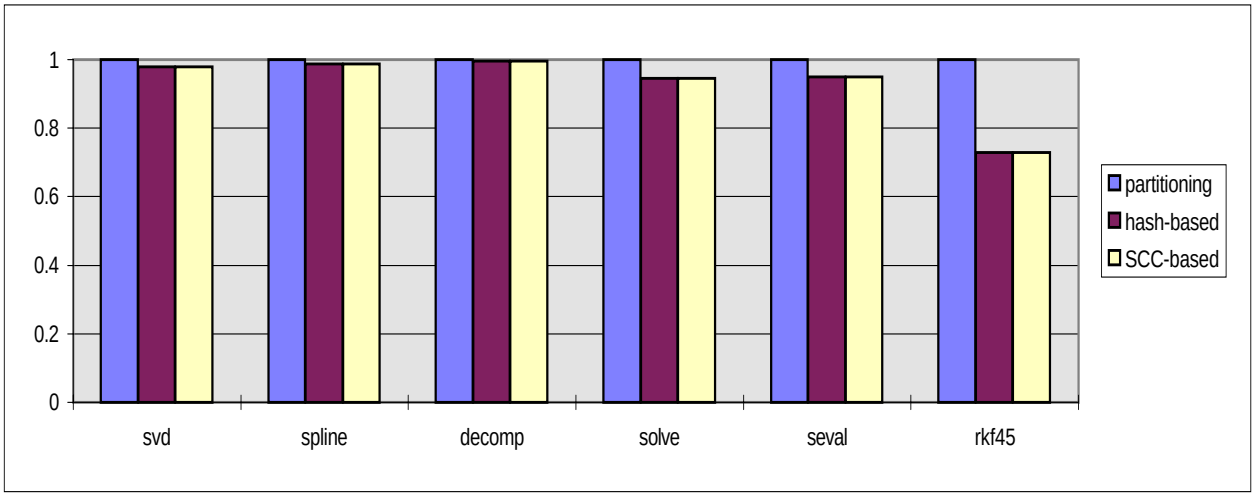


Figure 5 Comparison of value numbering techniques – FMM benchmark

4 Experimental Results

Even though we can prove that SCC-based value numbering is never worse than hash-based value numbering or value partitioning, an equally important question is how much this theoretical distinction matters in practice. To assess the real impact of these techniques, we have implemented all of the optimizations in our experimental Fortran compiler. Comparisons were made using routines from a suite of benchmarks consisting of over 50 routines drawn from the SPEC benchmark suite and from Forsythe, Malcolm, and Moler's book on numerical methods [10].

Our optimizer is composed of a sequence of passes that operate on ILOC – our intermediate language. ILOC is a pseudo-assembly language for a RISC machine with an arbitrary number of symbolic registers. The back end generates code that is capable of counting the number of ILOC operations executed. Routines are optimized using the sequence of global reassociation [4], value numbering (the type is indicated in the legend), lazy code motion [12, 9], global constant propagation [15], operator strength reduction [2], value numbering, global constant propagation, global peephole optimization, dead code elimination [8, Section 7.1], copy coalescing, and a pass to eliminate empty basic blocks. We repeat the global constant propagation and value numbering passes to clean up after operator strength reduction. Figures 6 and 5 show only those routines where there was variation in the number of ILOC operations executed. Each column represents dynamic counts of ILOC operations, normalized against value partitioning.

We should point out that eliminating more redundancies does not necessarily result in reduced execution time. This is due to the way different optimizations interact. In our experiment, removing more redundancies had a negative impact on only three routines: `bilsla`, `ihbtr`, and `orgpar`. In these routines, propagating more constants created more opportunities for operator strength reduction. Since we are measuring the number of instructions executed rather than actual execution times, this effect may be deceptive.

We also compared the time required by each of the techniques for some of the larger routines in the test suite. These results are shown in Table 1. The number of blocks, SSA names, and operations are given to indicate the size of the routine being optimized. The SCC technique is significantly faster than partitioning; it is competitive with hashing until a routine has enough SCCs to make iteration its dominant behavior.

5 Conclusion

We have presented a new value numbering algorithm that is centered around the strongly connected components of the SSA graph. It discovers at least as many equalities as previously known techniques because it is a global algorithm that includes constant folding and algebraic simplification. The algorithm is easy to understand and implement, and it runs in $\mathbf{O}(N \times D)$ time, where N is the number of nodes in the SSA graph and D is the loop connectedness of the SSA graph. We experimentally compared the improvements made by our technique over existing ones when applied to real programs in the context of an optimizing compiler.

routine	*blocks*	*SSA names*	*operations*	*hash-based*	*SCC-based*	*partitioning*
`tomcatv`	131	2212	2663	0.03	0.05	0.13
`ddeflu`	109	5494	4502	0.04	0.35	1.81
`debflu`	116	5856	3951	0.05	0.41	2.17
`deseco`	251	13164	11771	0.17	0.65	4.11
`twldrv`	266	23486	15615	0.30	2.82	13.49
`fpppp`	2	18127	22462	0.51	0.49	1.60

Table 1 Running times of value numbering techniques

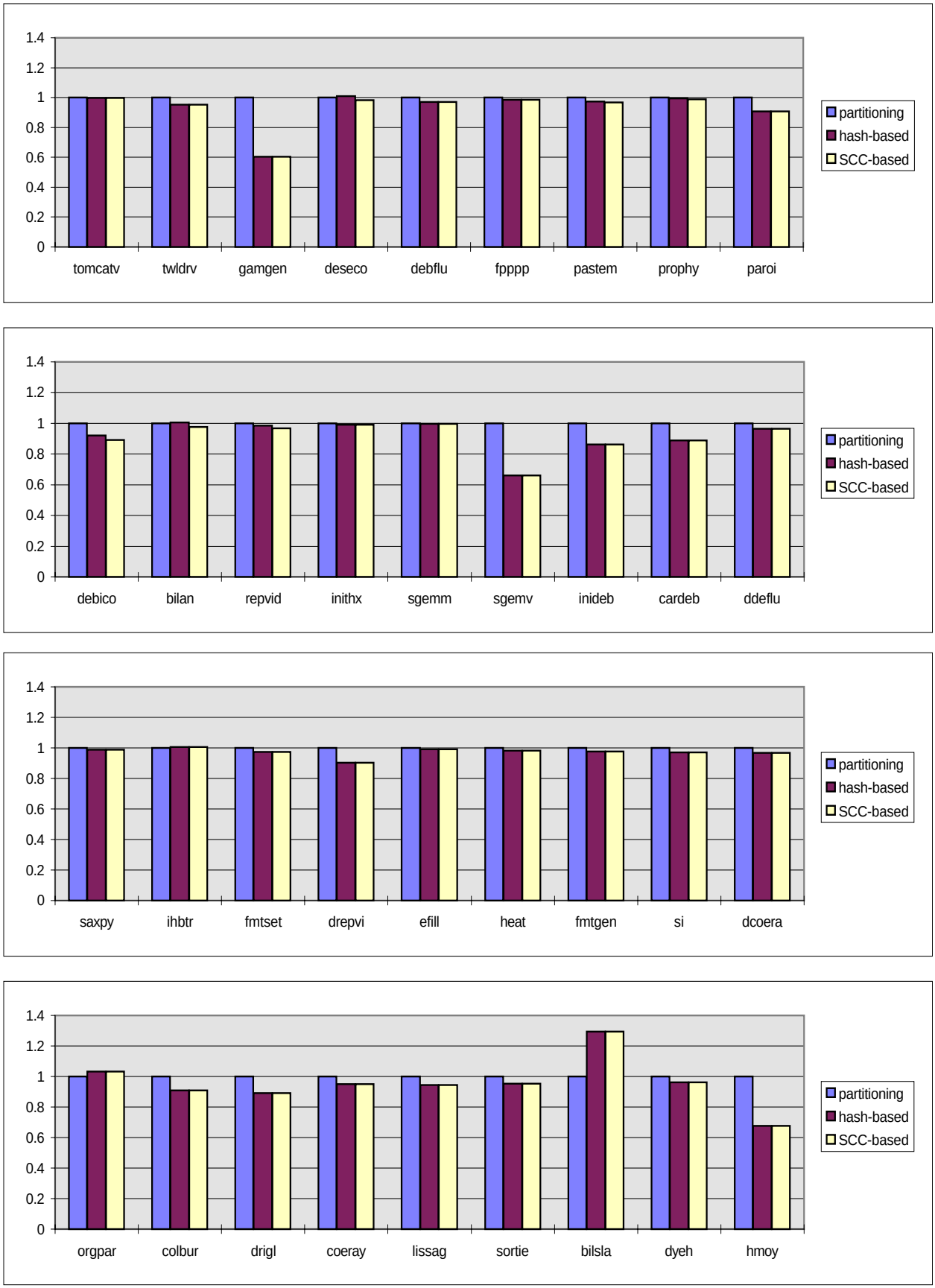


Figure 6 Comparison of value numbering techniques – SPEC benchmark

6 Acknowledgements

Our colleagues in the Massively Scalar Compiler Project at Rice have played a large role in this work. Without their implementation efforts, we could not have completed this work. We are especially grateful to Tim Harvey and Linda Torczon who acted as sounding boards during the development of the algorithm. David Spott and David Wallace of Sun Microsystems have also shown a great deal of interest in this work. We would also like to thank Vivek Sarkar and IBM for supporting Taylor Simpson through the IBM Cooperative Fellowship.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[2] Frances E. Allen, John Cocke, and Ken Kennedy. Reduction of operator strength. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[3] 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, San Diego, California, January 1988.

[4] Preston Briggs and Keith D. Cooper. Effective partial redundancy elimination. *SIGPLAN Notices*, 29(6):159–170, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation.*

[5] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. Technical Report CRPC-TR95517-S, Center for Research on Parallel Computation, Rice University, November 1994. Submitted to *Software – Practice and Experience.*

[6] Cliff Click. *Combining Analyses, Combining Optimizations.* PhD thesis, Rice University, 1995.

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

[8] 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.

[9] Karl-Heinz Drechsler and Manfred P. Stadel. A variation of Knoop, Rüthing, and Steffen's "lazy code motion". *SIGPLAN Notices*, 28(5):29–38, May 1993.

[10] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations.* Prentice-Hall, Englewood Cliffs, New Jersey, 1977.

[11] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[12] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. *SIGPLAN Notices*, 27(7):224–234, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[13] Donald E. Knuth. An empirical study of Fortran programs. *Software – Practice and Experience*, 1:105–133, 1971.

[14] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[15] 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.