Value Numbering

Preston Briggs
Keith Cooper
Taylor Simpson

CRPC-TR94517-S
November 1994

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

Value Numbering

Preston Briggs
Tera Computer Company

Keith D. Cooper
L. Taylor Simpson
Rice University

1 Introduction

Value numbering is a code optimization technique with a long history in both literature and practice. 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. In particular, value numbering accomplishes four objectives. It assigns an identifying number (a *value number*) to each value computed in the code in a way that two values have the same number if the compiler can prove they are equal for all possible program inputs. It recognizes certain algebraic identities, like $i = i + 0$ and $j = j \times 1$, and uses them to simplify the code and to expand the set of values known to be equal. It uses value numbers to find redundant computations and remove them. It discovers constant values, evaluates expressions whose operands are constants, and propagates them through the code. There are several ways to accomplish each of these goals, and the methods can be applied across different scopes. This paper describes different techniques for assigning numbers and handling redundancies. It includes experimental evaluation of the relative effectiveness of these different approaches.

In value numbering, the compiler can only assign two expressions the same number if it can prove that they always produce equal values. Two techniques for proving this equivalence appear in the literature.

- The first approach hashes an operator and the value numbers of its operands to produce a number for the resulting value. Hashing ensures that identical values with the same operation receive the same value number. The hash-based techniques are on-line methods that update the program immediately. Their efficiency relies on the expected-case constant-time behavior of hashing.[1]
- The second approach divides the expressions in a procedure into equivalence classes by value, called *congruence classes*. Two values are congruent if they are computed by the same operator and the corresponding operands are congruent. These methods are called *partitioning* algorithms. The partitioning algorithm runs off-line; it must run to completion before updating the code. It can be made to run in $\mathbf{O}(E \log_2 N)$ time, where N and E are the number of nodes and edges in the routine's static single-assignment (SSA) graph.

Once value numbers have been assigned, redundancies must be discovered and removed. Many techniques are possible, ranging from *ad hoc* removal through data-flow techniques.

This paper makes several distinct contributions. These include: (1) an algorithm for hash-based value numbering over a routine's dominator tree, (2) an extension of Alpern, Wegman, and Zadeck's partition-based global value numbering algorithm to perform AVAIL-based removal of expressions, and (3) an experimental comparison of these techniques in the context of an optimizing compiler.

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

[1] Cai and Paige give an off-line, linear time time algorithm that uses multiset discrimination as an alternative to hashing [5].

2 Hash-Based Value Numbering

Cocke and Schwartz describe a local technique that uses hashing to discover redundant computations and fold constants [6]. Each unique value is identified by its *value number*. Two computations in a block have the same value number if they are provably equal. In the literature, this technique and its derivatives are called "value numbering."

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 the computation's value. If it has already been computed in the block, it will already exist in the table. The recomputation can be replaced with a reference to an 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 variables get assigned new values, the compiler must carefully keep track of the location of each expression in the hash table. Consider the code fragment on the left side of Figure 1. At statement (1), the expression $X+Y$ is found in the hash table, but it is available in B and not in A, since A has been redefined. At statement (2), the situation is worse; $X+Y$ is in the hash table, but it is not available anywhere. We can handle this by attaching a list of variables to each expression in the hash table and carefully keeping it up to date.

As described, the technique works for single basic blocks. It can also be applied to an expanded scope, called an extended basic block. An extended basic block is a set of blocks B_1, B_2, $\ldots$, B_n where B_i is the only predecessor of B_{i+1}, for $1 \leq i < n$, and B_1 does not have a unique predecessor. Value numbering over extended blocks works precisely because each value that flows into B_i $(i \neq 1)$ must flow through B_{i-1} and nowhere else. Blocks B_2 through B_n can be processed by initializing their hash tables with the results of processing the previous block. This description suggests that a scoped hash table similar to one that would be used for nested scope languages would be appropriate. Rather than copying the hash table, new entries can be removed after a block is processed. In reality, the compiler must do more than delete information added by the new block. It must restore the name list for each expression and the mapping from variables to value numbers. In practice, this adds a fair amount of overhead and complication to the algorithm, but it does not change its asymptotic complexity.

$$\begin{array}{lll}
 & A \leftarrow X+Y & A_0 \leftarrow X+Y \\
 & B \leftarrow X+Y & B_0 \leftarrow X+Y \\
 & A \leftarrow 1 & A_1 \leftarrow 1 \\
(1) & C \leftarrow X+Y & C_0 \leftarrow X+Y \\
 & B \leftarrow 2 & B_1 \leftarrow 2 \\
 & C \leftarrow 3 & C_1 \leftarrow 3 \\
(2) & D \leftarrow X+Y & D_0 \leftarrow X+Y \\
 & & \\
 & \textbf{Original} & \textbf{SSA Form}
\end{array}$$

Figure 1 Value Numbering Example

2.1 Static Single Assignment Form

Many of the difficulties encountered during value numbering of extended basic blocks can be overcome by constructing the static single assignment (SSA) form of the routine [7]. Each SSA name is assigned a value by only one operation in a routine; therefore, no name is ever reassigned, and no expression ever becomes inaccessible. These advantages become apparent if the code in Figure 1 is converted to SSA form. At statement (1), the expression $X + Y$ can be replaced by A_0 because the second assignment to A was given the name A_1. Similarly, the expression at statement (2) can be replaced by A_0. Also, the transition from single to extended basic blocks is simpler because we can, in fact, use a scoped hash table where only the new entries must be removed.

2.2 Dominator-Tree Value Numbering

Another key feature of SSA form is the information it provides about the way values flow into each block. A value can enter a block B in one of two ways: either it is defined by a ϕ-node at the start of B or it flows through B's parent in the dominator tree (*i.e.,* B's immediate predominator) [9]. Notice that for extended basic blocks, B_{i-1} is the immediate predominator of B_i, for $2 \leq i \leq n$. These observations led us to an algorithm for value numbering over the dominator tree.

The algorithm processes each block by initializing the hash table with the information resulting from value numbering its parent in the dominator tree. To accomplish this, we again use a scoped hash table. The value numbering proceeds by recursively walking the dominator tree. The left side of Figure 2 shows high-level pseudo-code for the algorithm.

To simplify the implementation of the algorithm, the SSA name of the first occurrence of an expression (in this path in the dominator tree) becomes the expression's value number. When a redundant computation of the expression is found, the compiler removes the operation and replaces all uses of the defined SSA name with the expression's value number. The compiler can use this replacement scheme over a limited region of code – in blocks dominated by the operation and in parameters to ϕ-nodes in the dominance frontier of the operation. In both cases, control must flow through the block where the first evaluation occurred (defining the SSA name's value).

The ϕ-nodes require special treatment. If any of the parameters of a ϕ-node have not been assigned a value number, then the compiler must assign a unique new value number to the result. On the other hand, if all of the parameters of the ϕ-node have value numbers, the compiler may be able to simplify the code. The following two conditions guarantee that all ϕ-node parameters in a block have been assigned value numbers.

1. When *value_number* is called recursively for the children of block b in the dominator tree, the children must be processed in reverse postorder. This ensures that all of a block's predecessors are processed before the block itself, unless the predecessor is connected by a back edge relative to the DFS tree.

2. The block must have no incoming back edges.

A ϕ-node can be eliminated if it is meaningless or redundant. A ϕ-node is *meaningless* if all its parameters have the same value number. A meaningless ϕ-node can be removed if the references to its result are replaced with the value number of its input parameters. A ϕ-node is *redundant* if it computes the same value as another ϕ-node in the same block. The compiler can identify redundant ϕ-nodes using a hashing scheme analogous to the one used for expressions. Without additional information about the conditions controlling the execution of different blocks, the compiler cannot compare ϕ-nodes in different blocks.

```
procedure value_number(Block $b$)
    Mark the beginning of a new scope
    for each $\phi$-node $p$ for name $n$ in $b$
        if $p$ is meaningless or redundant
            Put the value number for $p$ into $VN[n]$
            Remove $p$
        else

            $VN[n] \leftarrow n$
            Add $p$ to the hash table
    for each assignment $a$ of the form $n \leftarrow exp$ in $b$
        if $exp$ is found in the hash table
            Put the value number for $exp$ into $VN[n]$
            Remove $a$
        else

            $VN[n] \leftarrow n$
            Add $exp$ to the hash table
    for each successor $s$ of $b$
        Adjust the $\phi$-node inputs in $s$
    for each child $c$ of $b$ in the dominator tree
        value_number($c$)
    Clean up the hash table after leaving this scope
```

After SSA Construction

```
procedure rename_and_value_number(Block $b$)
    Mark the beginning of a new scope
    for each $\phi$-node $p$ for name $n$ in $b$
        if $p$ is meaningless or redundant
            Push the value number for $p$ onto $S[n]$
            Remove $p$
        else
            Invent a new value number $v$ for $n$
            Push $v$ onto $S[n]$
            Add $p$ to the hash table
    for each assignment $a$ of the form $n \leftarrow exp$ in $b$
        if $exp$ is found in the hash table
            Push the value number for $exp$ onto $S[n]$
            Remove $a$
        else
            Invent a new value number $v$ for $n$
            Push $v$ onto $S[n]$
            Add $exp$ to the hash table
    for each successor $s$ of $b$
        Adjust the $\phi$-node inputs in $s$
    for each child $c$ of $b$ in the dominator tree
        rename_and_value_number($c$)
    Clean up the hash table after leaving this scope
    for each $\phi$-node or assignment $a$ in the original $b$
        for each name $n$ defined by $a$
            pop $S[n]$
```

During SSA Construction

Figure 2 Dominator-Tree Value-Numbering Algorithms

After value numbering the ϕ-nodes and instructions in a block, the algorithm visits each successor block and updates any ϕ-node inputs that come from the current block. This involves determining which ϕ-node parameter corresponds to input from the current block and overwriting the parameter with its value number. Notice the resemblance between this step and the corresponding step in the SSA construction algorithm. This step must be performed before value numbering any of the block's children in the dominator tree, if the compiler is going to analyze ϕ-nodes.

To illustrate how the algorithm works, we will apply it to the code fragment in Figure 3. The first block processed will be B_1. Since none of the expressions on the right-hand sides of the assignments have been seen, the names u_0, v_0, and w_0 will be assigned their SSA name as their value number.

The next block processed will be B_2. Since the expression $c_0 + d_0$ was defined in block B_1 (which dominates B_2), we can delete the two assignments in this block by assigning the value number for both x_0 and y_0 to be v_0. Before we finish processing block B_2, we must fill in the ϕ-node parameters in its successor block, B_4. The first argument of ϕ-nodes in B_4 corresponds to input from block B_2, so we replace u_0, x_0, and y_0 with u_0, v_0, and v_0, respectively.

Block B_3 will be visited next. Since every right-hand-side expression has been seen, we assign the value numbers for u_1, x_1, y_1 to be u_0, w_0, and w_0, respectively, and remove the assignments. To finish processing B_3, we fill in the second parameter of the ϕ-nodes in B_4 with u_0, w_0, and w_0, respectively.

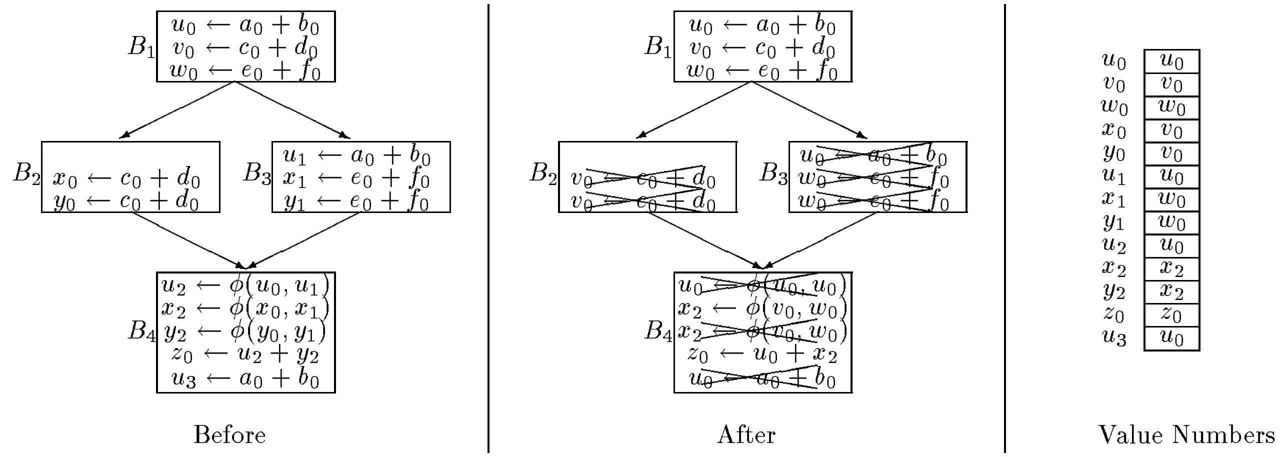


Figure 3 Dominator-Tree Value-Numbering Example

The final block processed will be B_4. The first step is to examine the ϕ-nodes. Notice that we are able to examine the ϕ-nodes only because we processed B_1's children in the dominator tree (B_2, B_3, and B_4) in reverse postorder and because there are no back edges flowing into B_4. The ϕ-node defining u_2 is meaningless because all its parameters are equal. Therefore, we eliminate the ϕ-node by assigning u_2 the value number u_0. Notice that this ϕ-node was made meaningless by eliminating the only assignment to u in a block with B_4 in its dominance frontier. In other words, when we eliminate the assignment to u in block B_3, we eliminate the reason the ϕ-node for u was inserted during the construction of SSA form. The second ϕ-node combines the values v_0 and w_0. Since this is the first appearance of a ϕ-node with these parameters, x_2 is assigned its SSA name as its value number. The ϕ-node defining y_2 is redundant because it is equal to x_2. Therefore, we eliminate this ϕ-node by assigning y_2 the value number x_2. When processing the assignments in the block, we replace each operand by its value number. This results in the expression $u_0 + x_2$ in the assignment to z_0. The assignment to u_3 is eliminated by giving u_3 the value number u_0.

Notice that if we applied single-basic-block value numbering to this example, the only redundancies we could remove are the assignments to y_0 and y_1. If we applied extended-basic-block value numbering, we could also remove the assignments to x_0, u_1, and x_1. Only dominator-tree value numbering can remove the assignments to u_2, y_2, and u_3.

2.3 Incorporating Value Numbering into SSA Construction

We have described dominator-tree value numbering as it would be applied to routines already in SSA form. However, it is possible to incorporate value numbering into the SSA construction process. There is a great deal of similarity between the value numbering process and the renaming process during SSA construction [7, section 5.2]. The renaming process can be modified as follows to accomplish renaming and value numbering simultaneously:

- For each name in the original program, a stack is maintained which contains subscripts used to replace uses of that name. To accomplish value numbering, these stacks will contain value numbers.

- Before inventing a new name for each ϕ-node or assignment, we first check if it can be eliminated. If so, we push the value number of the ϕ-node or assignment onto the stack for the defined name.

For comparison, the algorithms for dominator-tree value numbering after SSA construction and during SSA construction are presented side by side in Figure 2.

3 Global Value Numbering

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 [7]. 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. 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.[2]

Partitioning and renumbering alone will not improve the running time of the routine; the compiler must also find and remove the redundant computations. We explore three possibilities: dominator-based removal, AVAIL-based removal, and partial redundancy elimination.

3.1 Dominator-Based Removal

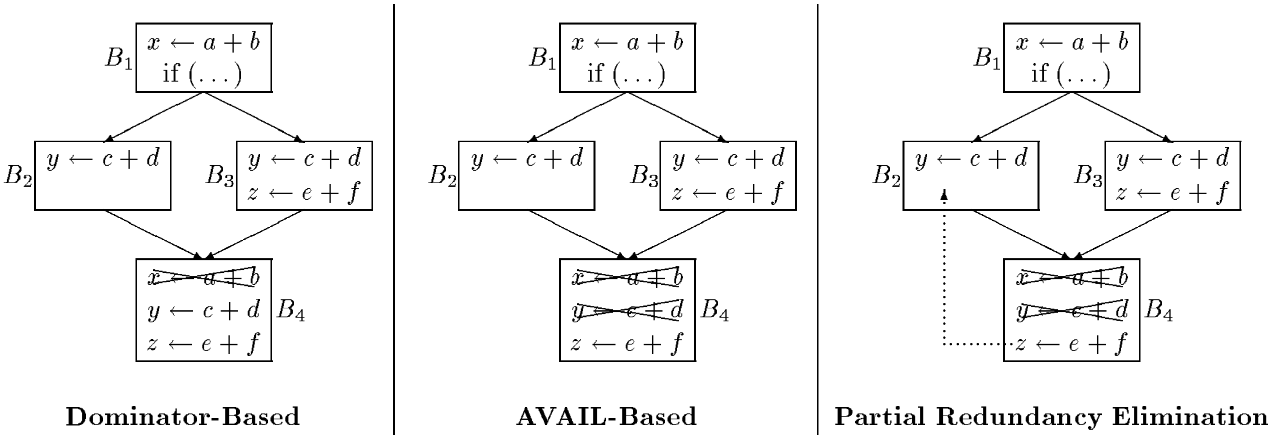
Alpern, Wegman, and Zadeck suggest removing computations that are dominated by another member of the congruence class [3]. The computation of x in Figure 4 is a redundancy that this method can eliminate. Since the computation of x in block B_1 dominates the computation in block B_4, the second computation can be removed.

To perform dominator-based removal, the compiler considers each congruence class and looks for pairs of members where one dominates the other. If we bucket sort the members of the class based on the preorder index in the dominator tree of the block where they are computed, then we can efficiently compare adjacent elements in the list and decide if one dominates the other. This decision is based on an ancestor test in the dominator tree. The entire process can be done in time proportional to the size of the congruence class.

3.2 AVAIL-Based Removal

The classical approach to redundancy elimination is to remove computations that are in the set of available expressions (AVAIL) at the point where they appear in the routine [2]. This approach uses data-flow analysis to determine the set of expressions available along all paths from the start of the routine. Notice that the calculation of x in Figure 4 will be removed because it is in the AVAIL set. In fact, any computation that

[2] Rosen, Wegman, and Zadeck describe a technique called *global value numbering* [11]. It is an interesting and powerful approach to redundancy elimination, but it should not be confused with global partitioning.



Dominator-Based **AVAIL-Based** **Partial Redundancy Elimination**

Figure 4 Eliminating Redundancies

would be removed by dominator-based removal would also be removed by AVAIL-based removal. This is because any block that dominates another is on all paths from the start of the routine to the dominated block. However, there are improvements that can be made by the AVAIL-based technique that are not possible using dominators. In Figure 4, y is calculated in both B_2 and B_3, so it is in the AVAIL set at B_4. Thus, the calculation of y in B_4 can be removed. However, since neither B_2 nor B_3 dominate B_4, dominator-based removal could not remove y.

Properties of the partitioning algorithm let us simplify the formulation of AVAIL. The traditional data-flow equations deal with lexical *names* while our equations deal with *values*. We need not consider the killed set for a block because no values are redefined in SSA form, and partitioning preserves this property. Consider the code fragment on the left side of Figure 5. Under the traditional data-flow framework, the assignment to X would kill the Z expression. However, if the assignment to X caused the two assignments to Z to have different values, then they would not be congruent to each other, and they would be assigned different names. Since the partitioning algorithm has determined that the two assignments to Z are congruent, the second one is redundant and can be removed. The only way the intervening assignment will be given the name X is if the value computed is congruent to the definition of X that reaches the first assignment to Z. The data-flow equations we use are shown in Figure 5.

$$\begin{aligned} Z &\leftarrow X + Y \\ X &\leftarrow \ldots \\ Z &\leftarrow X + Y \end{aligned}$$

Example

$$\text{AVAIL\_in}_i = \begin{cases} \emptyset, & \text{if } i \text{ is the entry block} \\ \displaystyle\bigcap_{j \in \text{pred}(i)} \text{AVAIL\_out}_j, & \text{otherwise} \end{cases}$$

$$\text{AVAIL\_out}_i = \text{AVAIL\_in}_i \cup \text{defined}_i$$

Data-Flow Equations

Figure 5 AVAIL-Based Removal

3.3 Partial Redundancy Elimination

Partial redundancy elimination (PRE) is an optimization introduced by Morel and Renvoise [10]. Partially redundant computations are redundant along some, but not necessarily all, execution paths. Notice that the computations of x and y in Figure 4 are redundant along all paths to block B_4, so they will be removed by PRE. On the other hand, the computation of z in block B_4 cannot be removed using AVAIL-based removal, because it is not available along the path through block B_2. The value of z is computed twice along the path through B_3 but only once along the path through B_2. Therefore, it is considered partially redundant. PRE can move the computation of z from block B_4 to block B_2. This will shorten the path through B_3 and leave the length of the path through B_2 unchanged. PRE has the added advantage that it moves invariant code out of loops.

4 Comparing the Techniques

Assume that X and Y are known to be equal in the code fragment in the left column of Figure 6. Then the partitioning algorithm will find A congruent to B and C congruent to D. However, a careful examination of the code reveals that if X is congruent to Y, then A, B, C, and D are all zero. The partitioning technique will not discover that A and B are equal to C and D, and it also will not discover that any of the expressions are equal to zero. On the other hand, the hash-based approach will 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 6. If we apply any of the hash-based approaches to this example, none of them will be able to prove that X_2 is equal to Y_2. This is because at the time a value number must be assigned to X_2 and Y_2, none of these techniques have visited X_3 or Y_3. They must therefore assign a unique value number to X_2 and Y_2. However, the partitioning technique will prove that X_2 is congruent to Y_2 (and thus X_3 is congruent to Y_3). The key feature of the partitioning algorithm which makes this possible is its initial optimistic assumption that all values defined by the same operator

```
$A \leftarrow X - Y$
$B \leftarrow Y - X$
$C \leftarrow A - B$
$D \leftarrow B - A$
```

Improved by Hash-Based Techniques

```
$X_0 \leftarrow 1$
$Y_0 \leftarrow 1$
while (...)
    $X_2 \leftarrow \phi(X_0, X_3)$
    $Y_2 \leftarrow \phi(Y_0, Y_3)$
    $X_3 \leftarrow X_2 + 1$
    $Y_3 \leftarrow Y_2 + 1$
```

Improved by Partitioning Techniques

Figure 6 Comparing the Techniques

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.

We should point out that eliminating more redundancies does not necessarily result in reduced execution time. When the compiler removes an operation in this fashion, it necessarily extends the live range of some other value, possibly hurting register allocation. On the other hand, the live range of the operands of the removed operation may be shortened, possibly improving register allocation. Our experience suggests that the negative impact on register allocation is negligible.

5 Experimental Results

Even though we can prove that each of the three global techniques and each form of hash-based value numbering is never worse than its predecessor, 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 routines drawn from the Spec benchmark and from Forsythe, Malcolm, and Moler's book on numerical methods [8].

The complete results are shown in Table 1. Each column represents dynamic counts of ILOC operations. Routines are optimized using the sequence of global reassociation [4], value numbering (the type is indicated in the column header), global constant propagation [12], global peephole optimization, dead code elimination [7, Section 7.1], copy coalescing, and a pass to eliminate empty basic blocks. All forms of value numbering were performed on the SSA form of the routine.

The first section of Table 1 compares the hash-based techniques. On average, code optimized using extended basic blocks performs 12.2% better than code optimized using single basic blocks; dominator-tree value numbering improves the code by another 5.4%. In our experiment, dominator-tree value numbering performs slightly better on average than global value numbering with dominator-based removal.

The second section of Table 1 compares the partitioning techniques. The AVAIL-based technique improves the code by 0.3% compared to the dominator-based technique, and PRE improves the code by another 12.0%. The ability of PRE to move invariant code out of loops contributes greatly to this improvement.

6 Summary

In this paper, we study a variety of redundancy elimination techniques. We have introduced a technique for applying hash-based value numbering to a routine's dominator tree. This technique is competitive in practice with the global value numbering techniques, while being faster and simpler. Additionally, we have improved the effectiveness of global value numbering by removing computations based on available values rather than dominance information and by applying partial redundancy elimination.

We presented experimental data comparing the effectiveness of each type of value numbering in the context of our optimizing compiler. The data indicates that our extensions to the existing algorithms can produce significant improvements in execution time.

routine		*Hash-Based*			*Global*	
	single	*extended*	*dominator*	*dominator*	*AVAIL*	*PRE*
tomcatv	436863008	417095649	411877418	411878446	411878446	187980202
twldrv	83877543	72748320	69913464	69913456	69850544	69236916
gamgen	156608	138200	138199	138199	138199	104632
iniset	93438	65252	56672	56672	56672	47426
deseco	18605	15920	15142	15215	15137	12941
prophy	6294	5446	4569	4569	4429	3804
pastem	5582	4606	3850	3850	3850	3477
debflu	5502	5086	4797	4797	4797	4622
bilan	4613	4304	3757	3757	3731	3132
paroi	4307	4035	3973	3973	3981	3614
fpppp	4046	4046	4046	4046	4046	4046
repvid	3997	3325	2745	2758	2731	2458
inithx	3829	3337	3074	3074	3074	2741
debico	3367	3120	3095	3095	3069	2637
integr	3313	2984	2640	2640	2640	2312
sgemv	1892	1690	1687	1687	1687	1002
sgemm	1614	1494	1493	1493	1493	954
inideb	1501	955	942	942	942	774
cardeb	1145	1076	1029	1029	1029	785
saxpy	1015	915	915	915	915	524
ddeflu	836	797	759	757	756	713
supp	831	831	828	828	822	822
fmtset	677	476	451	451	451	406
subb	636	636	636	636	636	636
ihbtr	496	437	435	435	435	410
x21y21	379	319	319	319	319	239
drepvi	369	326	276	277	275	273
saturr	320	331	328	328	320	317
efill	311	272	235	235	232	230
fmtgen	292	243	219	219	211	188
si	249	177	176	177	177	166
heat	209	208	177	176	176	177
dcoera	192	174	165	165	165	165
lclear	186	146	146	146	146	109
orgpar	173	147	144	144	144	120
yeh	169	144	132	132	132	132
colbur	152	134	121	123	123	123
coeray	139	121	112	112	112	104
drigl	120	117	112	112	112	111
lissag	116	100	100	100	100	89
aclear	114	90	90	90	90	69
sortie	92	90	84	84	84	83
sigma	55	55	55	55	55	55
svd	7230	6365	5998	6012	5990	4472
fmin	2075	1135	946	946	946	871
zeroin	1406	912	836	836	836	742
spline	1070	921	907	907	907	759
decomp	932	763	756	756	749	645
fehl	753	705	705	705	705	516
urand	227	222	221	221	221	221
solve	221	195	197	199	199	179
seval	114	107	80	80	80	79
rkf45	58	58	58	58	58	58

Table 1 Experimental Results

7 Acknowledgments

Our interest in this problem began with suggestions from both Jonathan Brezin of IBM and Bob Morgan of DEC. Independently, they suggested that we investigate value numbering over dominator regions. Bruce Knobe of Intermetrics also urged us to look at extending value numbering to ever larger regions.

Our colleagues in the Massively Scalar Compiler Project at Rice have played a large role in this work. In particular, we owe a debt of gratitude to Cliff Click, Tim Harvey, Linlong Jiang, John Lu, Philip Schielke, Rob Shillner, Lisa Thomas, Linda Torczon, and Edmar Wienskoski. Without their tireless implementation efforts, we could not have completed this study.

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] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[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] Jiazhen Cai and Robert Paige. "Look Ma, no hashing, and no arrays neither". In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 143–154, Orlando, Florida, January 1991.

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

[7] 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.

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

[9] Matthew S. Hecht. *Flow Analysis of Computer Programs.* Programming Languages Series. Elsevier North-Holland, Inc., 52 Vanderbilt Avenue, New York, NY 10017, 1977.

[10] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[11] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, San Diego, California, January 1988.

[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.